

Design, Benchmark and Determine the Performance of a Low Cost 3D Printed 6 Degrees of Freedom (DOF) Robotic Arm

A THESIS

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE AWARD OF THE DEGREE

Bachelor of Engineering (Mechatronics)

By

BROCK COOPER

SID:5791340

SUPERVISOR: DR PRASHAN PREMARATNE



UNIVERSITY
OF WOLLONGONG
AUSTRALIA

SCHOOL OF MECHANICAL, MATERIALS,
MECHATRONIC AND BIOMEDICAL ENGINEERING

NORTHFIELDS AVE, WOLLONGONG NSW 2522

May 22, 2022

Abstract

This thesis explores the development of a 3D printed 6 DOF robotic arm (BCR1), including designing a hardware controller alongside integrating an open-source median controller Robot Operating System.

The project investigates keeping the robotic arm low-cost using readily off the shelf components and rapid prototyping manufacturing techniques, Whilst keeping the design compact and similar resemblance to existing industrial robotic arms. The design process used was broken down into three key areas; the arm design, hardware control box (motors, drivers, microcontroller) and software control integration (Robot Operating System).

The BCR1 was developed to demonstrate the feasibility of a low-cost 3D printed robotic arm using a repeatability benchmark analysis, using a custom control program written to travel through a series of trajectory positions repeated for 30 cycles for each of the x, y and z-axis.

The most repeatable axis on the arm was determined to be the x-axis with repeatability within $1.136mm$. Through the benchmark analysis, the total repeatability of the arm is approximately $\pm 5.488mm$.

The BCR1 constructed from open-source hardware and software has kept future development open for new possibilities for improvements.

Acknowledgements

Firstly I would like to acknowledge my supervisor Dr Prashan Premaratne for whom this project wouldn't exist if it weren't for him constantly encouraging me to strive and do my best to achieve the highest results. I now find myself very passionate about learning new ways of integrating and controlling robotic arms whilst finding applications that they could be applied.

I want to thank my parents for supporting me throughout my life so far, especially the last five and a half years of my studies, without them, I wouldn't be who I am today.

To my brother (Chad), I want to thank you for always supporting me and encouraging me. He grounded me and always helped me reflect on the importance of my studies.

To my girlfriend (Hannah), thank you for your patience throughout this long journey and for helping me stay positive and achieve my goals.

To one of my best friends Ben Van Magill, who I look up to as a mentor. He has always inspired me to strive and encouraged me to learn new things that I wouldn't have pushed myself to get where I am today.

Statement of Originality

I, Brock Cooper, declare that this thesis, submitted as part of the requirements for the award of Bachelor of Engineering, in the School of Mechanical, Materials, Mechatronic and Biomedical Engineering, University of Wollongong, is wholly my own work unless otherwise referenced or acknowledged. The document has not been submitted for qualifications or assessment at any other academic institution.

As author of this thesis, I also hereby grant, subject to any prior confidentially agreements, SECTE permission, to use, distribute, publish, exhibit, record, digitize broadcast, reproduce and archive this work for the purposes of further research and teaching.

(strike out that which does not apply)

This thesis;

~~IS~~

IS NOT

subject to a prior confidentially agreement.

Brock Cooper
SID: 5791340

Dr Prashan Premaratne

List of Figures

2.1	Example ROS Network Configuration	4
2.2	Kinematic Representation	5
2.3	RViz Simulator	6
2.4	Catkin Workspace Folder Structure	9
2.5	12V DC Encoder Motor	10
2.6	Nemba 17 Stepper Motor and Coils Configuration	11
2.7	Servo Motor and PWM Position Control	11
2.8	ATmega2560 Pinout	12
2.9	L298N Full H-Bridge Motor Driver	13
2.10	Mechanical Limit Switch on 3D Printer (x-axis)	14
2.11	Optical Sensor Endstop on 3D Printer (z-axis)	14
2.12	Hall Effect Sensor	15
2.13	Robot Joints [27]	15
2.14	Desktop FDM 3D Printing	18
2.15	Pose Repeatability and ISO Cube	19
3.1	Existing 4 DOF Robotic Arm	21
3.2	Model Design	22
3.3	Coordinate Frame	23
3.4	Motor Types	25
3.5	Motor Torque Diagram	25
3.6	Link 1 Design	26
3.7	Bearing Arrangement	27
3.8	Motor Arrangement	28
3.9	Coupling Designs	28
3.10	Hall Effect Sensor Joint 2	29
3.11	Robot End Effector Position in Home Position	30
3.12	Control Box	31
3.13	Arm Design (BCR1) to be ROS enabled	32
3.14	MoveIt Setup Assistant	32
3.15	Arm being controlled in RViz using inverse kinematics	33
3.16	ROS Active Nodes for Demo Controller using rqt Topics	34

3.17	ROS Control Data Flow [16]	35
3.18	ROS Active Nodes for BCR1 using rqt Topics	35
3.19	Example Experimental Apparatus	36
4.1	Experimental Apparatus Setup	38
4.2	Dial Indicator Measurements	39
4.3	Repeatability in X-axis	40
4.4	Repeatability in Y-axis	41
4.5	Repeatability in Z-axis	41
4.6	Combined Resultant Repeatability of BCR1	42
4.7	Repeatability in Each Plane 3D Space	42
C.1	Nodes and Topics Overview for Demo Controller	58
C.2	Nodes and Topics Overview for BCR1	59
H.1	Control Box Wiring Diagram	81

List of Tables

2.1	PID Term Characteristics [29]	16
2.2	Ziegler–Nichols Method [30]	16
3.1	Denavit–Hartenberg Parameters for BCR1	23
3.2	Motor Specifications	26
4.1	Standard Deviation of X,Y & Z Axis	41
4.2	BCR1 Costing	43
D.1	List of ROS Distributions and Lifespan	60

Nomenclature

Abbreviation	Description
CAD	Computer Aided Design
DOF	Degrees of Freedom
FDM	Fused Deposition Modelling (3D printing)
ROS	Robot Operating System
URDF	Unified Robotic Description Format
RViz	3D Visualization Tool for ROS
MoveIt	Motion Planning Framework for ROS
XML-RPC	Remote procedure call protocol XML passed via HTTP
CNC	Computer Numerical Control
DC	Direct Current
PWM	Pulse-Width Modulation
TCP	Tool Center Point
PID	Proportional Integral Derivative
GUI	Graphical User Interface
SS	Steady State
CL	Closed Loop
BCR1	3D Printed Robot Arm Name (Brock Cooper Robo 1)

List of Changes

Section	Statement of Changes	Page Number
Thesis Title	Changed the title to match new added objectives	-
Abstract	Altered to summarize what has been done	i
Chapter 1	Added new objectives and reworded	1
Chapter 2	Addition of Figures Equations and Tables	3
Chapter 2	General Sentence Structure Editing	3
Chapter 2	Addition of extra research material	3
Chapter 3	Addition of new material robot design	18
Chapter 3	General editing and sentence structure	18
Chapter 3	Restructuring of subchapters	18
Chapter 4	Addition of results and experimental data	36
Chapter 5	Reworded conclusion for achieved objectives	42

Contents

Abstract	i
Acknowledgements	ii
Statement of Originality	iii
Nomenclature	vii
List of Changes	viii
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	2
2 Literature Review	3
2.1 Introduction	3
2.2 Motion Planning and Software	3
2.2.1 ROS Types	4
2.2.2 MoveIt	4
2.2.3 RViz	6
2.2.4 Rosserial	7
2.2.5 RQT Graphic User Interface	7
2.2.6 ROS Graph Level	7
2.2.7 ROS Controllers	7
2.2.7.1 Controllers	8
2.2.7.2 Hardware Interfaces	8
2.2.8 Catkin Workspace	8
2.3 Hardware	9
2.3.1 Motors	9
2.3.1.1 12V DC Encoder Geared Motor	9
2.3.1.2 Geared Stepper Motor	10
2.3.1.3 Servo Motor	11
2.3.2 Control Board (Arduino ATmega 2560)	12

2.3.3	L298 Motor Controller	12
2.3.4	Endstops	13
2.3.4.1	Mechanical	13
2.3.4.2	Optical	14
2.3.4.3	Hall Effect	14
2.4	Types of Robotic Joints	15
2.4.0.1	Prismatic	15
2.4.0.2	Revolute	15
2.5	PID Controller	15
2.6	Rapid Prototyping - Manufacturing	17
2.7	Repeatability Testing	18
2.7.1	Pose Repeatability	19
3	Design and Experimental Setup	20
3.1	Introduction	20
3.2	Realisation of Robot Arm Design	20
3.2.1	Robot Arm Design (BCR1)	21
3.2.2	BCR1 Kinematics	22
3.2.3	Motor Selection Criteria	24
3.3	Link Design	26
3.3.1	Bearing Arrangement	27
3.3.2	Motor Arrangement	27
3.3.3	Coupling Design	28
3.3.4	Homing Limit Sensors	29
3.3.5	BCR1 Forward Kinematics Home Position	29
3.4	Hardware Control Box	30
3.5	ROS Implementation	31
3.5.1	URDF Robot MoveIt Package	31
3.5.2	Simulating in RViz	33
3.5.3	ROS Hardware Interface	34
3.5.4	Repeatability Testing Design	35
4	Results, Analysis and Evaluation	38
4.1	Experimental Setup	38
4.2	Repeatability Results	40
4.2.1	Repeatability Analysis	41
4.3	Costing	43
5	Discussion and Conclusion	44
5.1	Key Findings	44

5.2	Arm Design	45
5.3	Conclusion	45
6	Future Work and Recommendations	46
6.1	BCR1 improvements	46
6.2	Software Development	46
6.3	Integration Into Applications	47
	References	48
A	Original Project Specifications	52
B	Logbook Summary Sheet	57
C	ROS Flowchart	58
D	ROS Distributions	60
E	MATLAB Code	61
E.1	BCR1 Forward Kinematics	61
E.2	BCR1 Repeatability Point Cloud Plot	62
F	Hardware Code	64
F.1	ROS Hardware Interface	64
F.1.1	bcr1_hw_interface.cpp	64
F.1.2	bcr1_hw_main.cpp	65
F.1.3	bcr1_hw_interface.h	67
F.2	Arduino Code	68
F.2.1	Main Program	68
F.2.2	Header Files	72
F.2.2.1	armCmd.h	72
F.2.2.2	bcr1Telemetry.h	73
F.2.3	Python MoveIt Commander	74
G	CAD Drawings	77
H	Wiring Schematic	81

Introduction

Robots are expanding rapidly in the range of applications they are being applied towards, from remote exploration to agriculture, search and rescue, and assembly lines. Robotic manipulators are becoming more widely used in aiding in real-world activities. Therefore, it is essential to use a benchmark that provides dual purpose for developers to improve their design whilst allowing the end-user to determine the appropriate armature for the desired application[1]. Robotic arms have been applied to industrial applications for quite some time now. The first industrial arm introduced by UNIMATE in 1961 has evolved into the robotic arm known as the PUMA arm [2]. Although technology advancements have progressed with new, more accessible, readily available manufacturing techniques, it has opened up the possibilities for consumer-based designs and desktop robotic arm applications at a more affordable price.

1.1 Motivation

The primary motivation behind the project is to design a low-cost 6 DOF robotic arm that has been inspired by an existing 4 DOF 3D printed robotic arm, improving and expanding the capabilities of the current design reflected in the new robotic arm. The robotic arm will also provide data that can be used to determine the new design's performance through a benchmarking methodology, which could evaluate best-suited tasks and environments. This will open up the opportunity for future development by analysing potential flaws that the new design may have. The project will use all open-source hardware and software, expanding the opportunity for future research by using open-source hardware. This will allow the integration of new sensors that could enhance the arm's functionality. Setting up a control interface and having a ground basis designed that could be used and applied to new designs in the future with the ROS integration more efficient and less time-consuming.

1.2 Objectives

This report aims to develop a new low-cost 3D printed robotic arm by inspiring an existing design constructed with a mix of 3 motors: a servo motor, stepper motor, and DC encoder motors. Achieving these results in a limited time will require the project to be broken down into sub-components, with quite a significant focus on applying a suitable control method that can manipulate multiple joints simultaneously. The controller can then carry out a set of specific measurement tasks to obtain data on the particular arm which could improve the design. Benchmarking criteria for robotic arms are in-depth and have numerous test parameters that demonstrate the arms' capabilities and operational functionality. Due to the limited time on this research project with various aspects, only one parameter will be selected to focus on in detail.

Literature Review

2.1 Introduction

This chapter breaks down the project into three main sections: the software and control theory, the hardware used and required, and research into existing testing methodologies. The information acquired through research from credible resources will aid in the successful implementation of the design to gather valuable data that can be critically synthesised. Due to the nature of the software, ROS is quite an in-depth and steep learning curve. The main focus of this chapter will be to develop an understanding of the functionality of ROS (open-source Robot Operating System).

2.2 Motion Planning and Software

ROS isn't a typical operating system yet is a flexible framework, providing a communication layer that allows seamless communication between a cluster/network of machines above a host machine. It offers a collection of libraries, tools and conventions for simplifying complex tasks [3].

ROS can be summarized as an operating system to achieve these main set criteria being:

- Peer-to-peer
- Tools-based
- Multi-lingual
- Thin
- Free and Open-Source

A peer-to-peer network enables the robot to have on-board machines that act as slave devices to receive data processed from off-board machines with higher processing power. This may be processing data such as image or speech recognition. The data sends from a host machine through a wireless LAN network. There can

be multiple hosts and slave devices that would be connected via ethernet to each other, as seen in Figure 2.1.

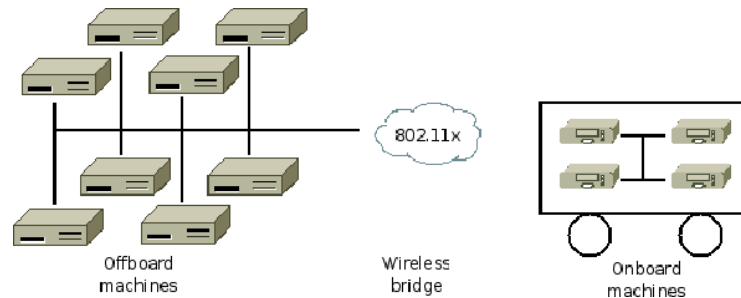


Figure 2.1: Example ROS Network Configuration [4]

ROS has been developed for multi-lingual use, meaning that it can support multiple coding languages such as C++, Python, Octave and LISP [4, 5]. With ROS designed as a messaging layer with configuration files stored and transferred as the XML-RPC, files are stored as an XML file, which is a markup language that is then transmitted using the HTTP protocol [6].

The operating system is designed to be tool-based and thin so that the system is built with smaller components rather than a monolithic development approach. While keeping a narrow approach, ROS uses code from various open-source projects such as simulators, drivers, navigation systems, vision algorithms, and many more. The thin nature of the program allows for keeping libraries and small executables separate, which makes it easier for testing. The open-source code for ROS has allowed for development, and debugging is available on GitHub [4].

2.2.1 ROS Types

The current version of ROS runs only on Unix-based platforms and is being tested primarily on Ubuntu and Mac OS X operating systems. There is potential for ROS to be installed on windows, but it will require Windows 10 IoT Enterprise and hasn't been thoroughly tested. Due to various Unix-based platforms, ROS has different versions called distributions. This is a set of packages designed for different versions of Linux, the current and previous versions can be seen in Table D.1 which can be seen to have a release date and end of life (EOL) date [7].

2.2.2 MoveIt

MoveIt is a motion planning software package that runs on top of ROS and is specifically designed to manipulate robot arms by calculating the necessary trajectories (this is the path followed by the end effector to a specific location). This software

utilises ROS messages and some common tools, specifically the robot visualiser RViz, and URDF robot format. This package implements complex previously developed algorithms that allow robots to calculate inverse kinematics, and forwards kinematics which makes for easy trajectory planning [8]. Forward kinematics determines the position and orientation of the tooltip once the parameters for each joint have been established. The equation for determining forward kinematics is seen in Equation 2.1 [9].

$$T_{k-1}^k = \begin{bmatrix} \cos(\theta_k) & -\cos(\alpha_k) * \sin(\theta_k) & \sin(\alpha_k) * \sin(\theta_k) & a_k * \cos(\theta_k) \\ \sin(\theta_k) & \cos(\alpha_k) * \cos(\theta_k) & -\sin(\alpha_k) * \cos(\theta_k) & a_k * \sin(\theta_k) \\ 0 & \sin(\alpha_k) & \cos(\alpha_k) & d_k \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.1)$$

Where:

- k Kinematic parameter for a specific joint
- θ_k Rotational angle for a joint about the z-axis
- α_k Rotational angle for the joint about the x-axis
- d_k Length translated along the z-axis
- a_k Length translated along the x-axis

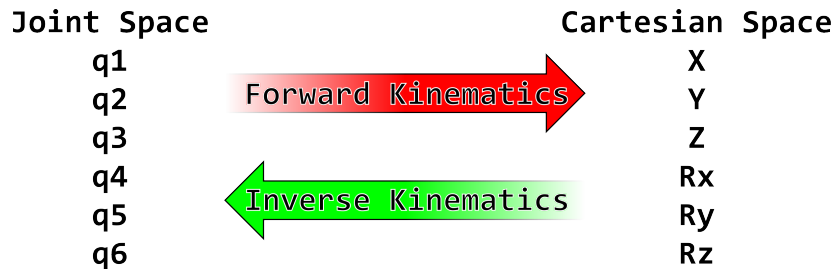


Figure 2.2: Kinematic Representation

A representation of forward kinematics and inverse can be seen in Figure 2.2 where joint space can be used to calculate cartesian space, using the joints being q1-q6 as the input. On the other hand, cartesian space can be converted into joint space, with multiple solutions depending on the position and orientation of the tool as the input.

To calculate the final tooltip position and orientation, it is the relation between the tool and base; by using the calculated relationship seen in Equation 2.1 the tool position can be determined by multiplying all the kinematic matrices seen in Equation 2.2.

$$T_{base}^{tool} = T_0^1 * T_1^2 * T_2^3 \dots T_{n-1}^n = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ r_{41} & r_{42} & r_{43} & r_{44} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & P_x \\ r_{21} & r_{22} & r_{23} & P_y \\ r_{31} & r_{32} & r_{33} & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.2)$$

2.2.3 RViz

RViz is a 3D visualisation simulation environment that allows developers/users to easily view the robot and see and how it's functioning in the environment in real-time, allowing for easier debugging, especially for complex quaternions and coordinate frames. RViz also shows the current arm position and configuration on a virtual model in the workspace, which can display live the position of the robot by using the ROS topics that publish sensor data. This can be seen in Figure 2.3 [10, 11].

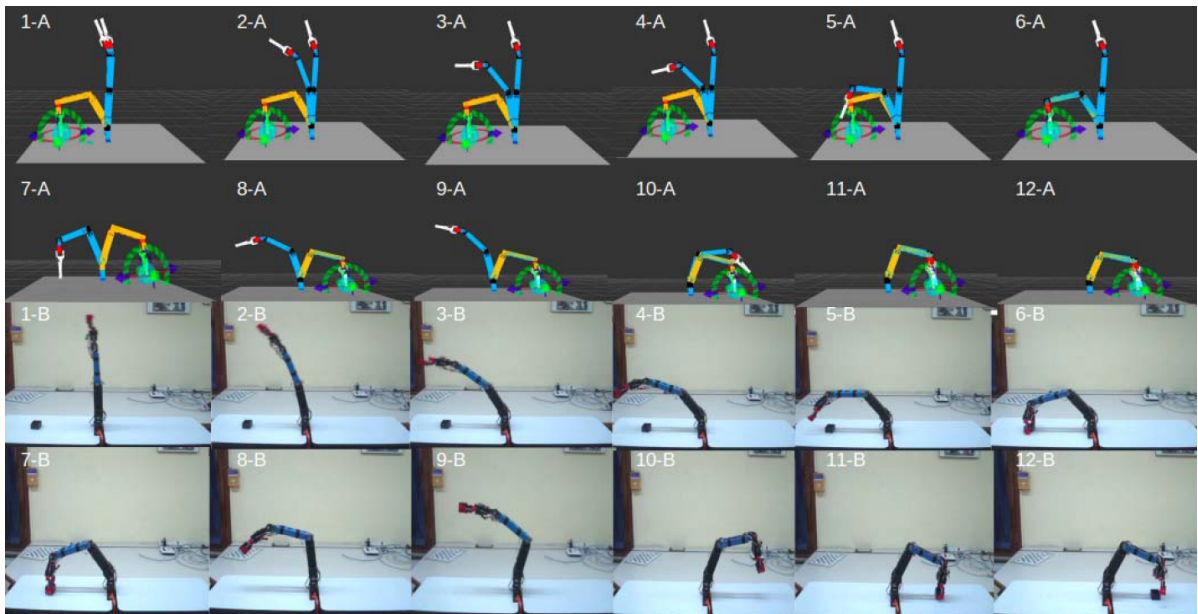


Figure 2.3: RViz simulator, sequence images (1-12)A display's the plan and execution path of the arm in RViz where images (1-12)B display's the current physical state of the robot corresponding to the current position in RViz [10]

2.2.4 Rosserial

The roserial package/library wraps serialised ROS messages and multiplexing multiple topics to send over a character device through a serial port or a network socket. An Arduino C++ library for Arduino is used to receive this data in the correct formats being a Float64, String or Arrays, which can then control real-world interfaces through a microcontroller such as an ATmega2560 [12].

2.2.5 RQT Graphic User Interface

ROS has a package called rqt that is a Qt-based framework for GUI development; there are many GUI's already readily available; these existing packages provide tools such as graphing (Group: Introspection), editing parameters for controllers (Group: Configuration), view topics and messages (Group: Topics), which can assist in setting up or observing robot states [13, 14].

2.2.6 ROS Graph Level

ROS has multiple processes which are processing data together. The breakdown of graph concepts are nodes, master node, messages, topics, roscore and rosout. The nodes are executable files that ROS uses to communicate with other nodes. Messages are a datatype used for publishing and subscribing data to a specific topic. Topics used are for nodes to publish and send messages/data to a topic or subscribe and receive data from a topic. The master node helps other nodes locate each other, which initiates with the roscore command; this starts the master node. The last main parameter is rosout, which enables a console log reporting mechanism [15].

2.2.7 ROS Controllers

The two main focuses with ROS control is the controllers that work in conjunction with the hardware interfaces, this allows the simulation environment to communicate with the physical hardware being the robot motors, actuators or other devices [16].

2.2.7.1 Controllers

A list of available ROS controllers:

- joint_state_controller
- position_controllers
- velocity_controllers
- effort_controllers
- joint_trajectory_controllers

2.2.7.2 Hardware Interfaces

A list of available ROS Hardware Interfaces:

- Joint Command Interface
 - Effort Joint Interface
 - Velocity Joint Interface
 - Position Joint Interface
- Joint State Interfaces
- Actuator State Interfaces

2.2.8 Catkin Workspace

ROS requires a catkin workspace which is a directory/folder structure that creates or modifies existing catkin packages, this simplifies the build and installation process for ROS packages. A catkin package can be built as a standalone project from the original ROS installation. In order to access and extend the ROS development environment, setup files are used so resources can be found in the installation folder; this is completed by running the bash script in the devel folder, as seen in the folder structure in Figure 2.4. Each of the root directories within the *catkin_ws* serves a different role; the Source (src) space contains the code for the catkin packages, and each folder in the source space contains one or more catkin packages; the space is unaffected by configuring, building or installing. Each of the catkin packages has its own *CMakeList.txt* that CMake invokes during the building process. The *build* space is where target files are placed before being built. The *devel* is where the built targets are set before being installed; they are arranged in the same structure as when they are installed in the *install* folder, which is where the targets once built are installed [17].

