

Robotic Fundamentals: Serial and Parallel Robot Kinematics

Finlo Heath
(Dated: January 8, 2023)

ABSTRACT

This report contains details, results and analysis for a series of simulations created to investigate the kinematics of serial and parallel robots. For the serial robot, the forward and inverse kinematics are derived and used to simulate a drawing task with both linear and arcing motion. The reachable workspace is derived. A bug2 algorithm is implemented for obstacle avoidance. For the parallel robot, the inverse kinematics and workspace are derived and displayed. Investigation into avoiding failure due to singularities is made. Simulations are created in MatLab, the original code can be found in the appendices of this report, or at the Author's GitHub repository: <https://github.com/Hinlo/MatLab-Simulation-of-Serial-and-Parallel-Robots>

CONTENTS

I. Introduction	3
II. Serial Robot: Investigations & Methodology	3
A. Forward Kinematics	3
1. Testing	4
B. WorkSpace	4
C. Inverse Kinematics	4
D. Task Completion	5
E. Trajectories and Obstacle Avoidance	5
1. Linear Motion	5
2. Free Motion	6
3. Obstacle Avoidance	6
III. Parallel Robot: Investigations & Methodology	6
A. Inverse Kinematics	6
B. WorkSpace	6
IV. Serial Robot: Results, Analysis and Discussion	7
A. Forward Kinematics	7
B. WorkSpace	8
C. Inverse Kinematics	8
D. Task Completion	10
E. Trajectories and Object Avoidance	10
1. Linear Motion	10
2. Free Motion	11
3. Obstacle Avoidance	11
V. Parallel Robot: Results, Analysis and Discussion	12
A. Inverse Kinematics	12
B. WorkSpace	12
VI. Part Three: Investigation of Methods to Avoid Issues When Operating the Robot Close to a WorkSpace Singularity	13
VII. Conclusions & Further Work	15
A. Serial Robot Forward Kinematics	16
B. Serial Robot Inverse Kinematics	22
C. Serial Robot Task Completion	25
D. Serial Robot Linear Trajectories	32
E. Serial Robot Free Trajectories	39
F. Serial Robot Obstacle Avoidance	45
G. Parallel Robot Inverse Kinematics	58
H. Parallel Robot WorkSpace	64
References	67

I. INTRODUCTION

Forward and inverse kinematics provide the ability to control and position robots in real space. Simulation allows testing well as quickly viewing the results of changes to code, without risk of damaging expensive equipment.

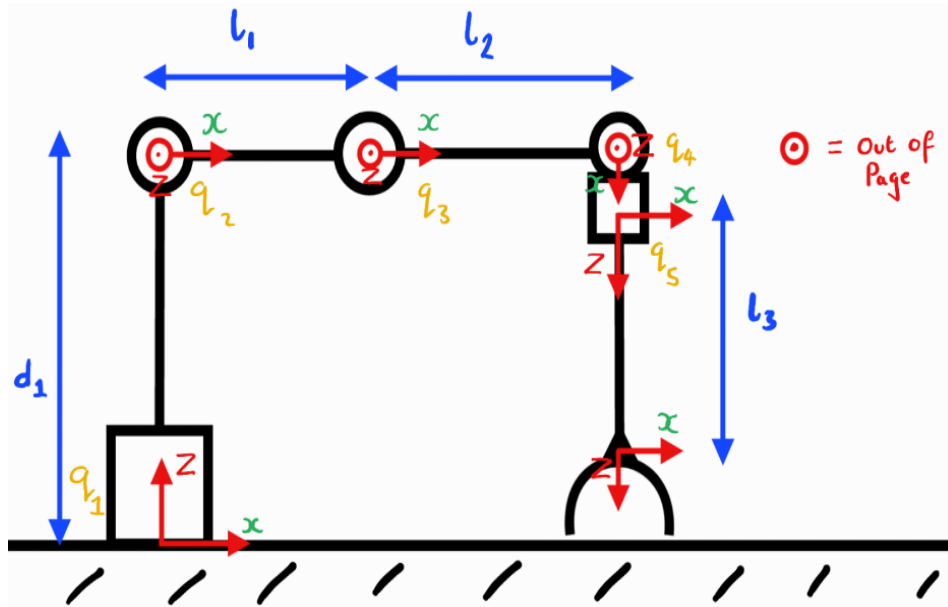
The serial robot simulated in this project is the LynxMotion ALD5 Robot Arm (*lynxmotion* n.d.). This robot has five degrees of freedom (DoF), counting the final rotating prismatic joint which does not affect the spatial position of the robot's end effector. The parallel robot simulated is in the style of a surgery robot.

This report details the methodology used to create each simulation, before presenting results, discussion and methods to avoid singularities. Conclusions are made including suggested further work. The appendices contain the full original code for each MatLab script created (Finlo Heath 2022).

II. SERIAL ROBOT: INVESTIGATIONS & METHODOLOGY

A. Forward Kinematics

Forward kinematics (FK) aims to derive the spatial position of each joint and the end effector of the robot, given the link lengths and joint angles. This project uses the Distal Denavit Hartenberg (DH) approach to FK (Wang et al. 2014). This requires creation of a diagram of the robot to fill the values in a "DH Table" as shown in FIG. 1. This method is more robust than the modified proximal method, as it does not require imaginary axes in the case that two joints have no physical distance between them.



(a) LynxMotion Arm at Zero Point Configuration

n	a	α	d	θ
1	0	90	d_1	q_1
2	l_1	0	0	q_2
3	l_2	0	0	q_3
4	0	90	0	q_4
5	0	0	l_3	q_5

(b) The Denavit Hartenberg Table for the LynxMotion Arm (angles in degrees)

FIG. 1. (a) LynxMotion robot arm annotated with the z -axis of rotation at each joint in the $x-z$ plane of the base. The robot is drawn in it's zero point configuration, meaning that each angle $q_1 - 5 = 0$. From this Diagram, table (b) is generated and substituted into the transformation matrix given in FIG. 2 to generate the transformation matrices between each consecutive joint.

Each row of the DH table is applied to the distal transformation matrix shown in FIG. 2, providing the transformation from one joint of the robot to the next. Multiplying in sequence, one can find the full transformation from the base of the robot to end effector, including both the rotation matrix and spatial translation; discernible within the transformation matrix as shown in FIG. 2 (b). In Appendix A, the transformation matrices are given in lines 25 - 53, with the DH table provided above in comment lines 14 - 20. Matrix multiplication - from right to left - is used to get the full transformation from base to end effector on line 62.

To acquire full robot arm motion, one must take the spatial translation component from not only the full base to end effector transformation matrix (lines 109 - 143) but also each of the preceding composite transformation matrices (lines 164 - 240). The spatial position of each joint is plotted at each angle value to give the overall change in motion of the whole arm.

$${}^{n-1}_n T = R_{z_{n-1}}(\theta_n) T_{z_{n-1}}(d_n) T_{x_n}(a_n) R_{x_n}(\alpha_n) =$$

$$\begin{bmatrix} c\theta_n & -c\alpha_n s\theta_n & s\alpha_n s\theta_n & a_n c\theta_n \\ s\theta_n & c\alpha_n c\theta_n & -s\alpha_n c\theta_n & a_n s\theta_n \\ 0 & s\alpha_n & c\alpha_n & d_n \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(a) The Distal Transformation Matrix

3×3 Rotation Matrix			1×3 Translation Vector
0	0	0	1

(b) Rotation and Translation Components of a Transformation Matrix

FIG. 2. (a) shows the generic distal transformation. Each element is substituted for the value in a given row of the DH table (FIG. 1 (b)) to find the transformation between two joints connected by a link (Wang et al. 2014). (b) distinguishes the rotation and translation components of a transformation matrix.

1. Testing

Edge cases allow testing that the FK work correctly. Providing each angle value as zero should output the robot arm in the zero point configuration displayed in FIG. 1 (a). Another useful reference position is $q_1, q_3, q_5 = 0$ and $q_2, q_4 = \pi/2$, which should place the robot in the straight upward orientation with spatial translation:

$$\begin{pmatrix} 0 \\ 0 \\ d1 + l1 + l2 + l3 \end{pmatrix}$$

A more concrete way of checking that both FK and inverse kinematics (IK) are working is to pick a desired end effector position, run the IK code to output the joint angles, then run these through the FK to check that this code returns the original end effector position.

B. Workspace

The Workspace of the robot is the full set of coordinates reachable by its end effector (Szep et al. 2009). The dexterous workspace is the subset of these coordinates which the robot can reach in all possible orientations. In Appendix A Lines 275 - 322, the reachable workspace is plotted. Using the full range of motion for each joint, a series of loops plot each point. Any points where the z coordinate is less than zero are skipped, preventing the end effector from moving below the ground surface.

C. Inverse Kinematics

IK takes an input of the end effector's orientation and position, outputting the five joint angles ($q_1 - q_5$). This can be done entirely using trigonometric analysis of the robot; this is illustrated in FIG. 3.

q_1 is derived using FIG. 3 (a). q_5 has no bearing on the spatial position of the robot, only the rotary orientation of the end effector. Hence it is given as in input.

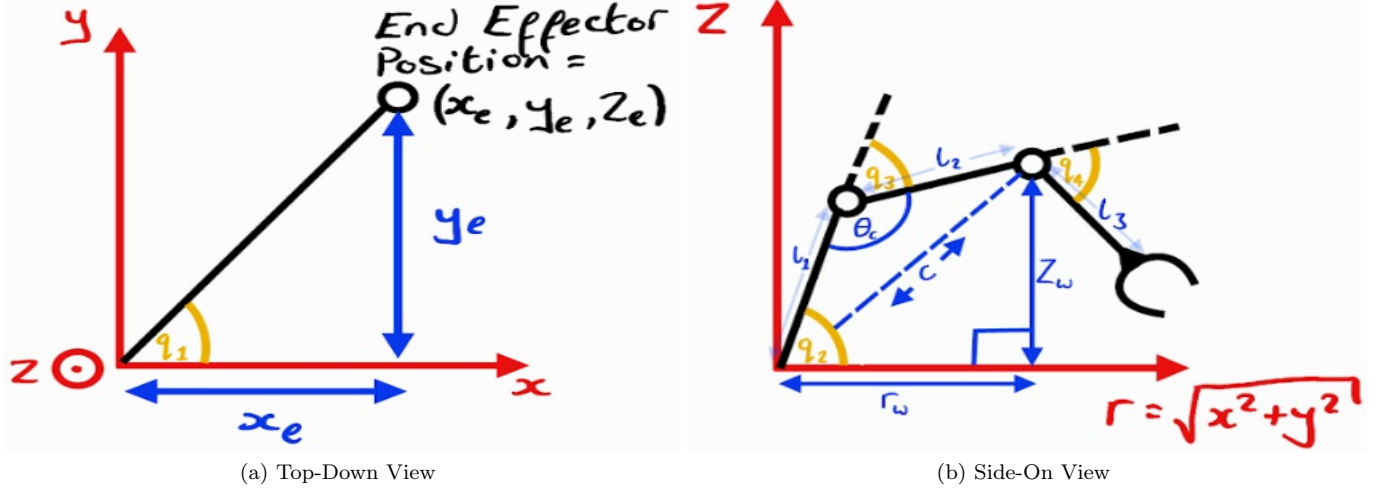


FIG. 3. The top down view allows one to find q_1 , the side on view allows for one to directly find q_2 and q_3 . q_4 is then found using the equation for ψ .

The remaining angles, q_2, q_3, q_4 are derived by modelling the mid section of the robot as a three joint planar robot and using trigonometric identities, displayed in FIG. 3 (b). With a few exceptions, there are multiple orientations the robot can have to reach a given position. This orientation is defined by

$$\psi = q_2 + q_3 + q_4$$

For this reason, as shown in Appendix B lines 26-27, the full range of possible values for ψ are used. Imaginary results are filtered out and a list of all real sets of possible angles are printed.

D. Task Completion

The chosen task was for the robot to draw (move to) the eight corners of a cuboid in 3D space. Completing the task is done using the following approach:

1. The points that the robot must move to are provided in a list.
2. IK are run to acquire the joint angles for the robot at each of these points
3. FK are run to acquire the position of each joint at these angles
4. The full robot orientation is plotted at each required point.

E. Trajectories and Obstacle Avoidance

Appendices D, E & F provide MatLab files for the three paths. The robot was modelled as operating at 10Hz, with a 1 second travel time between each point, meaning the trajectories display ten intermittent points between each corner of the cuboid. While a real robot must operate at far higher frequencies, this was deemed sufficient for this simulation.

1. Linear Motion

Linear motion means the robot end effector moves in a straight line. This was achieved by adding an extra step between steps 1 and 2 in section IID. straight lines are defined by equations of the form $y = mx + c$. A function was created which takes consecutive points in the list of those the robot must move to, returning the line gradient and

constant.

Ten points are sampled along the line between each of the original points to create a new robot trajectory array. Inverse and forward kinematics are run in sequence as before and the robot motion is displayed in a figure.

2. Free Motion

Free motion means the robot does not have to obey a specific path between each point. To achieve this, the first two steps of the method in section IID are followed. Consecutive sets of angles output by the IK are samples for ten intermittent points. FK are run on this expanded set of angles and the output is displayed, showing arcing as opposed to linear motion between points.

3. Obstacle Avoidance

The free motion trajectory was used as the starting point for the obstacle avoidance. As well as defining the points the robot must move to, the obstacle is defined as well. The chosen obstacle is a small cuboid in the path of the robot's trajectory. The inverse and forward kinematics are run, to find the full path of the robot with no diversion.

A function was created which samples the trajectory of the robot and discerns where it first impacts and leaves the obstacle. The function returns a path for the robot to move from its impact point, over the surface of the obstacle, to the point at which it leaves the obstacle, emulating the bug 2 algorithm (Yufka and Parlaktuna 2009). This new path is inserted in place of the previous path.

III. PARALLEL ROBOT: INVESTIGATIONS & METHODOLOGY

A parallel robot operates with multiple arms at different positions on the base, rather than a single base to end effector chain. The parallel robot in this study operates in the horizontal ($x - y$) plane. It has three identical arms each comprised of two joints attaching to a separate corner of the triangular end effector and base.

A. Inverse Kinematics

IK were solved by modelling the robot as three identical serial robot arms with different bases and at different angles to the end effector. This allows one to solve the IK for a planar three DoF arm and apply this to each of the arms in sequence. For each arm, the position of its base is taken as the origin in that arm's frame, and the position of the end effector is shifted to its coordinate in that arm's frame.

The angle of the arm to the end effector is the third joint angle, q_3 and is fixed based on the end effector angle, a . For the first arm, $q_3 = a + \frac{\pi}{6}$ as this is the angle from the x -axis to the centre of the end effector. The end effector is an equilateral triangle, hence $\frac{\pi}{6}$ is the angle between its edge and the vector to its centre point. For the second arm, this angle is the same except offset by an additional $\frac{2\pi}{3}$. The third is offset by $\frac{4\pi}{3}$. This occurs as the angle is always measured from the x -axis and the arms are set $\frac{2\pi}{3}$ apart on the end effector.

Once the relative end effector position and angle of the arm to the end effector is found, a function outputs the possible values for the first and second joint angles. These form an "elbow" joint which can each have two values for a given position. This function can be viewed in Appendix G lines 343 - 360.

B. Workspace

The file created in section III A was adjusted to take a large range of end effector positions. All imaginary results are removed, all real results are plotted to show the workspace for a given end effector orientation. Greater end effector angles require more taut links resulting in a smaller workspace.

IV. SERIAL ROBOT: RESULTS, ANALYSIS AND DISCUSSION

Across all simulations, link lengths of the robot are set to $0.1m$.

A. Forward Kinematics

The full analytical solution for the FK of the LynxMotion robot arm is a large 4×4 transformation provided at the start of Appendix A. One can be sure it is correct as it provides the expected results when given edge case angles. Two examples of this are shown in FIG. 4.

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(a) T_e for $q_1 = q_2 = q_3 = q_4 = q_5 = 0$ (b) T_e for $q_1 = 0, q_2 = 90, q_3 = 0, q_4 = 90, q_5 = 0$

FIG. 4. (a) shows the specified T_e matrix for the zero point of the robot, matching the configuration shown in FIG. 1 (a); z axis translation cancels out, leaving only translation along the x axis. (b) shows the specified T_e matrix when the robot angles are set for it's max height; all four links point translation along the z axis.

