ROCO224

Design and control of Manipulators

BFA MK-1

Matt Shaw(10548340), Jon Smith(10549502), Charlotte Perry(10547638) and Estilla Hefter(10543525)

5-14-2018

CONTENTS

Github repository: https://github.com/Jonsmith9847/ROCO-224-BFA

BRIEF

BACKGROUND:

Currently there are large-scale robot arms being used in space operations, for example on the International Space Station (ISS). These robot manipulators are used to help assemble the station, move astronauts around (with foot restraints), and to dock some spacecraft to the station. These manipulators are typically controlled by astronauts inside the space station, however there has been significant interest in teleoperating such robot arms, in orbit and perhaps on the moon, from ground stations on earth. This would be enabled by multiple camera views of the arm in operation.

Space comes with its own set of unique challenges that require a significant change in the way systems are designed, built and operated. Our core design relies upon the system being able to work within proximity of its own structure (self-intersection) and pack down into a low-volume package for transport. The software implementation requires that the system is tele-operated and has forward kinematics so that the final arm state can be calculated real time.

The real-time calculation of arm position is important so that the user can quickly estimate the arm position without having to wait for the real arm to move. The advantages of using robot manipulators to replace EVA's are wide and far reaching. Some examples include:

- Increased safety for Astronauts (fewer EVAs)
- Ground engineer controllable
- Can perform pre-planned manoeuvres
- Frees up Astronaut time for less menial maintenance tasks

The core challenge of operating in space is the effects of transmission delay making real time control of robot manipulators challenging. Using Kinematics, the controller will could determine if a movement is safe before sending the movement command. Our system will be designed to work at within lunar orbit, therefore we will simulate a transmission delay of 1.3 Seconds for control inputs.

Our research has shown that there are already space manipulators in use such as the Mobile Servicing System (MSS) on board the ISS. The MSS is used to perform assembly, maintenance and capture/docking of resupply vessels. This system is controllable from the ground and on board.

GOAL:

Our goal was to design a small robot manipulator for simulating space teleoperation and complete a simple task with only a limited camera view. On top of that we need to add the time delay that occurs when controlling a system on the Moon from Earth.

The main goal is to control the arm from a separate room, only using the camera view that we set up before the start of the task.

INTRODUCTION

DESCRIPTION OF HARDWARE

Actuators: Dynamixel RX-28/MX-28R

We used Dynamixel servo-motors to move the arm and operate the end effector. The servos where chosen because they use PID control and provide positional feedback. This enables us to assume the end position of the arm with a reasonable degree of accuracy. They also use a standard connector which encapsulates all power and communication requirements.

Raspberry Pi 3 model B

We are using the Pi as an on-system controller for the robot arm. The Pi is responsible for running ROS and controlling the Dynamixel motors. ROS will read the Joint angle commands and publish them to the motors. The Pi will be connected to the ROS network via ethernet.

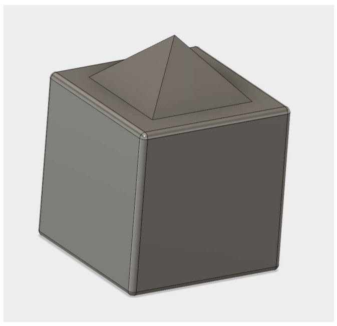
Wireless router

The wireless router is used to connect the remote laptop/control system to the arm. The router will provide both the Pi and control system with an IP address. The IP addresses will then be used to SSH into
the robot arm and send ROS messages to the network.

DESCRIPTION OF TASK

The task we set ourselves was to pick up and stack cubes (space components) using the robot arm from a remote location. The remote location cannot be in sight of the robot arm and must use a simulated lunar transmission delay.

The test cubes are shown below:



The test cubes include auto-registration in the form of a pyramid structure to assist with the stacking of cubes. This design is inspired by how most space docking ports feature auto-registration to complete the final alignment and fitting.

A brief overview of our approach to the problem

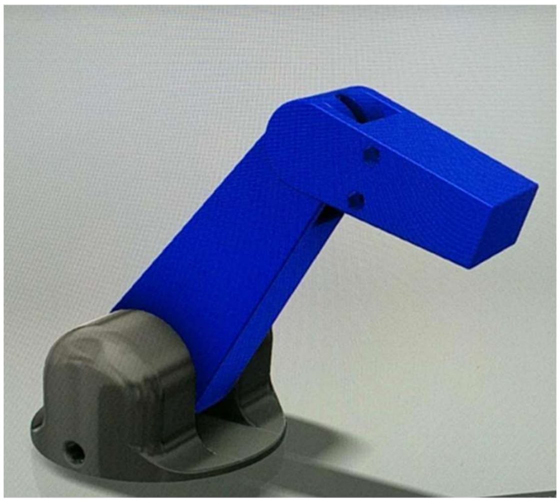
We decided to build the arm first ensuring that we had an appropriate test platform for which we can then design a software implementation to match. We began by using CAD to produce prototype arm parts. The parts helped inform our final design ideas. The hardware would be assembled completely as early in the process as possible so that a final software solution could be implemented.

Design Process

Initial Ideas

Pre-designing

We decided to try to design a preliminary design with some of our own motors for testing purposes. We designed the base turret of the arm, making it one piece with removeable caps to conceal the servo and wiring arrangement. However, after the test print of this piece which is shown below, we realised the design will be inefficient for taking apart and putting back together and caused unnecessary weight to be put on the servos.



So, we ended up not using this design and went back to designing an arm to accommodate the RX-28 servos.

After receiving the motors:

The decision on the arm form-factor that would be used for the space manipulator was a turret-mounted RRR arm with a further servo to rotate the end-effector. This is very similar in concept to traditional industrial manipulators such as the Mitsubishi MELFA series arms. This form-factor would give 6 degrees of freedom in operation which is suitable for complex operations in the orbital working environment.

As the arm will need to be transported into orbit, where available volume is at a premium, we decided to explore the possibility of making the arm fully self-intersecting. This would improve operational dexterity of the arm and allow the arm to fold into a small package for transportation. Furthermore, this requirement for ease of transportation led us to consider options for fully integrating the control systems for the arm into the arm structure, leaving the minimum of external peripherals. This would, for example, mean securing the Dynamixel-to-USB adapter and Dynamixel power supply board within the arm.

To improve structural integrity and stability, we decided to position all the servos in-plane at the joints rather than off-axis. This will, by extension, improve the stiffness of the arm, making the arm's dynamic behaviour more accurate to the kinematic model.