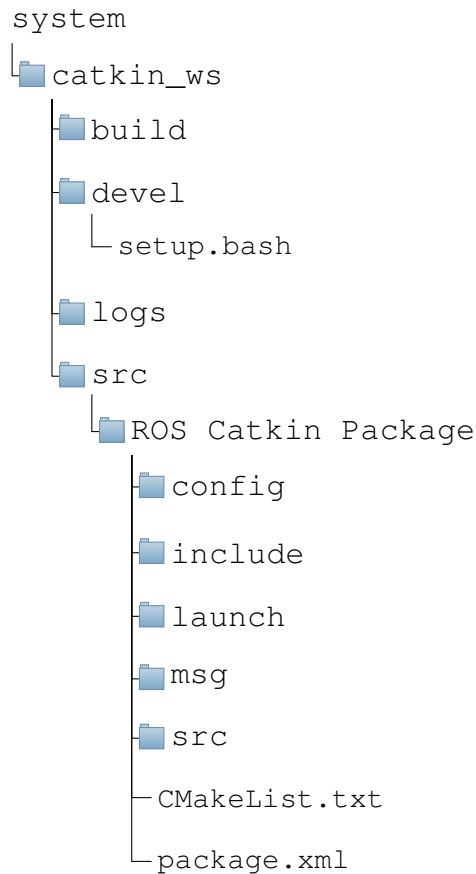


Figure 2.4: Catkin Workspace Folder Structure

2.3 Hardware

The hardware involved in this project will incorporate some of the existing hardware types and adapt it to be ROS-enabled. Therefore an understanding will need to be developed of the current components used on the arm to incorporate them into the code structure to operate as intended.

2.3.1 Motors

The existing arm consists of 3 different motor types that would require a variety of control methods. Understanding how each motor operates will be crucial in calibrating the new arm to communicate and run with ROS.

2.3.1.1 12V DC Encoder Geared Motor

The majority of the joints are comprised of 12V DC encoder motors. These motors allow for closed-loop positional control, enabling accurate control of the shaft location of the motor. Encoders are devices designed to generate a square

wave signal; these signals are created using either mechanical, resistive or optical means. The electric motors on the existing arm generate a pulse signal using an optical sensor called hall effect sensors which are attached to the end of the motor's shaft before the gearbox, as seen in Figure 2.5. A hall effect sensor is a device that can detect the presence or magnitude of a magnetic field. Encoders normally have three types of modes, being X1, X2 and X4, and this defines the minimum angle that the encoder can read. X1 is the base, then X2 means the resolution is double, and the minimum angle reading is halved, whilst X4 mode is four times the resolution. This can be seen using Equation 2.3 [18].

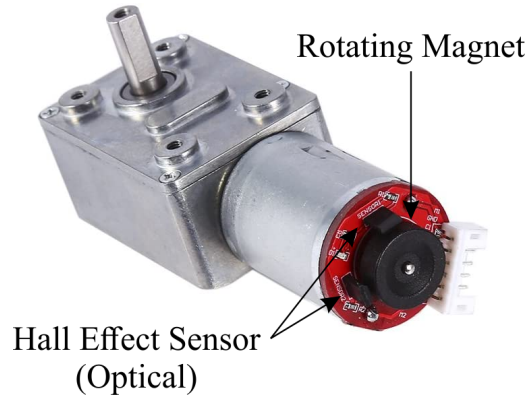


Figure 2.5: DC Worm Gear Motor 12V High Torque Reduction Gear Box with Encoder [19]

$$\text{minimum output shaft angle (radians)} = \frac{360\pi}{xN * 180} * G_{ratio} \quad (2.3)$$

Where:

- x Encoder Mode
- N Number of Pulses generated per shaft resolution
- G_{ratio} Gearbox Ratio

2.3.1.2 Geared Stepper Motor

The arm's wrist is a geared Nemma 17 Stepper motor seen in Figure 2.6a, a DC motor specifically designed to move in discrete stepping intervals. These are constructed using multiple coils that are arranged into groups called phases being A , \bar{A} , B and \bar{B} . These motors have high torques at low speeds with precise incremental control, although they are not very efficient as they draw high currents when trying to hold a position. The Nemma 17 stepper motor is common in various desktop CNC machines and 3D printers. The standard Nemma 17 stepper is a 200 count step per revolution, meaning that each step is 1.8° . Coil phases are essential in determining

the appropriate control method. A unipolar eight wire configuration stepper always energises the phases the same. In contrast, a Bipolar configuration uses four wires that require a H-bridge driver to reverse the current flow in the phases that alternate respectively. This can be seen in Figure 2.6b [20].

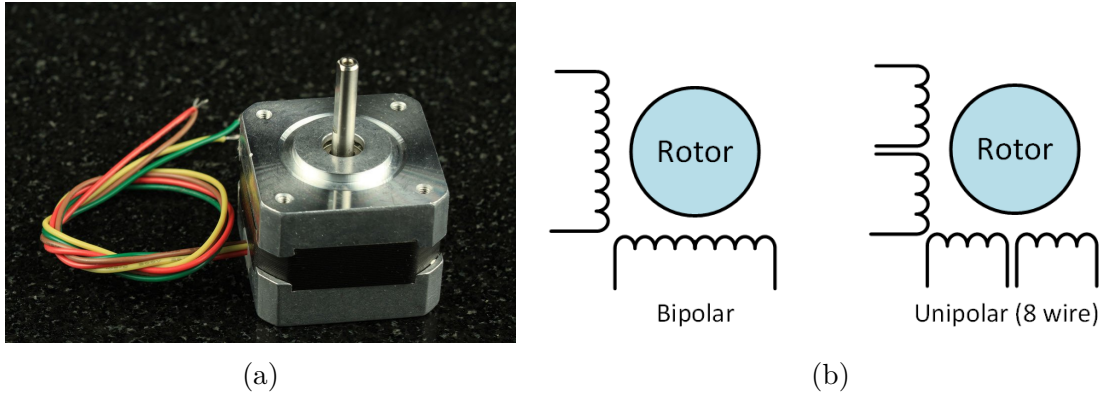


Figure 2.6: Nemma 17 Stepper Motor and Coils Configuration [20]

2.3.1.3 Servo Motor

Gripper control is actuated on the arm using a servo motor, as seen in Figure 2.7a, which comprises a DC motor and potentiometer integrated with a control circuit. The motor is connected to a series of gears that allow higher torque on the output shaft. The potentiometer's resistance changes when the motor rotates, allowing the control circuit to regulate the required directional movement. Once the servo actuator has moved to the desired location, the power supplied to the motor is stopped (although if an acting force were to be applied, the motor would provide power to hold its current position). A servo motor can usually rotate 90 degrees on either side of its neutral position, which allows for a total rotational range of 180°. The min pules will rotate the servo towards 0° where the max pulse will turn the arm towards 180° as seen in Figure 2.7b [21].

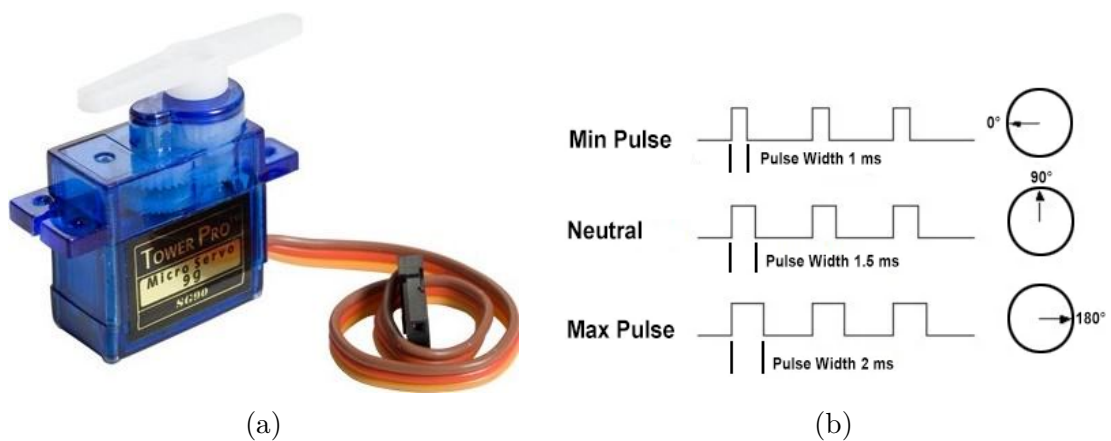


Figure 2.7: Servo Motor and PWM Position Control [21]

2.3.2 Control Board (Arduino ATmega 2560)

When it comes to controlling the physical hardware of the robotic arm, a affordable yet effective choice is the Arduino ATmega 2560, as seen in Figure 2.8. With the development of electronics over the years, microcontrollers have been essential in controlling real-world physical hardware, from household appliances to cars, robots and numerous other devices or equipment. Intel Corporation developed the first microcontroller in 1971, which was called the i4004, a 4-bit microcontroller [22].

The ATmega2560 is a suitable choice for the robotic arm as it has a large amount of required digital and analogue input and output pins necessary for controlling all the motors. The board has 86 pins, with 72 being digital IO (D0-D53), 16 analog pins (A0-A15) that allow up to 10-bit resolution with the analog to digital conversion and 15 of the digital IO capable of generating an 8-bit PWM signal.

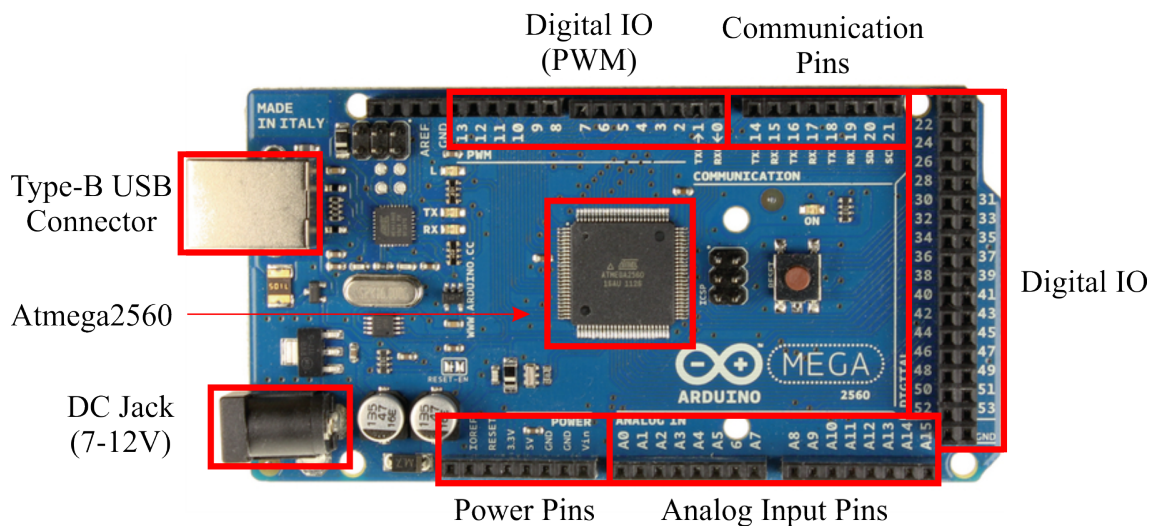


Figure 2.8: ATmega2560 Pinout [23]

2.3.3 L298 Motor Controller

A suitable motor controller that is diverse enough to control 12V DC motors and stepper motors is the L298N, as seen in Figure 2.9. This full H-bridge driver allows control of both speed and direction by using an input PWM signal. This setup has two outputs that can be used to control two DC motors or handle one bipolar stepper motor. As the PWM signal is modulated, it changes the output voltage supplied to the motor, controlling the speed at which the motors rotate.

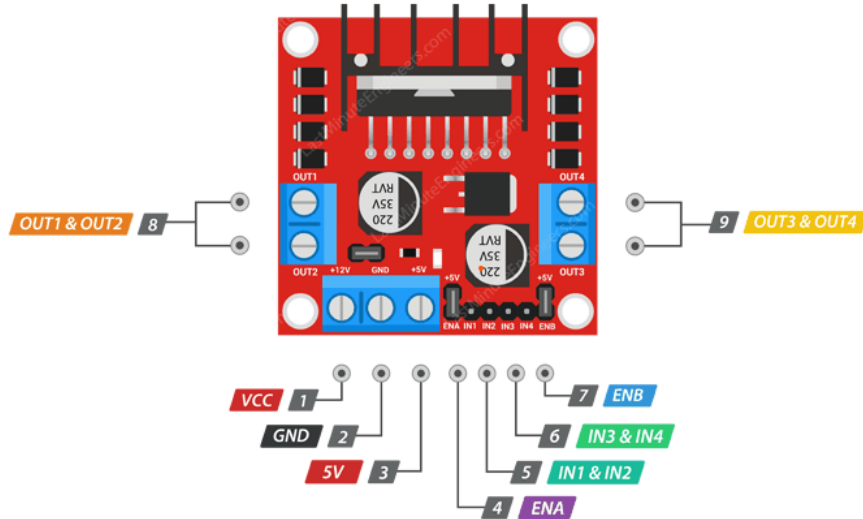


Figure 2.9: L298N Full H-Bridge Motor Driver [24]

The breakout board can be supplied with 5V or 12V, depending on the motor being driven. With four inputs being ENA and ENB, this enables the motor when pulled high with a jumper. Then the IN1, IN2 (Motor A) and IN3, IN4 (Motor B) pins are used to control the motor's direction and speed. If both pins are pulled high or low for a motor, it will stop, or if a pin is pulled high and low, then the motor spins in a specific direction [24].

2.3.4 Endstops

Endstops are a device used to help with positional control, which are used on large industrial machines and CNC devices down to desktop machines. These devices are placed so that when the machine moves to a particular position, the end-stop is either pulled high or low, letting the machine know that it is at its limits or home position. For the machine to understand it is at its designated location, a device is required that triggers the signal needed when the machine has reached a particular position. The three different typical end-stops found in CNC machines and robotic arms are mechanical, optical and hall effect.

2.3.4.1 Mechanical

Mechanical end-stops are the simplest and most commonly found end-stops. These end-stops are physical switches with a lever attached that pivots and triggers a physical button, as seen in Figure 2.10. They come in a variety of sizes and require very little force to trigger them, meaning the machine can trigger them without crashing into the limits of the machine. The typical wiring of a mechanical

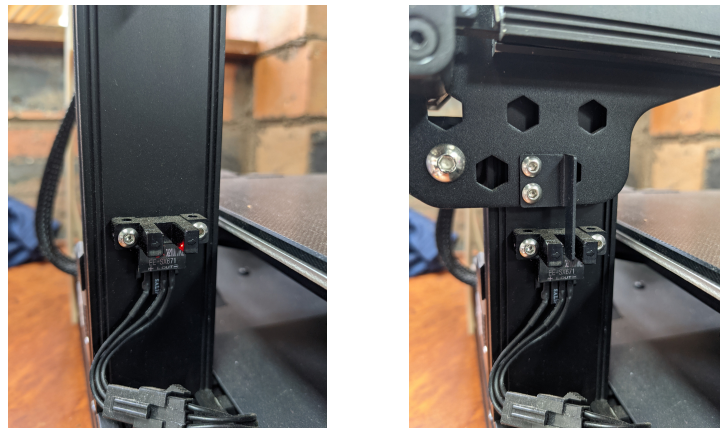
switch only requires two wires. When the switch is closed, the contact will be open or closed depending on if the switch is normally open or closed [25].



Figure 2.10: Mechanical Limit Switch on 3D Printer (x-axis)

2.3.4.2 Optical

Optical sensors are a device that uses a photo-interrupter in a U-shaped configuration, as seen in Figure 2.11a. A beam of infrared light is emitted from one side of the sensor and received from an optical detector from the other side. This type of end-stop is triggered when an object breaks the path of light of the sensor, evident in Figure 2.11b. This sensor is helpful as there is no need for physical interaction with the end-stop increasing the lifespan and decreasing any potential risk of components breaking on the CNC machine or robotic arm [25].



(a) Sensor Open

(b) Sensor Closed

Figure 2.11: Optical Sensor Endstop on 3D Printer (z-axis)

2.3.4.3 Hall Effect

A hall effect end-stop comprises two parts; a hall effect sensor connected to a circuit seen in Figure 2.12. This allows it to interpret the signal by amplifying the output generated called the “Hall Voltage”, which is a small change in voltage induced by the presence of a magnetic field. Therefore, the second part required is

a magnet that needs to be mounted on the carriage or moving component to trigger the sensor. The advantage of this system is similar to an optical sensor not requiring any physical interaction to activate the sensor [25, 26].

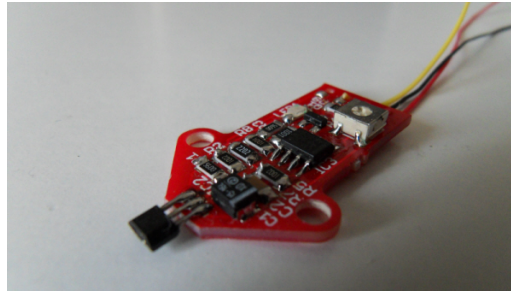


Figure 2.12: Hall Effect Sensor [25]

2.4 Types of Robotic Joints

Two types of joints that are used in industrial robotic arms:

2.4.0.1 Prismatic

A prismatic, as seen in Figure 2.13a joint is a motion that provides a linear sliding movement that only allows the joint to move in positional space, allowing only the robot to move in the X, Y or Z axis.

2.4.0.2 Revolute

A revolute joint, as seen in Figure 2.13b a mechanism that can rotate around a common point. This joint allows a robot to position itself in the X, Y or Z plane while also controlling orientation Roll, Pitch or Yaw.

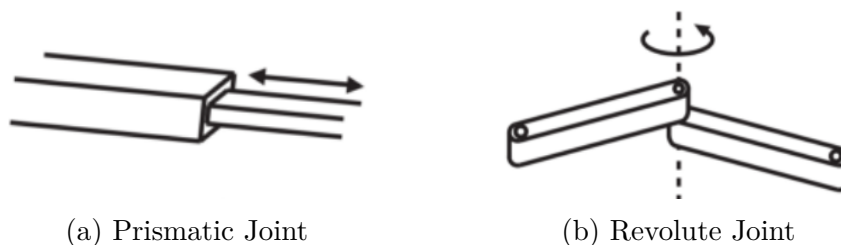


Figure 2.13: Robot Joints [27]

2.5 PID Controller

A PID controller is a controller designed and used in specific control applications that can be used to regulate flow, pressure, temperature speed or other processes

that require a variable output. A PID controller is a feedback system used to control variables with a stable, accurate output. In Equation 2.4 can be seen the velocity transfer function of a DC motor that of which can be integrated to give the position transfer function which is evident in Equation 2.5 where $G(s)$ is was multiplies by $\frac{1}{s}$ [28]. Transfer functions are important in applying to PID controllers as they take specific variables from the motor specifications to derive a specific transfer function for a DC motor.

$$G(s) = \frac{\theta(s)}{V(s)} = \frac{K_\tau}{(L * J)s^2 + (B * L + J * R)s + (B * R + K_a * K\tau)} \quad (2.4)$$

$$F(s) = \frac{\theta(s)}{U(s)} = \frac{K_\tau}{s((L * J)s^2 + (B * L + J * R)s + (B * R + K_a * K\tau))} \quad (2.5)$$

A PID controller consists of three variables that can change how a system responds to an input; these variables are known as the proportional, integral and derivative. In Table 2.1 can be seen how the effect of changing each variable in the system and the response it has on the output.

Table 2.1: PID Term Characteristics [29]

CL RESPONSE	RISE TIME	OVERSHOOT	SETTLING TIME	S-S ERROR
Kp	Decrease	Increase	Small Change	Decrease
Ki	Decrease	Increase	Increase	Decrease
Kd	Small Change	Decrease	Decrease	No Change

PID controllers typically need to be tuned with the parameters in the Table 2.1. This can be done in two typical ways being with an automatic tuning process where a program/algorithm will tune a model or real time data of a system. Another method is using the Ziegler–Nichols Method which is a manual tuning process that has been developed for tuning PID controllers.

Table 2.2: Ziegler–Nichols Method [30]

Controller	K_p	T_i	T_d	K_i	K_d
P controller	$0.5K_u$	0	0	0	0
PI controller	$0.45K_u$	$0.8T_u$	0	$0.54K_u/T_u$	0
Classic PID controller	$0.6K_u$	$0.5T_u$	$0.125T_u$	$1.2K_u/T_u$	$0.075K_u/T_u$

Where:

- K_u Ultimate Gain
- T_u Oscillation Periods

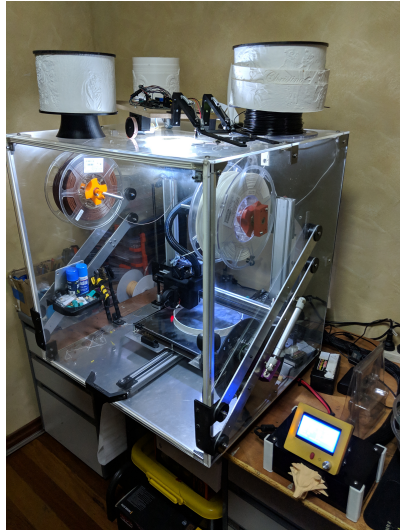
In order to tune the PID loop for the ROS controller with the motors that will be used for the new arm design, the Ziegler–Nichols Method approach will be taken as the manufacturing doesn't provide all the motor characteristics for the transfer functions of the DC motors. The Ziegler–Nichols Method involves increasing the gain K_p until the ultimate gain is achieved with stable oscillation. The period T_u can be measured and applied to determine the approximate values for P,I, and D terms using Table 2.2 that can be used in the ROS controller [30].

2.6 Rapid Prototyping - Manufacturing

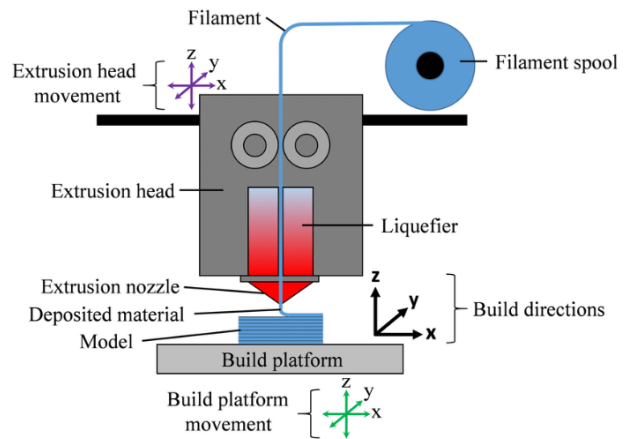
Rapid prototyping is becoming a more popular way of development and manufacturing in industry, allowing people and companies to reduce lengthy production times to create a working concept or final product in accelerated turnaround periods. Most robotic arms use traditional manufacturing methods being machining, cast or injection moulded parts/components. These methods use complex tool-path generation with costly and lengthy turnaround periods, resulting in a very costly end product.

3D printing technologies can be dated back to it's construction in 1986, where the first printing process was formed called the stereo-lithography machine (SLA) [31]. Furthermore, technological advancements brought the next form of 3D printing: Selective Laser Sintering (SLS). Then finally, the development of the commonly known and accessible 3D printing process is Fused Deposition Modelling (FDM) machines. 3D printing is not only limited to plastics but can be metals such as Titanium, Stainless Steel, Aluminium, etc.

