**Development**

2.1 Code: Si Han, Report: Both 2.2 Code: Si Han, Report: Saivydas

3.1 Code: Both, Report: Saivydas 3.2 Code: Both, Report: Saivydas

*GitHub:* [*https://github.com/SaiVillani/IVR*](https://github.com/SaiVillani/IVR)

**Robot Vision**

2.1: Joint state estimation

{Algorithmic explanation, and reasoning}

The algorithm first uses detects colour function via blob detection, to detect the relevant colour and zone in on the centre of the sphere, using BGR format for red, green, blue, and yellow. There are two callback functions, these allow us to tune in to the camera streams, as well as re-direct our cameras. For example, in callback2 when camera 2 fails to detect the red sphere the estimated created from the image of camera 1 are used instead. Beyond this, we use a standard pixel to meter conversion function that is used to detect the joint angles. 2 implementations for the detection of joint angles are included, a basic version which relies on trigonometry, and many if clauses for corner cases, which as you will see in the graphs result in violent pulls and flat lines, as well as a chamfer matching implementation that is built on a detecting the rotation of the links by matching the outline of the links to improve our estimations.

Given that Joint angle 4 was so difficult, it wasn’t enough to simply switch between camera 2 and camera 1 when the red sphere is not visible. This is because, even when camera 1 can see all 3 relevant components compared to camera 2 it still fails to detect motion in the x axis because the picture is completely adjacent to the robot. So, for camera 1 it is very difficult to tell if the red sphere and thus joint angle 4 is moving away or towards it. So, we attempted to implement linear extrapolation by estimating the position of the red sphere via our knowledge of the position of the blue and yellow spheres, which are much easier to spot, thus trigonometry can be used to estimate the position of the red sphere. This allows us to better deal with this mirroring effect of misinterpreting the sign of the angle, by transforming a given picture to our extrapolated picture via the orthogonal vector which allows us to map the coordinates from real to extrapolated.

{Graphical results, and comments}

Chart, line chart

Description automatically generatedThe estimations of joint angle 2 are not optimal. The Chamfer matching estimations seem to be the most consistent, particularly on the slopes, but not near the troth of the curve, this is likely to be due to the blockage of [DETAILS].

Chart

Description automatically generatedThe estimation of joint angle 3 are significantly better for both the simpler method and Chamfer matching model. However, both of our estimation models will flatten out near the peak or troth of the actual angle, this is due a hard coded assumption, that when [Details] we know it must be a right angle. The estimation recovers from the hard code quickly.

Chart, line chart

Description automatically generatedThe estimation of joint angle 4 is the worst by far. The simple estimation method seems to completely oppose the sign of the curve, we encountered a similar problem in vision2 as well. The chamfer method seems to be better at picking up the correct direction, and occasionally follows closely to the true angle, however, it often wildly deviates from the actual pattern of motion. [details].

{Corner cases, explanations of some bad results, show investigation, ask Si Han}

We did perform investigations, which included snap shots of the exact times when either camera would not be able to either see a sphere because it was blocked by another, or because it failed to capture the relevant axis of motion. For example [cases when 1 covers other]. Additionally [camera 1 cannot help with ja4 because it cannot tell of x axis on y axis.]

2.2: Joint state estimation

The algorithm is similar, however fixing the second joint has made things harder again because [DETAILS].

{Graphical results, and comments}

{Corner cases, explanations of some bad results, show investigation, ask Si Han}

**Robot Control**

*3.1: Forward Kinematics*

The algorithm uses the simple stuff that we learned from forward kinematics. The forward kinematics calculation was done with the following notation, as well as these Denavit-Hartenberg parameters.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Link number | D (meters) | (degrees) | r | α |
| L1 | 4.0 | 90 | 0 | 90 |
| L3 | 3.2 | -90 | Not sure | -90 |
| L4 | 2.8 | 90 | Not sure | 0 |

|  |  |  |
| --- | --- | --- |
| Comparison number | Actual Position [z, x, y] | Calculated Position [z, x, y] |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |
| 7 |  |  |
| 8 |  |  |
| 9 |  |  |
| 10 |  |  |

Speak of different cases, error drift. There are inacurracies why??

It could be because we are not waiting enough between them? The robot needs more time. Trajectories are really inacurrate??

*3.2: Inverse Kinematics*

The inverse kinematics demands we infer the position of the joints based on the end effector. As such we required implementing quite a few more things.

{Graphical results, and comments}

{Corner cases, explanations of some bad results, show investigation, ask Si Han}

Things to discuss:

Average error, error drift, and deviation from the signal.