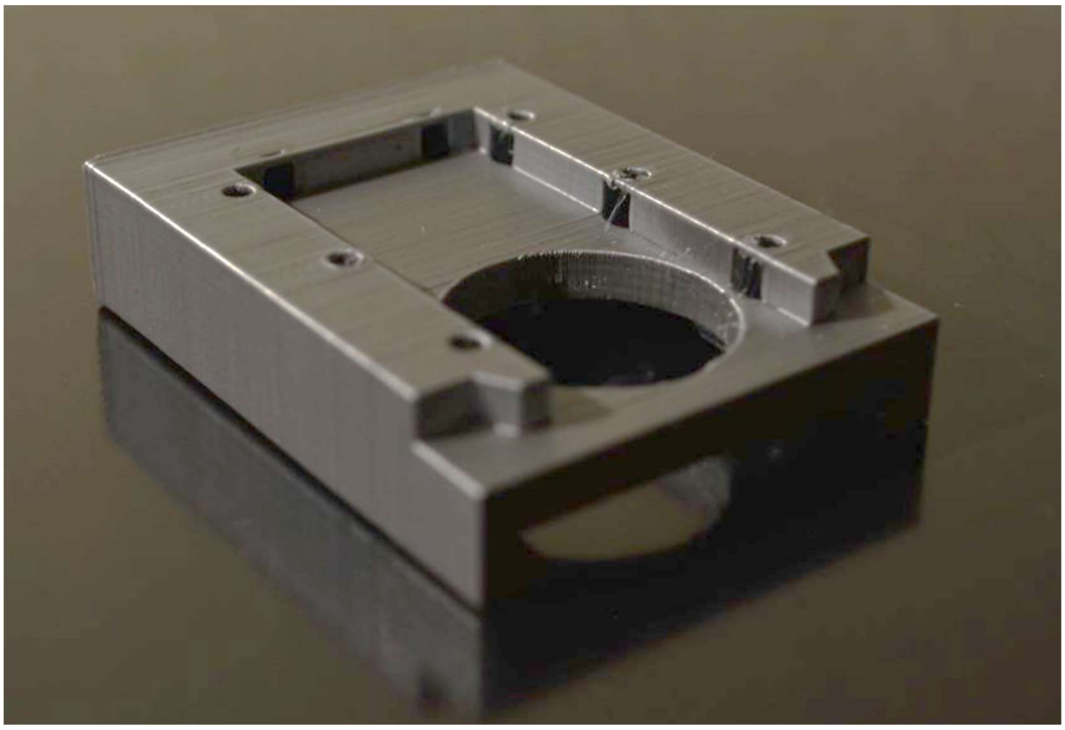
CAD PROCESS

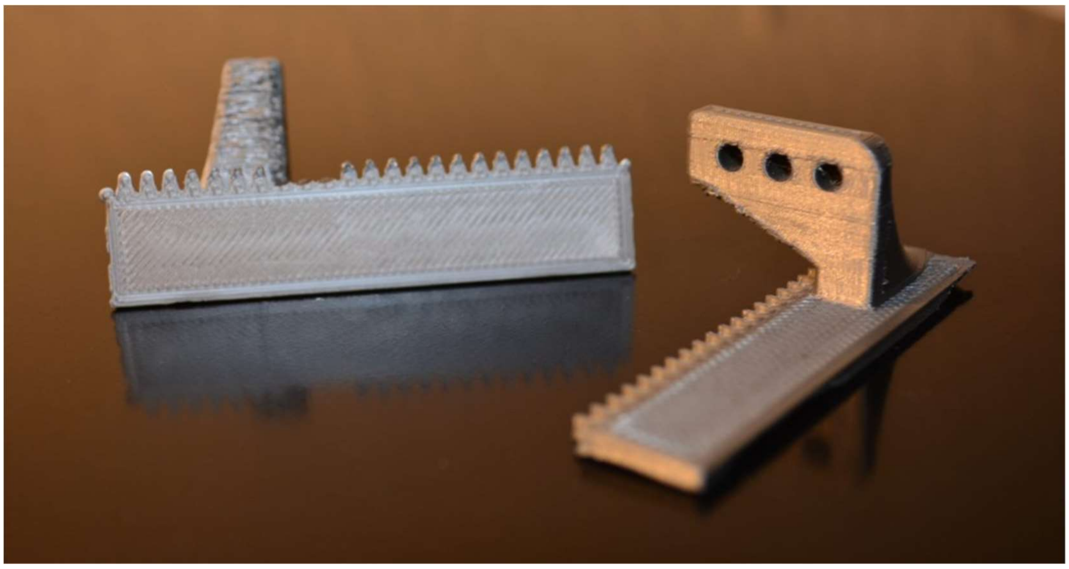
The design was modelled using a combination of Autodesk Inventor Professional 2018 and Autodesk Fusion360. The components designed in Inventor employed the parametric capabilities of the package as far as possible to streamline the process. Inventor also contains a library of standard parts (e.g. fasteners) which are adherent to relevant standards. This allows the designer to check *clearances for such parts in the assembly environment as the components of the design are pieced together.*

Gripper End-Effector Design

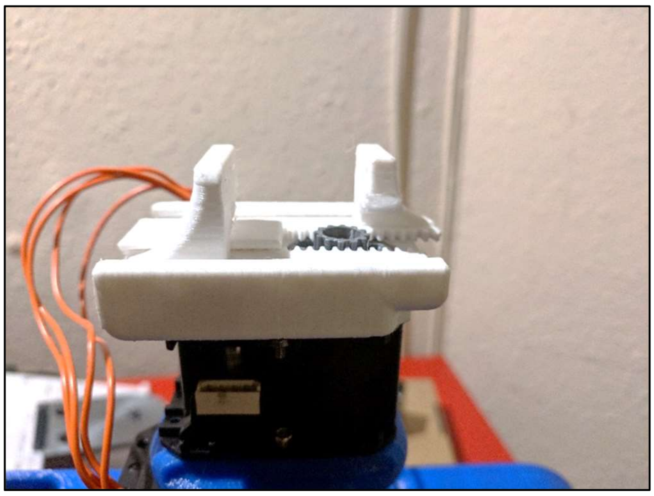
The gripper was inspired by the rack and pinion design used in some robot manipulators. The design produces a strong, compact and simple end-effector. The design needed to be small for space operations. This allows the arm to be easily stowed in a compact configuration. Furthermore, space activities often require the robot to work within proximity of its own structure. The grabber assembly was made entirely from 3D Printed parts.

We produced a 'version 1' design in PLA for testing. The design revealed that the PLA rack teeth would occasionally break when under high load. Furthermore, the design featured pointed corners that made the robot less user friendly.





Version 2 combined three improvements. The pointed corners of where rounded to improve user interaction. We also changed the print material to PETG. This lead to an increase of strength preventing gear teeth from breaking. The final adjustment involved the reduction of radial forces between gears so that a clutch like behaviour is exhibited when the gripper is jammed.



First complete design

The lead designer began the process from the base of the planar, RRR section of the manipulator; the most critical section of the arm in terms of torque limitations of the servos. The starting point was a freely-available model of the Dynamixel RX-28 servo sourced from GRABCAD.

From the profile of the servo mounting holes, a turret-base component was designed which would form the base of the planar section of the arm. The turret mounted to the servo horn of the first servo and provided a mounting rail which aligns the rotational axis of the first and second servos. The first iteration of the design had ribbed rails for greater stiffness.

The first link of the arm was designed with the self-intersection capabilities in mind. Made up of a split profile, where the two halves are secured by the servo horn bolts and a pair of M6 bolts with captive nuts. As the length of this first link was likely to have to be changed post-manufacture, the length of the link was split as close as reasonable to the axis of the base servo and a dovetail joint included to allow the length of the first link to be changed accordingly.

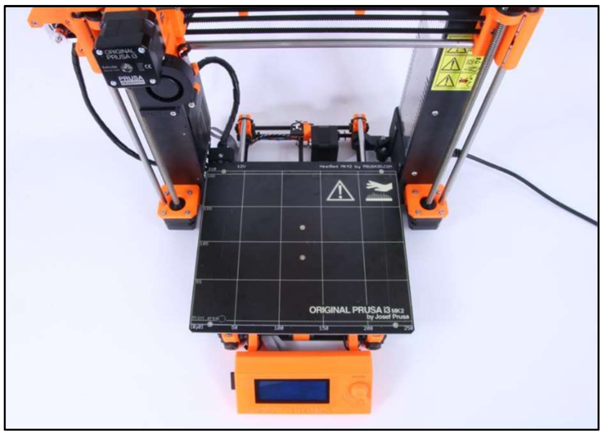
Whilst exploring the potential for full self-intersection of the links nearest the end-effector through the first link, I set the dimension between the two servo axes on the first link to a sufficient distance to allow the second link to rotate continuously.

The second link has its two joint servo bodies mounted to the link, forming much of the link’s structure. Similar to the first link, the second link has a split profile, with 2 identical shells connecting the servos.