The principles of 3D printing involve taking a 3D model in the form of an STL or 3MF file, sliced into layers/cross-sections along the z-axis at a specified thickness. The tool head then extrudes molten plastic that is fed through a nozzle following a specified path in the generated GCODE file to build the model layer by layer, allowing time for the previous layer to cool as seen in Figure 2.14b. The printer that will be used for printing the components for the project is a custom designed enclosed 3D printer that can be seen in Figure 2.14a



(a) 3D Printer Used for Project



(b) 3D Printing Process [32]

Figure 2.14: Desktop FDM 3D Printing

2.7 Repeatability Testing

Due to the nature of robotic arms, several performance criteria can be focused on when accessing and testing the performance of robotic arms. According to ISO9283:1998(E) standards "Manipulating industrial robots – Performance criteria and related test methods" there are 14 criterion that can be tested, these are as follows [33]:

- pose accuracy and pose repeatability
- multi-directional pose accuracy variation
- distance accuracy and distance repeatability
- position stabilization time
- position overshoot
- drift of pose characteristics
- exchangeability
- path accuracy and path repeatability
- path accuracy on reorientation
- cornering deviations
- path velocity characteristics
- minimum posing time
- static compliance
- weaving deviations

One of the project's primary focuses is benchmarking a low-cost 3D printed robotic arm with a more complex control method. Due to many characteristics involved when benchmarking a robotic arm and the limited time available for the completion of the project, the main focus of this report will be determining the pose repeatability of the new arm.

2.7.1 Pose Repeatability

Pose repeatability is the process of measuring set points located in the robot's working space, called its ISO cube. This is a cube where the arms TCP can move about and most frequently operate within. The ISO cube should have an imaginary inclined plane with at least five measurement points, as seen in Figure 2.15b. The process involves measuring each position for a minimum of 30 cycles during the tool moves between the five defined points. The test must be conducted at 100% of the manufacturer's specified velocity. It is optional to run further tests at 50% or 10%. It can be seen in the barycenter, which is represented by the letter G in Figure 2.15a is a proposed sphere where it is the mean value that displays the coordinates that correspond to the mean values calculated from the experimental data [34].

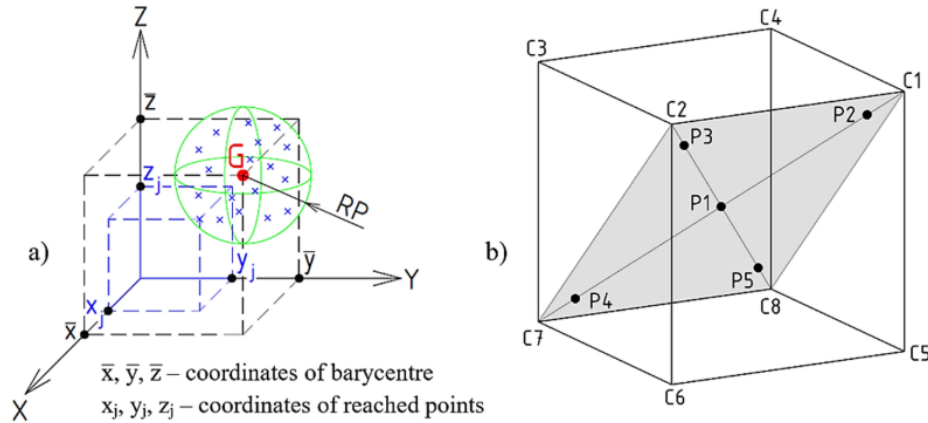


Figure 2.15: Pose Repeatability and ISO Cube [34]

To calculate the pose repeatability using the standard deviation calculated from the data collected through experimental results uses the Equation 2.6 where Pq is the point measured within the ISO cube on the inclined plane.

$$RP_{Pq} = \pm S_{Pq} = \pm 3 \sqrt{\frac{1}{n-1} \sum_{j=1}^n (Pq_j - \bar{P}q)^2} \quad (2.6)$$

Design and Experimental Setup

3.1 Introduction

This section explores the concepts of developing a low-cost 3D printed robotic arm that will be used to conduct an experimental analysis to determine the repeatability of a 3D printed robotic arm. The project also aims to implement a suitable control method that will be able to plan and solve complex inverse kinematics. This will be required to undertake the benchmarking for the repeatability of the 3D printed ROS enabled arm. This section will be broken down into crucial skills and techniques to carry out the experimental analysis. The chapter will also discuss any design considerations taken into account to achieve the best results that can be critically analysed.

3.2 Realisation of Robot Arm Design

The realisation and inspiration of the robot arm design came from an existing design provided by my supervisor. This arm design was a 4 DOF arm as seen in Figure 3.1 which consisted of one 12V DC motor, two 24V DC Encoded Motors and one Nemma 17 stepper motor. After analysing this existing design provided insight and inspiration to re-design a new arm and improve its current flaws and weaknesses. One of the main issues with the arm was the base rotational joint that positioned the robot's end effector in the X and Y planes, having issues with slipping with the current setup.

In addition, the design was limited to positional space and unable to have the ability to satisfy the roll, pitch and yaw of the TCP. The project's aim will incorporate using ROS as a control interface, allowing for a more flexible design, and taking full advantage of the complex inverse solver algorithms. Using three different motors makes it more complicated to integrate the hardware with the software. While the base joint not having an encoder makes it nearly impossible to determine the angle of joint 1, using consistent motors with the ability to monitor positions with encoders will be adapted in the re-design.

Another main issue with the existing arm was link 1 being held at an angle. This couldn't be adjusted. When a gripper was attached, it would be sitting at a fixed angle and unable to approach the items to be picked up in a perpendicular motion.



Figure 3.1: Existing 4 DOF Robotic Arm

3.2.1 Robot Arm Design (BCR1)

As ROS can handle tasks solving complex inverse kinematics of robots consisting of six or more joints, the robot model will consist of a minimum of six joints. A typical six-axis industrial robotic arm consists of six revolute joints where joints 1-3 position the TCP in positional space, known as the robot elbow. Where joints 4-6 are used for tool orientation, known as the robot's wrist. A model will be developed and drawn using the software package SOLIDWORKS a CAD package developed by Dassault Systèmes (a French software corporation) [35].

A model design was drawn as seen in Figure 3.2; this joint configuration was based on existing industrial robot designs where joint 1 can position the TCP in the X-Y plane and joints 2 & 3 can control the reach in the X, Y and Z axis. Then joints 4-6 being the wrist of the robot that can handle the roll, pitch and yaw of the end effector.

3.2.2 BCR1 Kinematics

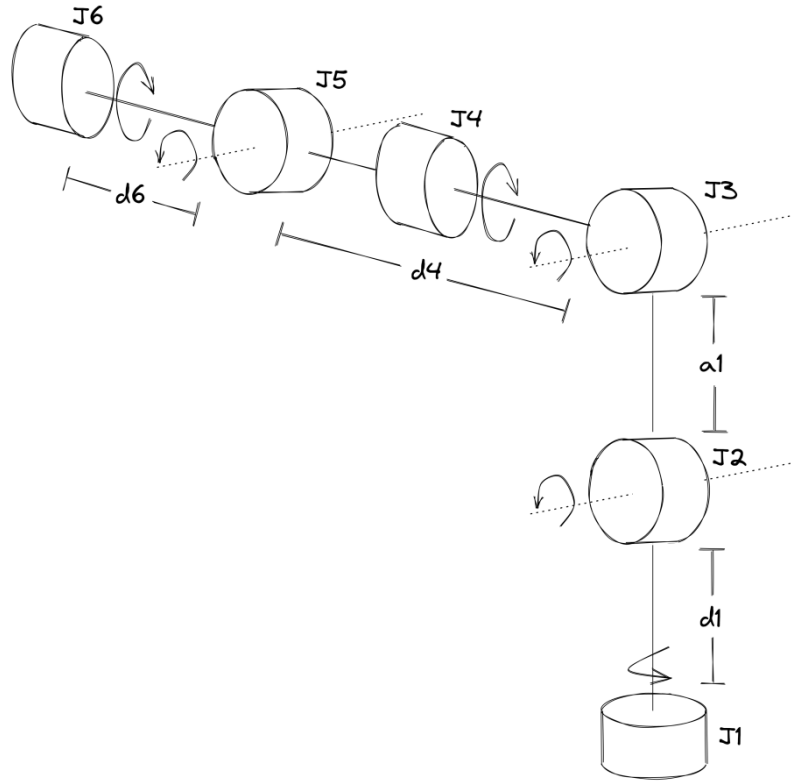


Figure 3.2: Model Design

To understand the fundamentals of the design a coordinate frame is attached to the robot model. A technique is followed to connect these reference frames called the Denavit–Hartenberg (D-H) convention, a series of rules exploring how to orientate the coordinate frames for each joint.

Steps are as follows:

1. Number joints starting from 1 to n, starting from base to tool
2. Assign coordinate frame to the base of the robot with the Z^0 axis aligned with the rotation of joint 1
3. Align Z-axis for each joint to be colinear with the joint rotation or translation; right-hand grip rule determining positive rotation.
4. Select X^k to be orthogonal to Z^k and Z^{k-1} , if Z^k and Z^{k-1} are colinear, X^k should point away from Z^{k-1} .
5. Y^k is selected using the right-handed orthonormal rule.
6. Repeat steps for joints if $k < n$.

From this, the Robot DH parameters can be determined as seen in Table 3.1, which will allow the calculation of the forward kinematics of the robot through the use of Equation 2.1. This will display the end-effector position and orientation in space, given the joint angles as the input.

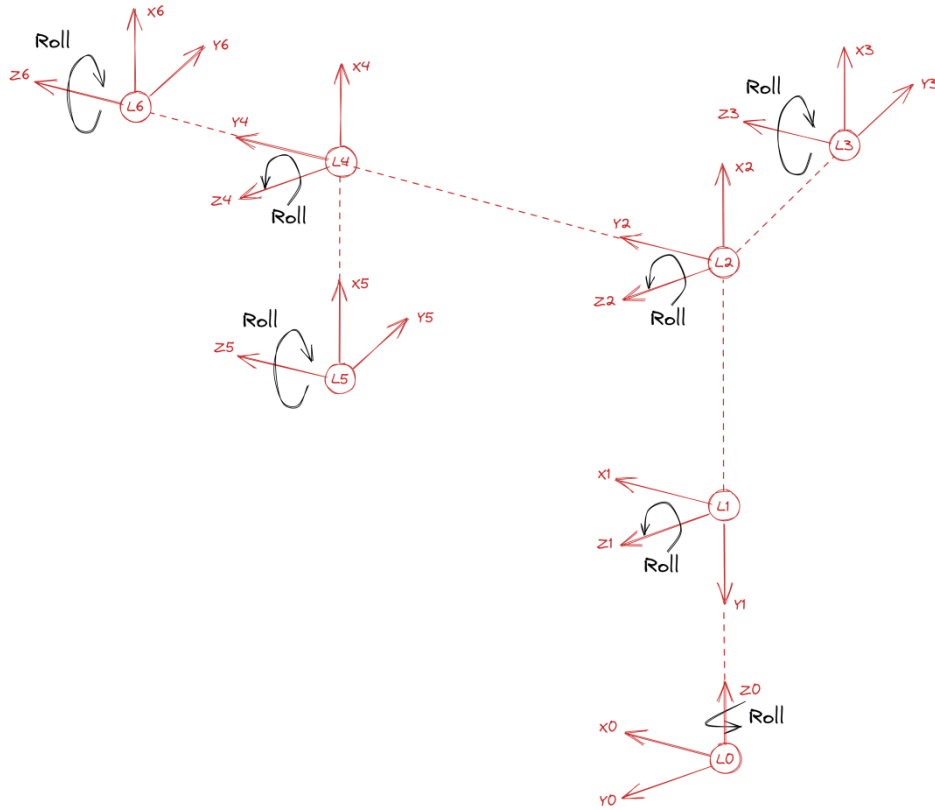


Figure 3.3: Coordinate Frame

Table 3.1: Denavit–Hartenberg Parameters for BCR1

Axis	θ	d	a	α	Home
1	q1	d1	0	$\pi/2$	0
2	q2	0	a1	0	$\pi/2$
3	q3	0	0	$\pi/2$	0
4	q4	d4	0	$-\pi/2$	0
5	q5	0	0	$\pi/2$	0
6	q6	d6	0	0	0

The MATLAB script in Appendix E can be used to calculate the general solution for T_0^6 that can be used to derive the tool configuration vector. The tool configuration vector can be solved through a numerical approach of an inverse kinematic solver.

This allows the tool position and orientation as an input, then outputting the appropriate joint angles to satisfy the condition. Since there are multiple solutions for a desired position and orientation of the TCP, a numerical solution will be used instead. Through the aid of built-in libraries, the heavy lifting is conducted by the MoveIt library. This library has built-in inverse kinematic solvers that take a numerical approach and find multiple results.

3.2.3 Motor Selection Criteria

One of the main goals of the design consideration is to keep the arm low cost and affordable. This narrows down the motor selection criteria and overall structure of the arm design. To keep the design simplistic and reduce complexity in joint designs. A low-cost DC motor will be the most viable option. After analysing the existing arm, the motors used were dual shaft motors that supported either side of the joint, linking two links together. This concept will reduce cost and weight in having to add extra bearings for a single shaft motor. The motor size will determine the robot link lengths. The motor seen in Figure 3.4a was selected as it has multiple gear ratios with the same package size, allowing for the same design and can assign a different gearbox depending on the torque required for each joint. Since there are two collinear joints in the design, being joint 4 & 6 as a single shaft motor as evident in Figure 3.4b with a bearing will be required. The specifications for gear reductions with rated torques can be seen in Table 3.2.

After designing the arm and determining the link dimensions seen in Appendix G using the specified motors in Figure 3.4 with the datasheets provided by the manufacturer, the motor rated torques could be determined to select the appropriate gear reduction. Joint 2 is the crucial joint in selecting the proper gear reduction to have enough torque to lift the arm in the fully extended position, as seen in Figure 3.5. Using the general formula for torque being $\mathcal{T} = Force * distance$, the Equation 3.1 can be derived for determining the torque at joint 2 where the arm is in the fully extended position.



(a) 12V Dual Shaft DC Encoded Motor (b) 12V Single Shaft DC Encoded Motor

Figure 3.4: Motor Types

Motor	Mass (kg)	Force (N)
M3	0.1	0.981
M4	0.12	1.18
M5	0.1	0.981
M6	0.12	1.18

Motor Positions are as follows: $d_1 = 0.178m$, $d_2 = 0.06m$, $d_3 = 0.18m$, $d_4 = 0.13m$

$$\begin{aligned}
 \mathcal{T}_{M2} &= d_1 * F_{M3} + d_{1+2} * F_{M4} + d_{1+2+3} * F_{M5} + d_{1+2+3+4} * F_{M6} \\
 \mathcal{T}_{M2} &= 0.178 * (0.981) + (0.178 + 0.06) * 1.18 + (0.178 + 0.06 + 0.18) * 0.981 \\
 &\quad + (0.178 + 0.06 + 0.18 + 0.13) * 1.18 \\
 &\Rightarrow \mathcal{T}_{M2} = 1.51N.m
 \end{aligned} \tag{3.1}$$

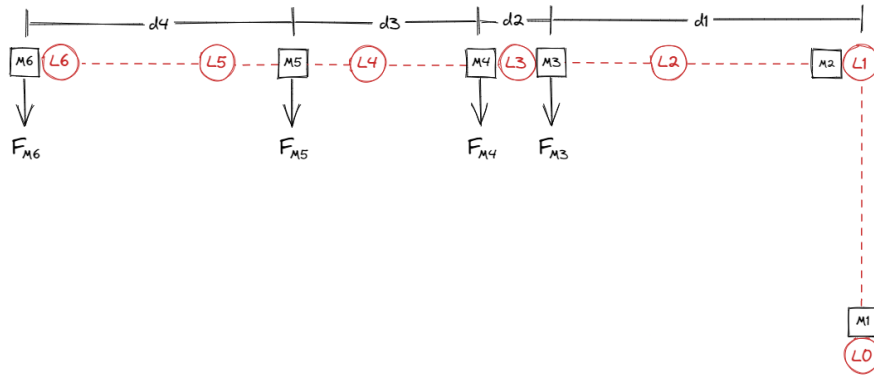


Figure 3.5: Motor Torque Diagram

The max torque by joint 2 was calculated to be $1.51N.m$ meaning that looking

at the motor specifications in Table 3.2 that the motor with a reduction of 600:1 would be a suitable fit for joint 2, as it is roughly half of the rated stall torque. Seeing that joint 2 will have a maximum speed of 7.5 rpm and considering the linear interpolation of joint movements of a robotic arm, a 600:1 gear reduction would be used for all the other joints.

Table 3.2: Motor Specifications

Reduction Ratio	Rated Voltage (V)	Speed (rpm)	Current (A)	Stall Torque (N.m)
1000	12	4	0.6	4.41
600	12	7.5	0.6	2.94
340	12	15	0.6	1.77
260	12	18	0.6	1.27
200	12	22	0.6	0.98
150	12	30	0.6	0.74
90	12	51	0.6	0.20
65	12	70	0.6	0.16
40	12	120	0.6	0.09

3.3 Link Design

To keep the BCR1 design relatively simple each link was based on a standard link and modified/resized. Each link was based off link two, where there are six main pieces. The motor housing comprises of four elements to encase the motor and allow it to be easily mounted within. The linking arms connect to the motor housing that extends for the following link to attach to, as seen in Figure 3.6b. The link assembled is seen in Figure 3.6a. The link was held together with the same bolt size being M3x6 mm and the required heat inserts. Selecting the dual shaft motors, reduces the cost of not requiring bearings for joints 2, 3, and 4, as the joints could be fully constrained on either side.

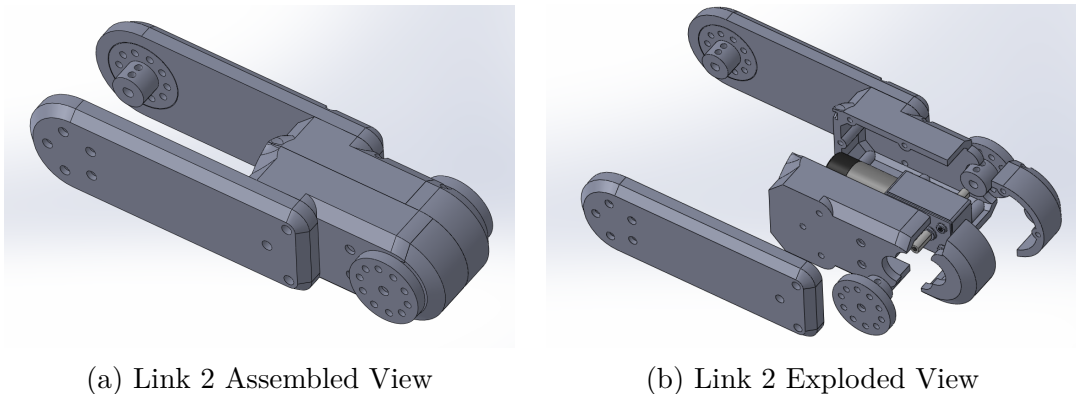


Figure 3.6: Link 1 Design

3.3.1 Bearing Arrangement

Due to having concentric joints for joints 1 & 4, the joints cannot be fully constrained, and the arm will apply a perpendicular torque from the weight of the links. Therefore, an added bearing was required for each concentric joint with an added coupling to constrain this joint.

The motor can constrain axial load. This is the force applied parallel with the motor.

A radial ball bearing was selected to handle the radial loads. The bearing arrangements is seen in Figure 3.7, whilst the bearing coupling that linked the two links together are seen in 3.9b.

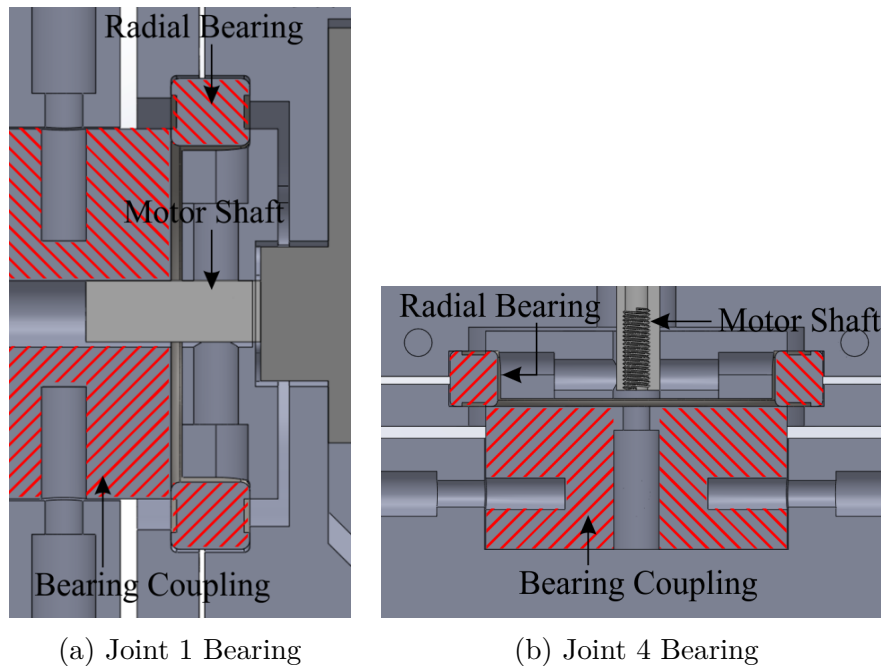


Figure 3.7: Bearing Arrangement

3.3.2 Motor Arrangement

Since the design aim was to keep the BCR1 low cost, the motor arrangement was straightforward as there were no belts, pulleys or geared shafts, meaning that each link was going to be attached using the previous motor. Thus keeping the motor's close to one another was the most crucial consideration to keep the mass mostly over the robot's base. In doing so, the joint 3 motor was arranged, so the motor was hanging off the back end of the arm as seen in Figure 3.8 in order to keep the mass as far back as possible.

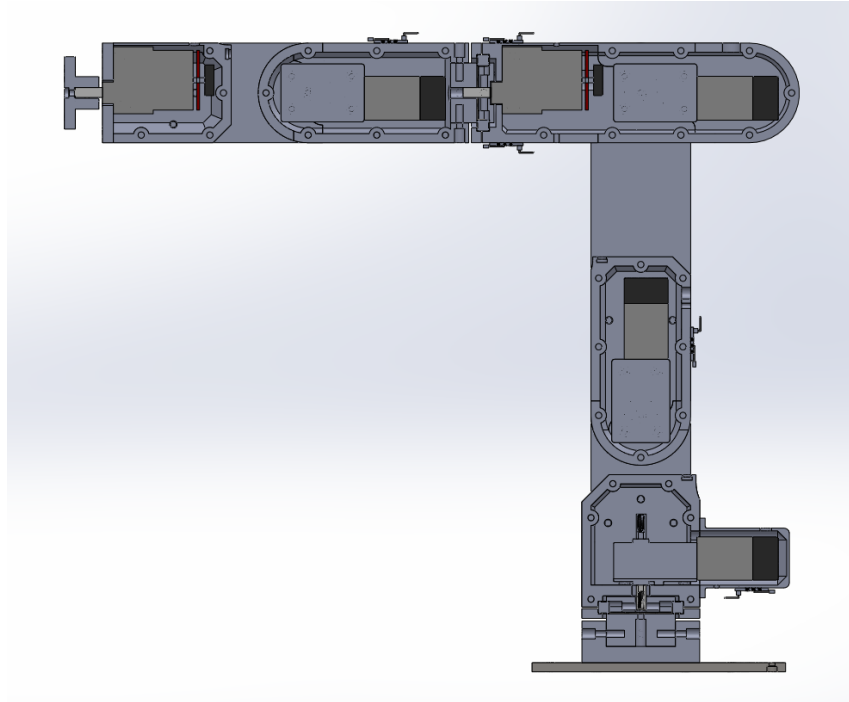
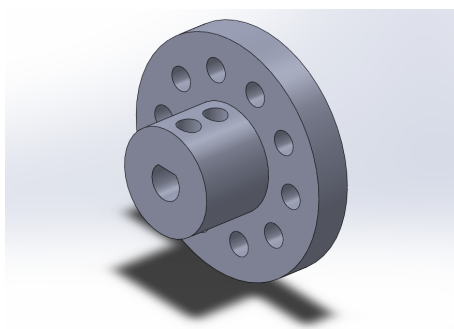


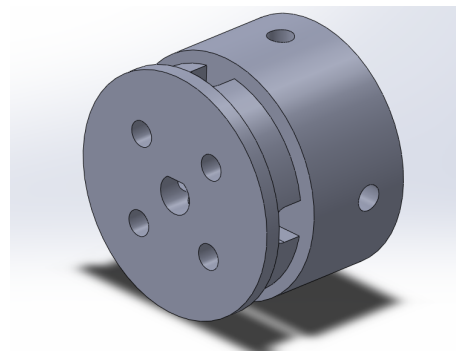
Figure 3.8: Motor Arrangement

3.3.3 Coupling Design

A coupling design was created to reduce costs that could be entirely 3D printed. Brass heat inserts were added for structural integrity, allowing the bolts to grab correctly, attaching the motors to each link with minimal to no backlash. There were two designed couplings required for two different motor types, the first being the dual shaft coupling as evident in Figure 3.9a and the other being for the single axle bearing coupling seen in Figure 3.9b.