Testing this with a series of positions, the following figure is generated:

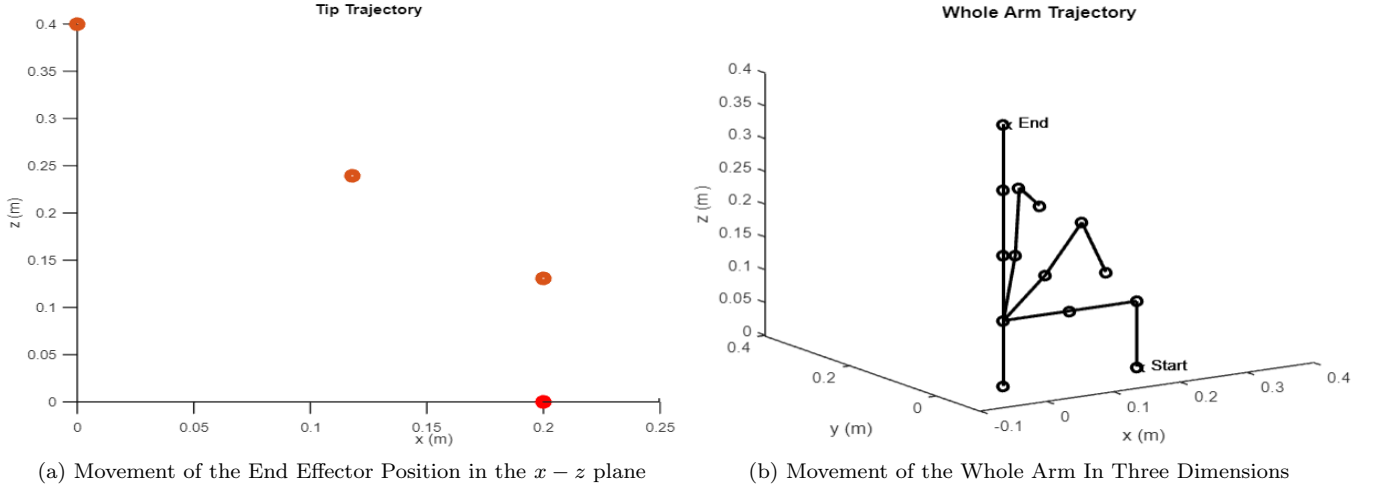


FIG. 5. (a) shows the motion of the end effector only and in the $x - z$ plane. (b) shows the same motion for the whole arm from a three dimensional viewpoint.

FIG. 5 shows the robot beginning at the configuration given by the matrix shown in FIG. 4 (a) and ending at the configuration given by the matrix in FIG. 4 (b)

B. WorkSpace

FIG. 6 Shows the comprehensive workspace of the LynxMotion arm, operating on a flat surface. The workspace is hemispherical since the robot cannot pass through the ground. For the main figure, FIG. 6 (a), each joint is incremented in steps of 15 degrees, steps of 30 degrees are used for (b) and (c); the 15 degree plot was too computationally intense for perspective changes. The true workspace is a continuous area enclosing the points shown in the figure.

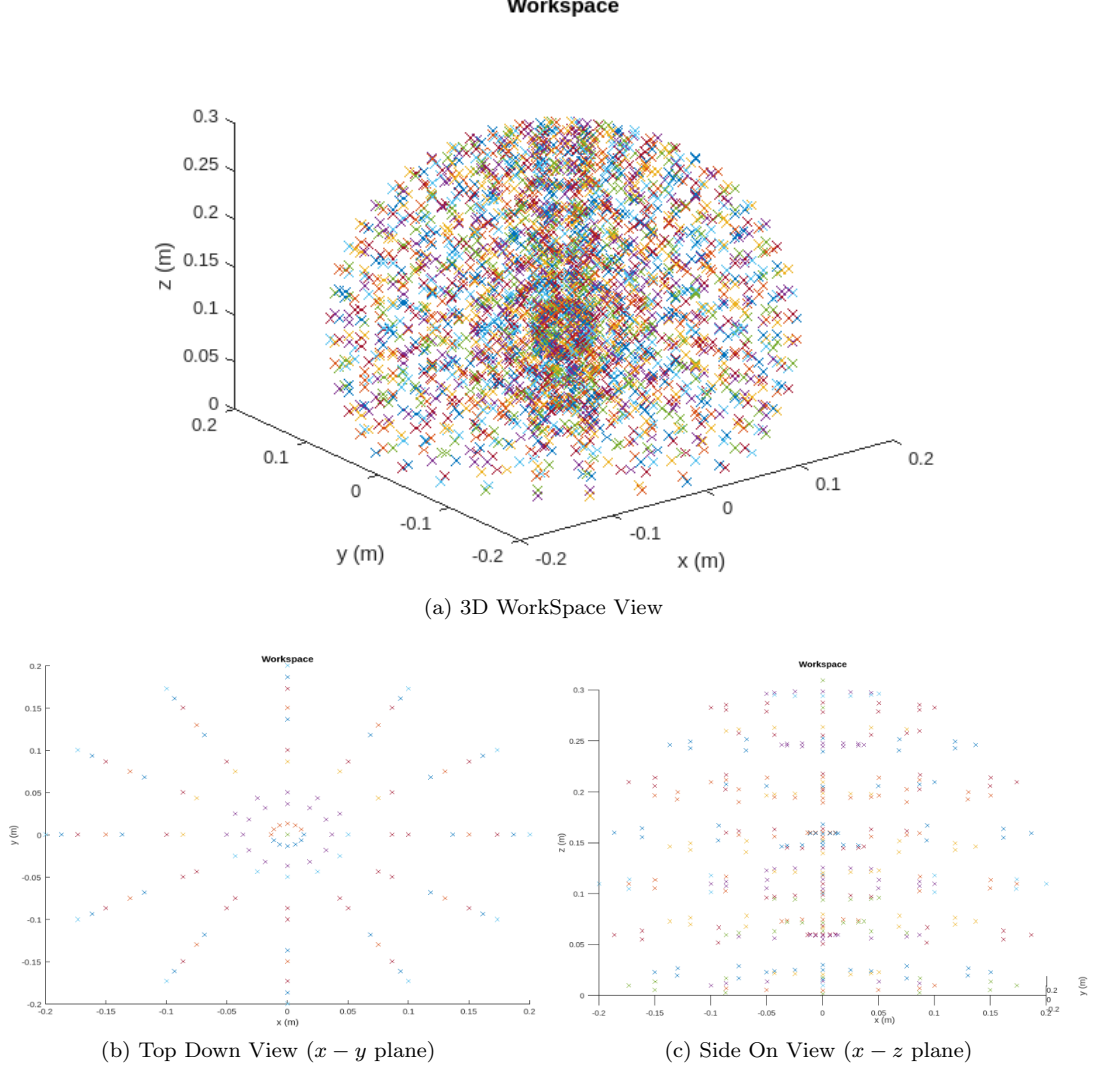


FIG. 6. (a) displays the three dimensional figure showing the workspace of the LynxMotion arm on a flat surface with all link lengths equal to 0.1m. (b) and (c) show a top down and $x-z$ planar view respectively.

C. Inverse Kinematics

The inverse kinematic output consists of five joint angles acquired from an input of the end effector spatial position and orientation. The position is given by coordinates x, y, z . The orientation is given by end effector rotation of μ and angle to the horizontal plane, ψ .

The following values for each joint angle were derived:

$$q1 = \arctan\left(\frac{y}{x}\right)$$

$$q2 = \arctan\left(\frac{z_w}{r_w}\right) + \arctan\left(\frac{l_2 \sin(q3)}{l_1 + l_2 \cos(q3)}\right)$$

$$q3 = \arctan\left(\pm \frac{\sqrt{1 - D^2}}{D}\right)$$

$q3$ is found first of $q2, q3$ & $q4$, the value of D, z_w & r_w are derived in Appendix B lines 70 - 78. As there can be two values of $q3$ for each orientation, it follows that there can be two values of $q2$ and $q4$. After finding values for $q2$ and $q3$, one can find possible $q4$ values for each valid ψ value by rearranging the ψ equation for $q4$,

$$q4 = \psi - q2 - q3$$

$$q5 = \mu$$

Example results for four end effector positions are given below, including the two edge cases we have looked at in section IV A:

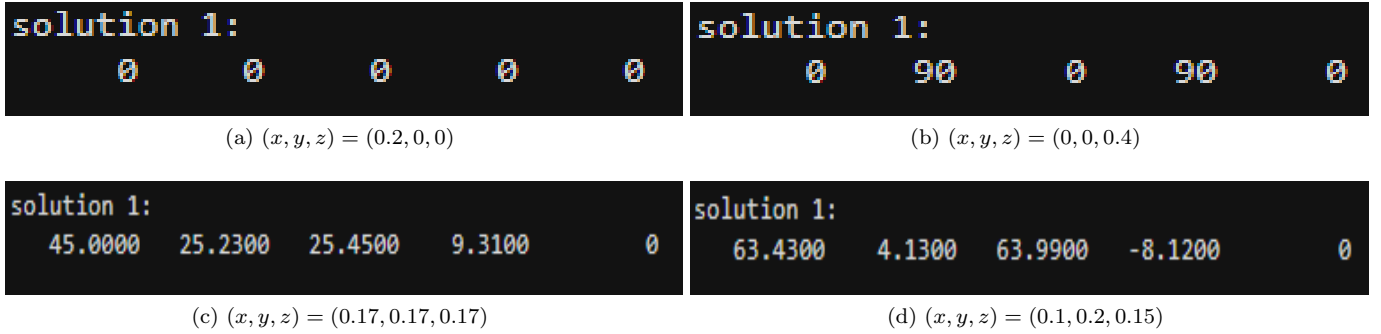


FIG. 7. Output of the Inverse Kinematics file shown in Appendix B. The sub-captions display the input coordinates, the joint angles are output in order $q1 - 5$. The final angle is always zero as $\mu = q5 = 0$ is chosen.

We can see that FIG. 7 (a) and (b) return the input angles given in FIG. 4, and extensive testing corroborates that the forward and kinematics output the inverse kinematic input and vice versa. For each figure in FIG. 7, with the exception of (b) which can be reached in only one orientation, the program outputs multiple possible solutions.

D. Task Completion

The task of plotting the points of a cuboid was successfully carried out by the arm, as shown in FIG. 8. This is also confirmed by the printed output of the file. A check is done to see if the desired points array is identical to the array output for the arm to follow. As long as the desired points are within the workspace, the arm is always successful.

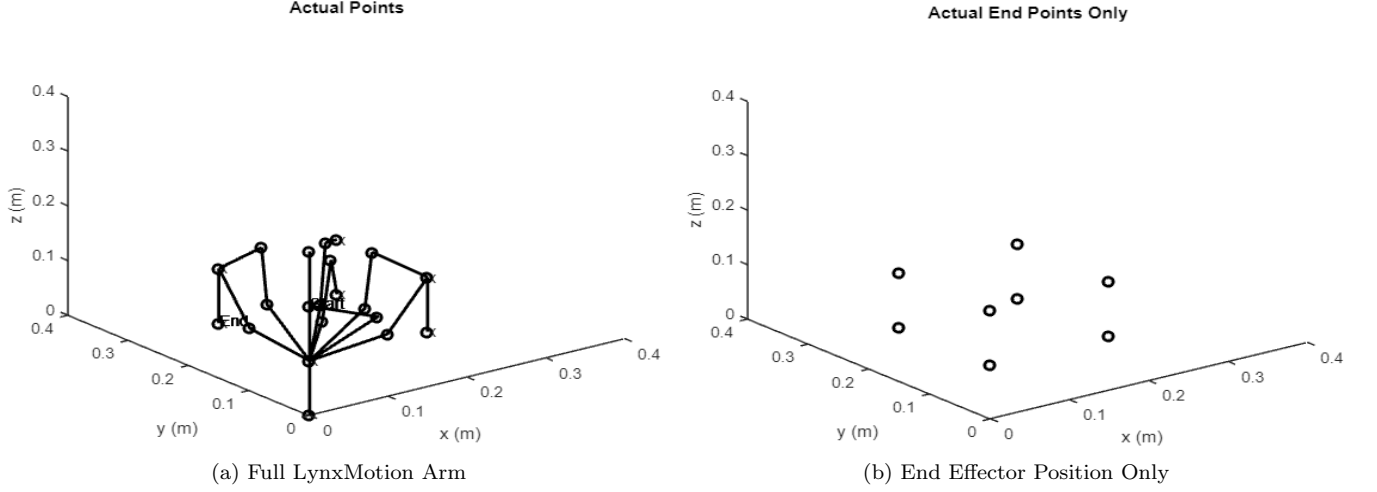


FIG. 8. This figure displays the arm outputting each vertex of a cuboid as required by the set task. (a) shows the full arm, while (b) confirms the endpoint position only.

E. Trajectories and Object Avoidance

When running the MatLab files, each of the trajectory plots are animated, showing robot motion. The task remains identical, but now the robot's path between each cuboid vertex is plotted.

1. Linear Motion

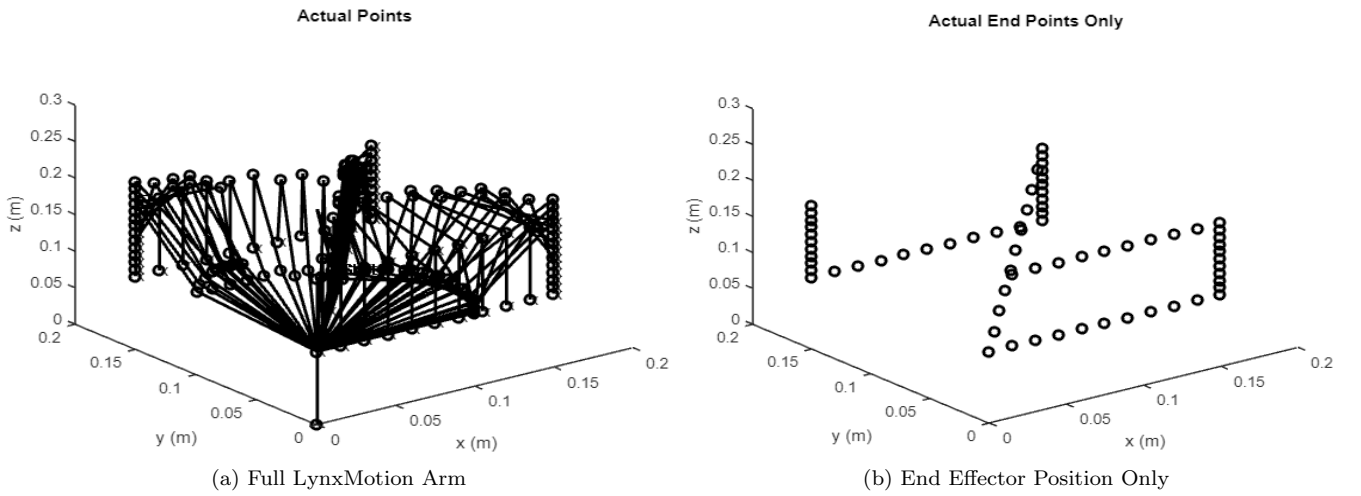


FIG. 9. The robot simulated moving in a linear path between each required vertex point. (a) shows the full arm, while (b) confirms the endpoint position only. The robot begins at $(x, y, z) = (0.0, 0.0, 0.2)$ and finishes at $(x, y, z) = (0.0, 0.15, 0.2)$.

Linear motion is simple to achieve in two dimensions but more complex in three. To speed up the program, the equation function filters to check for lines which pass along each axis first as the gradient and constant are more quickly discerned. This is shown in Appendix D lines 404 - 445.

2. Free Motion

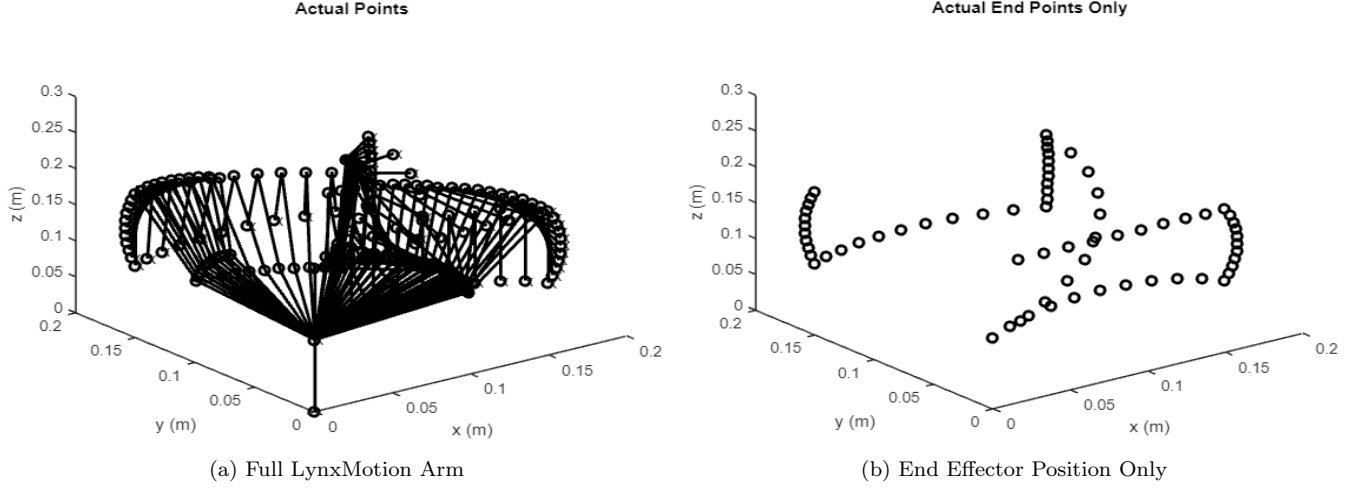


FIG. 10. The robot simulated moving in an arcing path between each required vertex point. (a) shows the full arm, while (b) confirms the endpoint position only. The robot begins at $(x, y, z) = (0.0, 0.0, 0.2)$ and finishes at $(x, y, z) = (0.0, 0.15, 0.2)$.

The arcing effect is created with the increment of each joint angle equally as the arm moves between two points.