First iteration of 3D-printed parts

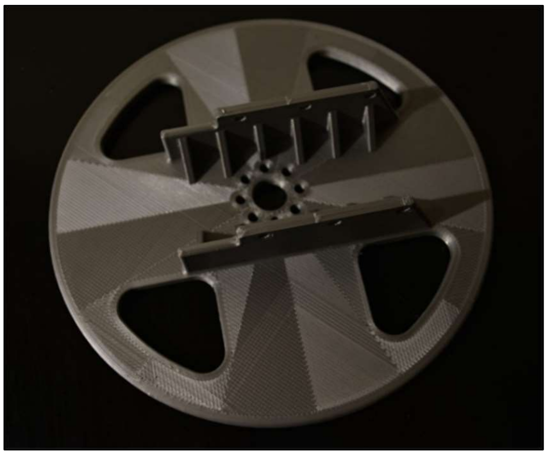
Thanks to the rapid prototyping method of 3D-printing, the components that had now been designed could be produced immediately once completed. This allowed for easy test fitting of the components to the servos to ensure the tolerancing approaches used to compensate for the 3D printer’s limitations were suitable.

The printer used for all the components was a Prusa Research i3 MK2. The first parts were printed in silver PLA with a 20% infill. Using a 0.15mm layer thickness, the print time was possibly longer than necessary, but returned very accurate parts with a good finish.



The first component produced (the turret base) failed during printing due to an issue with bed adhesion. After a brief redesign to increase the surface area of the part contacting the bed, the turret was printed again and could be successfully fitted to the servo horn and mounting holes of the Dynamixels. Sadly, this print had also suffered from bed adhesion issues and had warped slightly as the material cooled. A third attempt at printing the part yielded a successful result which would be suitable for the final assembly.

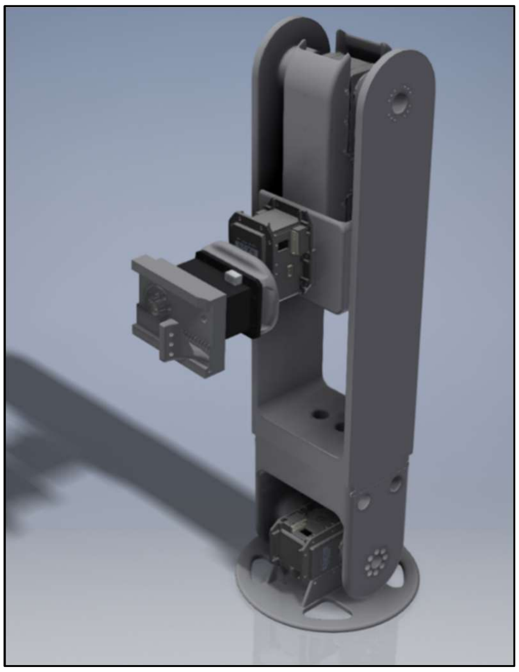
Next, the two components of the lower half of the first link were also printed. An issue discovered with these components was that the holes four mounting to the servo horn were far too deep for any standard length M2 machine screw. They did allow test fitting of the captive nyloc nuts with the selected tolerance of 0.3mm separation around the nut which was successful.



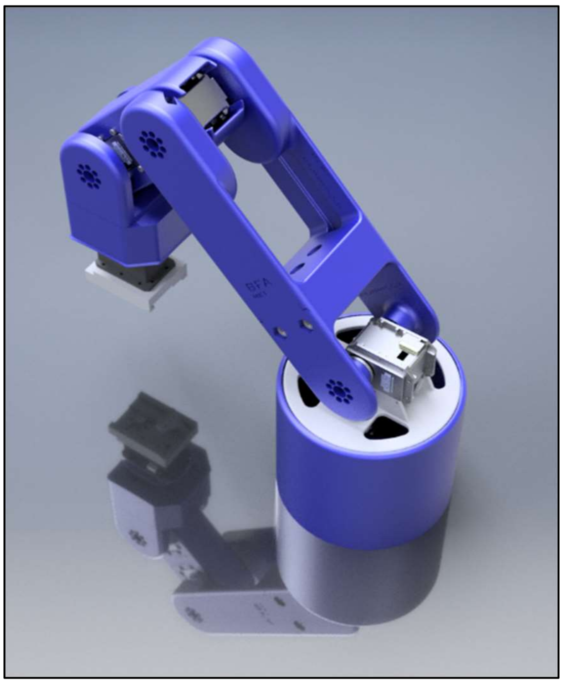
Design Improvements

Following the first round of printing, some changes to the overall design were made to rectify some issues regarding hole depths and to improve the aesthetics of the manipulator.

1.The Self-Intersecting arm design



2. The improved design



A cable management solution was incorporated into the structure of the first link components to carry the Dynamixel cables between the second and third servos. Whilst these parts were being edited, the decision was made to abandon the full self-intersection for a compromise folding design, where the second link can intersect the first, keeping the manipulator compact for easy transportation.

At this stage, a base for the turret component was also added to the design to retain the servo providing the rotation axis for the planar section of the arm. This component was designed with the intention of having further modular sections below it for containing the manipulators peripherals, as mentioned in the initial ideas section.

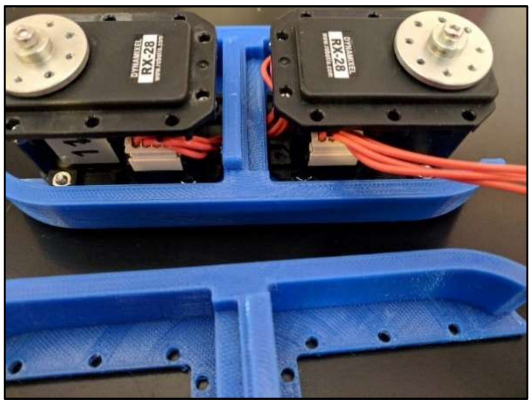
ASSEMBLY AND TESTING

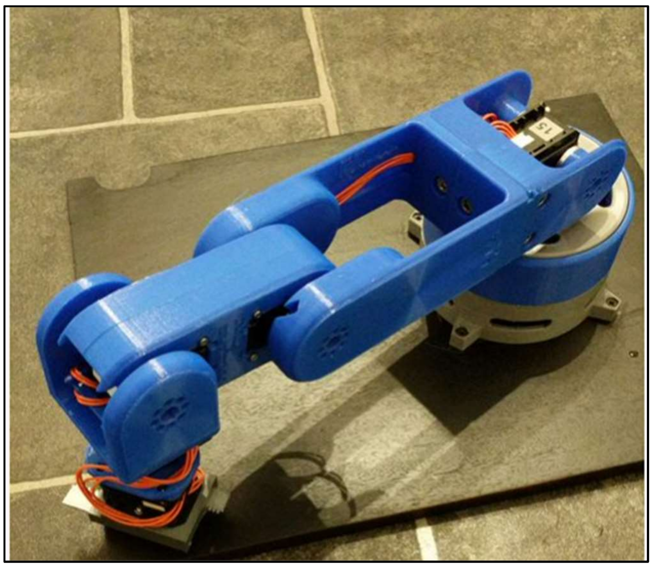
First assembly and cable routing

Initially, the manipulator was assembled from the base turret to the end effector, as the base had yet to be printed for installation. Also, we did not yet have access to the raspberry Pi board that would be used to run the servo controller nodes, so the design could not be fully assembled anyway.

The assembly was hampered initially by the lack of suitable fasteners and cables of sufficient length to connect the Dynamixels together through each link of the arm.