(a) Dual Shaft Motor Coupling Design



(b) Single Shaft Motor Coupling Design

Figure 3.9: Coupling Designs

3.3.4 Homing Limit Sensors

The chosen end-stop used to home the robot and determine the joint positions when rebooting were hall effect sensors, as they are easy to align and don't require any physical contact to trigger them. The hall effect sensor modules were mounted to the outside of a link, and a magnet was then embedded in the previous link as seen in Figure 3.10a. The sensor is triggered as the link rotates backwards until the hall effect sensor is in range of the magnet, evident in Figure 3.10b.

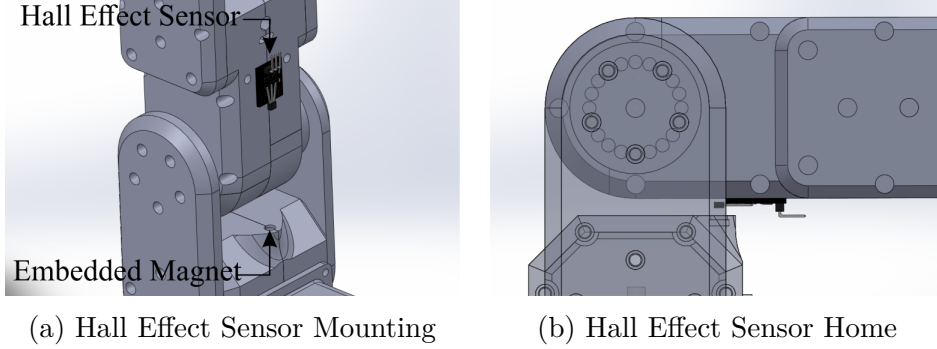


Figure 3.10: Hall Effect Sensor Joint 2

3.3.5 BCR1 Forward Kinematics Home Position

The rotation matrices in Equation 3.2 can be used to input the joint angles that are multiplied to determine the tool position and orientation in relation to the base.

$$\begin{aligned}
 T_0^1 &= \begin{bmatrix} C1 & -C(\frac{\pi}{2})S1 & S(\frac{\pi}{2})S1 & 0 \\ S1 & C(\frac{\pi}{2})C1 & -S(\frac{\pi}{2})C1 & 0 \\ 0 & S(\frac{\pi}{2}) & C(\frac{\pi}{2}) & d1 \\ 0 & 0 & 0 & 1 \end{bmatrix} & T_1^2 &= \begin{bmatrix} C(\frac{\pi}{2}2) & -S(\frac{\pi}{2}2) & 0 & a2C(\frac{\pi}{2}2) \\ S(\frac{\pi}{2}2) & C(\frac{\pi}{2}2) & 0 & a2S(\frac{\pi}{2}2) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 T_2^3 &= \begin{bmatrix} C3 & -C(\frac{\pi}{2})S3 & S(\frac{\pi}{2})S3 & 0 \\ S3 & C(\frac{\pi}{2})C3 & -S(\frac{\pi}{2})C3 & 0 \\ 0 & S(\frac{\pi}{2}) & C(\frac{\pi}{2}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} & T_3^4 &= \begin{bmatrix} C4 & -C(\frac{\pi}{2})S4 & -S(\frac{\pi}{2})S4 & 0 \\ S4 & C(\frac{\pi}{2})C4 & S(\frac{\pi}{2})C4 & 0 \\ 0 & -S(\frac{\pi}{2}) & C(\frac{\pi}{2}) & d4 \\ 0 & 0 & 0 & 1 \end{bmatrix} & (3.2) \\
 T_4^5 &= \begin{bmatrix} C5 & -C(\frac{\pi}{2})S5 & S(\frac{\pi}{2})S5 & 0 \\ S5 & C(\frac{\pi}{2})C5 & -S(\frac{\pi}{2})C5 & 0 \\ 0 & S(\frac{\pi}{2}) & C(\frac{\pi}{2}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} & T_5^6 &= \begin{bmatrix} C6 & -S6 & 0 & 0 \\ S6 & C6 & 0 & 0 \\ 0 & 0 & 1 & d6 \\ 0 & 0 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

Using Equation 2.2 an example case for determining the TCP position in the robot home position can be calculated using the values for the joint angles of 0 and link lengths from Appendix G. Using the MATLAB script attached in Appendix E to perform the matrix multiplication to calculate the position and orientation of the end effector. This can be verified through the CAD model as seen in Figure 3.11

showing the correct X, Y and Z coordinates as well as the tool orientation as seen in Equation 3.3.

$$T_{base}^{tool} = T_0^1 * T_1^2 * T_2^3 * T_3^4 * T_4^5 * T_5^6 = \begin{bmatrix} 0 & 0 & 1 & 318.1 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 319.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.3)$$

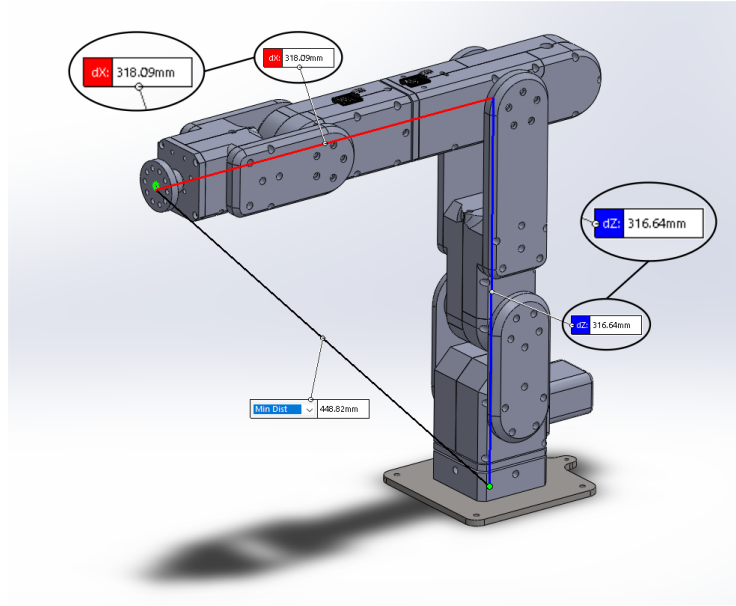


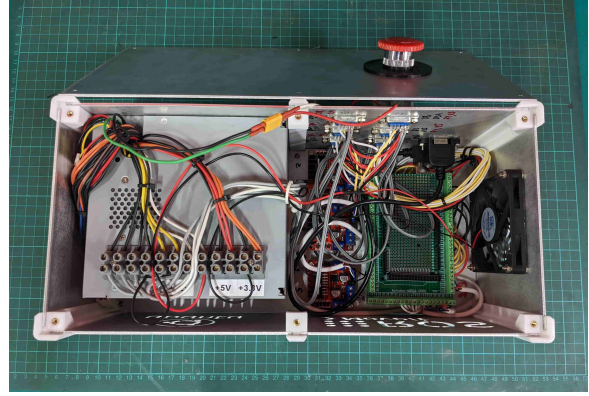
Figure 3.11: Robot End Effector Position in Home Position

3.4 Hardware Control Box

The hardware control acts for two purposes, the first use case is for housing all the electrical components. These are the power supply, motor controllers, and micro-controller (ATmega2560) mounted on nylon standoffs to ensure complete electrical isolation from the aluminium housing. The other primary purpose of the control box is to act as a mounting point for the 3D printed robotic arm (BCR1) which allows it to be firmly mounted using thumbscrews. The added weight allows the arm to reach fully extended with enough mass at the base of the arm to prevent it from falling over. The overall wiring schematic is seen in Figure H.1, through the use of six 9 Way D-SUB connectors seen in Figure 3.12a, it allowed for the motors to be connected to the housing whilst allowing the arm to be removed at any stage easily. This flexible design allows for the control box to possibly be used for different types of robotic arms or a new, improved design.



(a) Control Box Back



(b) Control Box Wiring

Figure 3.12: Control Box

3.5 ROS Implementation

Since ROS is mainly developed and tested on Linux, this was easy to select the newest distribution of Linux, being Ubuntu 20.04 Focal Fossa. There were options to go with a virtual machine that allows a user to emulate an operating system on top of an existing system. Due to having access to a desktop, installing another hard drive, and setting up Linux with a dual boot system was the most viable solution to avoid any driver errors when communicating via serial that could pose an issue when using an emulation/virtual machine.

3.5.1 URDF Robot MoveIt Package

One of the most crucial elements is developing the URDF file required for ROS to understand the robot configuration and display it in the RViz workspace. An open-source GitHub project maintained by Brawner [36] has been developed as a SolidWorks plugin that can be used to export assemblies of robotic arms with the required STL formats and descriptions. Using the photo as seen in Figure 3.13a and the SolidWorks model as seen in Figure 3.13b. This saves having to write the description formats manually.

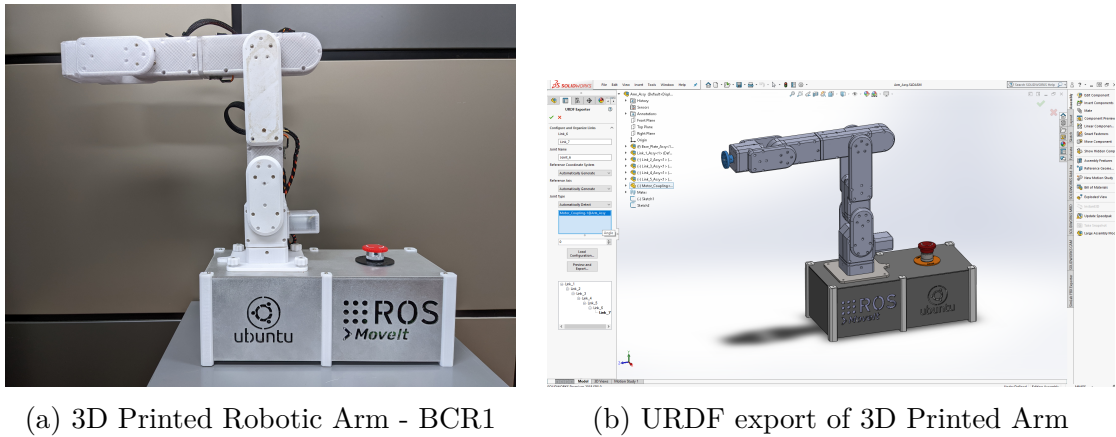


Figure 3.13: Arm Design (BCR1) to be ROS enabled

MoveIt setup assistant can then read the files necessary to finish off the configuration process required for ROS, as seen in Figure 3.14. The MoveIt setup assistant allows importing the previously exported URDF package created in SolidWorks and enables the user to define all the joints. A self-collision matrix is generated within the software to allow the robot to determine which joints can collide during this process. Default robot poses can be added with a choice of inverse kinematic solver along with the position, velocity and acceleration controller required to generate planned robot trajectory paths.

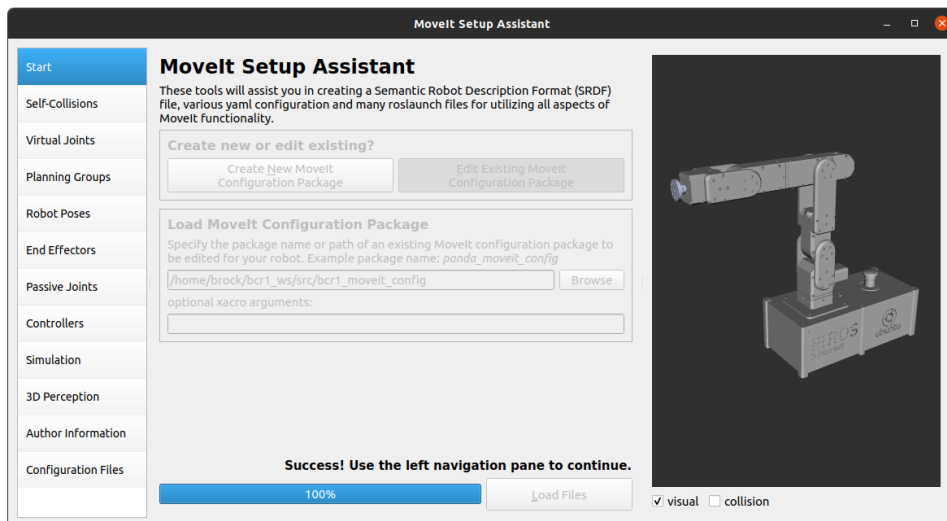
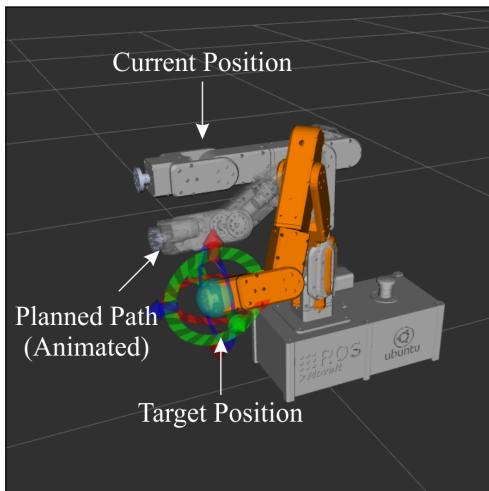


Figure 3.14: MoveIt Setup Assistant

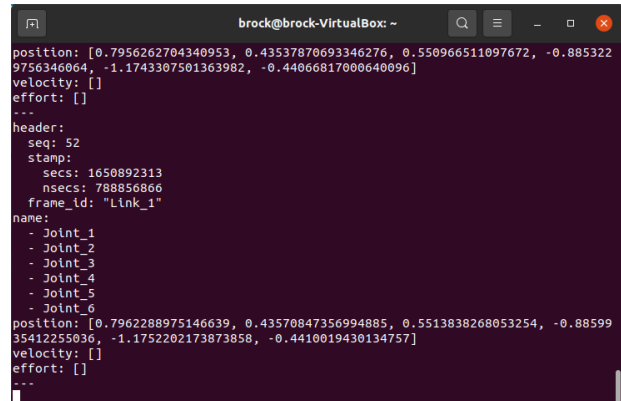
The MoveIt setup assistant generates the necessary files into a ROS package that can be built in the catkin workspace. A demo launch file can be run to test the robot configuration with the fake controller manager.

3.5.2 Simulating in RViz

Once a package is created for the arm, the default launch file can be run with a roscore node and RViz, allowing the robot to be loaded into the RViz visualiser. The robot can be manipulated using the axis limiting motion in the specified axis, or the end effector can be moved around using the blue ball at the end of the robot, as seen in Figure 3.15a. The proposed trajectory is animated from the current position to the target position highlighted in orange. To test the configuration of the arm a joint state topic can be subscribed to, as seen in Figure 3.15b when the robot moves, the joint angles are displayed live from the “*move_group/fake_controller_joint_states*” which are outputted in radians. This default publisher runs when a robot isn’t connected by default. Running this demo launch files ensures that the URDF is reading correctly and there weren’t any issues when launching the robot after setting up the robot using the MoveIt Setup Assistant. Once the model is checked, a hardware interface controller will be integrated to work with the physical hardware meaning a new launch file will be created to run the new desired nodes.



(a) Simulated 3D Printed Arm RViz



(b) Subscriber Node Current Joint Angles

Figure 3.15: Arm being controlled in RViz using inverse kinematics

For the current working setup of the ROS application for the BCR1 arm, there are three active nodes in the system, as seen in Figure 3.16. The “*/robot_state_publisher*” allows the system to publish the robot state to the “*/tf*” topic, which is a transformation library that enables the tracking of multiple coordinate frames that are available to the whole system and allows RViz to take this as an input and display the robot state in 3D. The “*/move_group*” has topics that are used for the end-user to define specific target poses or joint states that additional written packages can feed in. Whilst the “*/move_group*” allows the program to read in and publish the joint conditions for a given URDF model. An overview of the entire

nodes and topics is seen in Figure C.1 where nodes are in oval shapes, and topics are drawn in rectangles, which all stem from the master node in the middle.

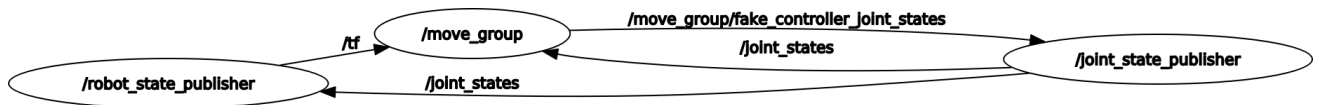


Figure 3.16: ROS Active Nodes for Demo Controller using rqt Topics

3.5.3 ROS Hardware Interface

To interface one of the available ROS controllers, a hardware interface is required to send (*hardware_interface::RobotHW::write*) and receive (*hardware_interface::RobotHW::read*) commands to and from the robot, being joints, sensors or actuators. In Figure 3.17 is seen, the data flow process for ROS communication between hardware and the simulation environment with the selected controller. To implement this controller, a boilerplate which is a template for a simple simulation interface designed and maintained for setting up a hardware interface for ROS controllers was used [37]. The main control of DC motors with an Arduino with the L298N dual full-bridge motor driver uses an analog write PWM signal from 0-255.

ROS has a controller for each ROS interface that links to a controller to communicate with the *hardware_interface::RobotHWrobot*, for this specific arm, a *joint_trajectory_controller* with an Effort Joint Hardware Interface will be used. The *joint_trajectory_controller* will be using the *effort_controller* namespace, this controller accepts a joint angle as an input and will output an effort based on the position of the joint and how far the joint has to move, which is passed through a PID loop. These values are then passed through the Effort Joint Hardware Interface. The PID parameters can all be entered into *controllers.yaml* file in the hardware control package, along with the maximum and minimum effort parameters being 0-255 can be inputted into the *joint_limits.yaml* from the *moveit_config*.

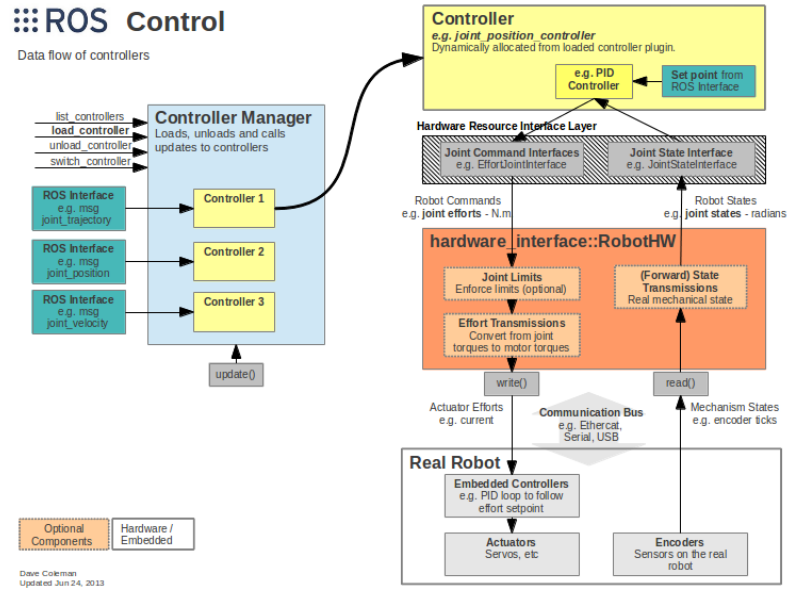


Figure 3.17: ROS Control Data Flow [16]

After setting up the hardware interface with a new custom launch file (*bcr1_HW_main.launch*) that is used to start the master node *bcr1_hw_main* and the other executable nodes being, */move_group*, */robot_state_publisher*, */serial_node* as seen in Figure 3.18 these are the core nodes to communicate joint angles and update the RViz simulation whilst being able to communicate with the hardware through the *rosserial* node. The message names are seen above each linking arrow with the two main custom messages being the */arduino/bcr1Telemetry* and */arduino/armCmd* these were written to send payload messages to and from the robot, with */arduino/bcr1Telemetry* sending the robot joint current angles and */arduino/armCmd* that sends the effort (PWM) signal required to move each joint. An entire detailed overview of all the active ROS nodes, topics and messages can be seen in Figure C.2.

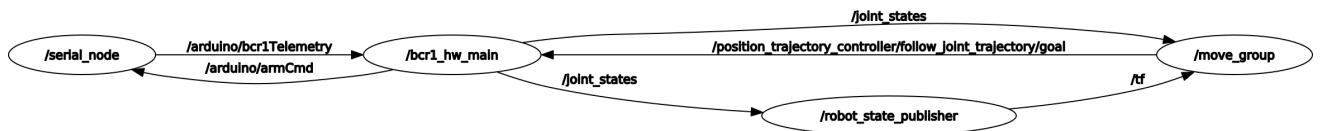


Figure 3.18: ROS Active Nodes for BCR1 using rqt Topics

3.5.4 Repeatability Testing Design

The repeatability design will consist of using a dial indicator as seen in Figure 3.19a which is a measuring device that is typically used for measuring a variety of precision engineering applications, such as straightness, levelness, shaft runout and

many more. The reason for choosing such a device for taking measurements for testing the repeatability of the robotic arm is that it is relatively cost-effective and easy to access such equipment.



(a) Analogue Dial Indicator



(b) Digital Dial Indicator Apparatus [34]

Figure 3.19: Example Experimental Apparatus

Both the arm and dial indicator will need to be mounted on a flat plane/surface with a ground base for taking all measurements. The arm will be able to move to specific positions where the dial indicator will be mounted, as seen in the example in Figure 3.19b. The arm will be able to approach the probe on the end of the dial indicator, where measurement will be recorded, before the arm completes another move to another position ensuring all joints move from their current state. The arm will come back to take another measurement and undergo this process for a minimum of 30 readings in each axis.