3. Obstacle Avoidance

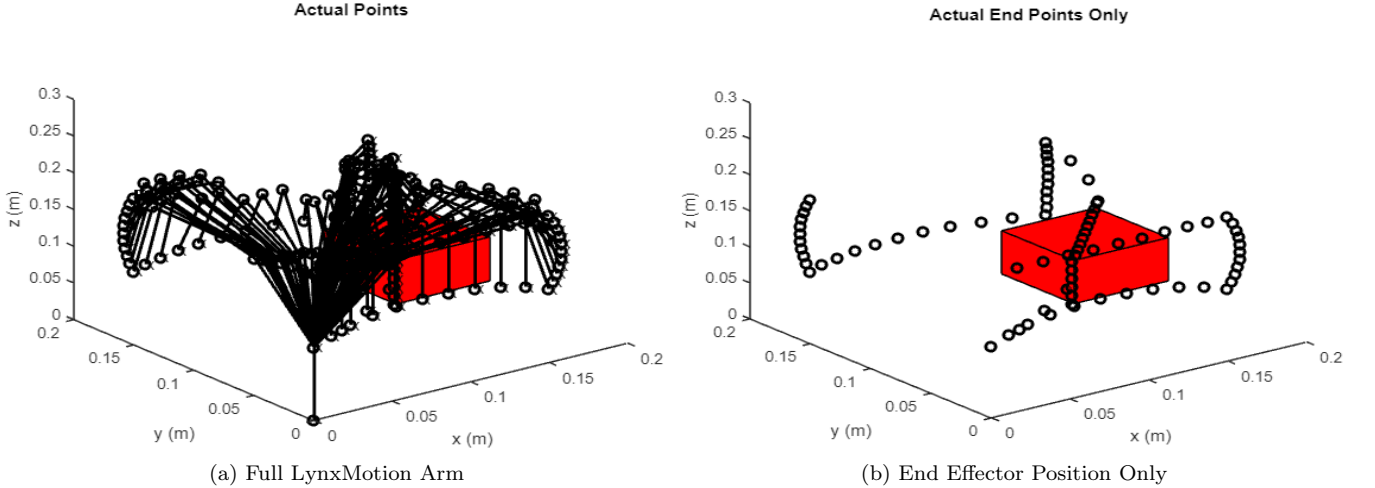


FIG. 11. Free motion of the robot, with bug2 obstacle avoidance implemented. (a) shows the full arm, while (b) confirms the endpoint position only. The robot begins at $(x, y, z) = (0.0, 0.0, 0.2)$ and finishes at $(x, y, z) = (0.0, 0.15, 0.2)$, diverting its path to move around the red cube.

The red cuboid interrupts the arc of the robot. Once it reaches the obstacle, the robot alters its path and traverses the obstacle surface. The solution is general, the obstacle position may be moved and the robot will still traverse

across it. While inefficient, it is necessary to re-run the entire inverse and forward kinematics once the end effector path across the obstacle has been determined. Without this, the joint positions of the robot would not change while it traversed the obstacle - the link lengths would not be constant and the simulation would be inaccurate.

V. PARALLEL ROBOT: RESULTS, ANALYSIS AND DISCUSSION

The parallel robot end effector is shown in dark blue. The three arms are cyan and the base is red. This simulation is completely general. FIG. 12 shows the two opposite configurations, six more can be made as combinations of these two. The end effector remains in the same position despite the differing arm angles.

A. Inverse Kinematics

Readers can test other inverse kinematic solutions by running the file given in Appendix G. The end effector is drawn by connecting the positions of the third joint on each of the arms. Since this always results in an equilateral triangle of side length $\sqrt{3} \times 0.17$ - the defined shape of the end effector - one can be sure the result is correct. If the user requests an end effector position which is outside the workspace for the given angle, the error message, "Chosen Position is outside of robot workspace.", is provided.

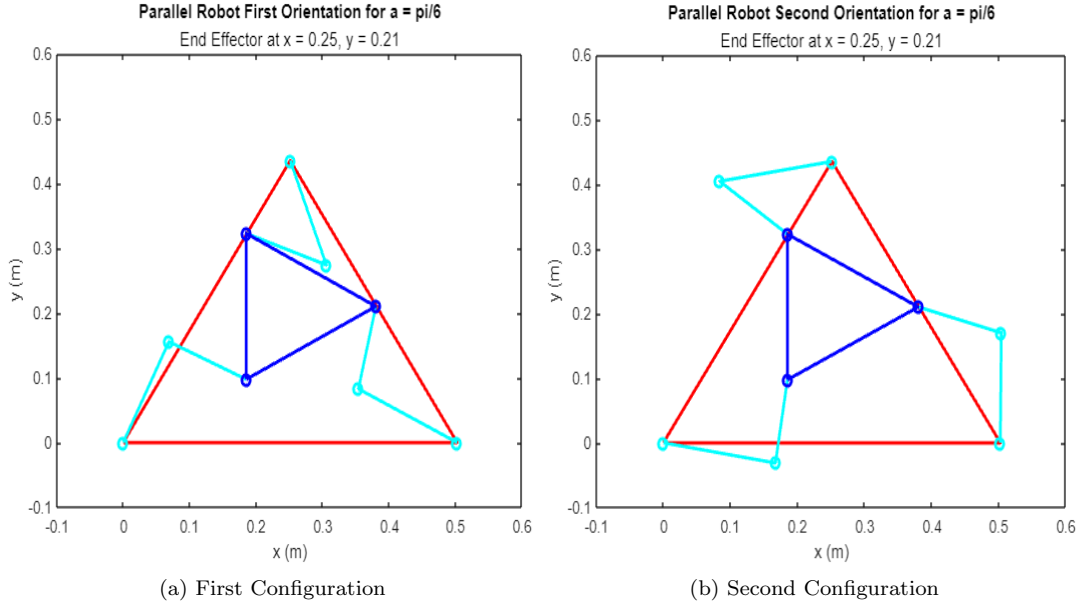


FIG. 12. Two configurations outputting the same end effector position. The two configurations each use the alternating “elbow” possible with the arm links, there are eight different configurations possible overall.

B. WorkSpace

In FIG. 13 we see that the end effector angle, a , has a significant effect on its workspace. The convex hull enclosing the points would suggest a circular workspace for $a = 0$, however the plotted points show that the end effector cannot reach segments of the circle toward the corners of the base triangle. This is because the arms opposite to these corners reach their full extension at these points. For $a = \frac{\pi}{6}$, the workspace shifts in the direction of the angle, increasing on the right side of the robot but decreasing it on the left.

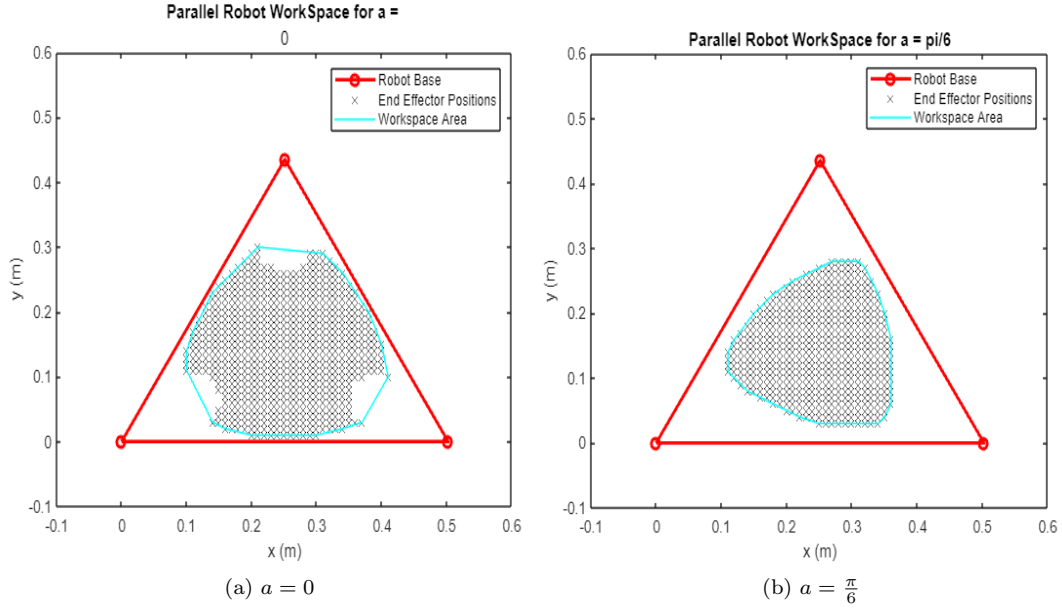


FIG. 13. The full workspace for the parallel robot with the end effector angles $a = 0$ and $a = \frac{\pi}{6}$ to the horizontal (x -axis). The workspace is further reduced as the angle is increased.

VI. PART THREE: INVESTIGATION OF METHODS TO AVOID ISSUES WHEN OPERATING THE ROBOT CLOSE TO A WORKSPACE SINGULARITY

Robots can reach singularities during real world operation. A singularity is defined as a point in the workspace at which the robot cannot operate as intended due to a mechanical limitation causing the loss of one or more DoF (Zhao et al. 2021). For example, in FIG. 5 (b), the end point of the arm is positioned straight up, along the z -axis. In this orientation, it can no longer move along the z -axis without moving in the $x - y$ plane too; a DoF has been lost. This is a boundary singularity. This is why many modern robots, such as Boston Dynamics' Atlas, move exclusively with bent legs (Griffin et al. 2018). In simulation, we can solve this issue by adding angular constraints to each of the joints as shown in FIG. 14. This method reduces the overall workspace of the robot, but it ensures the LynxMotion arm avoids the only full extension singularity in its workspace.

```

1  % Define our set of angles for each joint
2  q1.set = [ 0 20 40 45 ]'*pi/180 ; % base can spin full 360 degrees
3  q2.set = [ 0 30 60 90 ]'*pi/180 ; % first p joint can move 0 to 180
4  q3.set = [ 0 10 20 30 ]'*pi/180 ; % second p joint can move -180 to 180
5  q4.set = [ 0 -10 -20 -30 ]'*pi/180 ; % third p joint can move -180 to 180
6  q5.set = [ 0 90 180 30 ]'*pi/180 ; % claw can spin full 360 degrees
7
8  % Robot hits straight arm singularity only if q2 = q4 = 90 and q3 = 0,
9  % as d1 link must always point straight up.
10
11 for i = 1:length(q2.set)
12     if q2.set(i) == 90 && q4.set(i) == 90 && q3.set == 0
13         q2.set(i) = 95
14         q4.set(i) = 85
15         q3.set(i) = 5
16     end
17 end

```

FIG. 14. Example code to prevent full extension (boundary) singularities in the LynxMotion robot arm.

A second singularity type is the internal singularity, where joint axes of the robot line up perfectly, meaning the links of the robot attempt to occupy the same spatial position. This can cause serious damage to the robot and danger to humans operating it. For this a more robust solution is needed. Singularities occur when the determinant of the Jacobian matrix is equal to zero, $\det(J(q)) = 0$, meaning it's inverse does not exist (Zhao et al. 2021). This is because the Jacobian matrix converts the joint angular velocity (\dot{q}) to end effector velocity (v_e), and singularities occur when infinitesimal changes in the end effector velocity cause large changes in joint velocity,

$$v_e = J(q)\dot{q}$$

These singularities can be avoided by checking the planned route of the robot against the Jacobian first, and throwing an error before the code is executed if at any point $\text{Det}(J) = 0$, an example is given in FIG. 15. The code in FIG.s 14 & 15 are developed from the Appendix A simulation.

```

1      % Check each end effector point to ensure it is not at a singularity. Throw an error if it is.
2      % End effector position and orientation is defined by T_e transformation matrix.
3
4      % Use matlab's native jacobian computation.
5      Jacobian_of_arm = jacobian([T_e], [q1,q2,q3,q4,q5])
6      % Calculate the general determinant
7      Det_J = det(Jacobian_of_arm)
8
9      % check each value of T_e.specified to see if Det(Jacobian_of_arm) == 0
10     for i = 1:length(q1_set(i))
11         Det_J.Specified = subs(yt_element,[d1 l1 l2 l3 q1 q2 q3 q4 q5], [d1_set l1_set l2_set l3_set ...
12             q1_set(i) q2_set(i) q3_set(i) q4_set(i) q5_set(i)])
13
14         % If it is, stop the simulation and throw an error message.
15         if Det_J.Specified == 0
16             error('One of your robot orientation angle sets (position %s) results in a singularity! ...
17                 Please change it and try again.', i)
18         end
19     end

```

FIG. 15. Example code to prevent both internal and boundary singularities in the LynxMotion robot arm.

VII. CONCLUSIONS & FURTHER WORK

The simulations created for this project are each successful and written as general solutions. This allows the input parameters to be changed to simulate all possible solutions for each input being provided.

Improvements could be made in efficiency and robustness. Addressing the former, Appendix F is over 800 lines long as both the inverse and forward kinematics are run twice. One could simplify this code by converting the forward and inverse kinematics sections to a function which need be written only once.

Regarding robustness, the linear motion simulation is successful for a wide range of possible inputs, but fails at certain edge cases. The straight line equation is derived as $z = mr + c$. m and c are gradient and z -axis intercept, $r = \sqrt{x^2 + y^2}$ is the magnitude of movement in the horizontal plane. This method is limited by the fact that it cannot distinguish direction in the horizontal plane. This limitation could be removed by checking the sign of the change in x and y coordinates and using conditional statements to discern the direction. This would be added in Appendix D, in the else statement at line 433.

The parallel robot simulations are both efficient and accurate, making these the most production ready in their current state. Overall, the set of simulations created for this project provide useful and accurate insight into the operation of the LynxMotion arm and parallel surgery robot. They allow for testing of new operations with zero risk to real hardware and demonstrate all possible solutions for a user to achieve a given robot configuration; a valuable tool for engineers working with robots of this type.

Appendix A: Serial Robot Forward Kinematics

Code available at: https://github.com/Hinlo/MatLab-Simulation-of-Serial-and-Parallel-Robots/blob/main/FINN_FK_LynxMotionArm.m
FINN_FK_LynxMotionArm.m

The general solution of the end effector forward kinematics transformation matrix (T_e) is given here. It is a 4×4 matrix where each row is contained within a set of square brackets and separated by a line break. Each column is separated by a comma and a new line. The full file prints this result.

```

1  T_e =
2
3  [sin(q1)*sin(q5) + cos(q5)*(cos(q4)*(cos(q1)*cos(q2)*cos(q3) -
4  cos(q1)*sin(q2)*sin(q3)) - sin(q4)*(cos(q1)*cos(q2)*sin(q3) +
5  cos(q1)*cos(q3)*sin(q2))),
6  cos(q5)*sin(q1) - sin(q5)*(cos(q4)*
7  (cos(q1)*cos(q2)*cos(q3) - cos(q1)*sin(q2)*sin(q3)) - sin(q4)*
8  (cos(q1)*cos(q2)*sin(q3) + cos(q1)*cos(q3)*sin(q2))),
9  cos(q4)*(cos(q1)*cos(q2)*sin(q3) + cos(q1)*cos(q3)*sin(q2)) + sin(q4)*
10 (cos(q1)*cos(q2)*cos(q3) - cos(q1)*sin(q2)*sin(q3)),
11 l3*(cos(q4)*(cos(q1)*cos(q2)*sin(q3) + cos(q1)*cos(q3)*sin(q2)) + sin(q4)*
12 (cos(q1)*cos(q2)*cos(q3) - cos(q1)*sin(q2)*sin(q3))) + l1*cos(q1)*cos(q2) +
13 l2*cos(q1)*cos(q2)*cos(q3) - l2*cos(q1)*sin(q2)*sin(q3)]
14
15 [- cos(q1)*sin(q5) - cos(q5)*(cos(q4)*(sin(q1)*sin(q2)*sin(q3) -
16 cos(q2)*cos(q3)*sin(q1)) + sin(q4)*(cos(q2)*sin(q1)*sin(q3) +
17 cos(q3)*sin(q1)*sin(q2))),
18 sin(q5)*(cos(q4)*(sin(q1)*sin(q2)*sin(q3) -
19 cos(q2)*cos(q3)*sin(q1)) + sin(q4)*(cos(q2)*sin(q1)*sin(q3) +
20 cos(q3)*sin(q1)*sin(q2))) - cos(q1)*cos(q5),
21 cos(q4)*(cos(q2)*sin(q1)*sin(q3) + cos(q3)*sin(q1)*sin(q2)) - sin(q4)*
22 (sin(q1)*sin(q2)*sin(q3) - cos(q2)*cos(q3)*sin(q1)),
23 l3*(cos(q4)*(cos(q2)*sin(q1)*sin(q3) + cos(q3)*sin(q1)*sin(q2)) - sin(q4)*
24 (sin(q1)*sin(q2)*sin(q3) - cos(q2)*cos(q3)*sin(q1))) + l1*cos(q2)*sin(q1) +
25 l2*cos(q2)*cos(q3)*sin(q1) - l2*sin(q1)*sin(q2)*sin(q3)]
26
27 [cos(q5)*(cos(q4)*(cos(q2)*sin(q3) + cos(q3)*sin(q2)) + sin(q4)*(cos(q2)*cos(q3) - sin(q2)*sin(q3))),
28 -sin(q5)*(cos(q4)*(cos(q2)*sin(q3) + cos(q3)*sin(q2)) + sin(q4)*(cos(q2)*cos(q3) - sin(q2)*sin(q3))),
29 sin(q4)*(cos(q2)*sin(q3) + cos(q3)*sin(q2)) - cos(q4)*(cos(q2)*cos(q3) - sin(q2)*sin(q3)),
30 d1 + l1*sin(q2) - l3*(cos(q4)*(cos(q2)*cos(q3) - sin(q2)*sin(q3)) - sin(q4)*
31 (cos(q2)*sin(q3) + cos(q3)*sin(q2))) + l2*cos(q2)*sin(q3) + l2*cos(q3)*sin(q2)]
32
33 [ 0, 0, 0, 1 ]

