Robot Visuomotor Control

By Patrick J Nkwocha M00720579

PDE3433 Advanced Robotics Assessment 3

Contents Page

1. Assignment Objective

For this project I have been tasked with writing a program to catch a flying object (ball) thrown into a robot workspace with a UR10 arm, without any provided cartesian values as instructions. This will be done inside the ROS ecosystem.

2. Understanding the problem at hand

To do this, one must carry out an evaluation of the scenario at hand. Initially, a ball trajectory would be generated, which abstracts the path in which the ball is travelling along. Naturally, the possibility of the robot manipulators end effector being in the exact position that coincides with the ball trajectory, in order to catch it would be very small. For this reason, the error (segment) between the desired position of the manipulator, and the current positioning would have to be considered and then mitigated by some sort of action (movement) via a written algorithm. This would also be necessary, as once the manipulator had caught the ball a new trajectory would be generated.

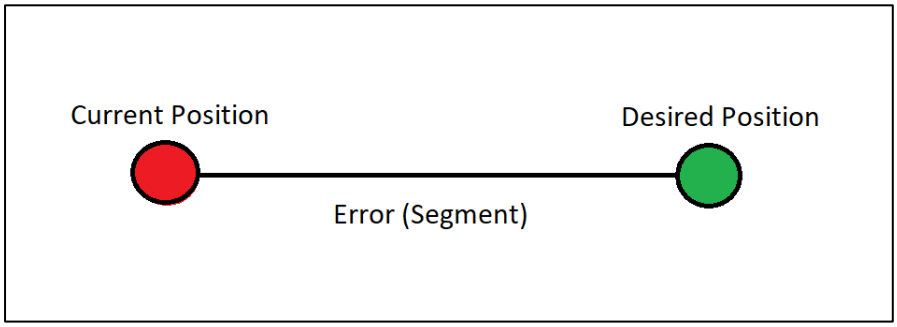

Figure 1 - Depiction of problem abstraction

To achieve this, the program will have to do the following:

- Interpret the robot manipulators position (pose)
- Compare its position with the desired position (ball's trajectory), to elicit the error.
- Update the manipulators position (movement) to mitigate the error.

All the while notifying the user of the value of the recurringly updated error, as well as the balls state (if it had been caught or not). This problem is what in artificial intelligence is referred to as a ***search problem.*** A search problem consists of:

- A ***state space*** – a set of all the possible states where you can be e.g., the cartesian points that make up the space in which the manipulator finds itself (its surroundings)
- A ***start state*** – the state from where the search begins e.g., the starting position of the robot manipulator (its home position).
- A ***goal test*** – a function that looks at the current state and returns whether or it is the goal state e.g., a query to interpret results. It should be noted that the brief also provided a threshold for a successful catch which would assist with formulating a goal test query.

Meanwhile the solution to the search problem would comprise of a sequence of actions that make up the plan, ultimately, which transform the start state into the goal state. This is achieved by using ***search algorithms***.

3. Interpreting a manipulator robot's positioning

The pose of a robot manipulator is a combination of both position (x, y, z) and orientation (Euler angles roll, pitch, yaw) [1], sometimes elicitation of the error can include both position or orientation partially, or both completely. In this case both were necessary to fulfil requirements.

From my understanding each error (cartesian and orientational) would be an abstraction of discrepancy among one of these parameters solely, or a result of a discrepancy amongst a combination of them. Therefore, I felt that querying the effect each parameter has on its corresponding error would eventually yield a solution. Formulation of my solution began based on the image below:

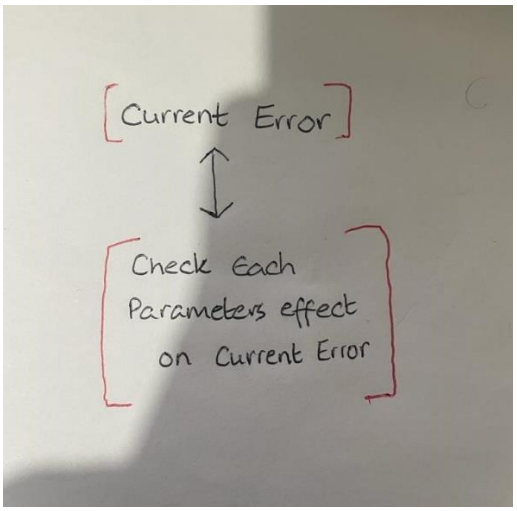

Figure 2 - Initial Idea for my algorithm

Where after observing the effect of the queried parameter my algorithm would behave accordingly.

4. My approach for a solution algorithm

After recognizing the task as a search problem, the best way to attempt at solving this problem was by making use of a search algorithm. My approach can be broken down into sections of setting the start state, retrieving information on the goal and current states, and the plan (getting closer to the goal state).