To control the arm and automate the process of taking multiple readings, a control program was required to manipulate the arm into different positions numerous times. Since ROS has many packages available, the MoveIt library has a package called *moveit_commander* that interfaces with the *move_group* topic that has a list of functions and definitions that can be written and used in Python or C++. Originally the robot position and tool orientation is displayed as a coordinate transformation and rotation matrix as seen in Equation 3.4. In order to use the *moveit* commander and specify positions and tool orientation, the required input is desired in quaternions (a mathematical notation used for representing orientations and rotations of an element in three-dimensional space). Quaternions consist of four components roll, pitch, yaw and W , a scalar defining the amount of rotation along that axis as seen in Equation 3.5.

$$T_{base}^{tool} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \end{bmatrix} \quad (3.4)$$

$$q = \begin{bmatrix} q_i \\ q_j \\ q_j \\ q_k \end{bmatrix} = q_i \mathbf{i} + q_j \mathbf{j} + q_k \mathbf{k} + q_r \quad (3.5)$$

Where:

- r real component being W
- i, j, k imaginary components for Roll, Pitch and Yaw

From the rotation matrix T_{base}^{tool} the basis for calculating quaternion are seen in Equation 3.6

$$\begin{aligned} q_r &= \frac{1}{2} \sqrt{1 + T_{11} - T_{22} - T_{33}} \\ q_i &= \frac{1}{4q_r} (T_{32} + T_{23}) \\ q_j &= \frac{1}{4q_r} (T_{13} + T_{31}) \\ q_k &= \frac{1}{4q_r} (T_{21} + T_{12}) \end{aligned} \quad (3.6)$$

In certain cases there are multiple ways to calculate q, although numerical accuracy can be reduced by avoiding cases in which the denominator is close to zero. Another mathematical representation are seen in Equation 3.7

$$\begin{aligned} q_i &= \frac{1}{2} \sqrt{1 + T_{11} - T_{22} - T_{33}} \\ q_j &= \frac{1}{4q_i} (T_{12} + T_{21}) \\ q_k &= \frac{1}{4q_i} (T_{13} + T_{31}) \\ q_r &= \frac{1}{4q_i} (T_{32} + T_{23}) \end{aligned} \quad (3.7)$$

Due to the numerical inaccuracy of solving the orientations manually, a more appropriate method was taken through the use of computational calculations since the *moveit_commander* is compatible with python. Making light work for converting to quaternion form using the python module *quaternion*. The python code is seen in Appendix F.2.3

Results, Analysis and Evaluation

With all the available results and data, the chapter aims to analyse the performance of the final BCR1 concerning the objectives and goals outlined at the beginning of the project seen in Chapter 1. Overview of benchmarking results and a breakdown of the overall cost of the BCR1 robotic arm.

4.1 Experimental Setup

The experimental setup design consisted of testing the repeatability of the arm in the x, y and z-axis for 30 cycles in each. An example of the setup can be seen in Figure 4.1 representing the test trajectory path where the arm would run through 30 times, testing the x-axis repeatability. This consisted of the arm starting in the rest position evident in Figure 4.1a where the arm would then move to an approach position Figure 4.1b in order to approach the dial indicator in a parallel path/motion seen in Figure 4.1c. The arm was moved back to the arm rest position for testing on all axis to ensure all the joints move through a range of motion to ensure the most valid repeatability results of the BCR1.



(a) BCR1 Arm Rest

(b) BCR1 Approach

(c) BCR1 Probe

Figure 4.1: Experimental Apparatus Setup

The end effector plate would then probe the end of the dial indicator from the approach position in Figure 4.2a to the probe position in Figure 4.2b where a recording of the dial indicator measurement was taken in excel. This path is repeated for 30 cycles and then repeated for 30 cycles on the y-axis and z-axis by changing each axis's tool orientation. The process was automated using the python

script using the MoveIt commander python API leaving a 15 second delay between each probe to read and record the dial indicator measurement.

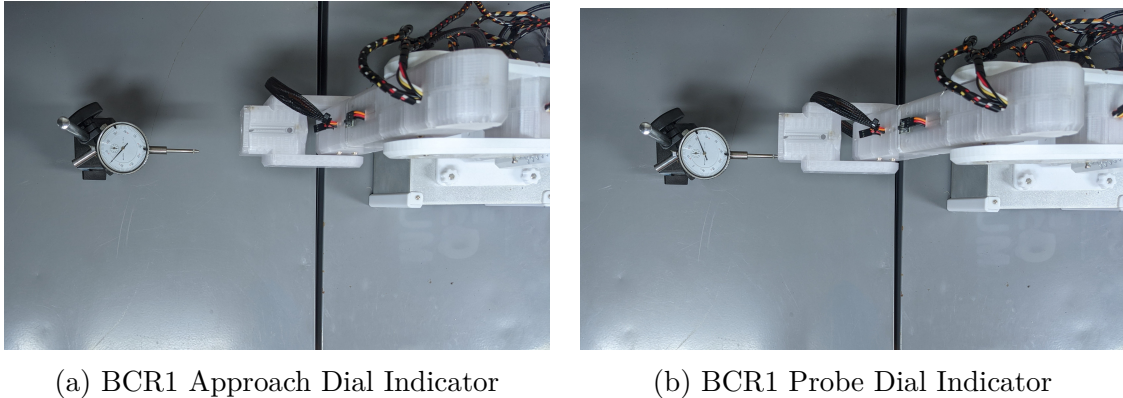


Figure 4.2: Dial Indicator Measurements

For the arm to move in the same trajectory path for the 30 cycles, an appropriate planning library was required of the five available planners the MoveIt library supports. A planning library is required for arms that have six or more joints due to the nature of the kinematic model; these arms can have multiple solutions for trajectory path planning to reach the desired goal whilst considering possible collisions that the robot could have through its range of motion. A trajectory motion planner determines how each joint should move through space to reach the target goal. The first and most popular planning library available is the Open Motion Planning Library (OMPL), an open-source motion planning library that implements a range of sampling-based motion planning algorithms that MoveIt can directly implement. The OMPL planner is set to default to the motion planner in the MoveIt configuration package for new robots. The second commonly used planner is the Pilz Industrial Motion Planner, a deterministic generator for linear and circular motions, whilst having the ability to merge multiple motion sequences together. This motion planner has been optimised for calculating the path for the shortest possible trajectory. The Stochastic Trajectory Optimisation for Motion Planning (STOMP) is an optimised-based motion planner developed to plan smooth trajectories designed for obstacle avoidance and optimisation for constraints. Search-Based Planning Library (SBPL) is a generic set of motion planners that use search-based planning but aren't fully integrated into the MoveIt library, with limited functionality. The last planner available for MoveIt is the Covariant Hamiltonian Optimization for Motion Planning (CHOMP). That uses a novel gradient-based trajectory optimisation technique, allowing for everyday simple motion planning, both trainable and straightforward, allowing for quick solvable solutions based on the implemented algorithm [38].

In order to benchmark the arm the technique required the arm to follow the

same trajectory. After analysing the different motion planning libraries available from the MoveIt package, it was decided the Pilz motion planner would be most suitable as it can calculate the path quickly with the shortest possible trajectory. While considering potential joint collisions, the popular default OMPL planner occasionally would calculate varying trajectories that were slightly longer, although having relatively the same planning time.

4.2 Repeatability Results

The experimental data gathered for each axis can be seen in Figure 4.3-4.5 where the points are plotted as deviations from the mean for each dataset. Each plot represents a pose repeated 30 times for each axis; it was noticeable the x-axis has the greatest repeatability, with the largest outlier being $1.136mm$ from the mean. This was expected to be the best axis for repeatability due to all the joint motors in this direction already have pre-applied torque to the motors reducing the effect of backlash in the motors affecting the repeatability. The Y-axis has a larger deviation from the mean with the largest outlier being $1.990mm$. The main reasoning behind the Y-axis having a slightly greater error from the X-axis is due to joint 1 of the arm that has no torque acting on the motor, therefore the backlash in the DC geared motors would be influencing these results. Whilst the Z-axis had the largest deviation from the mean being $5.342mm$, the main reason for this large repeatability error was caused by joint 2, as the motor torque is slightly underpowered and not always capable of consistently reaching its target goal.

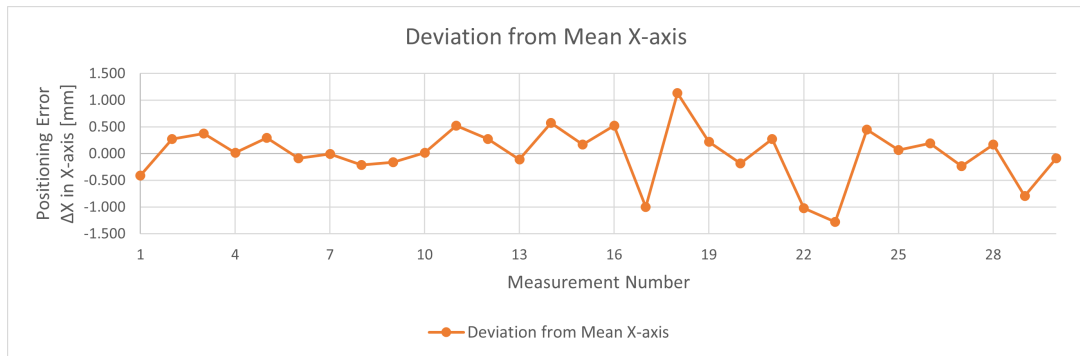


Figure 4.3: Repeatability in X-axis

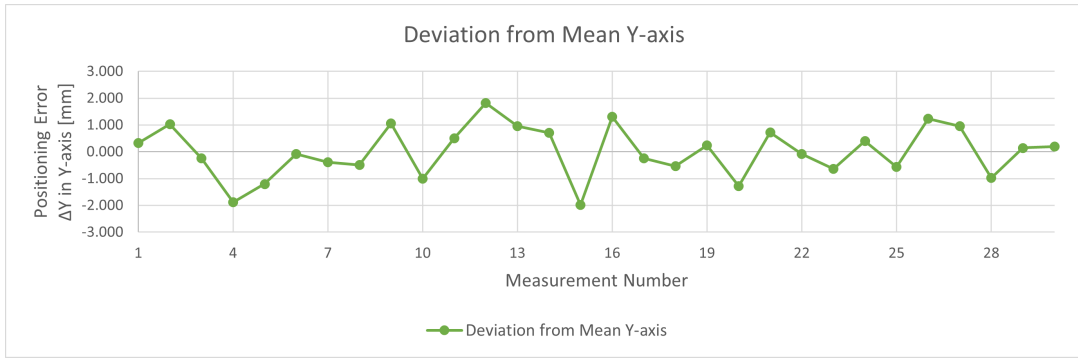


Figure 4.4: Repeatability in Y-axis

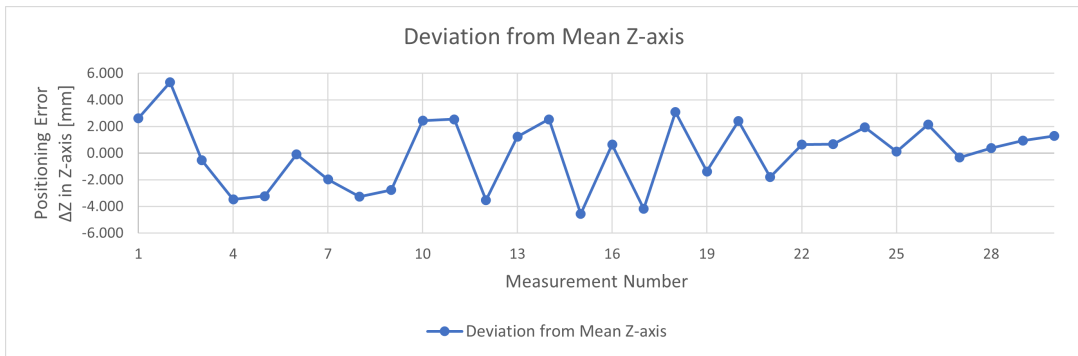


Figure 4.5: Repeatability in Z-axis

The standard deviation for each axis can be seen in Table 4.1 displaying the dispersion of results with the z-axis having the greatest distribution of data.

Table 4.1: Standard Deviation of X,Y & Z Axis

Axis	Standard Deviation
X	0.513
Y	0.953
Z	2.522

4.2.1 Repeatability Analysis

In three-dimensional Cartesian space, points have three components to find the coordinate position. To find the distance between two points in a straight line, the distance formula as seen in Equation 4.1 can be applied to the results for the x,y and z-axis.

In applying this to the raw data, the total repeatability of the BCR1 for the test position in 3D space can be observed, with the max deviation of the arm being $5.448mm$ radius from the median test point position. Therefore, the arm within this vicinity can reach a target pose within $\pm 5.488mm$. This is observed as the largest possible deviation being an outlier as seen in Figure 4.6 from 30 data points with the average deviation being $\pm 2.376mm$.

$$R = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \quad (4.1)$$

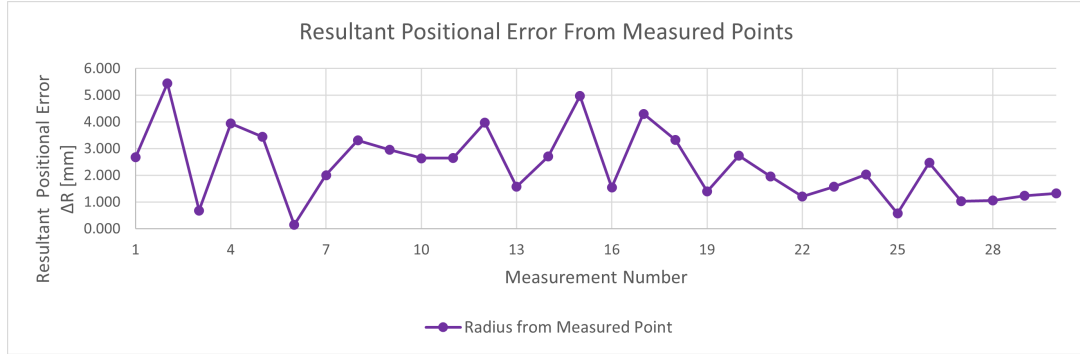


Figure 4.6: Combined Resultant Repeatability of BCR1

Further analysis of the data is evident in plane form, showing each point the arm reached and the distance away from the mean of the data. In Figure 4.7 is seen in 3D space, the points surrounding the data mean; the larger green dot. The further away a point is from the mean test point in a straight line, the dot colour converges to red. Through visualising the data in this form, it is evident that the X, Y plane has the greatest repeatability whilst the X, Z plane and Y, Z plane have more significant variance in the distance from the test mean. This is due to Z-axis position error from lack of torque at low control voltages caused by low joint velocities calculated through the ROS controller.

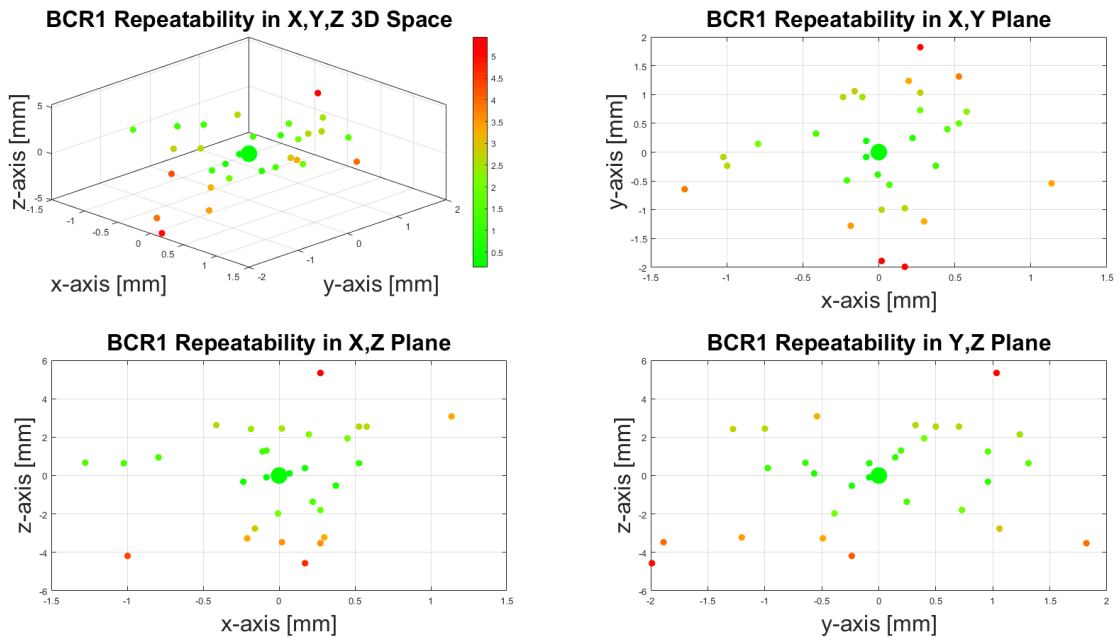


Figure 4.7: Repeatability in Each Plane 3D Space

4.3 Costing

One of the key areas was to keep the design affordable with off the shelf components and rapid manufacturing techniques. Through research and sourcing of materials, the overall arm cost was able to meet this criterion coming in at a final cost of \$268.78 AUD as seen in Table 4.2. This was achieved in multiple ways including, finding an overseas supplier for the majority of the electrical components and bearings, and using a consistent design that requires very few different types of hardware and keeping bolt sizes consistent. One cost-saving component was having all the links 3D printed, which was the project’s aim; in doing so, it kept the design low cost, as designing parts to be injection moulded is a costly process and can cost up to \$5000+ AUD. In addition, the cost of the arm was kept lower by reducing the amount of filament required for the components by designing parts that require no support in the FDM printing process. This was able to keep the weight of the filament needed to $\approx 1200\text{g}$. The overall printing time for all the components took $\approx 65\text{hrs}$ with the approximate power consumption of the machine being 90 watts. The total print time consumed $\approx 5.85\text{kWh}$. With the average cost of power in Australia being 27c/kWh, the total printing cost came to a total of \$1.58 from the total consumed energy.

Table 4.2: BCR1 Costing

Item	Quantity	Per Unit	Total	Supplier
Micro DC 12V 10 (600:1)	4	\$16.3	\$65.2	AliExpress
Micro DC 12V 10 (600:1)	2	\$10.7	\$21.4	AliExpress
RS, 6806	1	\$3.16	\$3.16	AliExpress
RS,6807	1	\$3.89	\$3.89	AliExpress
Hall Effect Sensor Module	6	\$4.95	\$29.7	JayCar
Arduino Atmega2560	1	\$22.03	\$22.03	AliExpress
L298N Motor Driver	3	\$2.12	\$6.36	AliExpress
Arduino Atmega2560 Terminal Shield	1	\$9.76	\$9.76	AliExpress
26 AWG Wire	3	\$4.95	\$14.85	JayCar
Aluminium Sheet (600x600) Optional	1	\$26.55	\$26.55	Ullrich Aluminium
USB-B Extension Cable	1	\$2.24	\$2.24	AliExpress
Heat Inserts (100pcs)	1	\$5.63	\$5.63	AliExpress
3D Printer Filament PETG (1kg)	2	\$28.99	\$57.98	Amazon
Total			\$268.75	

Discussion and Conclusion

This chapter made an overall understanding and conclusions from the key findings through the project development and research. It evaluates the effectiveness of the project and if it met the set criteria initially outlined.

5.1 Key Findings

Through this research, multiple outcomes were discovered and achieved. The main results include designing a low-cost 3D printed 6 DOF robotic arm, ROS control Interface and benchmarking the repeatability of the final design. Through these critical findings, there is a basis for future development that can be achieved. The most significant outcome of this project was the ROS Hardware Interface controller. This is because there is very little ROS documentation on how to implement hardware interfaces, particularly an effort controller with closed loop control, as there aren't many robots out there that use low-cost hardware. Instead, have more advanced controllers with pre-existing ROS packages written for them. There are existing ROS enabled low-cost arms; they are open-loop systems that don't have any feedback and use the fake controller that the MoveIt Setup Assistant generates. Not being able to determine the actual arm position, for example, if a motor were to be interrupted, such as a stepper motor skipping steps, the arm wouldn't be able to correct for this type of behaviour. In contrast, the BCR1 setup can determine the actual robot position by tracking each of the six motor encoders. After designing the BCR1 and setting up an appropriate controller, one key finding was that DC motors are not necessarily the best solution for a robotic arm. Due to the control method in controlling DC motor speeds, a lower voltage supply is required, but a lower voltage supply to the motor means the torque is proportionally less than the rated torque. On the other hand, a stepper motor has higher torque at lower rpm and decreases with speed. Another option would be a BLDC Brushless motor which has higher starting torques with the mechanical advantage of not requiring brushes for commutation but instead using electronic control controlling commutation [39].

5.2 Arm Design

The overall arm design was quite simplistic, with it being easy to assemble with very little hardware required. The arms design was made so that it was able to be fully printed without the need for any support material. In turn, it saved costs on the filament and reduced print time, saving on power costs as well. Each link could be assembled and easily joined together using the coupling design that allowed each link to interconnect. The control box also allowed for universal use case scenarios, possibly using it for future designed arms.

5.3 Conclusion

In summation, the conducted research project involved various aspects that were key to the project's success, with each core area dependent on the previous research and project development. Although the project was a success, there is still room for improvement in multiple areas.

Throughout the project, it demonstrates that ROS is a suitable controller for a robotic system with active tools and existing packages readily available. Although, it is challenging to implement into real-life applications with very little documentation on integrating software with real-life hardware. ROS is designed as a communication layer median for a robotic arm/system. There is usually an added software layer for creating control plans and trajectories, whether this is path planning for welding, pick and place operations, human-assisted interaction or toolpath operations. Furthermore, development can be done to add more functionality to the arm see Chapter 6.

The overall arm design was lacking in some areas. Still, it served the initial purpose of benchmarking to determine the repeatability of the arm, which was able to be carried out despite some results having more significant errors. The project's aim was met and gave insight into future improvements that can be made to the BCR1 design. This concept was initially discussed at the start of the project using this research as a building block for future research into low-cost 3D printed ROS enabled robotic arms.

Furthermore, the project displayed that an industrial solution control system can be applied to a small scale design and implemented using all open-source hardware readily available with off the shelf components. Implementing ROS has opened up the development opportunities for integrating the arm into various workspaces upon developing the higher-level control interface, whether it is a GUI interface or integration with existing software.

Future Work and Recommendations

This chapter outlines various topics for further future work and design that were not considered within the project due to time constraints and complexity. The three key focuses are improvements to the design, software development and use case scenarios. More optimisation can be done through this work, improving the arm over time.

6.1 BCR1 improvements

One of the first improvements could be on joints 1 & 4 that require using a bearing. The main concern was the shaft attaching to the bearing adaptor whilst modifying a thrust bearing on joint 1 to handle the moment/torque caused by the arm extending. The radial bearing has a bit of play, not being able to constrain the joint fully. The second flaw in the design is the backlash in the DC geared motors having around ± 1.9 deg, which causes an additional error that affects the repeatability of the arm. This would require sourcing possibly more expensive motors or contacting the supplier to see if it is possible to work to a tighter tolerance on the motor specifications. A critical design missing from the BCR1 Robot design was an end effector/tool; a mounting plate was added to the robot's end for a future tool to be designed and mounted. One crucial issue with the arm was joint 2 being slightly underpowered. Two issues caused this: the power supply and the motor specifications of the stall torque being slightly less than specified. Therefore changing the motor in joint 2 to a larger gear reduction motor would help eliminate this issue.

6.2 Software Development

A key area for development would be writing software packages for ROS to have greater control over the arm, allowing for better control for a range of applications,

essentially permitting the robot to understand the working environment. This could include writing packages to integrate camera vision for 2D vision, lidar for 3D vision, Force Torque Sensors, Collision Detector Sensors, or even part detection sensors. Furthering the software development on the arm will open up the opportunity to use the arm in various applications, whether repetitive or small one-off handling tasks that would normally require an operator to interfere with the job/task. One main useful software design feature that would aid the robot would be a graphical user interface (GUI) having the ability to enter the X, Y and Z coordinates along with the rotational matrix. Also, having other buttons and functions for gripper control, homing, and a teach pendant can save end-effector positions in a program that can then run through the interface.