As the RX-28 servo uses a standard Molex SPOX-series connector for the power and RS-485 interconnection, we opted to purchase some of the terminals and enclosures to make up our own cables of a suitable length for the application. These were made during the assembly process, as some of the connections required parts of the arm to be disassembled to gain access to the Dynamixel connectors. Lengths of 22awg wire were inserted into the cable management pathways through the manipulator and then positioned at the connector. The range of motion was then tested by manually moving the joints which affected the slack in the cable to ensure the cable would not be placed under excess strain during operation. The cables were then made up on the bench and continuity-tested with a DMM before being installed.





Once a sufficient supply of fasteners had been acquired, the manipulator could be made structurally sound to allow for some motion testing using ROS.

The fasteners selected were ISO-4762 adherent socket-head cap screws of sizes M2.5 and M2 to interface with the threads on the servo horns and the captive nuts on the servo casings.

During the assembly stage, one component that was discarded was the cowling that covered the end-effector rotation servo. Issues with interference between the component and the servo casing meant that it would have to be modified to fit correctly. In this instance it was decided that the function of the component (as a cable management device) could be replicated using a cable tie.

Testing using ROS

Initial tests of the servos using ROS were successful for all links apart from the first segment of the arm. It was discovered that the single servo had insufficient torque to hold the arm in an upright position without overloading.

The mass of the robot arm beyond the first joint is 726g. The arm length is 440mm (44cm). Therefore, the maximum torque applied to the motor assuming an even distribution of weight is

$$\tau = 22cm \times 0.726Kg = 15.972\ Kgcm$$

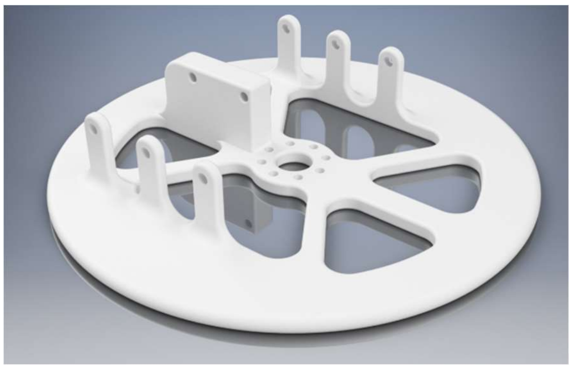
The Dynamixel stall torque is 37.73 Kg-cm

However, for stable operation the datasheet recommends maximum torque of 7.55 Kg-cm.

Therefore, to achieve this we doubled up the Dynamixels in the base segment to produce a final maximum torque of 7.986 Kg-cm. This operation is just outside the useable limits. Therefore, we will restrict the range of motion so that the robot arm does not remain fully extended for extended periods of time. It is reasonable to consider that in low-gravity environments the torque experienced by the motors will be significantly reduced.

The Dynamixel controller package for ROS facilitates the use of a pair of servos for a single joint (one as a master, one as a mirrored slave) we decided to redesign the base turret and bottom section of the first link to use a pair of motors, thereby fulfilling the torque requirements.

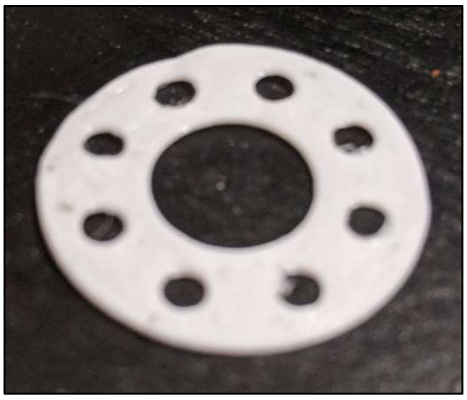
Modifying the base turret



To incorporate the two servos into the base turret, a different approach for the mounting system was required as access to the outside faces of the servos would no longer be possible. The first redesign used a set of legs to secure the drive-side of the servos to the turret using the standard captive nut method. The non-drive side was then secured by threading one of the M2.5 screws directly into the printed part through the mounting hole, accessed from the drive-side.

The lower half of the first segment of the arm also had to be widened to accommodate the width of two servos side-by-side.

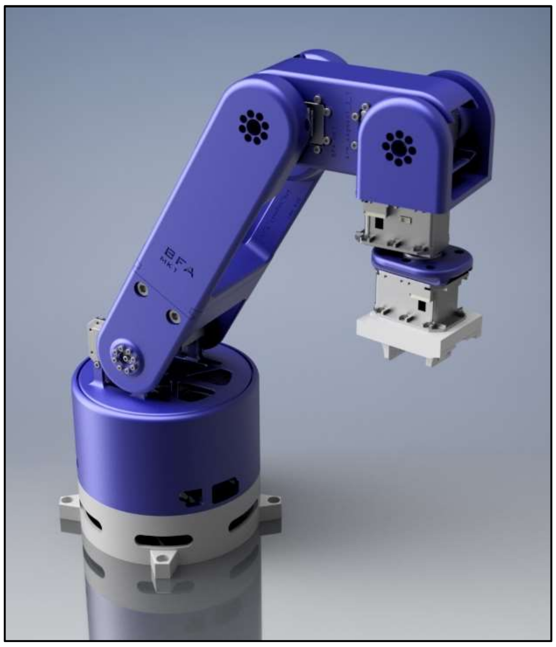
Following assembly of the first turret component, it was discovered that there was an interference issue between the first segment ends and the heads of the servo mounting bolts. A further redesign of the turret fixed this issue.



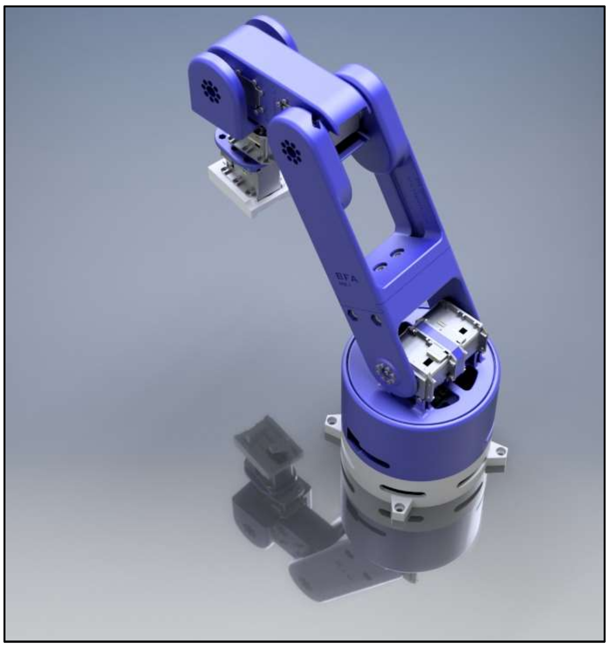
This was not the end of the problems with the dual servo mount, however. Whilst testing the servo controller for the dual motor, it was discovered that the turret could flex as the arm segment was moved around, allowing the two servos to twist their axes out of alignment and overload. This issue was fixed by adding a spacer that tied the two servos together, hence using the servo bodies to stiffen the assembly.