```

The full forward kinematics file is given here:

```

1  %% lynxmotion arm has 5 dof!
2  % NOTE: Defined angles start from x = 0, rotating +ve or -ve direction, about the z
3  % axis always!!
4  %ok<*NOPTS> do not pester me about semi colons
5  %ok<*SAGROW> do not pester me about lists
6  % click line numbers to enforce a pause at that line during run.
7  % remember, cos(x) is radians, cosd(x) would be degrees.
8  % remember, arrays count from 1 not 0 in matlab.
9  % clear the output
10 clear all %ok<*CLALL>
11 close all
12 clc
13
14 %% DH TABLE
15 % a    alpha    d    theta
16 % 0    90       d1   q1
17 % l1   0        0    q2
18 % l2   0        0    q3
19 % 0    90       0    q4

```



```

20 % 0      0      13  q5
21 %% Set our variables
22 syms d1 l1 l2 l3; % length variables
23 syms q1 q2 q3 q4 q5; % angle variables
24 %% Show our 5 transformation matrices
25 T_01 = [
26     cos(q1), 0, sin(q1), 0]
27     sin(q1), 0, -cos(q1), 0]
28     [      0, 1,      0, d1]
29     [      0, 0,      0, 1]];
30
31 T_12 = [
32     cos(q2), -sin(q2), 0, l1*cos(q2)]
33     sin(q2), cos(q2), 0, l1*sin(q2)]
34     [      0,      0, 1,      0]
35     [      0,      0, 0,      1]];
36
37 T_23 = [
38     cos(q3), -sin(q3), 0, l2*cos(q3)]
39     sin(q3), cos(q3), 0, l2*sin(q3)]
40     [      0,      0, 1,      0]
41     [      0,      0, 0,      1]];
42
43 T_34 = [
44     cos(q4), 0, sin(q4),      0]
45     sin(q4), 0, -cos(q4),      0]
46     [      0, 1,      0,      0]
47     [      0, 0,      0,      1]];
48
49 T_45 = [
50     cos(q5), -sin(q5), 0, 0]
51     sin(q5), cos(q5), 0, 0]
52     [      0,      0, 1, 13]
53     [      0,      0, 0, 1]];
54
55 %% Multiply them in the required order (right to left)
56 % You can show or comment out the components.
57
58 % uncomment to see step by step multiplication of T matrices.
59 % T_34*T_45
60 % T_23*T_34*T_45;
61 % T_12*T_23*T_34*T_45;
62 T_e = T_01*T_12*T_23*T_34*T_45;
63
64 % Show our final transformation matrix
65 T_e
66
67 % now get a specified result, for example:
68
69 T_e.Specified = subs(T_e, [q1, q2, q3, q4, q5, l1, l2, l3, d1], ...
70     [0*(pi/180), 90*(pi/180), 0*(pi/180), 0*(pi/180), 0*(pi/180), 1, 1, 1, 1])
71
72 %% The following code simulates the forward kinematics of a the lynxmotion
73 % arm using my forward kinematics calculated above.
74 % Figure 2 shows its movement from start to end
75 % position, Figure 1 shows the location of its end effector at points of
76 % its trajectory and Figure 3 shows the maximum potential workspace of the
77 % arm's end effector.
78
79 disp('The following code simulates the forward kinematics of a simple 2DOF')
80 disp('serial manipulator. Figure 2 shows ts movement from start to end position,')
81 disp('Figure 1 shows the location of its end effector at points of its trajectory')
82 disp('and Figure 3 shows the maximum potential workspace of its end effector')
83
84 %% A series of joint angles
85 % The following variables are defined in the form of column-vectors with
86 % 4 rows each. Each row represents a different position (angle) of the joint.
87 % e.g. inititally we hae 60 degrees for q1 and -30 for q2.
88 % USING .set TO MARK WHEN WE HAVE DEFINED OUR SYMBOLIC VARIABLES.
89 q1.set = [ 0 20 40 45 ]'*pi/180 ; % base can spin full 360 degrees

```

```

89 q2_set = [ 0 30 60 90]*pi/180 ; % first p joint can move 0 to 180
90 q3_set = [ 0 10 20 30 ]*pi/180 ; % second p joint can move -180 to 180
91 q4_set = [ 0 -10 -20 -30]*pi/180 ; % third p joint can move -180 to 180
92 q5_set = [ 0 90 180 30 ]*pi/180 ; % claw can spin full 360 degrees
93
94
95 %% Links Lengths
96 dl_set = 0.1 ; % example lengths, start with all as 10cm
97 l1_set = 0.1 ;
98 l2_set = 0.1 ;
99 l3_set = 0.1 ;
100
101 %% Trigonometric abbreviations
102 % not required at the moment but see original uwe script if you want to add
103 % them.
104
105
106 %% Tip position
107 % These equations are taken from our general T_e matrix above.
108
109 % CHECK: I think these are going to be the x,y,z translation values from
110 % T_e and ignore the rot values for orientation of the end effector.
111 xt_element = T_e(1,4); % the x transformation is the 4th element of the 1st row of our general ...
    transformation matrix, T_e.
112 yt_element = T_e(2,4); % the y transformation is the 4th element of the 2nd row of our general ...
    transformation matrix, T_e.
113 zt_element = T_e(3,4); % the z transformation is the 4th element of the 3rd row of our general ...
    transformation matrix, T_e.
114
115 % to add the angles and lengths, we will use a quick loop.
116 % lengths are just one value, but angles have an array, so we'll loop
117 % through the angle arrays.
118 xt = [];
119 for i = 1:length(q1_set)
120     xt_value = subs(xt_element,[d1 l1 l2 l3 q1 q2 q3 q4 q5], [d1_set l1_set l2_set l3_set ...
        q1_set(i) q2_set(i) q3_set(i) q4_set(i) q5_set(i)]); % substitute in the set values.
121     xt(i,1) = xt_value ;
122 end
123
124
125
126 % repeat for y
127 yt = [];
128 for i = 1:length(q1_set)
129     yt_value = subs(yt_element,[d1 l1 l2 l3 q1 q2 q3 q4 q5], [d1_set l1_set l2_set l3_set ...
        q1_set(i) q2_set(i) q3_set(i) q4_set(i) q5_set(i)]); % substitute in the set values.
130     yt(i,1) = yt_value ;
131 end
132
133
134
135 % repeat for z
136 zt = [];
137 for i = 1:length(q1_set)
138     zt_value = subs(zt_element,[d1 l1 l2 l3 q1 q2 q3 q4 q5], [d1_set l1_set l2_set l3_set ...
        q1_set(i) q2_set(i) q3_set(i) q4_set(i) q5_set(i)]); % substitute in the set values.
139     zt(i,1) = zt_value ;
140 end
141
142 % pt is a 4 by 3 double as required for the formatting below.
143 pt = [ xt yt zt ] ;
144
145 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
146 %% Plot the trajectory of the end-effector
147 figure (1)
148 %line below just dictates where the image appears on the screen in matlab
149 %desktop
150 set(1,'position',[680 558 560 420])
151
152

```

```

153
154 % IN 3D...
155 plot3(pt(1,1), pt(1,2), pt(1,3), 'ro') % plot the first position of the robot's end effector
156 hold on
157 plot3(pt(2:4,1),pt(2:4,2),pt(2:4,3), 'o') % plot the 3 following positions of the robot's ...
    end effector
158 title('Tip Trajectory') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');
159
160
161 %% Plot the robotic arm, in 4 different positions
162 % this currently just moves q1, the base. Do we want to rotate more than
163 % one joint? We defo don't want to rotate just the base.
164 figure (2)
165 %line below just dictates where the image appears on the screen in matlab
166 %desktop
167 set(2,'position',[116 190 560 420])
168
169 %base is at origin
170 base = 0 ;
171
172
173 % next joint given by T01 translations (movement from base to joint 1)
174 % need to make into a 4 by 3 array.
175 Tj1_x = zeros(4,1) ; % x translation is zero
176 Tj1_y = zeros(4,1) ; % y translation is zero
177 Tj1_z = subs(zeros(4,1), 0, d1_set); % z translation is d1
178
179 % next joint given by T01T12 translations - movement from base to joint 2
180 % make our general elements for each array
181 movement = T_01*T_12 ;
182 xt_element = movement(1,4) ;
183 yt_element = movement(2,4) ;
184 zt_element = movement(3,4) ;
185 % fill the joint 2 4x1 arrays
186 Tj2_x = [] ;
187 Tj2_y = [] ;
188 Tj2_z = [] ;
189 for i = 1:length(q1_set)
190     xt_value = subs(xt_element, [l1 d1 q1 q2], [l1_set d1_set q1_set(i) q2_set(i)]) ;
191     yt_value = subs(yt_element, [l1 d1 q1 q2], [l1_set d1_set q1_set(i) q2_set(i)]) ;
192     zt_value = subs(zt_element, [l1 d1 q1 q2], [l1_set d1_set q1_set(i) q2_set(i)]) ;
193     Tj2_x(i,1) = xt_value ;
194     Tj2_y(i,1) = yt_value ;
195     Tj2_z(i,1) = zt_value ;
196 end
197 % above worked, now continue for the rest.
198
199
200 % next joint given by T01T12T23 translations - movement from base to joint 3
201 % make our general elements for each array
202 movement = T_01*T_12*T_23 ;
203 xt_element = movement(1,4) ;
204 yt_element = movement(2,4) ;
205 zt_element = movement(3,4) ;
206 % fill the joint 2 4x1 arrays
207 Tj3_x = [] ;
208 Tj3_y = [] ;
209 Tj3_z = [] ;
210 for i = 1:length(q1_set)
211     xt_value = subs(xt_element,[d1 l1 l2 q1 q2 q3], [d1_set l1_set l2_set q1_set(i) q2_set(i) ...
        q3_set(i)]) ;
212     yt_value = subs(yt_element,[d1 l1 l2 q1 q2 q3], [d1_set l1_set l2_set q1_set(i) q2_set(i) ...
        q3_set(i)]) ;
213     zt_value = subs(zt_element,[d1 l1 l2 q1 q2 q3], [d1_set l1_set l2_set q1_set(i) q2_set(i) ...
        q3_set(i)]) ;
214     Tj3_x(i,1) = xt_value ;
215     Tj3_y(i,1) = yt_value ;
216     Tj3_z(i,1) = zt_value ;
217 end
218

```

```

219
220 % next joint given by T_01T_12T_23T_34 translations - movement from base to joint 4
221 % note there's no translation in this joint so no change from above.
222 % make our general elements for each array
223 movement = T_01*T_12*T_23*T_34 ;
224 xt_element = movement(1,4) ;
225 yt_element = movement(2,4) ;
226 zt_element = movement(3,4) ;
227 % fill the joint 2 4x1 arrays
228 Tj4_x = [] ;
229 Tj4_y = [] ;
230 Tj4_z = [] ;
231 for i = 1:length(q1_set)
232     xt_value = subs(xt_element,[d1 l1 l2 l3 q1 q2 q3 q4], [d1_set l1_set l2_set l3_set q1_set(i) ...
        q2_set(i) q3_set(i) q4_set(i)]) ;
233     yt_value = subs(yt_element,[d1 l1 l2 l3 q1 q2 q3 q4], [d1_set l1_set l2_set l3_set q1_set(i) ...
        q2_set(i) q3_set(i) q4_set(i)]) ;
234     zt_value = subs(zt_element,[d1 l1 l2 l3 q1 q2 q3 q4], [d1_set l1_set l2_set l3_set q1_set(i) ...
        q2_set(i) q3_set(i) q4_set(i)]) ;
235     Tj4_x(i,1) = xt_value ;
236     Tj4_y(i,1) = yt_value ;
237     Tj4_z(i,1) = zt_value ;
238 end
239
240 % We already calculated our end effector positions - that's the pt array.
241
242
243
244
245
246 % generate the graph. Note: only 4 joints will be visible as there is
247 % no spatial distinction between joints three and 4.
248 for i = 1:4 % zeros is the base position, doesn't change.
249     xx = [base; Tj1_x(i); Tj2_x(i); Tj3_x(i); Tj4_x(i); pt(i,1) ] ;
250     yy = [base; Tj1_y(i); Tj2_y(i); Tj3_y(i); Tj4_y(i); pt(i,2) ] ;
251     zz = [base; Tj1_z(i); Tj2_z(i); Tj3_z(i); Tj4_z(i); pt(i,3) ] ;
252
253     pause(1) % pause long enough that we see the first position
254     plot3(xx,yy,zz,'ko-','Linewidth',2)
255     axis equal
256     hold on
257
258     % label axes, start point, end point.
259     xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)')
260     text(pt(1,1),pt(1,2),pt(1,3), 'x') ; text(pt(1,1) + 0.002,pt(1,2) + 0.002,pt(1,3) + ...
        0.002,'ptStart') ;
261     text(pt(4,1),pt(4,2),pt(4,3), 'x') ; text(pt(4,1) + 0.002,pt(4,2) + 0.002,pt(4,3) + ...
        0.002,'ptEnd') ;
262     axis([ -0.1 0.4 -0.1 0.4 0 0.4 ])
263     hold off % CHANGE TO HOLD ON TO SEE ALL LINES AT ONCE.
264     pause(2)
265 end
266
267 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
268 %% Workspace - once working, set to finer increments to get workspace plot.
269
270 % Here we must extend the range of the set angles to encompass all possible
271 % angles.
272
273 % SETTING ARRAYS: start:interval:end. x = 1:2:7 -> x = [1,3,5,7]
274 % to be able to plot, keep intervals constant.
275 q1_set = (0:30:360)*pi/180 ; % base angle can go 0 to 360, using intervals of 5 atm.
276 q2_set = (-90:30:90)*pi/180 ; % joint has range -90 to 90
277 q3_set = (-180:30:180)*pi/180 ; % joint has range -180 to 180
278 q4_set = (-180:30:180)*pi/180 ; % joint has range -180 to 180
279 q5_set = 0; % rot of end effector has range 0 to 360 but this doesn't
280 % affect workspace.
281 % we don't care about q5 as workspace is about how far the robot
282 % can reach, the final orientation of the end effector doesn't matter
283 % therefore, set q5 as a constant.