6.3 Integration Into Applications

There is a wide range of applications for light tasks that the BCR1 arm could be applied towards. Such as a small production line, handling a small PCB assembly process, moving PCBs from a staging plate of a pick n' place machine and into a solder reflow oven. With 3D printing becoming quite a growing hobby, profession and professional industry, the BCR1 could open opportunities for desktop 3D printer makers and hobbyists. The arm is a low-cost design, and nearly all 3D printed. Opening up to the community allowing the arm to be applied towards being a companion for a 3D printer. This could involve removing finished prints or even entire print beds if this is an available option. Whilst also interacting with an active print. If this involves pausing the print to add a particular part, whether it be a nut for a thread or some added weight to give the print more weight, reducing the need to rely on a user to be around when the print reaches the target layer where the interaction needs to occur.

With readily available platforms and print management platforms open source, allowing integration to be possible. Octoprint is an open-source 3D printer web interface designed to control and interface with desktop 3D printers. The interface design has been developed so it is fully extendable, allowing users to write plugins in python that can be installed onto the Octoprint instance. Since ROS allows for python communication through the MoveIt commander, it could be possible to map the bed regarding the robot's position and enable it to understand where parts are located on the build plate.

References

- [1] F. Negrello, H. S. Stuart, and M. G. Catalano, “Hands in the real world,” *Frontiers in Robotics and AI*, vol. 6, no. 147, 2020.
- [2] M. E. Moran, “Evolution of robotic arms,” *Journal of robotic surgery*, vol. 1, p. 103–111, 2007.
- [3] “Ros.org | about ros,” Ros.org, 2021. [Online]. Available: <https://www.ros.org/about-ros/>
- [4] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, “Ros: an open-source robot operating system,” vol. 3, 01 2009.
- [5] A. Ademovic, “An introduction to robot operating system: The ultimate robot application framework,” Toptal Engineering Blog, 2016. [Online]. Available: <https://www.toptal.com/robotics/introduction-to-robot-operating-system#:~:text=The%20main%20languages%20for%20writing,preferred%20due%20to%20better%20performance.>
- [6] R. McCollin, “A complete guide on xmlrpc.php in wordpress (what it is, security risks, how to disable it),” Kinsta, 07 2020. [Online]. Available: <https://kinsta.com/blog/xmlrpc-php/>
- [7] “Ros/introduction - ros wiki,” Ros.org, 2018. [Online]. Available: <http://wiki.ros.org/ROS/Introduction>
- [8] R. K. Megalingam, A. Sahajan, A. Rajendraprasad, S. K. Manoharan, and C. P. K. Reddy, “Ros based six-dof robotic arm control through can bus interface.” 2021 5th International Conference on Intelligent Computing and Control Systems (ICICCS): Institute of Electrical and Electronics Engineers Inc., Conference Proceedings, pp. 739–744. [Online]. Available: <https://ieeexplore-ieee-org.ezproxy.uow.edu.au/document/9432341>
- [9] H. Wu, H. Handroos, and P. Pessi, “Mechatronics design and development towards a heavy-duty waterhydraulic welding/cutting robot,” *Mechatronics for Safety, Security and Dependability in a New Era*, pp. 421–426, 2007. [Online]. Available: <https://www.sciencedirect.com/topics/engineering/forward-kinematics>

- [10] S. Hernandez-Mendez, C. Maldonado-Mendez, A. Marin-Hernandez, H. V. Rios-Figueroa, H. Vazquez-Leal, and E. R. Palacios-Hernandez, “Design and implementation of a robotic arm using ros and moveit!” in *2017 IEEE International Autumn Meeting on Power, Electronics and Computing (ROPEC)*. Institute of Electrical and Electronics Engineers Inc., Conference Proceedings, pp. 1–6.
- [11] “rviz - ros wiki,” Ros.org, 2013. [Online]. Available: <http://wiki.ros.org/rviz#Overview>
- [12] F. Joshua, “frankjoshua roserial arduino lib,” GitHub, 2020. [Online]. Available: https://github.com/frankjoshua/roserial_arduino_lib
- [13] “rqt/plugins - ros wiki,” Ros.org, 2016. [Online]. Available: <http://wiki.ros.org/rqt/Plugins>
- [14] “rqt - ros wiki,” Ros.org, 2014. [Online]. Available: <http://wiki.ros.org/rqt>
- [15] E. Robotics, “Understanding ros nodes · erle robotics gitbook,” Gitbooks.io, 2018. [Online]. Available: https://erlerobotics.gitbooks.io/erlerobot/content/en/ros/tutorials/understanding_ros_nodes.html
- [16] S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. Rodríguez Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Lüdtke, and E. Fernández Perdomo, “ros_control: A generic and simple control framework for ros,” *The Journal of Open Source Software*, 2017. [Online]. Available: <http://www.theoj.org/joss-papers/joss.00456/10.21105.joss.00456.pdf>
- [17] “catkin/workspaces - ros wiki,” Ros.org, 2017. [Online]. Available: <http://wiki.ros.org/catkin/workspaces>
- [18] E. P. Company, “White paper - the basics of how an encoder works,” Encoder.com, 2011. [Online]. Available: <https://www.encoder.com/wp2011-basics-how-an-encoder-works#:~:text=Encoders%20convert%20motion%20to%20an,count%2C%20speed%2C%20or%20direction.>
- [19] “Amazon.com: Dc worm gear motor 12v high torque reduction gear box with encoder srong self-locking 0.236 in output shaft (10rpm) : Herramientas y mejoras del hogar,” Amazon.com, 2021. [Online]. Available: <https://www.amazon.com/Walfront-Torque-Reduction-Encoder-Self-Locking/dp/B073S5GM6Q>

- [20] B. Earl, “All about stepper motors,” Adafruit Learning System, 05 2014. [Online]. Available: <https://learn.adafruit.com/all-about-stepper-motors/what-is-a-stepper-motor>
- [21] “How servo motors work | servo motors high efficiency and power,” Jameco.com, 2021. [Online]. Available: <https://www.jameco.com/Jameco/workshop/Howitworks/how-servo-motors-work.html>
- [22] “History of microcontrollers | toshiba electronic devices and storage corporation,” Semicon-storage.com, 2021. [Online]. Available: <https://www.electronicshub.org/arduino-mega-pinout>
- [23] “Arduino mega pinout | arduino mega 2560 layout, specifications,” Electronics Hub, 01 2021. [Online]. Available: <https://www.electronicshub.org/arduino-mega-pinout/>
- [24] L. M. Engineers, “Interface l298n dc motor driver module with arduino,” Last Minute Engineers, 11 2018. [Online]. Available: <https://lastminuteengineers.com/l298n-dc-stepper-driver-arduino-tutorial/>
- [25] Toglefritz, “Types of endstops,” Toglefritz’s Lair, 07 2015. [Online]. Available: <https://toglefritz.com/types-of-endstops/#:~:text=There%20are%20three%20different%20types,%2C%20optical%2C%20and%20Hall%20effect.>
- [26] “Everything you need to know about hall effect sensors | rs components | rs australia,” Rs-online.com, 2014. [Online]. Available: <https://au.rs-online.com/web/generalDisplay.html?id=ideas-and-advice/hall-effect-sensors-guide#:~:text=So%2C%20how%20does%20a%20Hall,sense%20the%20position%20of%20objects.>
- [27] khalil idrissi, “Degrees of freedom of a robot - khalil idrissi - medium,” Medium, 04 2020. [Online]. Available: https://medium.com/@khalil_idrissi/degrees-of-freedom-of-a-robot-c21624060d25
- [28] A. Reddy, “Transfer function of armature controlled dc motor,” Electrical Engineering Info, 05 2017. [Online]. Available: <https://www.electricalengineeringinfo.com/2017/05/transfer-function-armature-controlled-dc-motor.html>
- [29] “Control tutorials for matlab and simulink - introduction: Pid controller design,” Umich.edu, 2022. [Online]. Available: <https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction§ion=ControlPID>

- [30] G. Ellis, “Four types of controllers,” *Control System Design Guide*, pp. 97–119, 2012.
- [31] “A detailed history of 3d printing,” 3D Insider, 03 2017. [Online]. Available: <https://3dinsider.com/3d-printing-history/>
- [32] J. T. Cantrell, S. Rohde, D. Damiani, R. Gurnani, L. DiSandro, J. Anton, A. Young, A. Jerez, D. Steinbach, C. Kroese, and P. G. Ifju, “Experimental characterization of the mechanical properties of 3d-printed abs and polycarbonate parts,” *Rapid Prototyping Journal*, vol. 23, no. 4, pp. 811–824, 2017.
- [33] “Manipulating industrial robots - Performance criteria and related test methods,” International Organization for Standardization, Standard, Apr. 1998. [Online]. Available: <https://www-saiglobal-com.ezproxy.uow.edu.au/online/Product/Index/isoc000739255>
- [34] G. Kuric Ivan, Tlach Vladimír, Ságová Zuzana, Císar Miroslav, “Measurement of industrial robot pose repeatability,” *MATEC Web Conf.*, vol. 244, pp. 1–9, 2018.
- [35] “Dassault systèmes,” World Economic Forum, 2022. [Online]. Available: <https://www.weforum.org/organizations/dassault-systemes>
- [36] ros, “ros solidworks urdf exporter solidworks to urdf exporter,” GitHub, 09 2021. [Online]. Available: https://github.com/ros/solidworks_urdf_exporter
- [37] “ros_control_boilerplate - ros wiki,” Ros.org, 2019. [Online]. Available: http://wiki.ros.org/ros_control_boilerplate
- [38] “Planners | moveit,” Ros.org, 2022. [Online]. Available: <https://moveit.ros.org/documentation/planners/>
- [39] “Brushless vs brushed dc motors: When and why to choose one over the other | article | mps,” Monolithicpower.com, 2020. [Online]. Available: <https://www.monolithicpower.com/en/brushless-vs-brushed-dc-motors>
- [40] “Rep 3 – target platforms (ros.org),” Ros.org, 2011. [Online]. Available: <https://www.ros.org/repos/rep-0003.html>

Original Project Specifications

This page is intentionally left blank.



**SCHOOL OF ELECTRICAL, COMPUTER AND TELECOMMUNICATIONS
ENGINEERING
ECTE456 PROJECT PROPOSAL FORM**

1. Candidate Details	
Name: Brock Cooper	Student No: 5791340
Supervisor: Dr Prashan Premaratne	
Title of Project: To benchmark and determine the performance of a 3D printed 6 degrees of freedom (DOF) robotic arm.	
Brief Overview: Robotic arms are used in various tasks and are designed for specific applications to automate certain tasks. Industry developed robotic arms can have different degrees of freedom (DOF) consisting of various joint types. These are the total number of independent displacements or motions that an armature can move. When designing robotic arms, theoretical calculations can be determined for the design's operational capacity by using the datasheets of the components. However, this won't provide insight into the real-world performance of the DIY 3D printed arm. Therefore, a series of standardised benchmarking tests designed to determine the performance of robotic arms will be applied. Through undertaking these experiments and critically analysing the data for the DIY robotic arm, it will demonstrate the capabilities of the 6 DOF robotic arm and outline applications the arm would be best suited towards. A 3D printed arm will be designed and integrated with ROS (Robot Operating System) for more advanced path/trajectory planning to operate more complex tasks with optimal path planning. This will allow for manipulating the arm to carry out an experimental design setup to test the arm's performance.	

2. Project Description:

Robotic arms have been used in industry for quite some time now and are becoming more accessible to everyday consumers. However, many closed source software doesn't allow the end-user to experiment with their concepts and customise. The main concern is having more advanced control over a robotic arm, which can aid in benchmarking for determining machine accuracy. The main focus will be determining the pose repeatability of the device through experimentation further determined through research and ISO 9283:1998 standards.

A major issue when designing 3D printed robotic arms is the control system and how to benchmark the final design accurately. Previous studies show that printed arms can articulate each joint to a certain angle but cannot have more complex inverse kinematic solvers. This would allow the user to have more control that would be beneficial for accurately benchmarking a 3D printed 6-axis robotic arm through a series of manipulation tests, moving all 6 joints to and from a position and determining the performance and capabilities. Using an advanced inverse kinematic solver with a more industrial approach will allow adapting industrial testing methods to open-source designs.

The project is essential as it addresses possible design flaws present in 3D printing Robotic arms and helps develop a more accurate, robust design. The data obtained through experimentation could be applied in future concepts to improve on any design flaws that are currently present and limit the operational performance. In turn, could help in the progress of the development of 3D printed robotic arms for rapid prototyping and manufacturing.

Project Outcomes/Objectives:

- Comprehensive Literature Review
- Developing CAD Model with Unified Robot Definition Format (URDF) model for RViz and MoveIT
- Create a C++ and Python program to integrate with ROS
- Design an affordable low cost 6 axes robotic arm that can be almost fully 3D printed.
- Understanding ROS communication protocol
- Setup testing rigs to conduct a performance test on the arm
- Develop a Report, Present Findings through Presentations, Video Demonstration and Posters
- Have a functional Robotic Arm that can be used to complete set tasks with complex path planning

3. Project Plan:

To critically analyse a 3D printed robotic arm, an extensive literature review will be required referencing different techniques and standards that are currently used during the Summer session. It is crucial to undertake a comprehensive review to have a fair testing ground that other research studies have undertaken. Whilst researching into using ROS as a control system. The main plan to follow consist of 3 major steps being:

Develop of 3D CAD model

The first object consists of breaking down the current robotic arm and developing a CAD model and assembly that will be required to make the unified robot description format (URDF). This file will be necessary to create a 3D model simulation using RViz and MoveIT plugin in ROS. The model will be generated using Solidworks, which will enable exporting the URDF model required for creating the ROS package.

3D Printing and Assembling Low-Cost Robotic Arm

3D printing will be a significant component of the design as all the parts of the arm will be printed from PETG plastic, this plastic will be used as it can be printed in both enclosed and non-enclosed printers due to the lower glass transition temperature. Meaning if an enclosed printer is not an option, it can still be printed with a stronger tensile strength than PLA.

Control method with complex inverse kinematic solver

To write a C/C++ program that can interface with Robot Operating System (ROS) using an ATmega2560 as the main control board for controlling each of the motors that articulate each joint. This program will consist of determining how ROS publishes data and being able to decode and use the information in Arduino to have accurate motor control. The added functionality of using ROS as a complex inverse kinematic solver also has advanced collision detection, eliminating the robot from overextending or hitting objects or colliding with it's own links. A PID controller will need to be implemented in either ROS or the Arduino Program to control the Motor positions more accurately.

Experimentation/method

To calculate the performance and error of the arm through a series of tests following similar procedures to past experiments. The primary method will require a digital/analogue dial indicator that will be used to measure the repeatability of the arm and used for calculating positional error. This method is being adopted as it follows the ISO 9283:1998 standards used to test industrial robotic arms performance.

The experimental data won't necessarily have any baseline to compare against being a one-off design but rather allow for analysing the robotic arm physically and determining any deviations that could be causing specific trends in the data. If the method is carried out in a similar format to the ISO9283:1998 standards, then the results should accurately represent the accuracy of the arm.

Main Objectives for Semester 2:

1. Add to literature Review
2. Setup ROS Controller and Integrate with 6 Motors
3. Finish off CAD model and create URDF Package
4. Create a ROS Launch Package of 6 Axis Robotic Arm
5. Incorporate a Homing Routine
6. Finish 3D Printing and Building Robotic arm
7. Finish Wiring all the Hardware together
8. Run through a series of Experiments and synthesise the results

Semester 2 Planner

To benchmark and determine the performance of a 3D printed 6 degrees of freedom (DOF) robotic arm.

Task	Week
Design CAD Model for 6 Axis Arm	2-5
Wire Up Control Box	3-5
3D Print All Arm Parts	5-6
Finish writing Code to Integrate 6 Motor Controllers for ROS	5-6
Assemble Arm 3D Printed Components	6-7
Wire all motors with connectors for control unit	6-7
Conduct Experimental Analysis	8-9
Write up results and methodology	9-12
Design Presentation with Video Demonstration	11-13
Design Poster	12-14

4. Resources Required: (Expand to a half a page maximum)**Software:**


- Robotic Operating system (ROS)
- Arduino
- Ubuntu Linux Focal Fossa 20.04
- Solidworks 2018 – Student License computer-aided design (CAD)

Hardware:

- ATmega 2560 (Arduino)
- 12V DC Encoder Motors x6
- 12V Computer PSU (Power Supply)
- 3D Printer – Print Test Mounts
- CNC Fibre Optic Laser Cutter
- Measurement Devices (eg. Dial Indicator)

Student Signature

Declaration by the student: I have understood the feedback provided to me by the supervisor.

	Signature	Date
Student Name: Brock Cooper		20/03/2022

Note: the typical overall page count should not exceed 15 pages

Logbook Summary Sheet

This page is intentionally left blank.

ROS Flowchart

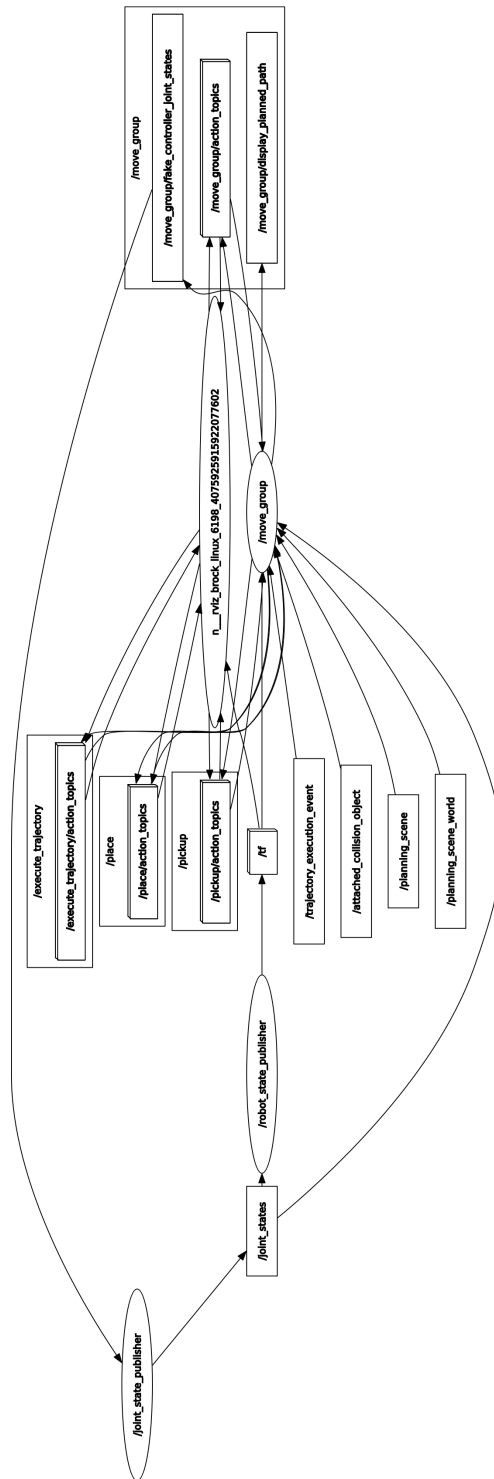


Figure C.1: Nodes and Topics Overview for Demo Controller

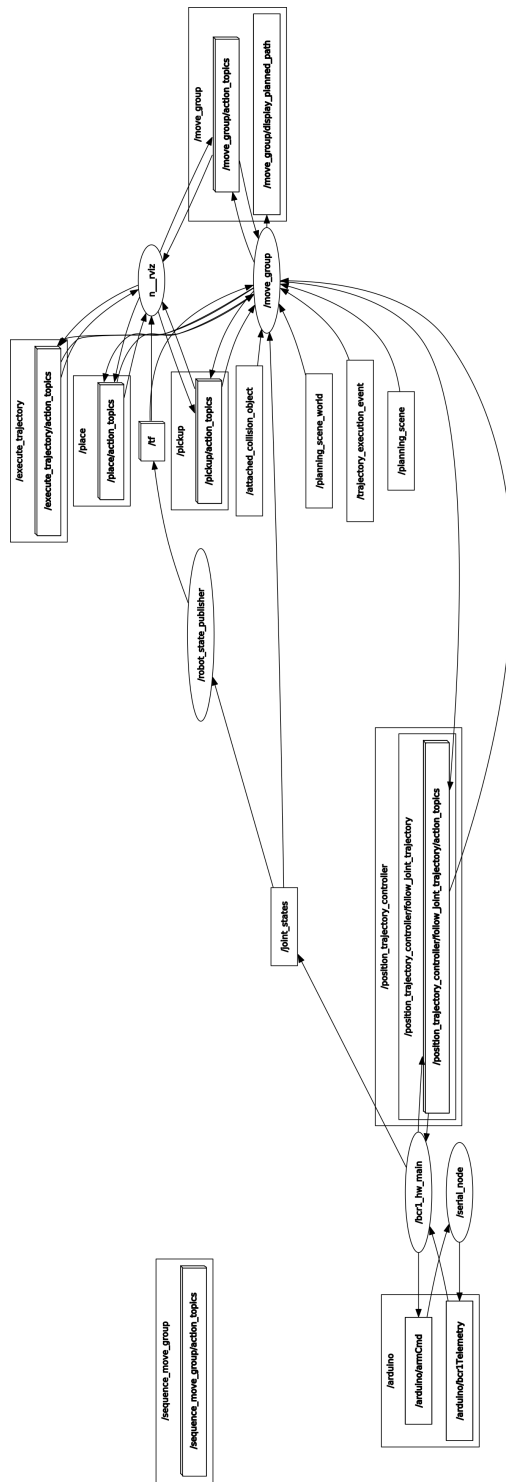


Figure C.2: Nodes and Topics Overview for BCR1

ROS Distributions

Table D.1: List of ROS Distributions and Lifespan [40].