4.1. Setting the start state

As the ball trajectory would be provided by the system my devised search algorithm would be somewhat informed, meaning that my algorithm would have some information on the goal state. This would make the search more efficient. Beforehand, I would have to establish a start state.

This is achieved by the following code:

```
# Notifying user that robot is moving
print("\nNow moving the robot to its home position...")

# Move the robot manipulator to home position
command = self.ur_con.generate_move_j(self.home_waypoint)

# Publish movement command to move robot
self.ur_script.publish(command)
```

Figure 3 - Code Snippet (Establishing the start state)

Here I use a list containing predetermined cartesian values (x, y, and z) and rotational force values (roll, pitch, yaw) and assign them to a "***home_waypoint***" variable. I then use a "***command***" variable which is then passed into a function which composes a message (topic). After which a script variable (***ur_script***) is used to publish the command (topic) in the ROS environment.

4.2. Retrieving information on the goal state and current state

Obtaining information on the goal state is done by something called a heuristic. A heuristic is a function which finds out how close a state is to the goal state. As stated, in this case the goal state is the generated ball trajectory. In order to get feedback on the discrepancy between the goal state and current state. The algorithm would also require a means of knowing the robot manipulators current state (position/pose). This was achieved by the following code:

```
# Notify user that the robot has reached the predetermined home position
print("The robot has successfully reached its home position")

# Get the robot pose (position)
pose = self.ur_con.get_pose()

# Get errors (distance between desired positioning - distance, and rotation)
errors = self.ur_con.check_errors(pose)

# Assign each variable for cartesian & rotaion error, and ball state (caught or not)
distanceError, rotationError, ballCaught = (errors.ee_trans_error), errors.ee_rot_error, errors.ball_caught

# Display default errors
print("\nDefault errors:")
print(" - Cartesian error:", (distanceError * 1000),"mm") # Directional error in millimeters
print(" - Orientational error:", rotationError,"degrees") # Rotational error in degrees
print(" - Ball caught:",ballCaught) # Boolean tell that tells us if the ball was caught or not
```

Figure 4 - Snippet depicting retrieval of error segment.

Here, the function (***get_pose***) which is inside of a class is called to get the current state of the robot manipulator. To check the discrepancy between both the current state and the goal (ball trajectory) another function is called (***check_errors***) in which the pose is passed into as an argument (parameter). The result is then returned, and its attributes, in particular the distance and rotation errors, as well as the ball state are derived for future processing within the plan, where it is used similarly to query the manipulators movement outcomes. This section of the code helps specify the initial error segment and serves as a goal test.

4.3. The Plan - Getting closer to the goal state.

The objective of the algorithm was to catch the ball within as little time as possible. To do so I decided to opt for a hill climbing heuristic search algorithm. Hill-climbing is a simple solution for the given problem and is often used in optimization problems where the goal is to find the best solution from a set of possible solutions [2].

For example, with the travelling salesman problem where the goal is to minimize the distance travelled by the salesman. In this case I would be taking the errors (cartesian error, and orientational error) and trying to minimise them.

Hill climbing algorithms work by:

- Evaluating the initial state. Querying if it is at the goal state, if so stopping and returning a success message. Otherwise, assigning the initial state as the current state.
- Looping until a solution state or a state within a threshold of the goal state is found.

4.3.1. Handling the cartesian error

To offset the search algorithm, I implemented a while loop which queries the Boolean value of the ball state (***ballCaught***), and only executes if the ball state is False (the ball has not been caught). In the case of it being False I begin by handling the cartesian error (distance error). Here I also used a while loop which only executes if the current distance error is greater than the distance threshold (0.05m). If so, I use the existing cartesian error (***distanceError***) and a for loop to iterate over the cartesian axes (***self.cartesianAxes = ["X", "Y", "Z"]***). I assigned the distance error as the increment value for the movement, in which the ***moveLAndFindError()*** function is called with the increment and cartesian axis passed as arguments to update the manipulator pose and derive a new distance error.