Final Design – Post testing

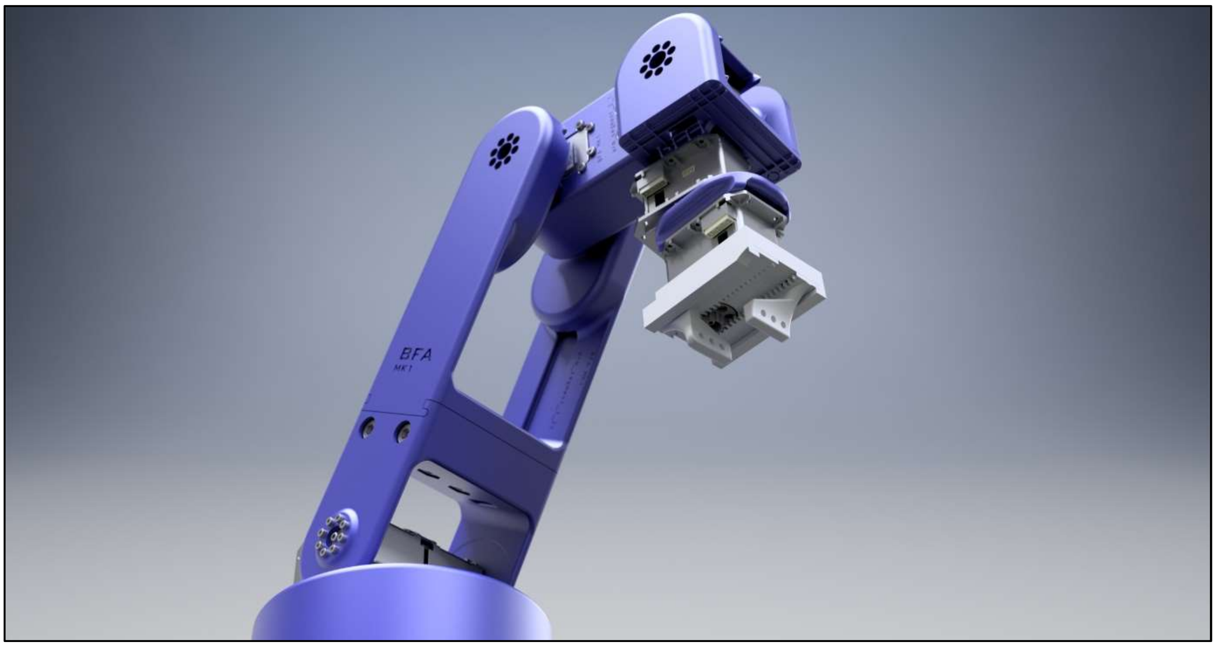
Having fixed the issues with the single-servo torque for the first segment of the arm, the design was now in its final state. Hence, a new assembly file was created in inventor to reflect all the changes made. Some rendered views are shown here:



4 Final Design Render



3 Final Design Render 2

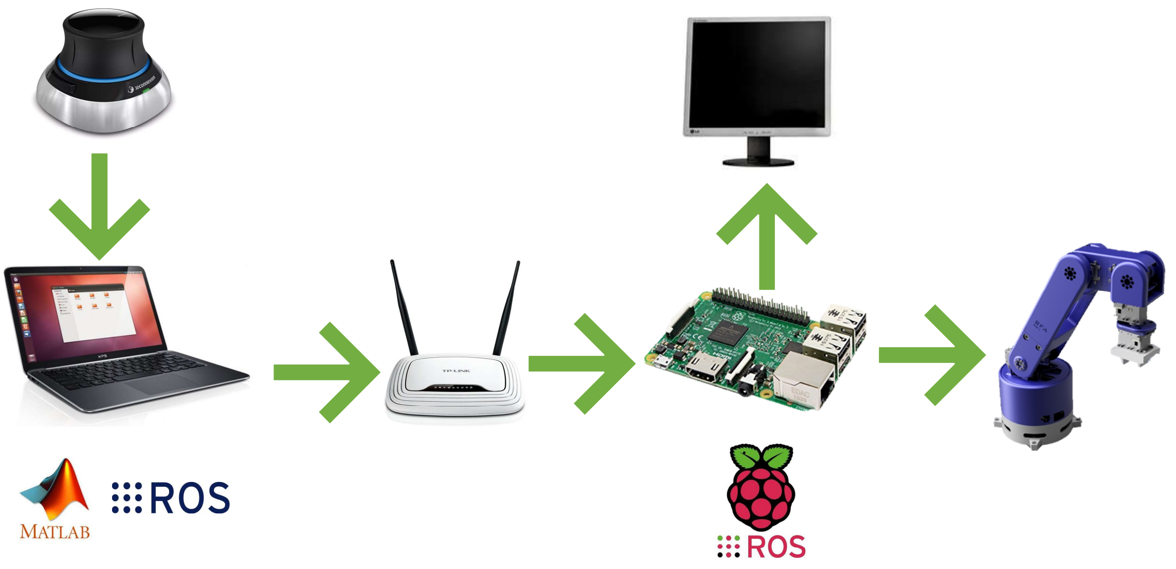


5 Final Design Render 3

IMPLEMENTATION

SOFTWARE IMPLEMENTATION

The software implementation for a space manipulator required the use of Matlab and ROS remotely to control a manipulator. The planned system design is outlined in the diagram below:

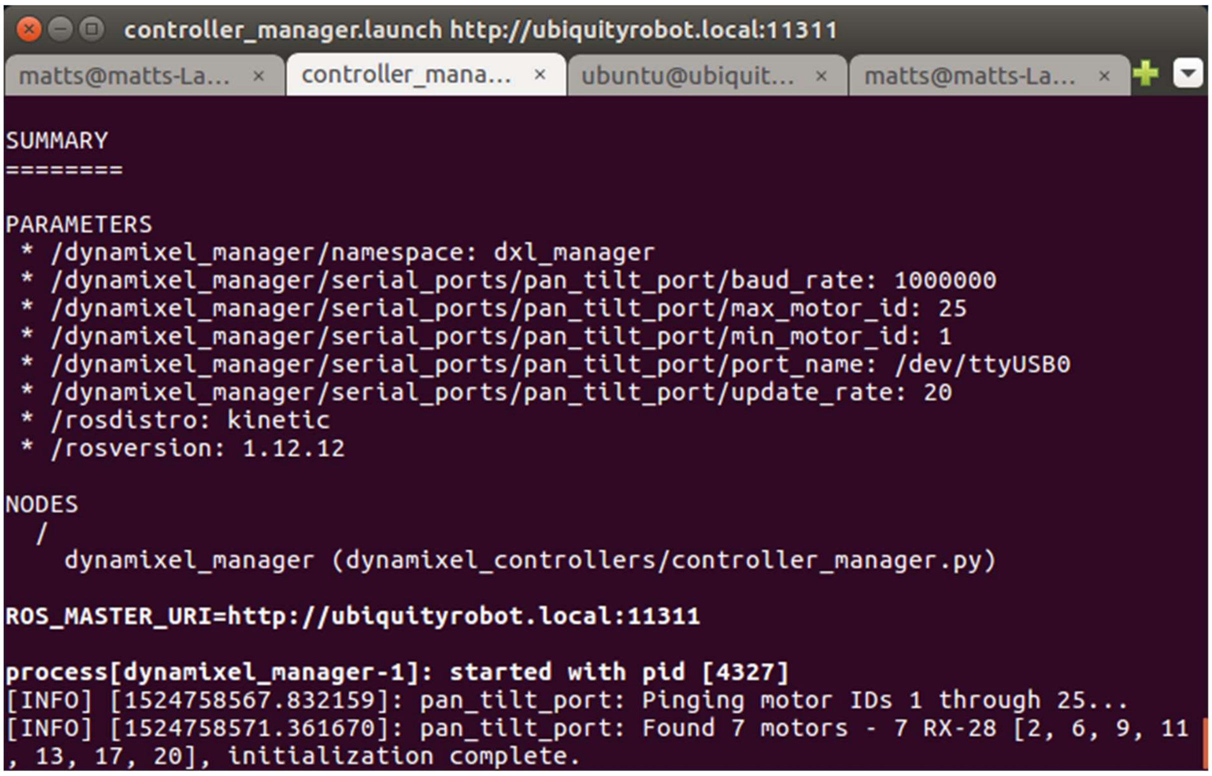


6 Overall System Map

The intended user input was a 3DConnexions Space Navigator 6DOF mouse, normally used for 3D design. We discovered there was existing MATLAB support for this device, but were not able to implement the inverse kinematics that were required for determining joint angles from input workspace coordinates. Then MATLAB runs forward kinematic calculations to determine the final joint angles. The Joint angles are then published to the ROS network remotely. The remote connection is achieved by using a router to run a DHCP server. Then using SSH the laptop can connect to the Raspberry Pi. The Pi is setup as the ROS master device in the network.

