Vision and Robotics Coursework

By s1948228 and s1834153

Workload Balance

We worked on the project simultaneously as we ran the project on one computer. Each question was undertaken together, with each person writing code when they had an idea or directing the other person to do so. We worked on the same computer so we did not believe we needed a commit history, however here is the link to our github regardless:

https://github.com/HTK55/ivr_assignment

Section 2.1

To calculate the joint angles, our algorithm first detects the center of each coloured circle, using camera 1 to get the z and y angles and camera 2 to get the z and x angles, then taking an average of the two z angles. There are times when a camera cannot see one or more of the circles (e.g. the circle is behind another circle), so we use the coordinates we calculated for this circle in the previous iteration. We then calculate the vectors between each circle, i.e. blue – yellow would be link2 and red – blue would be link3.

To calculate the angle of joint2, using just the x and z coordinates of link2, first we calculate the cross product of the unit vector y and link2, to get the correct orientation of the axis. Then we take this cross product vector and calculate the angle between it and the unit vector x, using the angle between two vectors equation (shown below, where vector 1 is x and vector 2 is y) to get angle α.

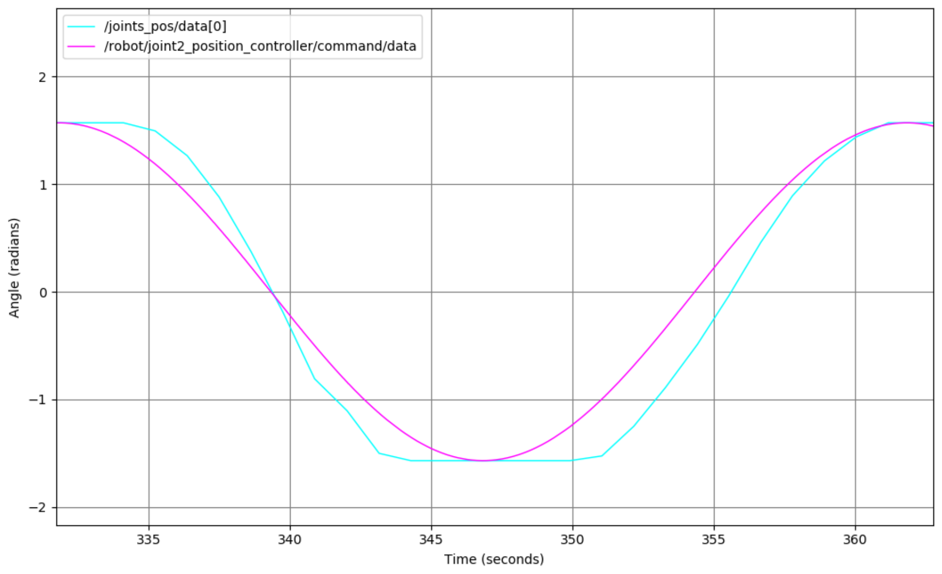
$\alpha = \cos^{-1} (x \bullet y \; / |x| * |y|)$

From this we can then calculate the angle of joint2. If the vector link2 is positive, joint2 = angle α – pi/2. If link2 is negative, joint2 = pi/2 - angle α. Then if joint2 > pi/2, or <-pi/2, just set it to pi/2 or –pi/2 respectively, as this means we have calculated the obtuse angle, so this is just pi/2 and –pi/2.

Then to calculate the angle of joint3 we just use the same angle between two equation, but between link2 and the y unit vector and then subtract this from pi/2 to get the correct angle.