If the new movement results in a smaller distance error than the previous error, the distance error variable is updated. Otherwise, an inverse movement is carried out by multiplying the increment by -2, subsequentially cancelling out the initial move and then moving in the opposite direction. After which, the updated distance error is queried again to check if it is smaller than the previous error and the distance error variable is updated accordingly. If the distance error cannot be reduced by either the positive or negative movement, it reverts the manipulator pose to the initial position and retrieves the initial distance error, rendering any current movement along the selected cartesian axis as ineffective. This cycle loops recurringly until the distance error is found to be less than the threshold, where I utilized a break to leave the while loop. Below is a snippet of the code depicting this.

```
while distanceError > self.distanceThreshold:

    # For loop for iterating through catesian axis list (cartesianAxes)
    for axis in self.cartesianAxes:

        # Assign distance error as movement increment
        increment = distanceError

        # Call function to update manipulator pose and retrieve new distance error
        updatedDistanceError = self.moveLAndFindError(axis, increment)

        # If statement to query if the new movement generated a smaller distance error than prior, if so assign result to distance error vari
        if updatedDistanceError < distanceError:

            distanceError = updatedDistanceError

        else:

            # Assign inverse increment variable by multiplying the increment by -2 (this cancels out the inital move, back to its origin posi
            inverseIncrement = increment * -2

            # Call function to update manipulator pose and retrieve new distance error
            updatedDistanceError = self.moveLAndFindError(axis, inverseIncrement)

            # Query if the new movement generated a smaller distance error than prior, if so assign result to distance error variable
            if updatedDistanceError < distanceError:

                distanceError = updatedDistanceError

            else:

                # Call function to revert manipulator pose and retrieve initial distance error
                distanceError = self.moveLAndFindError(axis, increment)

            print(" - | Queried the {} axis. | The distance error is {} mm. |".format(axis, distanceError))

            # Add 1 to number of steps for hill climb
            numOfSteps += 1

            # Check if the distance error is less than or equal to the distance threshold
            if distanceError <= self.distanceThreshold:
                print("\nThe cartesian error has been successfully handled!")
                break
```

Figure 5 – Snippet depicting handling of cartesian error.

4.3.2. *Handling the orientational error*

After noticing how my handling of the cartesian error was affective I decided to implement a similar method for handling the orientational error. Similarly, I begin with a while loop which only executes if the current rotation error is greater than the orientation threshold (5°). If so, I use the existing orientational error (***rotationError***) and a for loop to iterate over the rotational forces (***self. rotationalForces = ["Roll", "Pitch", "Yaw"]***). I then assign the rotation error as the deviation value for the rotation movement, in which the ***rotateEndEffectorFindError()*** function is called with the deviation and rotational force passed as arguments to update the manipulator pose and derive a new rotation error. To mitigate the rotation error, I used the same logic as described for the handling of the distance error. Below is a snippet depicting this.

```
while rotationError > self.orientationThreshold:

    # For loop for iterating through catesian axis list (cartesianAxes)
    for force in self.rotationalForces:

        # If statement to query forceName and determine force axis
        if force == "Roll":
            forceAxis = "X"
        elif force == "Pitch":
            forceAxis = "Y"
        else:
            forceAxis = "Z"

        # Assign rotation error as angular movement deviation
        deviation = rotationError

        # Call function to update manipulator pose and retrieve new rotation error
        updatedRotationError = self.rotateEndEffectorFindError(force, deviation)

        # If statement to query if the new movement generated a smaller rotation error than prior, if so assign result to rotation error variable
        if updatedRotationError < rotationError:

            rotationError = updatedRotationError

        else:

            # Assign inverse deviation variable by multiplying the deviation by -2, subsequently moving (rotating) the robot in the opposite direction
            inverseDeviation = deviation * -2

            # Call function to update manipulator pose and retrieve new rotation error
            updatedRotationError = self.rotateEndEffectorFindError(force, inverseDeviation)

            if updatedRotationError < rotationError:

                rotationError = updatedRotationError

            else:

                # Call function to revert manipulator pose and retrieve initial rotation error
                rotationError = self.rotateEndEffectorFindError(force, deviation)

        print(" - | Queried force - {} ({} axis). | The rotation error (angular deviation) is {} degrees |".format(force, forceAxis, rotationError))
```

Figure 6 – Snippet depicting handling of orientational error.

5. Conclusion

In conclusion I was able to achieve the objective of repeatedly catching the flying object (ball) via my solution. It should be noted that after carrying out some research I do believe there are various approaches to fulfilling the objective. For one alternatively, I might have used a machine learning model e.g., linear regression, or knearest neighbours to predict the pose for catching the ball based on the error. My reasoning for not going for this approach was I felt despite it theoretically making sense, implementation would be far more complex as I would have to retrain large sets of data after a new trajectory is generated. Another approach could be using inverse kinematics to calculate the joint angles required to move the end-effector to the desired position. I found this could be achieved using information which we know about the UR10 robot [3] and what information we derive from its positioning via the program. I believe that this approach could potentially complete the search in a quicker time than my solution by implementing a mathematical equation, along with a similar iterative algorithm. However, my solution is still able to complete its search in a use-case timely period.

6. References

[1] Palanisamy, T. (no date) Robotics position and orientation, Share and Discover Knowledge on SlideShare. Available at: https://www.slideshare.net/ThiyagaRajan56/robotics-position-and-orientation-83712713.

[2] Introduction to hill climbing: Artificial intelligence (2023) GeeksforGeeks. Available at: https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/.

[3] Universal UR10 (no date) Robots Done Right. Available at: https://robotsdoneright.com/Universal/Universal-UR10.html.