For our actuator implementation we followed the Dynamixel Controller tutorials on the ROS wiki, creating suitable configuration (.yaml) files that matched the motor IDs and joint constraints that we were using in our manipulator design.

2 key components make up the actuator control architecture present on the raspberry Pi: the controller manager and the meta controller. The meta controller creates topics for motor commands and publishes diagnostic data that can be used for debugging. The controller manager interfaces with the USB FTDI controller for the servos and directs the ROS topic information from the meta controller to the destination motor and vice versa.



```
SUMMARY
========

PARAMETERS
 * /dynamixel_manager/namespace: dxl_manager
 * /dynamixel_manager/serial_ports/pan_tilt_port/baud_rate: 1000000
 * /dynamixel_manager/serial_ports/pan_tilt_port/max_motor_id: 25
 * /dynamixel_manager/serial_ports/pan_tilt_port/min_motor_id: 1
 * /dynamixel_manager/serial_ports/pan_tilt_port/port_name: /dev/ttyUSB0
 * /dynamixel_manager/serial_ports/pan_tilt_port/update_rate: 20
 * /rosdistro: kinetic
 * /rosversion: 1.12.12

NODES
  /
    dynamixel_manager (dynamixel_controllers/controller_manager.py)

ROS_MASTER_URI=http://ubiquityrobot.local:11311

process[dynamixel_manager-1]: started with pid [4327]
[INFO] [1524758567.832159]: pan_tilt_port: Pinging motor IDs 1 through 25...
[INFO] [1524758571.361670]: pan_tilt_port: Found 7 motors - 7 RX-28 [2, 6, 9, 11
, 13, 17, 20], initialization complete.
```

7 Controller Manager process detecting Dynamixels

The Raspberry Pi receives the joint angle commands over the wireless network and publishes them to the motor meta-controller topics. We used Mixcell to configure the motor baud rates and determine their respective IDs. Using this information, we produced a launch file to setup the motors alongside a launch file to setup the Dynamixel controller manager. The controller manager is shown below:

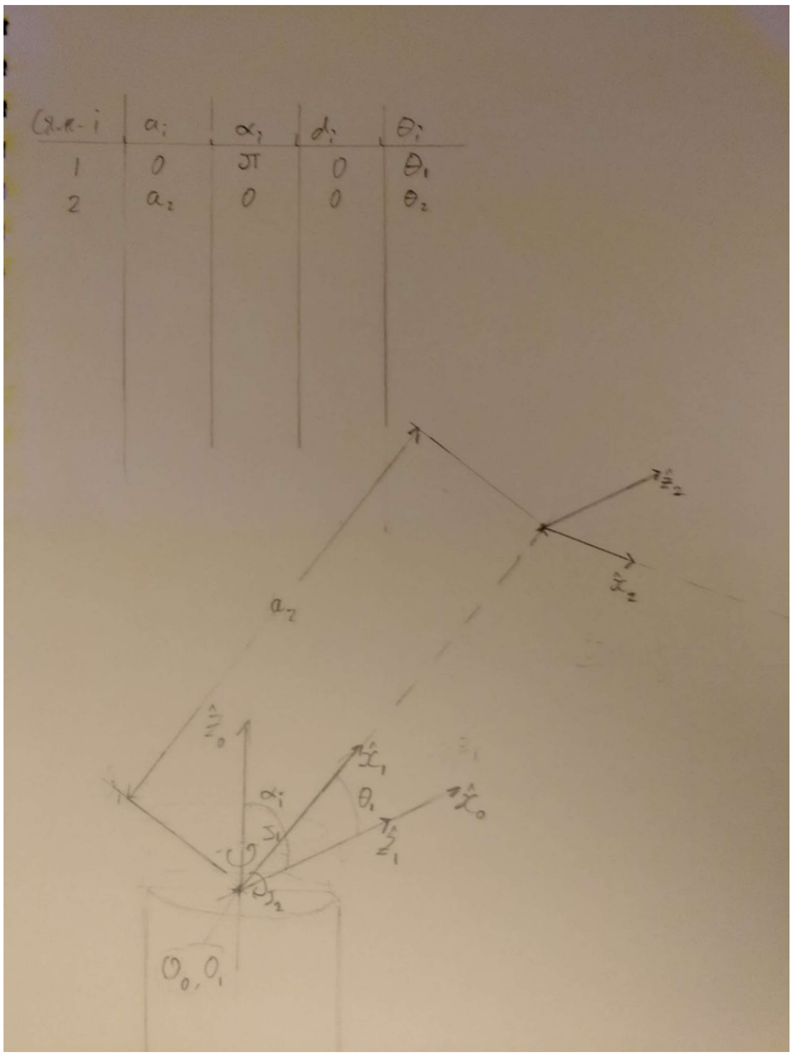
```
<!-- -*- mode: XML -*- -->

<launch>
    <node name="dynamixel_manager" pkg="dynamixel_controllers"
type="controller_manager.py" required="true" output="screen">
        <rosparam>
            namespace: dxl_manager
            serial_ports:
                pan_tilt_port:
                    port_name: "/dev/ttyUSB0"
                    baud_rate: 1000000
                    min_motor_id: 1
                    max_motor_id: 25
                    update_rate: 20
        </rosparam>
    </node>
</launch>
```

The ‘tilt.yaml’ file is used to configure the motors and their respective topics. The first three controllers are shown below. We setup j2 as a dual motor controller because two motors are used to support the first arm segment.

```
j1_controller:
  controller:
    package: dynamixel_controllers
    module: joint_position_controller
    type: JointPositionController
  joint_name: turret
  joint_speed: 1.0
  motor:
    id: 6
    init: 0
    min: 0
    max: 4095

j2_controller:
  controller:
    package: dynamixel_controllers
    module: joint_position_controller_dual_motor
    type: JointPositionControllerDual
  joint_name: seg_1
  joint_speed: 1.0
  motor_master:
    id: 2
    init: 0
    min: -1023
    max: 1023
  motor_slave:
    id: 13

j3_controller:
  controller:
    package: dynamixel_controllers
    module: joint_position_controller
    type: JointPositionController
  joint_name: seg_2
  joint_speed: 1.0
  motor:
    id: 9
    init: 0
    min: 0
    max: 4095
```

FORWARD KINEMATICS

Determining DH Parameters



[illegible]	a_i	α_i	d_i	θ_i
1	0	π	0	θ_1
2	a_2	0	0	θ_2

A hand-drawn diagram was the most effective solution to determining the Denavit-Hartenberg configuration of the manipulator:

For the table of parameters, standard DH convention was used. It is also worth noting that these kinematics do not account for end-effector swivel or grabber joints as the DH forward kinematics only give a position in the workspace coordinate frame. They do not account for the rotation of the end effector or the position of the grabber.