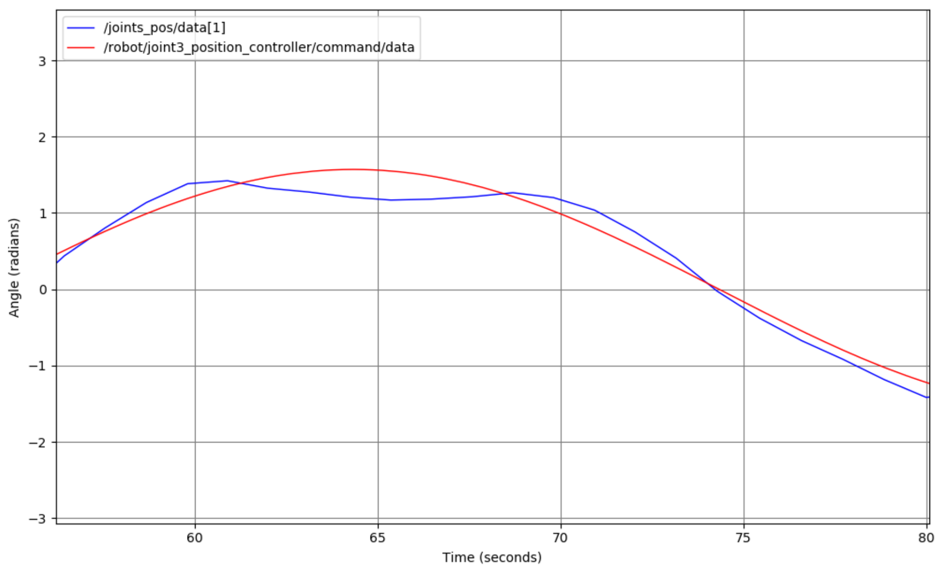
Then finally to calculate the angle of joint4 we use the same equation between link2 and link3, but only in the y and z planes. Then to handle the case again where we calculate the obtuse angle, when α > pi/2 we make joint4 = pi/2 - α, or when α < -pi we make joint4 = α + pi. We also encounter the error here, where the rotation of link3, makes the angle of joint4 falsely negative, to check when this occurs, we project link3 onto our previously calculated cross product for link2, then if this value is smaller than 0, we reverse the sign of joint4.

Estimation of joint2: ("/joint_pos/data[0]" is our calculated joint2 here though we changed it to "/joints_pos/joint_angle_2" in our final code)



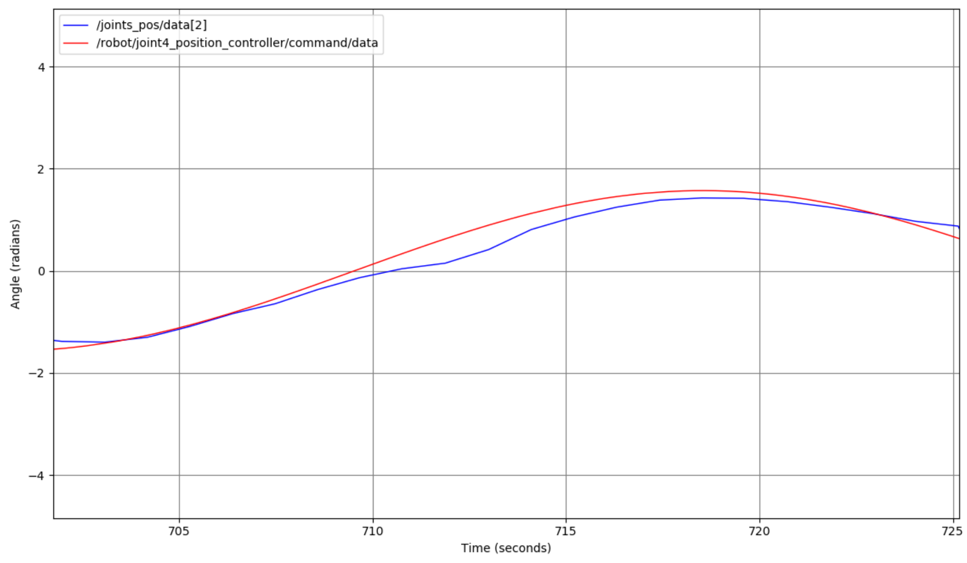
The estimation of joint 2 was consistent throughout however did have occurrences as you can see from the figure above there is 'an extension' of the local minima seen here. This is likely due to the accuracy of the detection process we implemented, as using the previously calculated coordinates is slightly inaccuarate. This process produced +/- 4 in all x,y,z values hence there is a consistent trend where the value cannot be accurately measured during this rotation.

Estimation of joint3: ("/joint_pos/data[1]" is our calculated joint3 here though we changed it to "/joints_pos/joint_angle_3" in our final code)



Joint 3 was also consistent as seen above throughout however would occasionally diverge from the graph sporadically. This occurred when the blue joint was either obscured by another joint or directly in front of the camera this is due to our approximate estimation of the blue or yellow joints estimation during this time.