ROS Distribution	Release Date	EOL	Supported Ubuntu System
ROS Noetic Ninjemys	May 23rd, 2020	May, 2025	Ubuntu Focal Fossa (20.04)
ROS Melodic Morenia	May 23rd, 2018	May, 2023	Ubuntu Artful (17.10) Ubuntu Bionic (18.04)
ROS Lunar Loggerhead	May 23rd, 2017	May, 2019	Ubuntu Xenial (16.04) Ubuntu Yakkety (16.10) Ubuntu Zesty (17.04)
ROS Kinetic Kame	May 23rd, 2016	Apr. 2021	Ubuntu Wily (15.10) Ubuntu Xenial (16.04)
ROS Jade Turtle	May 23rd, 2015	May, 2017	Ubuntu Trusty (14.04) Ubuntu Utopic (14.10) Ubuntu Vivid(15.04)
ROS Indigo Igloo	Jul. 22nd, 2014	Apr. 2019	Ubuntu Saucy (13.10) Ubuntu Trusty (14.04 LTS)
ROS Hydro Medusa	Sept. 4th, 2013	May, 2015	Ubuntu Precise (12.04 LTS) Ubuntu Quantal (12.10) Ubuntu Raring (13.04)
ROS Groovy Galapagos	Dec. 31, 2012	Jul. 2014	Ubuntu Oneiric (11.10) Ubuntu Precise (12.04 LTS) Ubuntu Quantal (12.10)
ROS Fuerte Turtle	Apr. 23, 2012	–	Ubuntu Lucid (10.04 LTS) Ubuntu Oneiric (11.10) Ubuntu Precise (12.04 LTS)
ROS Electric Emys	Aug. 30, 2011	–	Ubuntu Lucid (10.04 LTS) Ubuntu Maverick (10.10) Ubuntu Natty (11.04) Ubuntu Oneiric (11.10)
ROS Diamondback	Mar. 2, 2011	–	Ubuntu Lucid (10.04 LTS) Ubuntu Maverick (10.10) Ubuntu Natty (11.04)
ROS C Turtle	Aug. 2, 2010	–	Ubuntu Jaunty (9.04) Ubuntu Karmic (9.10) Ubuntu Lucid (10.04 LTS) Ubuntu Maverick (10.10)
ROS Box Turtle	Mar. 2, 2010	–	Ubuntu Hardy (8.04 LTS) Ubuntu Intrepid (8.10) Ubuntu Jaunty (9.04) Ubuntu Karmic (9.10)

MATLAB Code

E.1 BCR1 Forward Kinematics

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  % 6 AXIS ROBOT THESIS ARM (bcr1)
3  % Author: Brock Cooper
4  % SN: 5791340
5  % Date: 4/03/2022
6  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
7  clear all
8  clc
9
10 syms theta d a alpha
11
12 T = vpa([(cos(theta)) (-cos(alpha))*(sin(theta)) (sin(alpha))*(sin(theta)) a*(cos(
13     theta)) ;...
14     (sin(theta)) (cos(alpha))*(cos(theta)) (-sin(alpha))*(cos(theta)) a*(sin(
15     theta)) ;...
16     0 (sin(alpha)) (cos(alpha)) d
17     ;...
18     0 0 0 0 1
19     ...
20     ]));
21
22 syms q1 q2 q3 q4 q5 q6;
23 syms d1 d2 d3 d4 d5 d6;
24 syms a1 a2 a3 a4 a5 a6;
25 syms alpha1 alpha2 alpha3 alpha4 alpha5 alpha6;
26 % syms pi
27
28 % INPUT PARAMETERS
29 % theta_in = [q1 q2+(pi/2) q3 q4 q5 q6]; % Enter in terms radians (pi)
30 % d_in = [d1 0 0 d4 0 d6]; % Enter in decimal
31 % a_in = [0 a2 0 0 0 0]; % Enter in decimal
32 % alpha_in = [pi/2 0 pi/2 -pi/2 pi/2 0]; % Enter in terms of Radians
33
34 % INPUT PARAMETERS
35 theta_in = [0 0+(pi/2) 0 0 0 0]; % Enter in terms radians (pi)
36 d_in = [141.5 0 0 183.6 0 134.5]; % Enter in decimal
37 a_in = [0 178.0 0 0 0 0]; % Enter in decimal
38 alpha_in = [pi/2 0 pi/2 -pi/2 pi/2 0]; % Enter in terms of Radians
39
40 % Calculating from Base to Wrist
41 disp("First 3 matrices from the base to the wrist are seen below:")
42 T1_0 = vpa(subs(T,{theta ,d,a ,alpha},{theta_in(1) , d_in(1) , a_in(1) , alpha_in(1)}))
43 T2_1 = vpa(subs(T,{theta ,d,a ,alpha},{theta_in(2) , d_in(2) , a_in(2) , alpha_in(2)}))
44 T3_2 = vpa(subs(T,{theta ,d,a ,alpha},{theta_in(3) , d_in(3) , a_in(3) , alpha_in(3)}))

```

```

41
42 tool_b1 = simplify(T1_0*T2_1)
43 disp('multiply by T3_2')
44 disp('Final Tool Base Wrist is:')
45 tool_bw = simplify(tool_b1*T3_2) % T_Base→Wrist
46
47 % Calculating from Wrist to Tool
48 disp("Second 3 matricies from the wrist to the Tool are seen below:")
49 T4_3 = vpa(subs(T,{theta,d,a,alpha},{theta_in(4), d_in(4), a_in(4), alpha_in(4)}))
50 T5_4 = vpa(subs(T,{theta,d,a,alpha},{theta_in(5), d_in(5), a_in(5), alpha_in(5)}))
51 T6_5 = vpa(subs(T,{theta,d,a,alpha},{theta_in(6), d_in(6), a_in(6), alpha_in(6)}))
52
53 tool_w1 = simplify(T4_3*T5_4)
54 disp('multiply by T6_5')
55 disp('Final Tool Base Wrist is:')
56 tool_wt = simplify(tool_w1*T6_5) % T_Wrist→Tool
57
58 disp('Final Base to Tool matrix is:')
59 tool_bt = simplify(tool_bw*tool_wt) % T_Base→Tool

```

E.2 BCR1 Repeatability Point Cloud Plot

```

1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 % BCR1 Repeatability Point Cloud
3 % Author: Brock Cooper
4 % SN: 5791340
5 % Date: 27/04/2022
6 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
7 close all
8 clear all
9 close all
10 load('arm_ws')
11 load('MyColormap')
12 xpos = ArmRepeatabilityDATAS3(1:end,1)
13 ypos = ArmRepeatabilityDATAS3(1:end,2)
14 zpos = ArmRepeatabilityDATAS3(1:end,3)
15
16 %%
17 % figure(1);
18 subplot(2,2,1);
19 plot(0,0, 'g', ...
20      'LineWidth', 0.05, ...
21      'MarkerSize', 60)
22 hold all;
23 grid on;
24 colormap(mymap)
25 c = sqrt(xpos.^2 + ypos.^2 + zpos.^2);
26 sz = 50
27 scatter3(xpos, ypos, zpos, sz, c, 'filled');
28 colorbar
29 fontSize = 24;
30
31 caption = sprintf('BCR1 Repeatability in X,Y,Z 3D Space');
32 title(caption, 'FontSize', fontSize);
33 xlabel('x-axis [mm]', 'FontSize', fontSize);
34 ylabel('y-axis [mm]', 'FontSize', fontSize);
35 zlabel('z-axis [mm]', 'FontSize', fontSize);

```

```

36 view(45,45)
37
38 %% plot x-y
39 subplot(2,2,2);
40 colorbar
41 plot(0,0,'.g',...
42      'LineWidth',0.05,...
43      'MarkerSize',60)
44 hold on;
45 grid on;
46 colormap(mymap)
47 c = sqrt(xpos.^2 + ypos.^2);
48 sz = 50
49 scatter(xpos, ypos, sz, c, 'filled');
50
51 caption = sprintf('BCR1 Repeatability in X,Y Plane');
52 title(caption, 'FontSize', fontSize);
53 xlabel('x-axis [mm]', 'FontSize', fontSize);
54 ylabel('y-axis [mm]', 'FontSize', fontSize);
55
56 %% plot x-z
57 subplot(2,2,3);
58 colorbar
59 plot(0,0,'.g',...
60      'LineWidth',0.05,...
61      'MarkerSize',60)
62 hold on;
63 grid on;
64 colormap(mymap)
65 c = sqrt(xpos.^2 + zpos.^2);
66 sz = 50
67 scatter(xpos, zpos, sz, c, 'filled');
68
69 caption = sprintf('BCR1 Repeatability in X,Z Plane');
70 title(caption, 'FontSize', fontSize);
71 xlabel('x-axis [mm]', 'FontSize', fontSize);
72 ylabel('z-axis [mm]', 'FontSize', fontSize);
73
74 %% plot y-z
75 subplot(2,2,4);
76 colorbar
77 plot(0,0,'.g',...
78      'LineWidth',0.05,...
79      'MarkerSize',60)
80 hold on;
81 grid on;
82 colormap(mymap)
83 c = sqrt(ypos.^2 + zpos.^2);
84 sz = 50
85 scatter(ypos, zpos, sz, c, 'filled');
86
87 caption = sprintf('BCR1 Repeatability in Y,Z Plane');
88 title(caption, 'FontSize', fontSize);
89 xlabel('y-axis [mm]', 'FontSize', fontSize);
90 ylabel('z-axis [mm]', 'FontSize', fontSize);

```

Hardware Code

F.1 ROS Hardware Interface

F.1.1 bcr1_hw_interface.cpp

```

1  /*****
2  bcr1_hw_interface.cpp
3  Author: Brock Cooper
4  SN: 5791340
5  Date: 12/03/2022
6  *****/
7
8  #include <bcr1_control/bcr1_hw_interface.h>
9
10 namespace bcr1_ns
11 {
12     bcr1HWInterface::bcr1HWInterface(ros::NodeHandle &nh, urdf::Model *urdf_model)
13         : ros_control_boilerplate::GenericHWInterface(nh, urdf_model)
14     {
15         telemetry_sub = nh.subscribe("/arduino/bcr1Telemetry", 1, &bcr1HWInterface::
16             telemetryCallback, this);
17
18         cmd_pub = nh.advertise<bcr1_control::armCmd>("arduino/armCmd", 1);
19
20         ROS_INFO("bcr1HWInterface constructed");
21     }
22
23     void bcr1HWInterface::telemetryCallback(const bcr1_control::bcr1Telemetry::
24         ConstPtr &msg)
25     {
26         /*
27         float32 [6] angle # degrees
28
29         States
30         std::vector<double> joint_position_;
31         std::vector<double> joint_velocity_;
32         std::vector<double> joint_effort_;
33         */
34         for (int i = 0; i < num_joints_; i++)
35         {
36             joint_position_[i] = msg->angle[i] * DEG_TO_RAD;
37         }
38     }
39     void bcr1HWInterface::init()

```

```

40  {
41    // Call parent class version of this function
42    GenericHWInterface::init();
43
44    ROS_INFO("bcr1HWInterface Ready.");
45  }
46
47  void bcr1HWInterface::read(ros::Duration &elapsed_time)
48  {
49    // No need to read since our write() command populates our state for us
50    ros::spinOnce();
51  }
52
53  void bcr1HWInterface::write(ros::Duration &elapsed_time)
54  {
55    // Safety
56    // enforceLimits(elapsed_time);
57
58    /*
59    float32 [6]  effort # 0-255 PWM forArduino
60    float32 [6]  angle # deg
61    uint32 msg_ctr # count sent msgs to detect missed messages
62
63    Commands
64    std::vector<double> joint_position_command_;
65    std::vector<double> joint_velocity_command_;
66    std::vector<double> joint_effort_command_;
67    */
68
69    eff_jnt_sat_interface_.enforceLimits(elapsed_time);
70
71    // joint_effort_limits_[0]=10;
72    static bcr1_control::armCmd arm_cmd;
73    for (int i = 0; i < num_joints_; i++)
74    {
75      arm_cmd.effort[i] = joint_effort_command_[i];
76      // arm_cmd.effort[i] = joint_position_[i]; // testing reading robot angles
77      // into effort message
78      arm_cmd.angle[i] = joint_position_command_[i] * RAD_TO_DEG;
79    }
80
81    cmd_pub.publish(arm_cmd);
82  }
83
84  void bcr1HWInterface::enforceLimits(ros::Duration &period)
85  {
86  }
87 } // namespace bcr1_ns

```

F.1.2 bcr1_hw_main.cpp

```

1  /*****
2  bcr1_hw_main.cpp
3  Author: Brock Cooper
4  SN: 5791340
5  Date: 12/03/2022

```

```

6  *****/
7
8  #include <bcr1_control/bcr1_hw_interface.h>
9
10 namespace bcr1_ns
11 {
12     bcr1HWInterface::bcr1HWInterface(ros::NodeHandle &nh, urdf::Model *urdf_model)
13         : ros_control_boilerplate::GenericHWInterface(nh, urdf_model)
14     {
15         telemetry_sub = nh.subscribe("/arduino/bcr1Telemetry", 1, &bcr1HWInterface::
            telemetryCallback, this);
16
17         cmd_pub = nh.advertise<bcr1_control::armCmd>("arduino/armCmd", 1);
18
19         ROS_INFO("bcr1HWInterface constructed");
20     }
21
22     void bcr1HWInterface::telemetryCallback(const bcr1_control::bcr1Telemetry::
        ConstPtr &msg)
23     {
24
25         /*
26         float32 [6] angle # degrees
27
28         States
29         std::vector<double> joint_position_;
30         std::vector<double> joint_velocity_;
31         std::vector<double> joint_effort_;
32         */
33         for (int i = 0; i < num_joints_; i++)
34         {
35             joint_position_[i] = msg->angle[i] * DEG_TO_RAD;
36         }
37     }
38
39     void bcr1HWInterface::init()
40     {
41         // Call parent class version of this function
42         GenericHWInterface::init();
43
44         ROS_INFO("bcr1HWInterface Ready.");
45     }
46
47     void bcr1HWInterface::read(ros::Duration &elapsed_time)
48     {
49         // No need to read since our write() command populates our state for us
50         ros::spinOnce();
51     }
52
53     void bcr1HWInterface::write(ros::Duration &elapsed_time)
54     {
55         // Safety
56         // enforceLimits(elapsed_time);
57
58         /*
59         float32 [6] effort # 0-255 PWM forArduino
60         float32 [6] angle # deg
61         uint32 msg_ctr # count sent msgs to detect missed messages
62

```

```

63     Commands
64         std::vector<double> joint_position_command_;
65         std::vector<double> joint_velocity_command_;
66         std::vector<double> joint_effort_command_;
67     */
68
69     eff_jnt_sat_interface_.enforceLimits(elapsed_time);
70
71     // joint_effort_limits_[0]=10;
72     static bcr1_control::armCmd arm_cmd;
73     for (int i = 0; i < num_joints_; i++)
74     {
75         arm_cmd.effort[i] = joint_effort_command_[i];
76         // arm_cmd.effort[i] = joint_position_[i]; // testing reading robot angles
77         // into effort message
78         arm_cmd.angle[i] = joint_position_command_[i] * RAD_TO_DEG;
79     }
80     cmd_pub.publish(arm_cmd);
81 }
82
83 void bcr1HWInterface::enforceLimits(ros::Duration &period)
84 {
85
86 }
87 } // namespace bcr1_ns

```

F.1.3 bcr1_hw_interface.h

```

1  /*****
2  bcr1_hw_interface.h
3  Author: Brock Cooper
4  SN: 5791340
5  Date: 12/03/2022
6  *****/
7
8  #ifndef BCR1_HW_INTERFACE_H
9  #define BCR1_HW_INTERFACE_H
10
11 #include <ros_control_boilerplate/generic_hw_interface.h>
12
13 #include <bcr1_control/armCmd.h>
14 #include <bcr1_control/bcr1Telemetry.h>
15
16 #include <effort_controllers/joint_position_controller.h>
17
18 #define DEG_TO_RAD 0.01745329251
19 #define RAD_TO_DEG 57.2957795131
20
21
22 namespace bcr1_ns
23 {
24 /** \brief Hardware interface for a robot */
25 class bcr1HWInterface : public ros_control_boilerplate::GenericHWInterface
26 {
27 public:
28     /**

```

```

29     * \brief Constructor
30     * \param nh – Node handle for topics.
31     */
32     bcr1HWInterface(ros::NodeHandle& nh, urdf::Model* urdf_model = NULL);
33
34     /** \brief Initialize the robot hardware interface */
35     virtual void init();
36
37     /** \brief Read the state from the robot hardware. */
38     virtual void read(ros::Duration& elapsed_time);
39
40     /** \brief Write the command to the robot hardware. */
41     virtual void write(ros::Duration& elapsed_time);
42
43     /** \brief Enforce limits for all values before writing */
44     virtual void enforceLimits(ros::Duration& period);
45
46     protected:
47     ros::Subscriber telemetry_sub;
48     void telemetryCallback(const bcr1_control::bcr1Telemetry::ConstPtr &msg);
49
50     ros::Publisher cmd_pub;
51 }; // class
52
53 } // namespace ros_control_boilerplate
54
55 #endif

```

F.2 Arduino Code

F.2.1 Main Program

```

1  /*****
2  ros_controller_arduino.ino
3  Author: Brock Cooper
4  SN: 5791340
5  Date: 12/03/2022
6  *****/
7  #if (ARDUINO >= 100)
8  #include <Arduino.h>
9  #else
10 #include <WProgram.h>
11 #endif
12 #include <ros.h>
13 #include <sensor_msgs/JointState.h>
14 #include <stdlib.h>
15
16 #include <bcr1_control/armCmd.h>
17 #include <bcr1_control/bcr1Telemetry.h>
18
19 const float pi = 3.14159265359;
20
21 void cmd_cb(const bcr1_control::armCmd& cmd_arm)
22 {
23     effort[0] = cmd_arm.effort[0];

```



```

24   effort [1] = cmd_arm.effort [1];
25   effort [2] = cmd_arm.effort [2];
26   effort [3] = cmd_arm.effort [3];
27   effort [4] = cmd_arm.effort [4];
28   effort [5] = cmd_arm.effort [5];
29 }
30
31
32 ros::NodeHandle nh;
33 ros::Subscriber<bcr1_control::armCmd> sub("/arduino/armCmd", cmd_cb);
34
35 bcr1_control::bcr1Telemetry msg;
36 ros::Publisher pub("/arduino/bcr1Telemetry", &msg);
37
38 // How many motors
39 #define NMOTORS 6
40
41 // Pins
42 const int enca [] = {3, 2, 18, 19, 20, 21};
43 const int encb [] = {34, 35, 36, 37, 38, 39};
44 const int pwm [] = {13, 12, 11, 10, 9, 8};
45 const int in1 [] = {22, 25, 26, 28, 31, 32};
46 const int in2 [] = {23, 24, 27, 29, 30, 33};
47 const int hall [] = {40, 41, 42, 43, 44, 45};
48
49 // Globals
50 long prevT = 0;
51 volatile int posi [] = {0, 0, 0, 0, 0, 0};
52 volatile int hall_s [] = {0, 0, 0, 0, 0, 0};
53 volatile int prevPwmVal [] = {0, 0, 0, 0, 0, 0};
54 volatile int pwmVal [] = {0, 0, 0, 0, 0, 0};
55 float effort [] = {0, 0, 0, 0, 0, 0};
56
57 void setup() {
58   Serial.begin(115200);
59
60   for (int k = 0; k < NMOTORS; k++) {
61     pinMode(enca[k], INPUT);
62     pinMode(encb[k], INPUT);
63     pinMode(pwm[k], OUTPUT);
64     pinMode(in1[k], OUTPUT);
65     pinMode(in2[k], OUTPUT);
66     pinMode(hall[k], INPUT);
67   }
68
69   // homing routine
70   hall_s[0] = !digitalRead(hall[0]);
71   hall_s[1] = !digitalRead(hall[1]);
72   hall_s[2] = !digitalRead(hall[2]);
73   hall_s[3] = !digitalRead(hall[3]);
74   hall_s[4] = !digitalRead(hall[4]);
75   hall_s[5] = !digitalRead(hall[5]);
76
77   delay(1000);
78   int k = 1;
79   while (k < NMOTORS) {
80     if (hall_s[k] == 0 || hall_s[k] == 1) {
81       hall_s[k] = !digitalRead(hall[k]);
82       Serial.println("Homing");

```

```

83     Serial.println(hall_s[k]);
84     if (hall_s[k] == 1) {
85         setMotor(-1, 0, pwm[k], in1[k], in2[k]);
86         Serial.println("entered");
87         k++;
88         Serial.println(k);
89         hall_s[k] = digitalRead(hall[k]);
90         delay(2);
91     }
92     if (k == 0 || k == 1) {
93         setMotor(0, -180, pwm[k], in1[k], in2[k]);
94     } else {
95         setMotor(0, 180, pwm[k], in1[k], in2[k]);
96     }
97 }
98 }
99
100 delay(1000);
101 nh.getHardware()->setBaud(115200);
102 nh.initNode();
103 nh.subscribe(sub);
104
105 nh.advertise(pub);
106
107 attachInterrupt(digitalPinToInterrupt(enca[0]), readEncoder<0>, RISING);
108 attachInterrupt(digitalPinToInterrupt(enca[1]), readEncoder<1>, RISING);
109 attachInterrupt(digitalPinToInterrupt(enca[2]), readEncoder<2>, RISING);
110 attachInterrupt(digitalPinToInterrupt(enca[3]), readEncoder<3>, RISING);
111 attachInterrupt(digitalPinToInterrupt(enca[4]), readEncoder<4>, RISING);
112 attachInterrupt(digitalPinToInterrupt(enca[5]), readEncoder<5>, RISING);
113
114 Serial.println("target pos");
115 }
116
117 void loop() {
118
119     nh.spinOnce();
120     delay(5);
121     // read effort
122     for (int k = 0; k < NMOTORS; k++) {
123         pwmVal[k] = effort[k];
124     }
125
126     // set target position
127     int target[NMOTORS];
128
129     // time difference
130     long currT = micros();
131     float deltaT = ((float) (currT - prevT)) / ( 1.0e6 );
132     prevT = currT;
133
134     nh.spinOnce();
135     delay(5);
136
137     // Read the position
138     int pos[NMOTORS];
139     noInterrupts(); // disable interrupts temporarily while reading
140     for (int k = 0; k < NMOTORS; k++) {
141         pos[k] = posi[k];

```

```

142     }
143     interrupts(); // turn interrupts back on
144
145     msg.angle[0] = float(pos[0]) / (6600 / 360) + 0;
146     msg.angle[1] = float(pos[1]) / (6600 / 360) - 72;
147     msg.angle[2] = float(pos[2]) / (6600 / 360) + 53;
148     msg.angle[3] = float(pos[3]) / (8910 / 360) - 0;
149     msg.angle[4] = float(pos[4]) / (6600 / 360) - 0;
150     msg.angle[5] = float(pos[5]) / (6600 / 360) - 0;
151     pub.publish(&msg);
152     delay(10);
153
154     for (int k = 0; k < NMOTORS; k++) {
155         setMotor(prevPwmVal[k], pwmVal[k], pwm[k], in1[k], in2[k]);
156     }
157
158     for (int k = 0; k < NMOTORS; k++){
159         Serial.print("pos");
160         Serial.println(pos[4]);
161     }
162
163     for (int k = 0; k < NMOTORS; k++) {
164         int prevPwmVal = effort[k];
165     }
166
167     nh.spinOnce();
168     delay(5);
169
170 }
171
172 \\set motor pwm value
173 void setMotor(int prevPwmVal, int pwmVal, int pwm, int in1, int in2) {
174     analogWrite(pwm, fabs(pwmVal));
175     if (pwmVal >= prevPwmVal) {
176         digitalWrite(in1, HIGH);
177         digitalWrite(in2, LOW);
178     }
179     else if (pwmVal <= prevPwmVal) {
180         digitalWrite(in1, LOW);
181         digitalWrite(in2, HIGH);
182     }
183     else {
184         digitalWrite(in1, LOW);
185         digitalWrite(in2, LOW);
186     }
187 }
188
189 \\ read encoders function
190 template <int j>
191 void readEncoder() {
192     int b = digitalRead(encb[j]);
193     if (b > 0) {
194         posi[j]++;
195     }
196     else {
197         posi[j]--;
198     }
199 }

```

F.2.2 Header Files

F.2.2.1 armCmd.h