DH Parameter Table

Link	a_i - link length	α_i - link twist	d_i – link offset	θ_i – angle variable
1	0	$\pi/2$	0	θ_1
2	a_2(200mm)	0	0	θ_2
3	a_3(90.2mm)	0	0	θ_3
4	a_4(143.750)	0	0	θ_4

Implementing DH in MATLAB

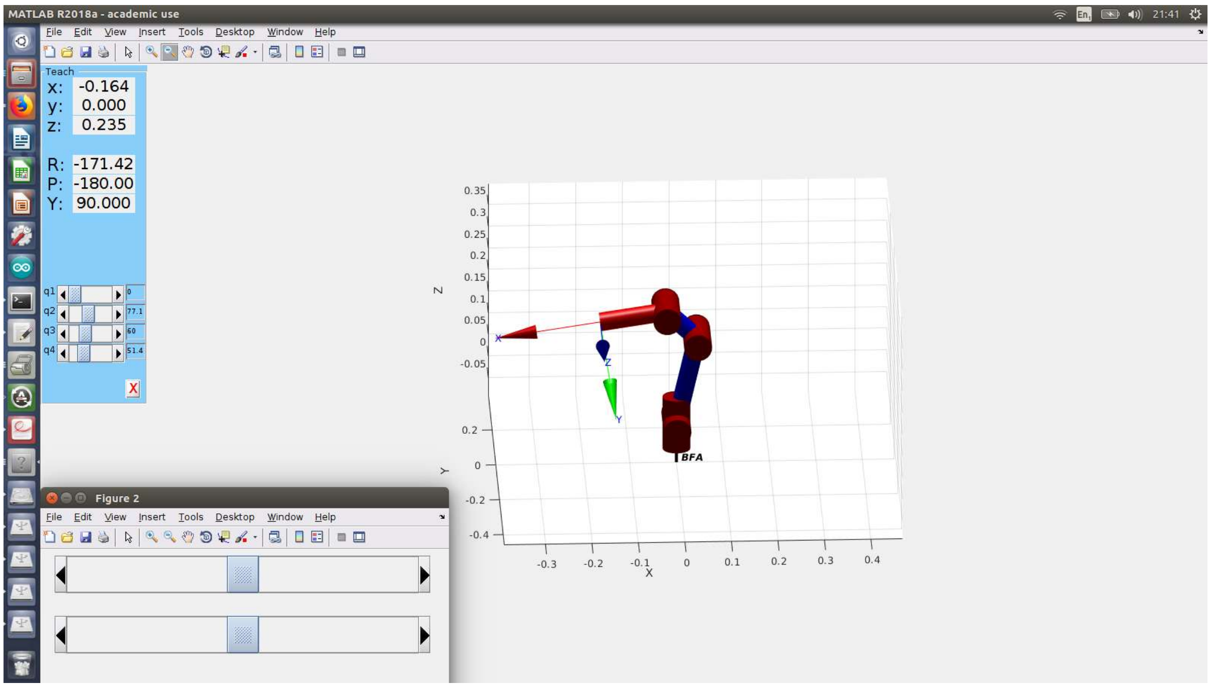
A basic simulation of the forward kinematics of the arm was implemented in MATLAB using Peter Corke's Robotics Toolbox (v10.2). Similar to one of the examples provided, The individual links are specified using values from the DH Parameter Table as 'Link' objects in a 1x4 matrix 'L'. This matrix is then used to generate a 'SerialLink' object that represents the manipulator. The teach pendant function is then used to move the joints in real time and view the coordinate values of the End-Effector with reference to the base-frame origin.

```
%*********************************************
%ROCO224 - BFA Forward kinematics DH test
%*********************************************
clear all
close all
clc
format short
%*********************************************
%Define DH params

%Link 1 - turret
a_1 = 0;
alpha_1 = pi/2;
d_1 = 0;
theta_1 = 0;
dh_1 = [theta_1 d_1 a_1 alpha_1 0 0];
%Link 2 - segment_1
a_2 = 0.2;
alpha_2 = 0;
d_2 = 0;
theta_2 = 0;
dh_2 = [theta_2 d_2 a_2 alpha_2 0 0];
%Link 3 - segment_2
a_3 = 0.0902;
alpha_3 = 0;
d_3 = 0;
theta_3 = 0;
dh_3 = [theta_3 d_3 a_3 alpha_3 0 0];
%Link 4 - segment_3
a_4 = 0.14375;
alpha_4 = 0;
d_4 = 0;
theta_4 = 0;
dh_4 = [theta_4 d_4 a_4 alpha_4 0 0];

%Create Link objects
L(1) = Link(dh_1);
L(2) = Link(dh_2);
L(3) = Link(dh_3);
L(4) = Link(dh_4);

BFA = SerialLink(L, 'name', 'BFA')
%Plot Robot animation
ws = [-1 1 -1 1 -0.14475 1];

figure;
BFA.plot([0,0,0,0],  'workspace', ws);
%Set joint limits
BFA.qlim = [[0 2*pi]; [-pi pi]; [0 2*pi]; [-pi pi]];
disp('Running Teach Pendant...')
BFA.teach
```

MATLAB Controller Implementation

Using the established SSH link to the raspberry Pi, we began to write a MATLAB script that used the variables generated by Peter Corke's teach pendant function to publish joint angles to the ROS meta-controller topics as the joint sliders are adjusted. This not only gives a visual representation of the arm's pose on-screen, but also returns the coordinate results of the forward kinematics; all in real time. We also included the realistic 1.3 second time delay into every transmission to the MATLAB ROS node, allowing us to simulate the transmission delay present in lunar-orbit teleoperation.

Additional sliders were also added to allow control of the gripper and end-effector joints.


MATLAB R2018a - academic use
21:41
File Edit View Insert Tools Desktop Window Help
Teach
Figure 2
File Edit View Insert Tools Desktop Window Help
BFA
X
Y
Z
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
-0.05
0.2
0
-0.2
-0.4
-0.3
-0.2
-0.1
0
0.1
0.2
0.3
0.4

x: -0.164
y: 0.000
z: 0.235

R: -171.42
P: -180.00
Y: 90.000

q1 0
q2 77.1
q3 60
q4 51.4

8 The control environment in MATLAB

The MATLAB script for the implementation can be found in Appendix/Final Matlab Code:1.1.

Experiments

Motion Experiments

Initially, we operated the arm using terminal commands directly input to the raspberry Pi. Joint angles were published individually to the relevant topics to determine joint limits and ensure that the arm operated as expected. This also allowed us to zero the servo horns to a desired link angle so that the arm would move to a suitable home position on start-up.

It was during these initial test stages that we discovered many of the issues present in our arm design, such as those with the dual-motor turret.

Teleoperation Experiments

Once the ROS network had been implemented using the DHCP server, we were able to repeat the process of publishing directly to the ROS topic for each joint, but this time wirelessly over SSH from a separate laptop running Ubuntu. From there we built the MATLAB code that allowed us to then move the joints using sliders from a remote location (provided within range of the Wireless network). During these experiments, we used a GoPro Hero 3+ camera connected wirelessly to an iPad to provide our live feed. Conveniently, the video feed delay was similar to the designed-in transmission delay required for accurate simulation of orbital transmission.

FINAL OPERATION DEMO

This short video demonstrates the arm in operation, being controlled remotely and manipulating the test cubes mentioned in the introduction:

https://youtu.be/LQe14lj3zQo

CONCLUSION

In conclusion we successfully built a teleoperated arm with Forward Kinematics that was controllable from a remote location using our own, wireless, ROS network. We successfully met our initial objectives and have overcome multiple design challenges throughout the process. If an updated version where to be made we would consider developing the software to use Inverse Kinematics. Inverse Kinematics would allow us to use a more complex control/user input device such as the 6DOF mouse we intended to use in the software implementation. The robot arm has demonstrated its ability to move and place objects over 25cm with remote control and limited viewing angles. Therefore, we can consider this project a success.