```

```

284
285 %% Angles Full Range of motion
286 % Not realistic but worth knowing
287
288 % q1_set = 0:30:360 ; % base angle can go 0 to 360, using intervals of 5 atm.
289 % q2_set = -90:30:90 ; % joint has range -90 to 90
290 % q3_set = -180:30:180 ; % joint has range -180 to 180
291 % q4_set = -180:30:180 ; % joint has range -180 to 180
292 % q5_set = 0; % rot of end effector has range 0 to 360 but this doesn't
293 %% Plot the workspace of the robot
294 figure (3)
295 %line below just dictates where the image appears on the screen in matlab
296 %desktop
297 set(3,'position',[1243 190 560 420])
298
299
300 for i = 1:length(q1_set) % for q1
301     for j = 1:length(q2_set) % for q2
302         for k = 1:length(q3_set) % for q3
303             for l = 1:length(q4_set) % for q4
304                 % define our x y z values for the given q 1-4 values
305                 xwork = subs(xt_element,[d1 l1 l2 l3 q1 q2 q3 q4 q5], [d1_set l1_set l2_set l3_set ...
306                     q1_set(1,i) q2_set(1,j) q3_set(1,k) q4_set(1,l) q5_set]); % substitute in the ...
307                     set values.
308                 ywork = subs(yt_element,[d1 l1 l2 l3 q1 q2 q3 q4 q5], [d1_set l1_set l2_set l3_set ...
309                     q1_set(1,i) q2_set(1,j) q3_set(1,k) q4_set(1,l) q5_set]); % we still want our ...
310                     end effector positions
311                 zwork = subs(zt_element,[d1 l1 l2 l3 q1 q2 q3 q4 q5], [d1_set l1_set l2_set l3_set ...
312                     q1_set(1,i) q2_set(1,j) q3_set(1,k) q4_set(1,l) q5_set]); % given by subbing ...
313                     into the element eqns.
314                 % plot this point!
315                 if zwork < 0 || abs( q2_set(1,j) + q3_set(1,k) + q4_set(1,l) ) > 2*pi % skip ...
316                     plotting this point if the z value is less than zero, e.g arm goes into the ...
317                     table. or if psi greater than 360
318                     continue
319                 end
320                 %
321                 disp("position set: " + num2str(i) + ", point : " + num2str(j) + num2str(k) + ...
322                     num2str(l) )
323                 %
324                 disp("x: " + double(xwork) + ", y: " +double(ywork) + ", z: " + double(zwork) + ...
325                     ", psi: " + round((q2_set(1,j) + q3_set(1,k) + q4_set(1,l)),2,"decimals") + ", mu: " + ...
326                     q5_set*180/pi)
327                 plot3(xwork,ywork,zwork,'x');
328                 hold on
329             end
330         end
331     end
332 end
333
334 % this can take some time, so display current percentage
335 % done while you wait.
336 percent_done = (i*j*k*l)/(length(q1_set)*length(q2_set)*length(q3_set)*length(q4_set))*100
337
338 end
339
340 title('Workspace') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');

```

Appendix B: Serial Robot Inverse Kinematics

Code available at: https://github.com/Hinlo/MatLab-Simulation-of-Serial-and-Parallel-Robots/blob/main/FINN_IK_LynxMotionArm.m

FINN_IK_LynxMotionArm.m

```

1  %% Inverse Kinematics Calculator for LynxMotion Arm
2  % key is to maintain list order across lists of each angle.
3  clear all %#ok<*CLALL> %#ok<*SAGROW>
4  close all
5  clc
6
7  %% Links Length
8  d1 = 0.1 ;
9  l1 = 0.1 ;
10 l2 = 0.1 ;
11 l3 = 0.1 ;
12
13 %% Desired position of end-effector - SET ME
14
15 % Cartesian Coords for x,y,z values.
16 px = 0.17;
17 py = 0.17 ;
18 pz = 0.17;
19
20 % mu and psi angles for orientation
21 mu = 0; % mu is rotation of end effector relative to "wrist"
22
23 % list of all possible psi values. they will be narrowed down later to only
24 % real values. technically could go negative angle but it just ends up
25 % repeating. e.g +90 is same as -270 degrees logically.
26 psi = 0:pi/6:2*pi; % RADIANS
27 psi = psi - pi/2; % angular correction to line up with zero point of FK.
28
29 % psi is angle between end effector and negative-z-axis, based on FK.
30 % psi = q2 + q3 + q4
31 % End effector desired position and orientation given by array EndEffector
32 End_Effector = [ px py pz ]' ;
33 % disp('Desired position =')
34 % disp("x,y,z")
35 % disp(End_Effector)
36 % disp("mu")
37 % disp(mu)
38 if norm(End_Effector) > l1+l2+l3+d1 % workspace is above ground
39     error('desired position is out of the workspace')
40 end
41
42
43 %% %%%%%%%%%%%%% Inverse Kinematics of LynxMotion Arm %%%%%%%%%%%%%
44 % For details on calculations, view notes in ReadMe.
45 % We must find values for the 5 joint angles below.
46 sym q1 ; % single value
47 q2 = [] ; % potentially many values
48 q3 = [] ; % potentially many values
49 q4 = [] ; % potentially many values
50 sym q5 ; % single value
51
52 %% Simple angles to find:
53
54 % Find q1
55 q1 = atan2(py,px) ;
56 % Find q5
57 q5 = mu ;
58
59 %% Define extra variables to help us find q2,3,4

```

```

60
61 % define (r,z) plane
62 r = sqrt(px^2 + py^2) ; % r is hypotenues in x-y plane.
63
64 % For each value of psi, there is a value of rw, zw and D
65 r_w = (1:length(psi)) ;
66 z_w = (1:length(psi)) ;
67 D = (1:length(psi)) ;
68 for i = 1:length(psi)
69 % define position of wrist in (r,z) plane
70     r_w(i) = r - l3*cos(psi(i)) ; %r_w and z_w will be real as cos or sin of a real number is a ...
        real number
71
72     z_w(i) = pz - d1 -l3*sin(psi(i)) ;
73
74     % define D - a placeholder variable for a large combination we derived in
75     % notes
76
77     D(i) = - (r_w(i)^2 + z_w(i)^2 - l1^2 - l2^2) / (2*l1*l2) ;
78     D(i) = round(D(i), 7) ; % rounding D avoids fake imaginary numbers due to rounding
79     % errors.
80 end
81 %% MAKE LISTS FOR EACH ANGLE AND ENSURE ALL THE SAME LENGTH. IF ONE OF THE ANGLES IN A SET OF 5 IS ...
    IMAGINARY, REMOVE THAT INDEX FOR EVERY ANGLE. SHOULD START WITH MANY BUT REDUCE TO ONLY A FEW.
82 %% get only real values
83
84 psi_real = [] ;
85 r_w_real = [] ;
86 z_w_real = [] ;
87 D_real = [] ;
88 for i = 1:length(psi)
89     if imag(sqrt(1-D(i)^2)) == 0 % for real values, append to real lists.
90         psi_real(end + 1) = round(psi(i),7) ; % round each list to get zeros rather than e^-10 or ...
            something.
91         r_w_real(end + 1) = round(r_w(i),7) ;
92         z_w_real(end + 1) = round(z_w(i),7) ;
93         D_real(end + 1) = round(D(i),7) ;
94     end
95 end
96
97 % we can see these four lists MUST have same length.
98 % disp(psi_real)
99 % disp(D_real)
100 % disp(z_w_real)
101 % disp(r_w_real)
102
103 %% Find q3 possibilities
104 % there are two possibilities for each value of D depending on
105 for i = 1:length(D_real)
106
107     q3(end+1) = atan2( sqrt(1-D_real(i)^2), -D_real(i)) ;
108     q3(end+1) = atan2(-sqrt(1-D_real(i)^2), -D_real(i)) ;
109 end
110
111 % disp("q3")
112 % disp(q3)
113 %% Explaining list order.
114 % matlab indexes from 1 not 0.
115 % if D has 3 values, we can see q3 will have 6, where
116 % D(1) corresponds to q3(1 & 2)
117 % D(2) corresponds to q3(3 & 4)
118 % D(3) corresponds to q3(5 & 6)
119 % we can see D(i) corresponds to values
120 % q3(2*i -1) and q3(2*i)
121 % we need to ensure this is *consistant* across all q lists below.
122 %% Find q2
123
124 % two options based on q3
125 for i = 1:length(D_real)
126     % for each r_w,z_w value, find first q2 value using first q3 value

```

```

127     q2(end+1) = atan2(z_w_real(i),r_w_real(i)) + atan2(l2*sin(q3(2*i)), l1+l2*cos(q3(2*i))) ;
128     % add +1 to
129     q2(end+1) = atan2(z_w_real(i),r_w_real(i)) + atan2(l2*sin(q3(2*i-1)), l1+l2*cos(q3(2*i-1))) ;
130 end
131 % disp("q2")
132 % disp(q2)
133 %% Find q4
134
135 % two options based on two sets of q2,q3
136 % use psi = q2 + q3 + q4
137 % +pi/2 is the geometry correction for zero point of q4
138 for i = 1:length(D_real) % round to remove zero discrepancies.
139     q4(end+1) = psi_real(i) - q2(2*i -1) - q3(2*i -1) +pi/2 ;
140     q4(end+1) = psi_real(i) - q2(2*i) - q3(2*i) +pi/2 ;
141 end
142
143
144 % disp("q4")
145 % disp(q4)
146 %% Remove duplicate values
147 % make a matrix out of the lists where one row is one solution
148
149 % we know matrix will have initial size: no. of items in q2, 3 or 4 x 5
150 % angles
151 SolutionMatrix = zeros(length(q2), 5);
152 for i = 1:length(q2)
153     SolutionMatrix(i,1) = q1 ;
154     SolutionMatrix(i,2) = q2(i) ;
155     SolutionMatrix(i,3) = q3(i) ;
156     SolutionMatrix(i,4) = q4(i) ;
157     SolutionMatrix(i,5) = q5 ;
158 end
159 % disp(SolutionMatrix)
160 % remove duplicate rows
161 Unique_Solutions = unique(SolutionMatrix,"rows","stable");
162 % stable prevents order being changed.
163
164
165 %% we also know, q4 can't be more than 360 degrees. filter these out.
166 valid_Solutions = [] ;
167 for i = 1:size(Unique_Solutions,1)
168     if abs(Unique_Solutions(i,4)) < 2*pi
169         valid_Solutions = [valid_Solutions; Unique_Solutions(i,:)] ;
170     end
171 end
172 %% Display Solutions
173
174 for i = 1:size(valid_Solutions,1) % for no. of rows aka no. of solutions
175
176     disp("solution " + num2str(i) + ": ")
177     disp(round(valid_Solutions(i,:)*180/pi,2,"decimals"))
178
179 end

```


Appendix C: Serial Robot Task Completion

Code available at: https://github.com/Hinlo/MatLab-Simulation-of-Serial-and-Parallel-Robots/blob/main/FINN_TASK_LynxMotionArm.m

FINN_TASK_LynxMotionArm.m

```

1  %% LynxMotion Arm Task
2  % This file uses inverse and forward kinematics to plan and show a task for
3  % the lynxmotion arm in matlab.
4
5  % the chosen task is to draw the vertices of a cuboid in 3d space.
6
7  % ***Task Steps***
8  % 1. define x,y,z positions for 5 or more points that would make a cuboid
9  % in 3d space.
10 % 2. use IK to find the q1-5 angle values for the arm at a each point.
11 % 3. use FK to plot the arm trajectory in 3d space and save the end
12 % effector points to display the smiley face.
13 clear all %#ok<*CLALL> %#ok<*SAGROW>
14 close all
15 clc
16
17 %% 1. define x,y,z positions that make a smiley face in 3d space.
18
19
20 % TASK: plot a cuboid in 3D space.
21 points.to_plot = [[0.0 0.0 0.2]; [0.0 0.0 0.1]; [0.15 0.0 0.2]; [0.15 0.0 0.1]; [0.15 0.15 0.1]; ...
22                 [0.15 0.15 0.2]; [0.0 0.15 0.2]; [0.0 0.15 0.1]] ;
23 % check the plotted points.
24 figure (1)
25 for i = 1:size(points.to_plot,1)
26     plot3(points.to_plot(i,1), points.to_plot(i,2), points.to_plot(i,3), 'ko-', 'Linewidth',2) % ...
27     % plot the first position of the robot's end effector
28     hold on
29 end
30 axis([ 0 0.4 0 0.4 0 0.4 ])
31 title('Desired Points') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');
32
33
34 %% 2. use IK to find the q1-5 angle values for the arm at a each point.
35 % Inverse Kinematics Calculator for LynxMotion Arm
36 % key is to maintain list order across lists of each angle.
37 % Links Length
38 d1 = 0.1 ;
39 l1 = 0.1 ;
40 l2 = 0.1 ;
41 l3 = 0.1 ;
42
43 % Run through the list of point
44 IK.OUTPUT = zeros(size(points.to_plot,1),5); % holding array for output q angles.
45 for j = 1:size(points.to_plot,1)
46     % Cartesian Coords for x,y,z values.
47     px = points.to_plot(j,1) ;
48     py = points.to_plot(j,2) ;
49     pz = points.to_plot(j,3) ;
50
51 % mu and psi angles for orientation
52 mu = 0; % mu is rotation of end effector relative to "wrist"
53
54 % list of all possible psi values. they will be narrowed down later to only
55 % real values. technically could go negative angle but it just ends up
56 % repeating. e.g +90 is same as -270 degrees logically.
57 psi = (0:pi/20:2*pi) ;

```

```

58 for i = 1: length(psi)
59     psi(i) = psi(i) - pi/2;
60 end
61
62
63 % psi is angle between end effector and negative-z-axis, based on FK.
64 % psi = q2 + q3 + q4
65 % End effector desired position and orientation given by array EndEffector
66 End_Effector = [ px py pz ]' ;
67 % disp('Desired position =')
68 % disp("x,y,z")
69 % disp(End_Effector)
70 % disp("mu")
71 % disp(mu)
72 if norm(End_Effector) > l1+l2+l3+d1 % workspace is above ground
73     error('desired position is out of the workspace')
74 end
75
76
77 %% %%%%%%%%% Inverse Kinematics of LynxMotion Arm %%%%%%%%%
78 % For details on calculations, view notes in ReadMe.
79 % We must find values for the 5 joint angles below.
80 sym q1 ; % single value
81 q2 = [] ; % potentially many values
82 q3 = [] ; % potentially many values
83 q4 = [] ; % potentially many values
84 sym q5 ; % single value
85
86 %% Simple angles to find:
87
88 % Find q1
89 q1 = atan2(py,px) ;
90 % Find q5
91 q5 = mu ;
92
93 %% Define extra variables to help us find q2,3,4
94
95 % define (r,z) plane
96 r = sqrt(px^2 + py^2) ; % r is hypotenues in x-y plane.
97
98 % For each value of psi, there is a value of rw, zw and D
99 r_w = (1:length(psi)) ;
100 z_w = (1:length(psi)) ;
101 D = (1:length(psi)) ;
102 for i = 1:length(psi)
103     % define position of wrist in (r,z) plane
104     r_w(i) = r - l3*cos(psi(i)) ; %r_w and z_w will be real as cos or sin of a real number is a ...
        real number
105
106     z_w(i) = pz - d1 -l3*sin(psi(i)) ;
107
108     % define D - a placeholder variable for a large combination we derived in
109     % notes
110
111     D(i) = - (r_w(i)^2 + z_w(i)^2 - l1^2 - l2^2) / (2*l1*l2) ;
112     D(i) = round(D(i), 7) ;% rounding D avoids fake imaginary numbers due to rounding
113     % errors.
114 end
115 %% get only real values
116
117 psi_real = [] ;
118 r_w_real = [] ;
119 z_w_real = [] ;
120 D_real = [] ;
121 for i = 1:length(psi)
122     if imag(sqrt(1-D(i)^2)) == 0 % for real values, append to real lists.
123         psi_real(end + 1) = psi(i);%round(psi(i),7) ; % round each list to get zeros rather than ...
            e^-10 or something.
124         r_w_real(end + 1) = r_w(i);%round(r_w(i),7) ;
125         z_w_real(end + 1) = z_w(i);%round(z_w(i),7) ;

```

```

126         D_real(end + 1) = D(i); %round(D(i),7) ;
127     end
128 end
129
130 % we can see these four lists MUST have same length.
131 % disp(psi_real)
132 % disp(D_real)
133 % disp(z_w_real)
134 % disp(r_w_real)
135
136 %% Find q3 possibilities
137 % there are two possibilities for each value of D depending on
138 for i = 1:length(D_real)
139
140     q3(end+1) = atan2( sqrt(1-D_real(i)^2), -D_real(i)) ;
141     q3(end+1) = atan2(-sqrt(1-D_real(i)^2), -D_real(i)) ;
142 end
143
144 % disp("q3")
145 % disp(q3)
146 %% Explaining list order.
147 % matlab indexes from 1 not 0.
148 % if D has 3 values, we can see q3 will have 6, where
149 % D(1) corresponds to q3(1 & 2)
150 % D(2) corresponds to q3(3 & 4)
151 % D(3) corresponds to q3(5 & 6)
152 % we can see D(i) corresponds to values
153 % q3(2*i -1) and q3(2*i)
154 % we need to ensure this is *consistant* across all q lists below.
155 %% Find q2
156
157 % two options based on q3
158 for i = 1:length(D_real)
159     % for each r_w,z_w value, find first q2 value using first q3 value.
160     % Switching order here corrects angles... why?
161     q2(end+1) = atan2(z_w_real(i),r_w_real(i)) + atan2(l2*sin(q3(2*i)), l1+l2*cos(q3(2*i))) ;
162     % add +1 to
163     q2(end+1) = atan2(z_w_real(i),r_w_real(i)) + atan2(l2*sin(q3(2*i-1)), l1+l2*cos(q3(2*i-1))) ;
164 end
165 % disp("q2")
166 % disp(q2)
167 %% Find q4
168
169 % two options based on two sets of q2,q3
170 % use psi = q2 + q3 + q4
171 for i = 1:length(D_real)
172     q4(end+1) = psi_real(i) - q2(2*i -1) - q3(2*i -1) +pi/2 ;
173     q4(end+1) = psi_real(i) - q2(2*i) - q3(2*i) +pi/2 ;
174 end
175
176 % disp("q4")
177 % disp(q4)
178 %% Remove duplicate values
179 % make a matrix out of the lists where one row is one solution
180
181 % we know matrix will have initial size: no. of items in q2,3 or 4 x 5
182 % angles
183 Solution_Matrix = zeros(length(q2), 5);
184 for i = 1:length(q2)
185     Solution_Matrix(i,1) = q1 ;
186     Solution_Matrix(i,2) = q2(i) ;
187     Solution_Matrix(i,3) = q3(i) ;
188     Solution_Matrix(i,4) = q4(i) ;
189     Solution_Matrix(i,5) = q5 ;
190 end
191 % disp(Solution_Matrix)
192 % remove duplicate rows
193 Unique_Solutions = unique(Solution_Matrix,"rows","stable");
194 % stable prevents order being changed.
195