```
1 #ifndef _ROS_bcr1_control_armCmd_h
2 #define _ROS_bcr1_control_armCmd_h
3
4 #include <stdint.h>
5 #include <string.h>
6 #include <stdlib.h>
7 #include "ros/msg.h"
8
9 namespace bcr1_control
10 {
11
12     class armCmd : public ros::Msg
13     {
14     public:
15         float effort[6];
16         float angle[6];
17
18     armCmd():
19         effort(),
20         angle()
21     {
22     }
23
24     virtual int serialize(unsigned char *outbuffer) const override
25     {
26         int offset = 0;
27         for( uint32_t i = 0; i < 6; i++){
28             union {
29                 float real;
30                 uint32_t base;
31             } u_efforti;
32             u_efforti.real = this->effort[i];
33             *(outbuffer + offset + 0) = (u_efforti.base >> (8 * 0)) & 0xFF;
34             *(outbuffer + offset + 1) = (u_efforti.base >> (8 * 1)) & 0xFF;
35             *(outbuffer + offset + 2) = (u_efforti.base >> (8 * 2)) & 0xFF;
36             *(outbuffer + offset + 3) = (u_efforti.base >> (8 * 3)) & 0xFF;
37             offset += sizeof(this->effort[i]);
38         }
39         for( uint32_t i = 0; i < 6; i++){
40             union {
41                 float real;
42                 uint32_t base;
43             } u_anglei;
44             u_anglei.real = this->angle[i];
45             *(outbuffer + offset + 0) = (u_anglei.base >> (8 * 0)) & 0xFF;
46             *(outbuffer + offset + 1) = (u_anglei.base >> (8 * 1)) & 0xFF;
47             *(outbuffer + offset + 2) = (u_anglei.base >> (8 * 2)) & 0xFF;
48             *(outbuffer + offset + 3) = (u_anglei.base >> (8 * 3)) & 0xFF;
49             offset += sizeof(this->angle[i]);
50         }
51         return offset;
52     }
53
54     virtual int deserialize(unsigned char *inbuffer) override
```

```

55     {
56         int offset = 0;
57         for( uint32_t i = 0; i < 6; i++){
58             union {
59                 float real;
60                 uint32_t base;
61             } u_efforti;
62             u_efforti.base = 0;
63             u_efforti.base |= ((uint32_t) (*(inbuffer + offset + 0))) << (8 * 0);
64             u_efforti.base |= ((uint32_t) (*(inbuffer + offset + 1))) << (8 * 1);
65             u_efforti.base |= ((uint32_t) (*(inbuffer + offset + 2))) << (8 * 2);
66             u_efforti.base |= ((uint32_t) (*(inbuffer + offset + 3))) << (8 * 3);
67             this->effort[i] = u_efforti.real;
68             offset += sizeof(this->effort[i]);
69         }
70         for( uint32_t i = 0; i < 6; i++){
71             union {
72                 float real;
73                 uint32_t base;
74             } u_anglei;
75             u_anglei.base = 0;
76             u_anglei.base |= ((uint32_t) (*(inbuffer + offset + 0))) << (8 * 0);
77             u_anglei.base |= ((uint32_t) (*(inbuffer + offset + 1))) << (8 * 1);
78             u_anglei.base |= ((uint32_t) (*(inbuffer + offset + 2))) << (8 * 2);
79             u_anglei.base |= ((uint32_t) (*(inbuffer + offset + 3))) << (8 * 3);
80             this->angle[i] = u_anglei.real;
81             offset += sizeof(this->angle[i]);
82         }
83         return offset;
84     }
85
86     virtual const char * getType() override { return "bcr1_control/armCmd"; };
87     virtual const char * getMD5() override { return "03
88         eab56b29be0236e78be72f0da0e475"; };
89 };
90
91 }
92 #endif

```

F.2.2.2 bcr1Telemetry.h

```

1 #ifndef _ROS_bcr1_control_bcr1Telemetry_h
2 #define _ROS_bcr1_control_bcr1Telemetry_h
3
4 #include <stdint.h>
5 #include <string.h>
6 #include <stdlib.h>
7 #include "ros/msg.h"
8
9 namespace bcr1_control
10 {
11
12     class bcr1Telemetry : public ros::Msg
13     {
14     public:
15         float angle[6];
16

```

```

17     bcr1Telemetry():
18         angle()
19     {
20     }
21
22     virtual int serialize(unsigned char *outbuffer) const override
23     {
24         int offset = 0;
25         for( uint32_t i = 0; i < 6; i++){
26             union {
27                 float real;
28                 uint32_t base;
29             } u_anglei;
30             u_anglei.real = this->angle[i];
31             *(outbuffer + offset + 0) = (u_anglei.base >> (8 * 0)) & 0xFF;
32             *(outbuffer + offset + 1) = (u_anglei.base >> (8 * 1)) & 0xFF;
33             *(outbuffer + offset + 2) = (u_anglei.base >> (8 * 2)) & 0xFF;
34             *(outbuffer + offset + 3) = (u_anglei.base >> (8 * 3)) & 0xFF;
35             offset += sizeof(this->angle[i]);
36         }
37         return offset;
38     }
39
40     virtual int deserialize(unsigned char *inbuffer) override
41     {
42         int offset = 0;
43         for( uint32_t i = 0; i < 6; i++){
44             union {
45                 float real;
46                 uint32_t base;
47             } u_anglei;
48             u_anglei.base = 0;
49             u_anglei.base |= ((uint32_t) (*(inbuffer + offset + 0))) << (8 * 0);
50             u_anglei.base |= ((uint32_t) (*(inbuffer + offset + 1))) << (8 * 1);
51             u_anglei.base |= ((uint32_t) (*(inbuffer + offset + 2))) << (8 * 2);
52             u_anglei.base |= ((uint32_t) (*(inbuffer + offset + 3))) << (8 * 3);
53             this->angle[i] = u_anglei.real;
54             offset += sizeof(this->angle[i]);
55         }
56         return offset;
57     }
58
59     virtual const char * getType() override { return "bcr1_control/bcr1Telemetry";
60     };
61
62     virtual const char * getMD5() override { return "
63     d5646d2d9986672237331f3ea363f45f"; };
64 }
65 #endif

```

F.2.3 Python MoveIt Commander

```

1 # Python 2/3 compatibility imports
2 from __future__ import print_function
3 from six.moves import input

```

```

4
5 import sys
6 import copy
7 import rospy
8 import moveit_commander
9 import moveit_msgs.msg
10 import geometry_msgs.msg
11
12 from std_msgs.msg import String
13 from moveit_commander.conversions import pose_to_list
14
15 from pyquaternion import Quaternion
16
17 import tf
18 import geometry_msgs
19
20 import time
21
22 moveit_commander.roscpp_initialize(sys.argv)
23
24 robot = moveit_commander.RobotCommander()
25
26 scene = moveit_commander.PlanningSceneInterface()
27
28 group_name = "bcr1_planning"
29 move_group = moveit_commander.MoveGroupCommander(group_name)
30
31 display_trajectory_publisher = rospy.Publisher(
32     "/move_group/display_planned_path",
33     moveit_msgs.msg.DisplayTrajectory,
34     queue_size=20,
35 )
36
37 print("===== Printing robot home position")
38 print(move_group.get_current_rpy())
39 print("")
40
41 def robot_pos(pose):
42     move_group.set_planner_id(planner_id = "PTP") # Pilz Planner
43     move_group.set_planning_time(seconds = 4)
44     # set scaling factor
45     move_group.set_max_acceleration_scaling_factor(value = 0.4)
46     move_group.set_max_velocity_scaling_factor(value = 0.4)
47     pose_goal = geometry_msgs.msg.Pose()
48     pose_goal.position.x = pose[0]
49     pose_goal.position.y = pose[1]
50     pose_goal.position.z = pose[2]
51
52     pose_goal.orientation.x = pose[3]
53     pose_goal.orientation.y = pose[4]
54     pose_goal.orientation.z = pose[5]
55     pose_goal.orientation.w = pose[6]
56
57     move_group.set_pose_target(pose_goal)
58
59     plan = move_group.go(wait=True)
60     # 'stop()' ensures that there is no residual movement
61     move_group.stop()
62     # clear targets after planning with poses.

```

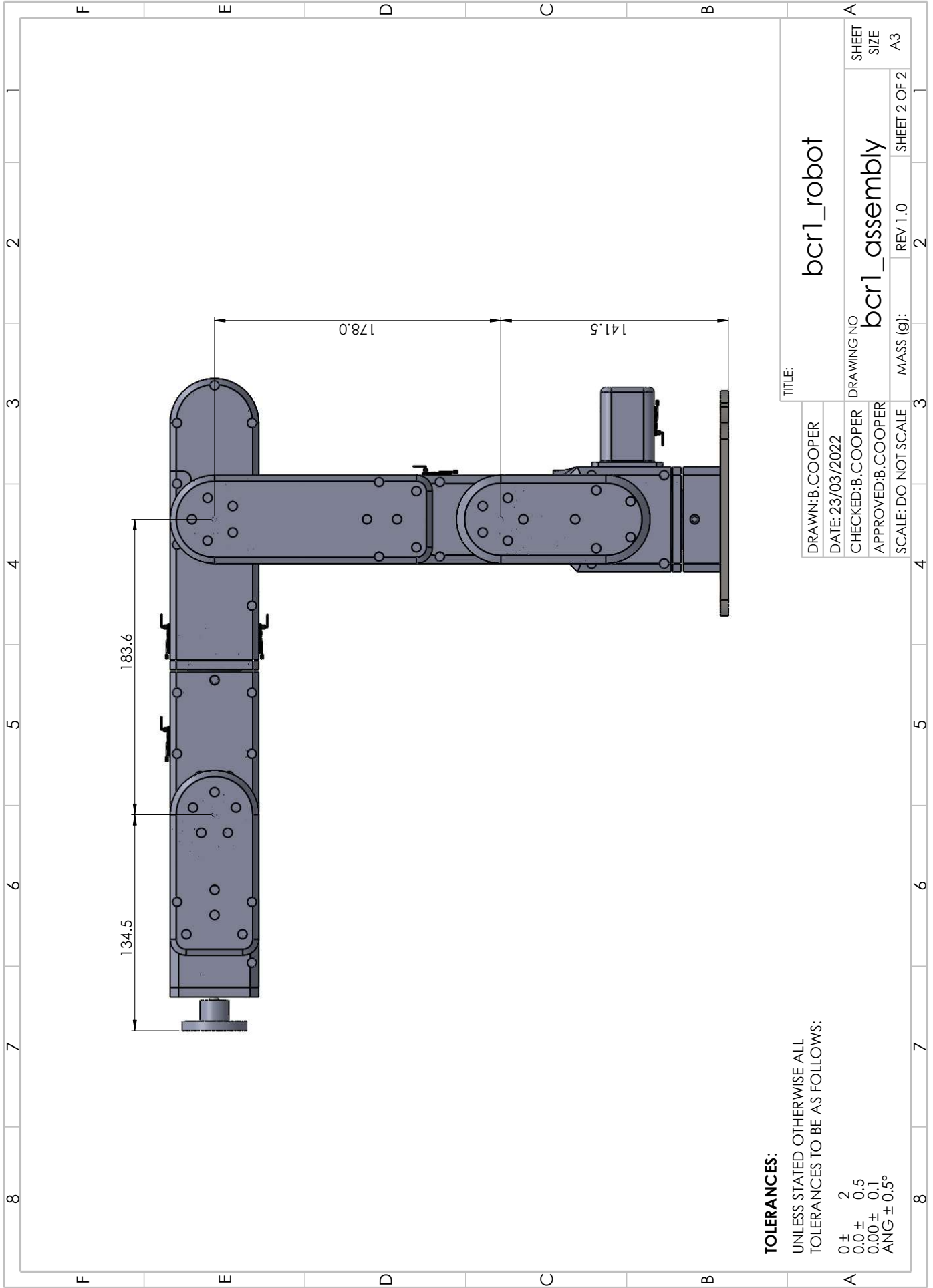
```

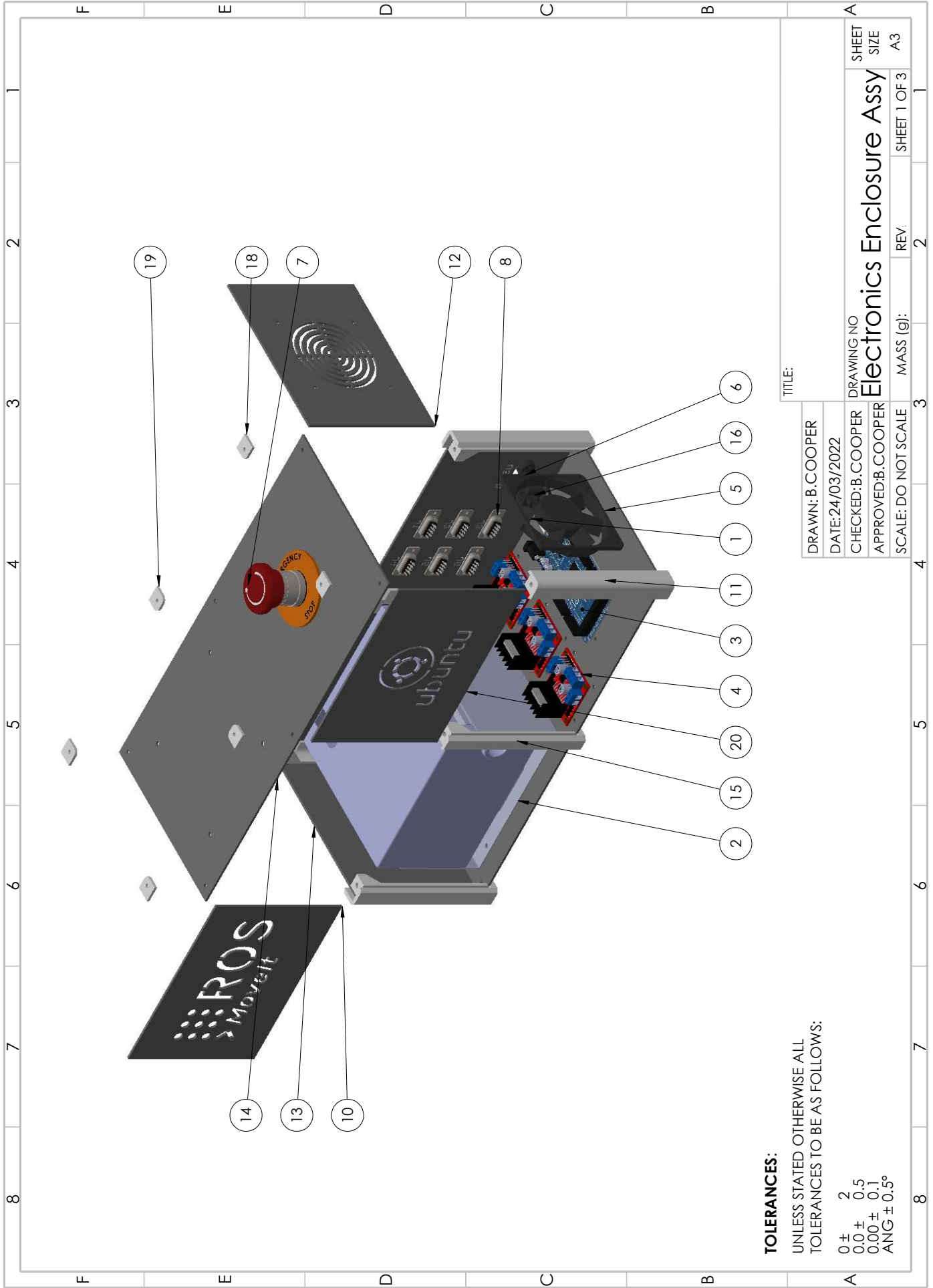
63
64 home = [0, 0, 0, 0, 0, 0]
65
66 i = 1
67 while i < 31:
68     # Position and Rotation for z-axis
69     robot_pos([0.25, 0.15, 0.22, 1, 0, 0, 0])
70     time.sleep(2)
71     robot_pos([0.25, 0.15, 0.22-0.12, 1, 0, 0, 0])
72     time.sleep(15)
73     robot_pos([0.25, 0.15, 0.22, 1, 0, 0, 0])
74     time.sleep(2)
75     move_group.go(home, wait=True)
76     move_group.stop()
77
78     # Position and Rotation for x-axis
79     robot_pos([0.25, 0.15, 0.22, 0.7071068, 0, 0.7071068, 0 ])
80     time.sleep(2)
81     robot_pos([0.25+0.12, 0.15, 0.22, 0.7071068, 0, 0.7071068, 0 ])
82     time.sleep(15)
83     robot_pos([0.25, 0.15, 0.22, 0.7071068, 0, 0.7071068, 0 ])
84     time.sleep(2)
85     move_group.go(home, wait=True)
86     move_group.stop()
87
88     # Position and Rotation for y-axis
89     robot_pos([0.25, 0.15, 0.22, 0.5, 0.5, 0.5, -0.5])
90     time.sleep(2)
91     robot_pos([0.25, 0.15+0.12, 0.22, 0.5, 0.5, 0.5, -0.5])
92     time.sleep(15)
93     robot_pos([0.25, 0.15, 0.22, 0.5, 0.5, 0.5, -0.5])
94     time.sleep(2)
95     move_group.go(home, wait=True)
96     move_group.stop()
97     time.sleep(2)
98     i=i+1
99
100 print("PROCESS SUCCEEDED")

```


CAD Drawings

This page is intentionally left blank.





TOLERANCES:

UNLESS STATED OTHERWISE ALL
TOLERANCES TO BE AS FOLLOWS:

- 0 ± 2
- 0.0 ± 0.5
- 0.00 ± 0.1
- ANG ± 0.5°

TITLE:

DRAWN: B.COOPER	DRAWING NO	SHEET
DATE: 24/03/2022		SIZE
CHECKED: B.COOPER	Electronics Enclosure Assy	A3
APPROVED: B.COOPER	REV:	SHEET 1 OF 3
SCALE: DO NOT SCALE	MASS (g):	

ITEM NO.	PART NUMBER	DESCRIPTION	QTY.
1	EE_MOUNTING_HOLES_BOTTOM	Enclosure Base Plate (2mm 5052 Aluminium)	1
2	ATX Power Supply	Dell ATX Power Supply (PSU)	1
3	arduino MEGA2650	ATmega 2560 Microcontroller	1
4	L298N Driver DC Motor, Stepper Motor.step	L298N Dual H-Bridge Motor Drivers	3
5	80mm brushless fan (SUNON KDE1208PTB2)	80mm brushless fan (SUNON KDE1208PTB2)	1
6	EE_PORTS_SIDE	D-SUB 9 Panel Mount (2mm 5052 Aluminium)	1
7	E-Stop With Housing	Emergency Stop Button	1
8	A_DB9	D-SUB 9 Connector	6
9	EE_PSU_SIDE	Power Supply Mounting Plate (2mm 5052 Aluminium)	1
10	EE_ROS_LOGO_FRONT	Front Panel ROS Logo (2mm 5052 Aluminium)	1
11	EE_Corner_Brackets	Corner Brackets (3D Printed PETG)	4
12	EE_FAN_SIDE	Fan Mount Plate (2mm 5052 Aluminium)	1
13	EE_BLANK_SIDE	Blank Side Plate (2mm 5052 Aluminium)	1
14	EE_TOP	Top Mounting Plate (2mm 5052 Aluminium)	1
15	EE_Straight_Brackets	Middle Brackets (3D Printed PETG)	2
16	USB - Type B - Female - Panel Mount	USB-B Extension Panel Mount	1
17	usb-B_panelMount	USB-B Panel Mount Fascia	1
18	Top_Panel_Corner_Washer	Top Panel Corner Washer (3D Printed PETG)	4
19	Top_Panel_Middle_Washer	Top Panel Middle Washer (3D Printed PETG)	2
20	EE_UBUNTU_LOGO_FRONT	Front Panel Ubuntu Logo (2mm 5052 Aluminium)	1

TOLERANCES:

UNLESS STATED OTHERWISE ALL TOLERANCES TO BE AS FOLLOWS:

- 0 ± 2
- 0.0 ± 0.5
- 0.00 ± 0.1
- ANG ± 0.5°

TITLE:

DRAWN: B.COOPER
DATE: 24/03/2022
CHECKED: B.COOPER
APPROVED: B.COOPER
SCALE: DO NOT SCALE

DRAWING NO	Electronics Enclosure Assy	SHEET
		SIZE
MASS (g):	REV:	A3
3	2	1

Wiring Schematic

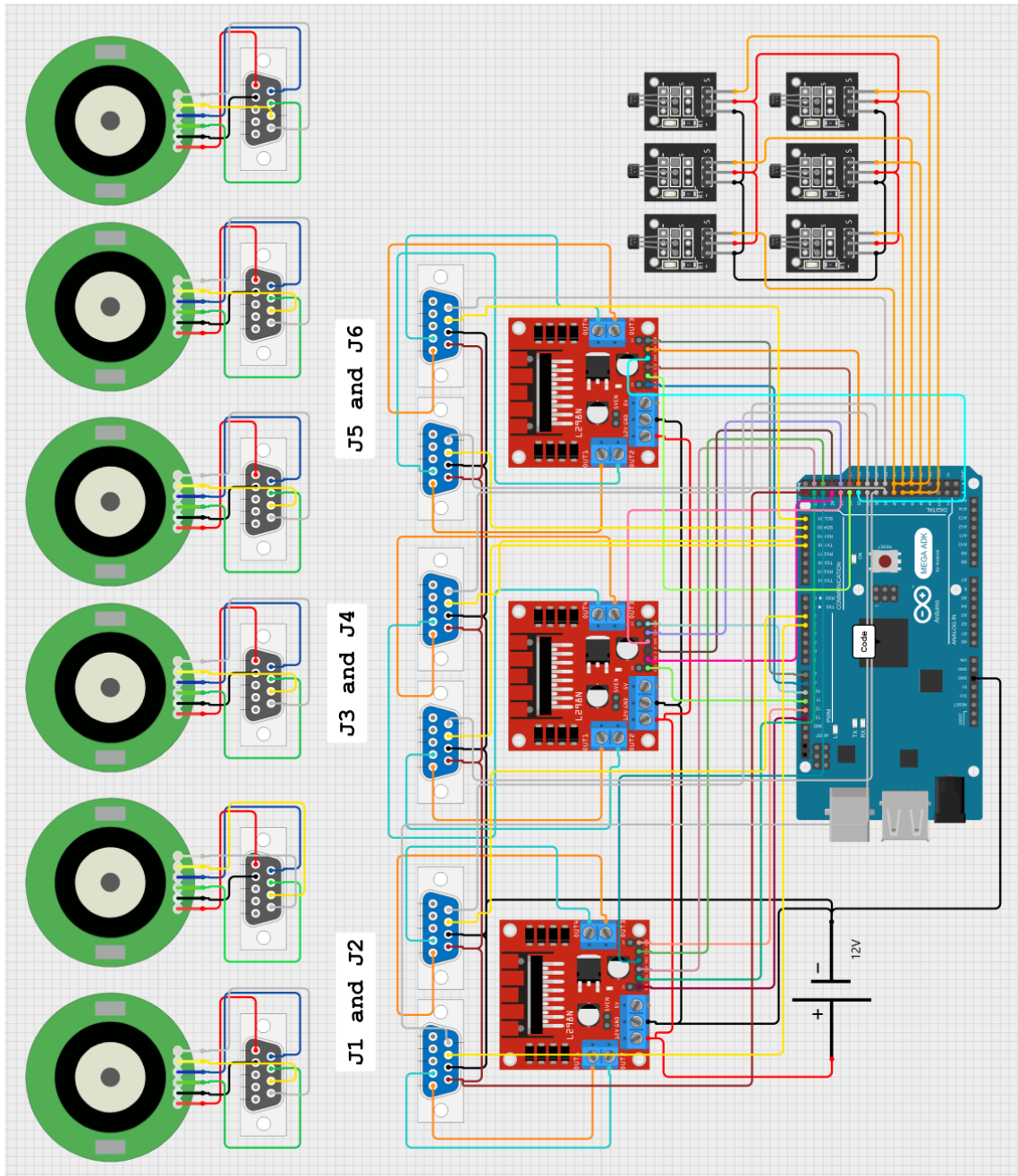


Figure H.1: Control Box Wiring Diagram