1.1 APPENDIX/FINAL MATLAB CODE:

```
%*********************************************
%ROCO224 - BFA Forward kinematics DH test
%*********************************************
clear all
close all
clc
format short
%*********************************************
rosshutdown;
rosinit('192.168.0.100');
pubJ1 = rospublisher('/j1_controller/command','std_msgs/Float64');
pubJ2 = rospublisher('/j2_controller/command','std_msgs/Float64');
pubJ3 = rospublisher('/j3_controller/command','std_msgs/Float64');
pubJ4 = rospublisher('/j4_controller/command','std_msgs/Float64');

msgJ1 = rosmessage('std_msgs/Float64');
msgJ2 = rosmessage('std_msgs/Float64');
msgJ3 = rosmessage('std_msgs/Float64');
msgJ4 = rosmessage('std_msgs/Float64');
```

```
%Define DH params

%Link 1 - turret
a_1 = 0;
alpha_1 = pi/2;
d_1 = 0;
theta_1 = 0;
dh_1 = [theta_1 d_1 a_1 alpha_1 0 0];
%Link 2 - segment_1
a_2 = 0.2;
alpha_2 = 0;
d_2 = 0;
theta_2 = 0;
dh_2 = [theta_2 d_2 a_2 alpha_2 0 0];
%Link 3 - segment_2
a_3 = 0.0902;
alpha_3 = 0;
d_3 = 0;
theta_3 = 0;
dh_3 = [theta_3 d_3 a_3 alpha_3 0 0];
%Link 4 - segment_3
a_4 = 0.14375;
alpha_4 = 0;
d_4 = 0;
theta_4 = 0;
dh_4 = [theta_4 d_4 a_4 alpha_4 0 0];

%Create Link objects
L(1) = Link(dh_1);
L(2) = Link(dh_2);
L(3) = Link(dh_3);
L(4) = Link(dh_4);

BFA = SerialLink(L, 'name', 'BFA')
%Plot Robot animation
ws = [-1 1 -1 1 -0.14475 1];

figure;
BFA.plot([0,0,0,0],  'workspace', ws);
%Set joint limits
BFA.qlim = [[0 2*pi]; [0 pi]; [0 pi]; [0 pi]];
disp('Running Teach Pendant...')
BFA.teach
    pubJ5 = rospublisher('/j5_controller/command','std_msgs/Float64');
    pubJ6 = rospublisher('/j6_controller/command','std_msgs/Float64');

    msgJ5 = rosmessage('std_msgs/Float64');
    msgJ6 = rosmessage('std_msgs/Float64');

    msgJ5.Data = 0;
    msgJ6.Data = 0;
    f = figure('Visible','Off');
    sldJ5 = 
uicontrol('Style','slider','Min',0,'Max',5,'Value',2.5,'Position',[20 
320 500 50],'String','J5');
    sldJ6 = 
uicontrol('Style','slider','Min',0,'Max',5,'Value',2.5,'Position',[20 
240 500 50],'String','J6');
    f.Visible = 'On';

while true
    q = BFA.getpos();

    msgJ1.Data = q(1);
    msgJ2.Data = q(2);
    msgJ3.Data = q(3) + pi;
    msgJ4.Data = q(4);
    msgJ5.Data = sldJ5.Value;
    msgJ6.Data = sldJ6.Value;

    send(pubJ1,msgJ1);
    send(pubJ2,msgJ2);
    send(pubJ3,msgJ3);
    send(pubJ4,msgJ4);
    send(pubJ5,msgJ5);
    send(pubJ6,msgJ6);
    pause(1.3);
end
```

PROGRAMMING INDUSTRIAL ROBOTS

INTRODUCTION

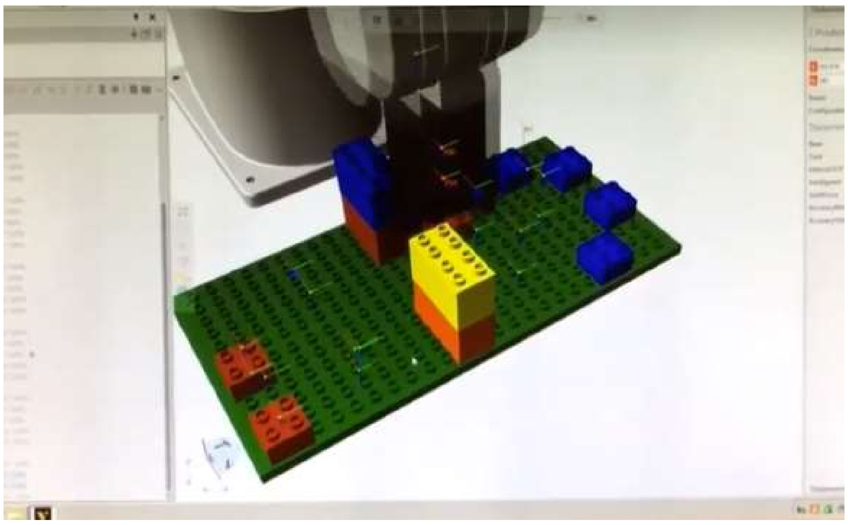
Description of robot

The Mitsubishi "RV-2AJ" has 5 degrees of freedom with 5 joints. The RV-2AJ is a large and rigid robot arm with powerful motors. This results in a very predictable and stable end effector position. It is widely understood that Industrial grade robots are dangerous because they are heavy objects moving at speed.

Description of task

Our task was to create a simulation in 3DAutomate. The task consisted of moving 8 LEGO Duplo blocks from one side of the base platform to the other while moving through a 2-block high gate. The pattern was randomly chosen from the ones provided to us in the lab manual.

Description of software used and comparison with other robot simulation software



3D-automate is a 3D simulation toolkit by Visual Components that's able to create realistic simulations for a given robot models. It's also used for off line programming to analyse and optimize specific tasks before its put to test in real life. The biggest advantage of this software is that it doesn't require any programming skill to use it. To create a simulation the user can just drag and drop the robot to the right place, teach its position to the software than create new routes to move the robot on by repeating the same action. The robot will move between the 2 points either in a curve or a straight line, depending on the user's preference/choice.

RELEVANCE

Brief comparison of simulated and real robot

The simulated and the real robot mostly acted in the same way. Both had the same movement limits as the simulated one had the exact same measurement as the real one. Even though they are capable of the same things the simulated one have a huge advantage over the real, physical arm. It cannot break. Rather than overloading the motors, breaking the physical components, the simulation will just show you an error that you can easily correct without any consequences.

Development

Describe how you worked as a team to solve the task

The first thing we did was setting up the start position of the bricks and the board. After that we played around with the software to test out the features and made sure everyone understands how to create points to move the robots inbetween. After we were all comfortable using the software, each of us did2 block placements in the code, involving picking up, moving it to the right place, then putting it down. While one of us were creating the points and adding it to the code the others supervised and payed extra attention to possible bugs, mistakes that the coder made and didn't notice.

Describe briefly problems encountered

The first time we uploaded the model the software was reacting very slowly, wasn't responsive at all. We thought the program couldn't handle the model's size so we deselected some of it and it did made a huge change.

The other problem we encountered is that as the LEGO board and the blocks were simulated the models could move around freely. In real life a block would stop moving after it made connection with the board or another block but in the simulation, it just went through it. Nothing was stopping it. This made it difficult to create the start position and the gate on the board. Also, when placing the blocks we could never be sure if they would be at the right place, or if the 4 hole on the bottom will actually fit in real life the way we put them on.

We also found it difficult to find the right placement for the LEGO board. It had to be in level with the base of the robot arm and it had to be close enough so it can reach it. The problem we had is that the board were too close to the base. After coding 3 blocks we realised that the robot arm is unable to reach the 4th block as the joints are cannot bend that far back.

Describe briefly lessons learned

We learnt how to use 3D automate to simulate movement of the industrial arm. Also we managed to overcome issues with excess rotation of the end effector while simulating due to not being able to reach the position correctly.

Results

We uploaded our code to the zip file with this report.

We successfully managed to get all 8 blocks from the initial position to the final stacked position, while passing through the gate to the other side. We completed this without any collisions.

Conclusion

To conclude we managed to manually control the industrial arm and programme it using 3D automate. We were successful in moving all 8 blocks without collisions through the gate into final positions.