```

```

196
197 %% we also know, q4 can't be more than 360 degrees. filter these out.
198 validSolutions = [] ;
199 for i = 1:size(UniqueSolutions,1)
200     if abs(UniqueSolutions(i,4)) < 2*pi
201         validSolutions = [validSolutions; UniqueSolutions(i,:)] ;
202     end
203 end
204
205 % stable prevents order being changed.
206 for i = 1:5
207     IK_OUTPUT(j,i) = validSolutions(1,i) ;
208 end
209 end
210
211 disp("Using IK: Array of q1-5 values:")
212 disp(IK_OUTPUT)
213
214
215
216 %% 3. use FK to plot the arm trajectory in 3d space
217 syms q1 q2 q3 q4 q5; % angle variables
218
219 % Show our 5 tranformation matrices
220 T_01 = [
221     [cos(q1), 0, sin(q1), 0]
222     [sin(q1), 0, -cos(q1), 0]
223     [0, 1, 0, d1]
224     [0, 0, 0, 1]];
225
226 T_12 = [
227     [cos(q2), -sin(q2), 0, l1*cos(q2)]
228     [sin(q2), cos(q2), 0, l1*sin(q2)]
229     [0, 0, 1, 0]
230     [0, 0, 0, 1]];
231
232 T_23 = [
233     [cos(q3), -sin(q3), 0, l2*cos(q3)]
234     [sin(q3), cos(q3), 0, l2*sin(q3)]
235     [0, 0, 1, 0]
236     [0, 0, 0, 1]];
237
238 T_34 = [
239     [cos(q4), 0, sin(q4), 0]
240     [sin(q4), 0, -cos(q4), 0]
241     [0, 1, 0, 0]
242     [0, 0, 0, 1]];
243
244 T_45 = [
245     [cos(q5), -sin(q5), 0, 0]
246     [sin(q5), cos(q5), 0, 0]
247     [0, 0, 1, l3]
248     [0, 0, 0, 1]];
249
250 % Get overall forward kinematics
251 T_e = T_01*T_12*T_23*T_34*T_45;
252
253 % extract the x,y,z elements of these
254 xt_element = T_e(1,4); % the x transformation is the 4th element of the 1st row of our general ...
    transformation matrix, T_e.
255 yt_element = T_e(2,4); % the y transformation is the 4th element of the 2nd row of our general ...
    transformation matrix, T_e.
256 zt_element = T_e(3,4); % the z transformation is the 4th element of the 3rd row of our general ...
    transformation matrix, T_e.
257
258 % Define all our angles in their own arrays
259 q1.set = [] ;
260 q2.set = [] ;
261 q3.set = [] ;
262 q4.set = [] ;

```

```

263 q5_set = [] ;
264 for i = 1:size(IK.OUTPUT,1) % i in range number of rows
265     % number of columns always 5 for q1-5
266     q1_set(i) = IK.OUTPUT(i,1) ;
267     q2_set(i) = IK.OUTPUT(i,2) ;
268     q3_set(i) = IK.OUTPUT(i,3) ;
269     q4_set(i) = IK.OUTPUT(i,4) ;
270     q5_set(i) = IK.OUTPUT(i,5) ;
271 end
272
273
274
275 % Now simply plot using our FK
276
277
278 % define variable for number of points
279 NoP = size(IK.OUTPUT,1) ; % same as number of sets of q values.
280
281 %base is at origin
282 base = 0 ;
283
284
285 % next joint given by T01 translations (movement from base to joint 1)
286 % need to make into a 4 by 3 array.
287 Tj1_x = zeros(NoP,1) ; % x translation is zero
288 Tj1_y = zeros(NoP,1) ; % y translation is zero
289 Tj1_z = subs(zeros(NoP,1), 0, d1) ; % z translation is d1
290
291 % next joint given by T01T12 translations - movement from base to joint 2
292 % make our general elements for each array
293 movement = T01*T12 ;
294 xt_element = movement(1,4) ;
295 yt_element = movement(2,4) ;
296 zt_element = movement(3,4) ;
297 % fill the joint 2 4x1 arrays
298 Tj2_x = [] ;
299 Tj2_y = [] ;
300 Tj2_z = [] ;
301 for i = 1:length(q1_set)
302     xt_value = subs(xt_element, [ q1 q2], [q1_set(i) q2_set(i)]) ;
303     yt_value = subs(yt_element, [ q1 q2], [q1_set(i) q2_set(i)]) ;
304     zt_value = subs(zt_element, [ q1 q2], [q1_set(i) q2_set(i)]) ;
305     Tj2_x(i,1) = xt_value ;
306     Tj2_y(i,1) = yt_value ;
307     Tj2_z(i,1) = zt_value ;
308 end
309 % above worked, now continue for the rest.
310
311
312 % next joint given by T01T12T23 translations - movement from base to joint 3
313 % make our general elements for each array
314 movement = T01*T12*T23 ;
315 xt_element = movement(1,4) ;
316 yt_element = movement(2,4) ;
317 zt_element = movement(3,4) ;
318 % fill the joint 3 4x1 arrays
319 Tj3_x = [] ;
320 Tj3_y = [] ;
321 Tj3_z = [] ;
322 for i = 1:length(q1_set)
323     xt_value = subs(xt_element,[q1 q2 q3], [q1_set(i) q2_set(i) q3_set(i)]) ;
324     yt_value = subs(yt_element,[q1 q2 q3], [q1_set(i) q2_set(i) q3_set(i)]) ;
325     zt_value = subs(zt_element,[q1 q2 q3], [q1_set(i) q2_set(i) q3_set(i)]) ;
326     Tj3_x(i,1) = xt_value ;
327     Tj3_y(i,1) = yt_value ;
328     Tj3_z(i,1) = zt_value ;
329 end
330
331
332 % next joint given by T01T12T23T34 translations - movement from base to joint 4

```

```

333 % note there's no translation in this joint so no change from above.
334 % make our general elements for each array
335 movement = T_01*T_12*T_23*T_34 ;
336 xt_element = movement(1,4) ;
337 yt_element = movement(2,4) ;
338 zt_element = movement(3,4) ;
339 % fill the joint 2 4x1 arrays
340 Tj4_x = [] ;
341 Tj4_y = [] ;
342 Tj4_z = [] ;
343 for i = 1:length(q1.set)
344     xt_value = subs(xt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
345     yt_value = subs(yt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
346     zt_value = subs(zt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
347     Tj4_x(i,1) = xt_value ;
348     Tj4_y(i,1) = yt_value ;
349     Tj4_z(i,1) = zt_value ;
350 end
351
352 % final joint given by T_01T_12T_23T_34T_45 translations - movement from base to joint 4
353 % note there's no translation in this joint so no change from above.
354 % make our general elements for each array
355 movement = T_01*T_12*T_23*T_34*T_45 ;
356 xt_element = movement(1,4) ;
357 yt_element = movement(2,4) ;
358 zt_element = movement(3,4) ;
359 % fill the joint 2 4x1 arrays
360 EE = [] ; % End Effector Positions list
361 for i = 1:length(q1.set)
362     xt_value = subs(xt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
363     yt_value = subs(yt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
364     zt_value = subs(zt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
365     EE(i,1) = xt_value ; % x values in column 1
366     EE(i,2) = yt_value ; % y values in column 2
367     EE(i,3) = zt_value ; % z values in column 3
368 end
369
370 % Check if end effector positions match points to plot.
371 disp("EE values (should match points to plot): ")
372 disp(EE)
373 disp("do they match? 0 = no, 1 = yes: ")
374 isequal(round(points.to.plot,2,"decimals"),round(EE,2,"decimals"))
375
376 figure (2)
377
378 % generate the graph. Note: only 4 joints will be visible as there is
379 % no spatial distinction between joints three and 4.
380 for i = 1:NoP % zeros is the base position, doesn't change.
381     xx = [base; Tj1_x(i); Tj2_x(i); Tj3_x(i); Tj4_x(i); EE(i,1) ] ;
382     yy = [base; Tj1_y(i); Tj2_y(i); Tj3_y(i); Tj4_y(i); EE(i,2) ] ;
383     zz = [base; Tj1_z(i); Tj2_z(i); Tj3_z(i); Tj4_z(i); EE(i,3) ] ;
384     axis([ 0 0.4 0 0.4 0 0.4 ])
385     plot3(xx,yy,zz,'ko-','Linewidth',2)
386     hold on % CHANGE TO HOLD ON TO SEE ALL LINES AT ONCE.
387     pause(2)
388
389     % label axes, start point, end point.
390     text(EE(1,1) + 0.002,EE(1,2) + 0.002,EE(1,3) + 0.002,'Start') ;
391     text(EE(end,1) + 0.002,EE(end,2) + 0.002,EE(end,3) + 0.002,'End') ;
392
393     % label end effector points
394     text(EE(i,1),EE(i,2),EE(i,3), 'x')
395 end
396 title('Actual Points') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');
397
398
399 figure (3) % end points only
400 for i = 1:NoP
401     plot3(EE(i,1),EE(i,2),EE(i,3),'ko-','Linewidth',2)
402     hold on

```

```
403 end
404 title('Actual End Points Only') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');
405 axis([ 0 0.4 0 0.4 0 0.4 ])
```

Appendix D: Serial Robot Linear Trajectories

Code available at: https://github.com/Hinlo/MatLab-Simulation-of-Serial-and-Parallel-Robots/blob/main/FINN_TASK_LIN_TRAJECTORIES_LynxMotionArm.m

FINN_TASK_LIN_TRAJECTORIES_LynxMotionArm.m

```

1 %% linear motion (straight lines)
2 clear all %#ok<*CLALL> %#ok<*SAGROW>
3 close all
4 clc
5 % link lengths
6 d1 = 0.1 ;
7 l1 = 0.1 ;
8 l2 = 0.1 ;
9 l3 = 0.1 ;
10
11 % assuming 10 HZ robot, performing each line in 1 second
12 %% TRYING LINEAR FIRST
13 %1. get start, end point and line eqn between.
14 %2. sample function for 10 points between each start and end point
15 %3. apply IK and FK
16
17 %% 1 & 2 -> find eqn for each line and add 10 points to the trajectory.
18 % function at bottom to do this.
19
20 points_per_line = 10 ;
21 points_to_plot = [[0.0 0.0 0.2]; [0.15 0.0 0.2]; [0.15 0.0 0.1]; [0.0 0.0 0.1]; [0.15 0.15 0.2]; ...
    [0.15 0.15 0.1]; [0.0 0.15 0.1]; [0.0 0.15 0.2]] ;
22 trajectories = zeros((size(points_to_plot,1) - 1)*points_per_line, 3) ;
23
24 % Use our function to get the gradient and constant, filter for lines along
25 % each axis.
26 for i = 1:(size(points_to_plot,1) - 1)
27     [grad, const, r_dir, r] = ...
        find_line_eqn(points_to_plot(i,1),points_to_plot(i+1,1),points_to_plot(i,2),points_to_plot(i+1,2),points_
        ;
28     for j = 1:points_per_line
29
30         if r_dir == 10
31             if i == 1 && j == 1
32                 trajectories(( (i-1)*10+j) ,3) = points_to_plot(i,3) + ...
                    (points_to_plot(i+1,3)-points_to_plot(i,3))*(j/points_per_line) ;
33                 continue
34             end
35             trajectories(( (i-1)*10+j) ,1) = trajectories(( (i-1)*10+j-1) ,1);
36             trajectories(( (i-1)*10+j) ,2) = trajectories(( (i-1)*10+j-1) ,2);
37             trajectories(( (i-1)*10+j) ,3) = points_to_plot(i,3) + ...
                (points_to_plot(i+1,3)-points_to_plot(i,3))*(j/points_per_line) ;
38             continue
39         end
40         x = points_to_plot(i,1) + r*cos(r_dir) * (j/points_per_line) ;
41         y = points_to_plot(i,2) + r*sin(r_dir) * (j/points_per_line) ;
42         r_temp = sqrt(x^2 + y^2) ;
43         z = grad*r_temp + const ;
44         trajectories(( (i-1)*10+j) ,1) = x ;
45         trajectories(( (i-1)*10+j) ,2) = y ;
46         trajectories(( (i-1)*10+j) ,3) = z ;
47     end
48 end
49 end
50 disp(trajectories)
51
52
53
54 %% 3

```



```

55 for j = 1:size(trajectories,1)
56     px = trajectories(j,1) ;
57     py = trajectories(j,2) ;
58     pz = trajectories(j,3) ;
59
60     % mu and psi angles for orientation
61     mu = 0; % mu is rotation of end effector relative to "wrist"
62     psi = (0:pi/20:2*pi) ;
63     for i = 1: length(psi)
64         psi(i) = psi(i) - pi/2;
65     end
66     EndEffector = [ px py pz ]' ;
67     if norm(EndEffector) > l1+l2+l3+d1 % workspace is above ground
68         error('desired position is out of the workspace')
69     end
70
71
72     %% %%%%%%%%% Inverse Kinematics of LynxMotion Arm %%%%%%%%%
73     % For details on calculations, view notes in ReadMe.
74     % We must find values for the 5 joint angles below.
75     sym q1 ; % single value
76     q2 = [] ; % potentially many values
77     q3 = [] ; % potentially many values
78     q4 = [] ; % potentially many values
79     sym q5 ; % single value
80
81     %% Simple angles to find:
82
83     % Find q1
84     q1 = atan2(py,px) ;
85     % Find q5
86     q5 = mu ;
87
88     %% Define extra variables to help us find q2,3,4
89
90     % define (r,z) plane
91     r = sqrt(px^2 + py^2) ; % r is hypotenues in x-y plane.
92
93     % For each value of psi, there is a value of rw, zw and D
94     r_w = (1:length(psi)) ;
95     z_w = (1:length(psi)) ;
96     D = (1:length(psi)) ;
97     for i = 1:length(psi)
98         % define position of wrist in (r,z) plane
99         r_w(i) = r - l3*cos(psi(i)) ; %r_w and z_w will be real as cos or sin of a real number is ...
100             a real number
101
102         z_w(i) = pz - d1 -l3*sin(psi(i)) ;
103
104         % define D - a placeholder variable for a large combination we derived in
105         % notes
106
107         D(i) = - (r_w(i)^2 + z_w(i)^2 - l1^2 - l2^2) / (2*l1*l2) ;
108         D(i) = round(D(i), 7) ; % rounding D avoids fake imaginary numbers due to rounding
109             % errors.
110     end
111     %% get only real values
112
113     psi_real = [] ;
114     r_w_real = [] ;
115     z_w_real = [] ;
116     D_real = [] ;
117     for i = 1:length(psi)
118         if imag(sqrt(1-D(i)^2)) == 0 % for real values, append to real lists.
119             psi_real(end + 1) = psi(i);%round(psi(i),7) ; % round each list to get zeros rather ...
120                 than e^-10 or something.
121             r_w_real(end + 1) = r_w(i);%round(r_w(i),7) ;
122             z_w_real(end + 1) = z_w(i);%round(z_w(i),7) ;
123             D_real(end + 1) = D(i);%round(D(i),7) ;
124         end
125     end

```

```

123     end
124
125     % we can see these four lists MUST have same length.
126     % disp(psi_real)
127     % disp(D_real)
128     % disp(z_w_real)
129     % disp(r_w_real)
130
131     %% Find q3 possibilities
132     % there are two possibilities for each value of D depending on
133     for i = 1:length(D_real)
134
135         q3(end+1) = atan2( sqrt(1-D_real(i)^2), -D_real(i)) ;
136         q3(end+1) = atan2(-sqrt(1-D_real(i)^2), -D_real(i)) ;
137     end
138
139     % disp("q3")
140     % disp(q3)
141     %% Explaining list order.
142     % matlab indexes from 1 not 0.
143     % if D has 3 values, we can see q3 will have 6, where
144     % D(1) corresponds to q3(1 & 2)
145     % D(2) corresponds to q3(3 & 4)
146     % D(3) corresponds to q3(5 & 6)
147     % we can see D(i) corresponds to values
148     % q3(2*i -1) and q3(2*i)
149     % we need to ensure this is *consistent* across all q lists below.
150     %% Find q2
151
152     % two options based on q3
153     for i = 1:length(D_real)
154         % for each r_w,z_w value, find first q2 value using first q3 value.
155         % Switching order here corrects angles... why?
156         q2(end+1) = atan2(z_w_real(i),r_w_real(i)) + atan2(l2*sin(q3(2*i)), l1+l2*cos(q3(2*i))) ;
157         % add +1 to
158         q2(end+1) = atan2(z_w_real(i),r_w_real(i)) + atan2(l2*sin(q3(2*i-1)), ...
159             l1+l2*cos(q3(2*i-1))) ;
160     end
161     % disp("q2")
162     % disp(q2)
163     %% Find q4
164
165     % two options based on two sets of q2,q3
166     % use psi = q2 + q3 + q4
167     for i = 1:length(D_real)
168         q4(end+1) = psi_real(i) - q2(2*i -1) - q3(2*i -1) +pi/2 ;
169         q4(end+1) = psi_real(i) - q2(2*i) - q3(2*i) +pi/2 ;
170     end
171     % disp("q4")
172     % disp(q4)
173     %% Remove duplicate values
174     % make a matrix out of the lists where one row is one solution
175
176     % we know matrix will have initial size: no. of items in q2,3 or 4 x 5
177     % angles
178     Solution_Matrix = zeros(length(q2), 5);
179     for i = 1:length(q2)
180         Solution_Matrix(i,1) = q1 ;
181         Solution_Matrix(i,2) = q2(i) ;
182         Solution_Matrix(i,3) = q3(i) ;
183         Solution_Matrix(i,4) = q4(i) ;
184         Solution_Matrix(i,5) = q5 ;
185     end
186     % disp(Solution_Matrix)
187     % remove duplicate rows
188     Unique_Solutions = unique(Solution_Matrix,"rows","stable") ;
189     % stable prevents order being changed.
190
191