Estimation of joint4: ("/joint_pos/data[2]" is our calculated joint4 here though we changed it to "/joints_pos/joint_angle_4" in our final code)



There was a larger degree of error occurring during this joint detection process. With simply a few large jumps, this occurred yet again whenever a joint was obscured, however more often the lack of fine detail within the detection process meant that occasional positive angles were calculated to be negative and vice versa. Overall It would produce half of the sin curve and the other half it would produce the negative in parts even if just for one second. This meant there was not often an occurrence of a full sin wave being displayed due to those 2 small details causing variation.

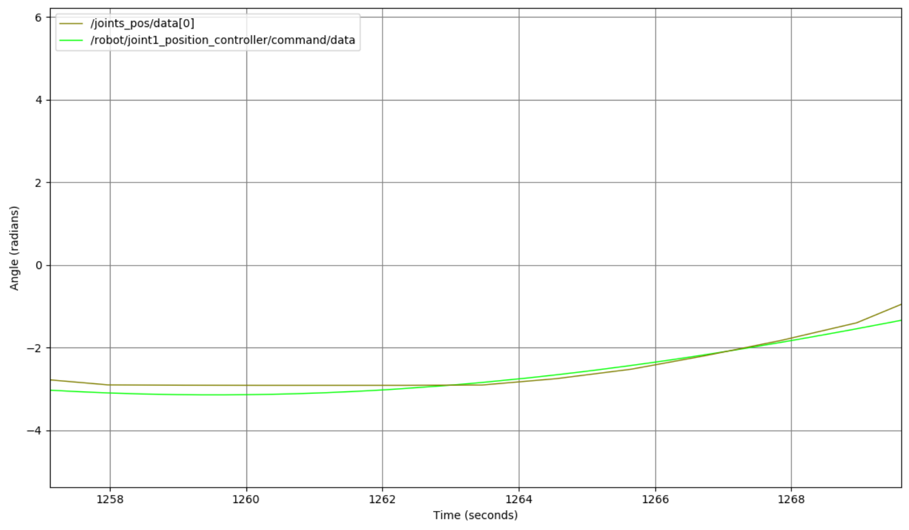
Section 2.2

In this algorithm we detect the coordinates of the coloured circles and obtain the links in the same way, adding an extra link now, link1 between yellow and green. Then to calculate the angle of joint1, we get just the x and y values of link2 and calculate the angle between this and the unit vector y (this could be either x or y here, but we chose y) using the same equation as before and subtract pi from it to get the correct angle of joint1.

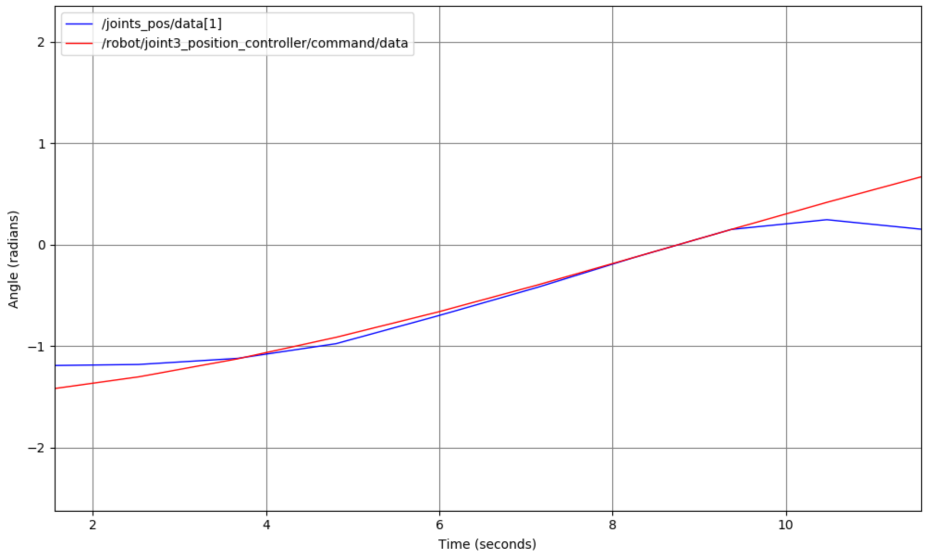
Then to calculate the angle of joint3, we employ the same method as in section 2.1. Because of consistent inaccuracy in calculation error here, we also had to add 0.15 to counteract the error.