```

```

192     %% we also know, q4 can't be more than 360 degrees. filter these out.
193     valid.Solutions = [] ;
194     for i = 1:size(Unique.Solutions,1)
195         if abs(Unique.Solutions(i,4)) < 2*pi
196             valid.Solutions = [valid.Solutions; Unique.Solutions(i,:)] ;
197         end
198     end
199     % stable prevents order being changed.
200     for i = 1:5
201         IK.OUTPUT(j,i) = valid.Solutions(1,i) ;
202     end
203 end
204 disp("Using IK: Array of q1-5 values:")
205 %disp(IK.OUTPUT);
206
207
208
209
210 %% 3. use FK to plot the arm trajectory in 3d space
211 syms q1 q2 q3 q4 q5; % angle variables
212
213 % Show our 5 transformation matrices
214 T_01 = [
215     [cos(q1), 0, sin(q1), 0]
216     [sin(q1), 0, -cos(q1), 0]
217     [0, 1, 0, d1]
218     [0, 0, 0, 1]];
219
220 T_12 = [
221     [cos(q2), -sin(q2), 0, l1*cos(q2)]
222     [sin(q2), cos(q2), 0, l1*sin(q2)]
223     [0, 0, 1, 0]
224     [0, 0, 0, 1]];
225
226 T_23 = [
227     [cos(q3), -sin(q3), 0, l2*cos(q3)]
228     [sin(q3), cos(q3), 0, l2*sin(q3)]
229     [0, 0, 1, 0]
230     [0, 0, 0, 1]];
231
232 T_34 = [
233     [cos(q4), 0, sin(q4), 0]
234     [sin(q4), 0, -cos(q4), 0]
235     [0, 1, 0, 0]
236     [0, 0, 0, 1]];
237
238 T_45 = [
239     [cos(q5), -sin(q5), 0, 0]
240     [sin(q5), cos(q5), 0, 0]
241     [0, 0, 1, l3]
242     [0, 0, 0, 1]];
243
244 % Get overall forward kinematics
245 T_e = T_01*T_12*T_23*T_34*T_45;
246
247 % extract the x,y,z elements of these
248 xt_element = T_e(1,4); % the x transformation is the 4th element of the 1st row of our general ...
249                    % transformation matrix, T_e.
250 yt_element = T_e(2,4); % the y transformation is the 4th element of the 2nd row of our general ...
251                    % transformation matrix, T_e.
252 zt_element = T_e(3,4); % the z transformation is the 4th element of the 3rd row of our general ...
253                    % transformation matrix, T_e.
254
255 % Define all our angles in their own arrays
256 q1.set = [] ;
257 q2.set = [] ;
258 q3.set = [] ;
259 q4.set = [] ;
260 q5.set = [] ;
261 for i = 1:size(IK.OUTPUT,1) % i in range number of rows

```

```

259     % number of columns always 5 for q1-5
260     q1.set(i) = IK.OUTPUT(i,1) ;
261     q2.set(i) = IK.OUTPUT(i,2) ;
262     q3.set(i) = IK.OUTPUT(i,3) ;
263     q4.set(i) = IK.OUTPUT(i,4) ;
264     q5.set(i) = IK.OUTPUT(i,5) ;
265 end
266
267
268 %
269 %Now simply plot using our FK
270
271
272 % define variable for number of points
273 NoP = size(IK.OUTPUT,1) ; % same as number of sets of q values.
274
275 %base is at origin
276 base = 0 ;
277
278
279 % next joint given by T01 translations (movement from base to joint 1)
280 % need to make into a 4 by 3 array.
281 Tj1_x = zeros(NoP,1) ; % x translation is zero
282 Tj1_y = zeros(NoP,1) ; % y translation is zero
283 Tj1_z = subs(zeros(NoP,1), 0, d1) ; % z translation is d1
284
285 % next joint given by T.01T.12 translations - movement from base to joint 2
286 % make our general elements for each array
287 movement = T.01*T.12 ;
288 xt_element = movement(1,4) ;
289 yt_element = movement(2,4) ;
290 zt_element = movement(3,4) ;
291 % fill the joint 2 4x1 arrays
292 Tj2_x = [] ;
293 Tj2_y = [] ;
294 Tj2_z = [] ;
295 for i = 1:length(q1.set)
296     xt_value = subs(xt_element, [ q1 q2], [q1.set(i) q2.set(i)]) ;
297     yt_value = subs(yt_element, [ q1 q2], [q1.set(i) q2.set(i)]) ;
298     zt_value = subs(zt_element, [ q1 q2], [q1.set(i) q2.set(i)]) ;
299     Tj2_x(i,1) = xt_value ;
300     Tj2_y(i,1) = yt_value ;
301     Tj2_z(i,1) = zt_value ;
302 end
303 % above worked, now continue for the rest.
304
305
306 % next joint given by T.01T.12T.23 translations - movement from base to joint 3
307 % make our general elements for each array
308 movement = T.01*T.12*T.23 ;
309 xt_element = movement(1,4) ;
310 yt_element = movement(2,4) ;
311 zt_element = movement(3,4) ;
312 % fill the joint 2 4x1 arrays
313 Tj3_x = [] ;
314 Tj3_y = [] ;
315 Tj3_z = [] ;
316 for i = 1:length(q1.set)
317     xt_value = subs(xt_element,[q1 q2 q3], [q1.set(i) q2.set(i) q3.set(i)]) ;
318     yt_value = subs(yt_element,[q1 q2 q3], [q1.set(i) q2.set(i) q3.set(i)]) ;
319     zt_value = subs(zt_element,[q1 q2 q3], [q1.set(i) q2.set(i) q3.set(i)]) ;
320     Tj3_x(i,1) = xt_value ;
321     Tj3_y(i,1) = yt_value ;
322     Tj3_z(i,1) = zt_value ;
323 end
324
325
326 % next joint given by T.01T.12T.23T.34 translations - movement from base to joint 4
327 % note there's no translation in this joint so no change from above.
328 % make our general elements for each array

```

```

329 movement = T_01*T_12*T_23*T_34 ;
330 xt_element = movement(1,4) ;
331 yt_element = movement(2,4) ;
332 zt_element = movement(3,4) ;
333 % fill the joint 2 4x1 arrays
334 Tj4_x = [] ;
335 Tj4_y = [] ;
336 Tj4_z = [] ;
337 for i = 1:length(q1.set)
338     xt_value = subs(xt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
339     yt_value = subs(yt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
340     zt_value = subs(zt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
341     Tj4_x(i,1) = xt_value ;
342     Tj4_y(i,1) = yt_value ;
343     Tj4_z(i,1) = zt_value ;
344 end
345
346 % final joint given by T_01T_12T_23T_34T_45 translations - movement from base to joint 4
347 % note there's no translation in this joint so no change from above.
348 % make our general elements for each array
349 movement = T_01*T_12*T_23*T_34*T_45 ;
350 xt_element = movement(1,4) ;
351 yt_element = movement(2,4) ;
352 zt_element = movement(3,4) ;
353 % fill the joint 2 4x1 arrays
354 EE = [] ; % End Effector Positions list
355 for i = 1:length(q1.set)
356     xt_value = subs(xt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
357     yt_value = subs(yt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
358     zt_value = subs(zt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
359     EE(i,1) = xt_value ; % x values in column 1
360     EE(i,2) = yt_value ; % y values in column 2
361     EE(i,3) = zt_value ; % z values in column 3
362 end
363
364 % Check if end effector positions match points to plot.
365 disp("EE values (should match points to plot): ")
366 disp(EE)
367 disp("do they match? 0 = no, 1 = yes: ")
368 isequal(round(trajectories,4,"decimals"),round(EE,4,"decimals"))
369
370 figure (2)
371
372 % generate the graph. Note: only 4 joints will be visible as there is
373 % no spatial distinction between joints three and 4.
374 for i = 1:NoP % zeros is the base position, doesn't change.
375     xx = [base; Tj1_x(i); Tj2_x(i); Tj3_x(i); Tj4_x(i); EE(i,1) ] ;
376     yy = [base; Tj1_y(i); Tj2_y(i); Tj3_y(i); Tj4_y(i); EE(i,2) ] ;
377     zz = [base; Tj1_z(i); Tj2_z(i); Tj3_z(i); Tj4_z(i); EE(i,3) ] ;
378     axis([ 0 0.2 0 0.2 0 0.3 ])
379     plot3(xx,yy,zz,'ko-','Linewidth',2)
380     hold on % CHANGE TO HOLD ON TO SEE ALL LINES AT ONCE.
381     pause(0.3)
382
383     % label axes, start point, end point.
384     text(EE(1,1) + 0.002,EE(1,2) + 0.002,EE(1,3) + 0.002,'Start') ;
385     text(EE(end,1) + 0.002,EE(end,2) + 0.002,EE(end,3) + 0.002,'End') ;
386
387     % label end effector points
388     text(EE(i,1),EE(i,2),EE(i,3), 'x')
389 end
390 title('Actual Points') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');
391
392
393 figure (3) % end points only
394 for i = 1:NoP
395     plot3(EE(i,1),EE(i,2),EE(i,3),'ko-','Linewidth',2)
396     hold on
397 end
398 title('Actual End Points Only') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');

```

```

399 axis([ 0 0.2 0 0.2 0 0.3 ])
400
401
402
403
404 %% Function to find gradient and constant of line equation
405 function [grad, const, r_dir, r] = find_line_eqn(x1, x2, y1, y2, z1, z2)
406
407     % find which vector r = in x,y plane magnitude and direction
408     % r_dir is angle from x axis along r, at the point of x1,y1 in the x,y
409     % plane.
410
411     % check for lines along each axis first
412     % line along z
413     if (x2 - x1) == 0 && (y2 - y1) == 0
414         grad = "null" ;
415         const = "null" ;
416         r_dir = 10 ; % max value of atan2 is < pi/2 therefore < 10
417         r2 = 0 ;
418         r1 = 0 ;
419         r = 0 ;
420         return
421     % line along y
422     elseif (x2 - x1) == 0
423         r1 = y1 ;
424         r2 = y2 ;
425         r = r2 - r1 ;
426         r_dir = pi/2 ;
427     % line along x
428     elseif (y2 - y1) == 0
429         r1 = x1 ;
430         r2 = x2 ;
431         r = r2 - r1 ;
432         r_dir = 0 ;
433     else
434         r1 = sqrt(x1^2 + y1^2) ;
435         r2 = sqrt(x2^2 + y2^2) ;
436         r = sqrt((y2-y1)^2 + (x2-x1)^2) ;
437         r_dir = atan2(y2-y1, x2-x1) ;
438         if r_dir < 0
439             r_dir = pi - r_dir;
440         end
441     end
442
443     grad = (z2-z1)/(r) ;
444     const = (z1*r2 - z2*r1)/(r) ;
445 end

```

Appendix E: Serial Robot Free Trajectories

Code available at: https://github.com/Hinlo/MatLab-Simulation-of-Serial-and-Parallel-Robots/blob/main/FINN_TASK_FREE_TRAJECTORIES_LynxMotionArm.m

FINN_TASK_FREE_TRAJECTORIES_LynxMotionArm.m

```

1 %% Free Trajectories (Arcing motion)
2
3 clear all %#ok<*CLALL> %#ok<*SAGROW>
4 close all
5 clc
6 % link lengths
7 d1 = 0.1 ;
8 l1 = 0.1 ;
9 l2 = 0.1 ;
10 l3 = 0.1 ;
11
12 % assuming 10 HZ robot, performing each line in 1 second
13 %% FREE MOTION
14 %1. do IK to find angles to reach each point
15 %2. sample angle intervals to show trajectories
16 %3. get expanded angles list
17 %4. run FK.
18
19 %% Define points to plot
20 points.to.plot = [[0.0 0.0 0.2]; [0.15 0.0 0.2]; [0.15 0.0 0.1]; [0.0 0.0 0.1]; [0.15 0.15 0.2]; ...
    [0.15 0.15 0.1]; [0.0 0.15 0.1]; [0.0 0.15 0.2]] ;
21
22 %% Run IK
23 for j = 1:size(points.to.plot,1)
24     px = points.to.plot(j,1) ;
25     py = points.to.plot(j,2) ;
26     pz = points.to.plot(j,3) ;
27
28     % mu and psi angles for orientation
29     mu = 0; % mu is rotation of end effector relative to "wrist"
30     psi = (0:pi/20:2*pi) ;
31     for i = 1: length(psi)
32         psi(i) = psi(i) - pi/2;
33     end
34     EndEffector = [ px py pz ]' ;
35     if norm(EndEffector) > l1+l2+l3+d1 % workspace is above ground
36         error('desired position is out of the workspace')
37     end
38
39
40     %% %%%%%%%%% Inverse Kinematics of LynxMotion Arm %%%%%%%%%
41     % For details on calculations, view notes in ReadMe.
42     % We must find values for the 5 joint angles below.
43     sym q1 ; % single value
44     q2 = [] ; % potentially many values
45     q3 = [] ; % potentially many values
46     q4 = [] ; % potentially many values
47     sym q5 ; % single value
48
49     %% Simple angles to find:
50
51     % Find q1
52     q1 = atan2(py,px) ;
53     % Find q5
54     q5 = mu ;
55
56     %% Define extra variables to help us find q2,3,4
57
58     % define (r,z) plane

```

```

59 r = sqrt(px^2 + py^2) ; % r is hypotenues in x-y plane.
60
61 % For each value of psi, there is a value of rw, zw and D
62 r_w = (1:length(psi)) ;
63 z_w = (1:length(psi)) ;
64 D = (1:length(psi)) ;
65 for i = 1:length(psi)
66 % define position of wrist in (r,z) plane
67 r_w(i) = r - l3*cos(psi(i)) ; %r_w and z_w will be real as cos or sin of a real number is ...
    a real number
68
69 z_w(i) = pz - d1 -l3*sin(psi(i)) ;
70
71 % define D - a placeholder variable for a large combination we derived in
72 % notes
73
74 D(i) = - (r_w(i)^2 + z_w(i)^2 - l1^2 - l2^2) / (2*l1*l2) ;
75 D(i) = round(D(i), 7) ;% rounding D avoids fake imaginary numbers due to rounding
76 % errors.
77 end
78 %% get only real values
79
80 psi_real = [] ;
81 r_w_real = [] ;
82 z_w_real = [] ;
83 D_real = [] ;
84 for i = 1:length(psi)
85 if imag(sqrt(1-D(i)^2)) == 0 % for real values, append to real lists.
86 psi_real(end + 1) = psi(i);%round(psi(i),7) ; % round each list to get zeros rather ...
    than e^-10 or something.
87 r_w_real(end + 1) = r_w(i);%round(r_w(i),7) ;
88 z_w_real(end + 1) = z_w(i);%round(z_w(i),7) ;
89 D_real(end + 1) = D(i);%round(D(i),7) ;
90 end
91 end
92
93 % we can see these four lists MUST have same length.
94 % disp(psi_real)
95 % disp(D_real)
96 % disp(z_w_real)
97 % disp(r_w_real)
98
99 %% Find q3 possibilities
100 % there are two possibilities for each value of D depending on
101 for i = 1:length(D_real)
102
103 q3(end+1) = atan2( sqrt(1-D_real(i)^2), -D_real(i)) ;
104 q3(end+1) = atan2(-sqrt(1-D_real(i)^2), -D_real(i)) ;
105 end
106
107 % disp("q3")
108 % disp(q3)
109 %% Explaining list order.
110 % matlab indexes from 1 not 0.
111 % if D has 3 values, we can see q3 will have 6, where
112 % D(1) corresponds to q3(1 & 2)
113 % D(2) corresponds to q3(3 & 4)
114 % D(3) corresponds to q3(5 & 6)
115 % we can see D(i) corresponds to values
116 % q3(2*i -1) and q3(2*i)
117 % we need to ensure this is *consistant* across all q lists below.
118 %% Find q2
119
120 % two options based on q3
121 for i = 1:length(D_real)
122 % for each r_w,z_w value, find first q2 value using first q3 value.
123 % Switching order here corrects angles... why?
124 q2(end+1) = atan2(z_w_real(i),r_w_real(i)) + atan2(l2*sin(q3(2*i)), l1+l2*cos(q3(2*i))) ;
125 % add +1 to
126 q2(end+1) = atan2(z_w_real(i),r_w_real(i)) + atan2(l2*sin(q3(2*i-1))), ...