Finally for joint4, we also used the same method as in 2.1.

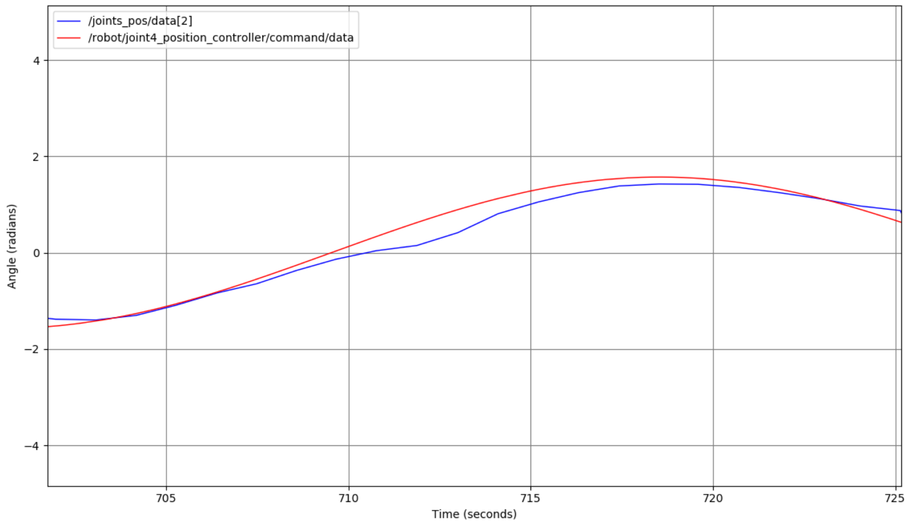
Estimation of joint1: ("/joint_pos/data[0]" is our calculated joint3 here though we changed it to "/joints_pos/joint_angle_1" in our final code)



Estimation of joint3:
("/joint_pos/data[1]" is our calculated joint3 here though we changed it to "/joints_pos/joint_angle_3" in our final code)

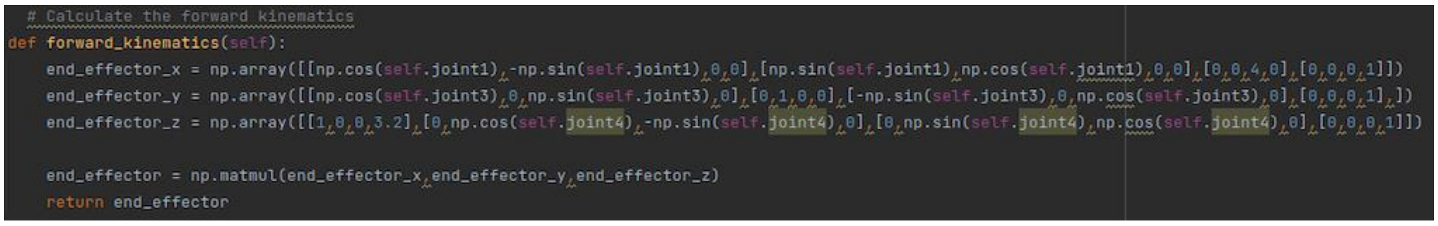


Estimation of joint4:
("/joint_pos/data[2]" is our calculated joint3 here though we changed it to "/joints_pos/joint_angle_4" in our final code)



Section 3.1

Here is a screenshot of our code calculating the matrix for forward kinematics, however we did not get any further in testing it.



```
  # Calculate the forward kinematics
def forward_kinematics(self):
    end_effector_x = np.array([[np.cos(self.joint1),-np.sin(self.joint1),0,0],[np.sin(self.joint1),np.cos(self.joint1),0,0],[0,0,4,0],[0,0,0,1]])
    end_effector_y = np.array([[np.cos(self.joint3),0,np.sin(self.joint3),0],[0,1,0,0],[-np.sin(self.joint3),0,np.cos(self.joint3),0],[0,0,0,1],])
    end_effector_z = np.array([[1,0,0,3.2],[0,np.cos(self.joint4),-np.sin(self.joint4),0],[0,np.sin(self.joint4),np.cos(self.joint4),0],[0,0,0,1]])

    end_effector = np.matmul(end_effector_x,end_effector_y,end_effector_z)
    return end_effector
```