```



```

11+12*cos(q3(2*i-1))) ;
127 end
128 % disp("q2")
129 % disp(q2)
130 %% Find q4
131
132 % two options based on two sets of q2,q3
133 % use psi = q2 + q3 + q4
134 for i = 1:length(D.real)
135     q4(end+1) = psi-real(i) - q2(2*i -1) - q3(2*i -1) +pi/2 ;
136     q4(end+1) = psi-real(i) - q2(2*i) - q3(2*i) +pi/2 ;
137 end
138
139 % disp("q4")
140 % disp(q4)
141 %% Remove duplicate values
142 % make a matrix out of the lists where one row is one solution
143
144 % we know matrix will have initial size: no. of items in q2,3 or 4 x 5
145 % angles
146 SolutionMatrix = zeros(length(q2), 5);
147 for i = 1:length(q2)
148     SolutionMatrix(i,1) = q1 ;
149     SolutionMatrix(i,2) = q2(i) ;
150     SolutionMatrix(i,3) = q3(i) ;
151     SolutionMatrix(i,4) = q4(i) ;
152     SolutionMatrix(i,5) = q5 ;
153 end
154 % disp(SolutionMatrix)
155 % remove duplicate rows
156 Unique.Solutions = unique(SolutionMatrix,"rows","stable") ;
157 % stable prevents order being changed.
158
159
160 %% we also know, q4 can't be more than 360 degrees. filter these out.
161 valid.Solutions = [] ;
162 for i = 1:size(Unique.Solutions,1)
163     if abs(Unique.Solutions(i,4)) < 2*pi
164         valid.Solutions = [valid.Solutions; Unique.Solutions(i,:)] ;
165     end
166 end
167 % stable prevents order being changed.
168 for i = 1:5
169     IK.OUTPUT(j,i) = valid.Solutions(1,i) ;
170
171     end
172 end
173 disp("Using IK: Array of q1-5 values:")
174 disp(IK.OUTPUT);
175
176
177 %% Expand angles to include motion between points
178
179 % define points per line and new list
180 points_per_line = 10 ;
181 expanded_angles = zeros((size(points.to_plot,1) - 1)*points_per_line, 5) ;
182
183 % populate new list
184 for i = 1:(size(points.to_plot,1) - 1)
185     for j = 1:points_per_line
186         expanded_angles(( (i-1)*10+j) ,1) = IK.OUTPUT(i,1) + ( IK.OUTPUT(i+1,1) - IK.OUTPUT(i,1) ) ...
            * j/points_per_line ;%q3
187         expanded_angles(( (i-1)*10+j) ,2) = IK.OUTPUT(i,2) + ( IK.OUTPUT(i+1,2) - IK.OUTPUT(i,2) ) ...
            * j/points_per_line ;%q3
188         expanded_angles(( (i-1)*10+j) ,3) = IK.OUTPUT(i,3) + ( IK.OUTPUT(i+1,3) - IK.OUTPUT(i,3) ) ...
            * j/points_per_line ;%q3
189         expanded_angles(( (i-1)*10+j) ,4) = IK.OUTPUT(i,4) + ( IK.OUTPUT(i+1,4) - IK.OUTPUT(i,4) ) ...
            * j/points_per_line ;%q3
190         expanded_angles(( (i-1)*10+j) ,5) = IK.OUTPUT(i,5) + ( IK.OUTPUT(i+1,5) - IK.OUTPUT(i,5) ) ...
            * j/points_per_line ;%q3

```

```

191
192     end
193 end
194 disp(" Array of Expanded Angles: ")
195 disp(expanded_angles)
196 %% 3. use FK to plot the arm trajectory in 3d space
197 syms q1 q2 q3 q4 q5; % angle variables
198
199 % Show our 5 transformation matrices
200 T_01 = [
201     [cos(q1), 0, sin(q1), 0]
202     [sin(q1), 0, -cos(q1), 0]
203     [0, 1, 0, d1]
204     [0, 0, 0, 1]];
205
206 T_12 = [
207     [cos(q2), -sin(q2), 0, l1*cos(q2)]
208     [sin(q2), cos(q2), 0, l1*sin(q2)]
209     [0, 0, 1, 0]
210     [0, 0, 0, 1]];
211
212 T_23 = [
213     [cos(q3), -sin(q3), 0, l2*cos(q3)]
214     [sin(q3), cos(q3), 0, l2*sin(q3)]
215     [0, 0, 1, 0]
216     [0, 0, 0, 1]];
217
218 T_34 = [
219     [cos(q4), 0, sin(q4), 0]
220     [sin(q4), 0, -cos(q4), 0]
221     [0, 1, 0, 0]
222     [0, 0, 0, 1]];
223
224 T_45 = [
225     [cos(q5), -sin(q5), 0, 0]
226     [sin(q5), cos(q5), 0, 0]
227     [0, 0, 1, l3]
228     [0, 0, 0, 1]];
229
230 % Get overall forward kinematics
231 T_e = T_01*T_12*T_23*T_34*T_45;
232
233 % extract the x,y,z elements of these
234 xt_element = T_e(1,4); % the x transformation is the 4th element of the 1st row of our general ...
235     transformation matrix, T_e.
236 yt_element = T_e(2,4); % the y transformation is the 4th element of the 2nd row of our general ...
237     transformation matrix, T_e.
238 zt_element = T_e(3,4); % the z transformation is the 4th element of the 3rd row of our general ...
239     transformation matrix, T_e.
240
241 % Define all our angles in their own arrays
242 q1_set = [];
243 q2_set = [];
244 q3_set = [];
245 q4_set = [];
246 q5_set = [];
247
248 for i = 1:size(expanded_angles,1) % i in range number of rows
249     % number of columns always 5 for q1-5
250     q1_set(i) = expanded_angles(i,1);
251     q2_set(i) = expanded_angles(i,2);
252     q3_set(i) = expanded_angles(i,3);
253     q4_set(i) = expanded_angles(i,4);
254     q5_set(i) = expanded_angles(i,5);
255 end
256
257 %
258 %Now simply plot using our FK
259
260
261

```

```

258 % define variable for number of points
259 NoP = size(expanded.angles,1) ; % same as number of sets of q values.
260
261 %base is at origin
262 base = 0 ;
263
264
265 % next joint given by T01 translations (movement from base to joint 1)
266 % need to make into a 4 by 3 array.
267 Tj1_x = zeros(NoP,1) ; % x translation is zero
268 Tj1_y = zeros(NoP,1) ; % y translation is zero
269 Tj1_z = subs(zeros(NoP,1), 0, d1) ; % z translation is d1
270
271 % next joint given by T01T12 translations - movement from base to joint 2
272 % make our general elements for each array
273 movement = T01*T12 ;
274 xt_element = movement(1,4) ;
275 yt_element = movement(2,4) ;
276 zt_element = movement(3,4) ;
277 % fill the joint 2 4x1 arrays
278 Tj2_x = [] ;
279 Tj2_y = [] ;
280 Tj2_z = [] ;
281 for i = 1:length(q1.set)
282     xt_value = subs(xt_element, [ q1 q2], [q1.set(i) q2.set(i)]) ;
283     yt_value = subs(yt_element, [ q1 q2], [q1.set(i) q2.set(i)]) ;
284     zt_value = subs(zt_element, [ q1 q2], [q1.set(i) q2.set(i)]) ;
285     Tj2_x(i,1) = xt_value ;
286     Tj2_y(i,1) = yt_value ;
287     Tj2_z(i,1) = zt_value ;
288 end
289 % above worked, now continue for the rest.
290
291
292 % next joint given by T01T12T23 translations - movement from base to joint 3
293 % make our general elements for each array
294 movement = T01*T12*T23 ;
295 xt_element = movement(1,4) ;
296 yt_element = movement(2,4) ;
297 zt_element = movement(3,4) ;
298 % fill the joint 2 4x1 arrays
299 Tj3_x = [] ;
300 Tj3_y = [] ;
301 Tj3_z = [] ;
302 for i = 1:length(q1.set)
303     xt_value = subs(xt_element,[q1 q2 q3], [q1.set(i) q2.set(i) q3.set(i)]) ;
304     yt_value = subs(yt_element,[q1 q2 q3], [q1.set(i) q2.set(i) q3.set(i)]) ;
305     zt_value = subs(zt_element,[q1 q2 q3], [q1.set(i) q2.set(i) q3.set(i)]) ;
306     Tj3_x(i,1) = xt_value ;
307     Tj3_y(i,1) = yt_value ;
308     Tj3_z(i,1) = zt_value ;
309 end
310
311
312 % next joint given by T01T12T23T34 translations - movement from base to joint 4
313 % note there's no translation in this joint so no change from above.
314 % make our general elements for each array
315 movement = T01*T12*T23*T34 ;
316 xt_element = movement(1,4) ;
317 yt_element = movement(2,4) ;
318 zt_element = movement(3,4) ;
319 % fill the joint 2 4x1 arrays
320 Tj4_x = [] ;
321 Tj4_y = [] ;
322 Tj4_z = [] ;
323 for i = 1:length(q1.set)
324     xt_value = subs(xt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
325     yt_value = subs(yt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
326     zt_value = subs(zt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
327     Tj4_x(i,1) = xt_value ;

```

```

328     Tj4_y(i,1) = yt_value ;
329     Tj4_z(i,1) = zt_value ;
330 end
331
332 % final joint given by T_01T_12T_23T_34T_45 translations - movement from base to joint 4
333 % note there's no translation in this joint so no change from above.
334 % make our general elements for each array
335 movement = T_01*T_12*T_23*T_34*T_45 ;
336 xt_element = movement(1,4) ;
337 yt_element = movement(2,4) ;
338 zt_element = movement(3,4) ;
339 % fill the joint 2 4x1 arrays
340 EE = [] ; % End Effector Positions list
341 for i = 1:length(q1_set)
342     xt_value = subs(xt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
343     yt_value = subs(yt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
344     zt_value = subs(zt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
345     EE(i,1) = xt_value ; % x values in column 1
346     EE(i,2) = yt_value ; % y values in column 2
347     EE(i,3) = zt_value ; % z values in column 3
348 end
349
350 % Check if end effector positions match points to plot.
351 % disp("EE values (should match points to plot): ")
352 % disp(EE)
353 % disp("do they match? 0 = no, 1 = yes: ")
354 % isequal(round(points_to_plot,4,"decimals"),round(EE,4,"decimals"))
355
356 figure (2)
357
358 % generate the graph. Note: only 4 joints will be visible as there is
359 % no spatial distinction between joints three and 4.
360 for i = 1:NoP % zeros is the base position, doesn't change.
361     xx = [base; Tj1_x(i); Tj2_x(i); Tj3_x(i); Tj4_x(i); EE(i,1) ] ;
362     yy = [base; Tj1_y(i); Tj2_y(i); Tj3_y(i); Tj4_y(i); EE(i,2) ] ;
363     zz = [base; Tj1_z(i); Tj2_z(i); Tj3_z(i); Tj4_z(i); EE(i,3) ] ;
364     axis([ 0 0.2 0 0.2 0 0.3 ])
365     plot3(xx,yy,zz,'ko-','Linewidth',2)
366     hold on % CHANGE TO HOLD ON TO SEE ALL LINES AT ONCE.
367     pause(0.3)
368
369 % label axes, start point, end point.
370 text(EE(1,1) + 0.002,EE(1,2) + 0.002,EE(1,3) + 0.002,'Start') ;
371 text(EE(end,1) + 0.002,EE(end,2) + 0.002,EE(end,3) + 0.002,'End') ;
372
373 % label end effector points
374 text(EE(i,1),EE(i,2),EE(i,3), 'x')
375 end
376 title('Actual Points') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');
377
378
379 figure (3) % end points only
380 for i = 1:NoP
381     plot3(EE(i,1),EE(i,2),EE(i,3),'ko-','Linewidth',2)
382     hold on
383 end
384 title('Actual End Points Only') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');
385 axis([ 0 0.2 0 0.2 0 0.3 ])

```

Appendix F: Serial Robot Obstacle Avoidance

Code available at: https://github.com/Hinlo/MatLab-Simulation-of-Serial-and-Parallel-Robots/blob/main/FINN_TASK_OBSTACLE.m

FINN_TASK_OBSTACLE.m

```

1      %% Bug 2 Obstacle Avoidance
2  % linear motion (straight lines)
3  clear all %#ok<*CLALL> %#ok<*SAGROW>
4  close all
5  clc
6  % link lengths
7  d1 = 0.1 ;
8  l1 = 0.1 ;
9  l2 = 0.1 ;
10 l3 = 0.1 ;
11
12 % assuming 10 HZ robot, performing each line in 1 second
13 % Using free motion rather than linear so as to track curved objects as
14 % well as straight line ones.
15
16 %% Object avoidance
17 % this file works in the following steps
18 % 1. object boundaries defined, currently a cube directly in the object
19 % path. Object task is still to plot the corners of a larger cube as in the
20 % previous free motion task.
21 % 2. Ik and Fk are performed for the task, the end effector positions are
22 % then sampled to see where they first impact and leave the object.
23 % 3. The end effector path is corrected using the obstacle avoidance function at the end
24 % to the bug 2 algorithm moving along the edge of the object.
25 % 4. IK and FK are re run with the new final path to ensure correct q1-5
26 % values and maintain link lengths.
27 % 5. Robot motion is plotted to show it still hits the desired points and
28 % avoids the object.
29
30
31 %% Define the object
32
33 % object is a smaller cuboid, detail x,y,z dimensions
34 % i = initial, f = final
35 ob_x_i = 0.06 ;
36 ob_x_f = 0.12 ;
37
38 ob_y_i = 0.01 ;
39 ob_y_f = 0.07 ;
40
41 ob_z_i = 0.12 ;
42 ob_z_f = 0.18 ;
43
44 % Define whole object using an array
45 % define points per line for each axis and object array
46 points_per_line = 10 ;
47 object = zeros(points_per_line+1,3) ;
48
49 % populate object array
50 for i = 1:(points_per_line + 1)
51     object(i,1) = ob_x_i + (i-1)*(ob_x_f - ob_x_i)/points_per_line ;
52     object(i,2) = ob_y_i + (i-1)*(ob_y_f - ob_y_i)/points_per_line ;
53     object(i,3) = ob_z_i + (i-1)*(ob_z_f - ob_z_i)/points_per_line ;
54 end
55 % disp(object)
56 %% Define points to plot
57 points_to_plot = [[0.0 0.0 0.2]; [0.15 0.0 0.2]; [0.15 0.0 0.1]; [0.0 0.0 0.1]; [0.15 0.15 0.2]; ...
58                 [0.15 0.15 0.1]; [0.0 0.15 0.1]; [0.0 0.15 0.2]] ;

```

```

59
60 %% Run IK
61 for j = 1:size(points.to_plot,1)
62     px = points.to_plot(j,1) ;
63     py = points.to_plot(j,2) ;
64     pz = points.to_plot(j,3) ;
65
66     % mu and psi angles for orientation
67     mu = 0; % mu is rotation of end effector relative to "wrist"
68     psi = (0:pi/20:2*pi) ;
69     for i = 1: length(psi)
70         psi(i) = psi(i) - pi/2;
71     end
72     EndEffector = [ px py pz ]' ;
73     if norm(EndEffector) > l1+l2+l3+d1 % workspace is above ground
74         error('desired position is out of the workspace')
75     end
76
77
78     %% %%%%%%%%% Inverse Kinematics of LynxMotion Arm %%%%%%%%%
79     % For details on calculations, view notes in ReadMe.
80     % We must find values for the 5 joint angles below.
81     sym q1 ; % single value
82     q2 = [] ; % potentially many values
83     q3 = [] ; % potentially many values
84     q4 = [] ; % potentially many values
85     sym q5 ; % single value
86
87     %% Simple angles to find:
88
89     % Find q1
90     q1 = atan2(py,px) ;
91     % Find q5
92     q5 = mu ;
93
94     %% Define extra variables to help us find q2,3,4
95
96     % define (r,z) plane
97     r = sqrt(px^2 + py^2) ; % r is hypotenues in x-y plane.
98
99     % For each value of psi, there is a value of rw, zw and D
100     r_w = (1:length(psi)) ;
101     z_w = (1:length(psi)) ;
102     D = (1:length(psi)) ;
103     for i = 1:length(psi)
104         % define position of wrist in (r,z) plane
105         r_w(i) = r - l3*cos(psi(i)) ; %r_w and z_w will be real as cos or sin of a real number is ...
            a real number
106
107         z_w(i) = pz - d1 -l3*sin(psi(i)) ;
108
109         % define D - a placeholder variable for a large combination we derived in
110         % notes
111
112         D(i) = - (r_w(i)^2 + z_w(i)^2 - l1^2 - l2^2) / (2*l1*l2) ;
113         D(i) = round(D(i), 7) ; % rounding D avoids fake imaginary numbers due to rounding
114             % errors.
115     end
116     %% get only real values
117
118     psi_real = [] ;
119     r_w_real = [] ;
120     z_w_real = [] ;
121     D_real = [] ;
122     for i = 1:length(psi)
123         if imag(sqrt(1-D(i)^2)) == 0 % for real values, append to real lists.
124             psi_real(end + 1) = psi(i);%round(psi(i),7) ; % round each list to get zeros rather ...
                than e^-10 or something.
125             r_w_real(end + 1) = r_w(i);%round(r_w(i),7) ;
126             z_w_real(end + 1) = z_w(i);%round(z_w(i),7) ;

```

```

127         D_real(end + 1) = D(i); %round(D(i),7) ;
128     end
129 end
130
131 % we can see these four lists MUST have same length.
132 % disp(psi_real)
133 % disp(D_real)
134 % disp(z_w_real)
135 % disp(r_w_real)
136
137 %% Find q3 possibilities
138 % there are two possibilities for each value of D depending on
139 for i = 1:length(D_real)
140
141     q3(end+1) = atan2( sqrt(1-D_real(i)^2), -D_real(i)) ;
142     q3(end+1) = atan2(-sqrt(1-D_real(i)^2), -D_real(i)) ;
143 end
144
145 % disp("q3")
146 % disp(q3)
147 %% Explaining list order.
148 % matlab indexes from 1 not 0.
149 % if D has 3 values, we can see q3 will have 6, where
150 % D(1) corresponds to q3(1 & 2)
151 % D(2) corresponds to q3(3 & 4)
152 % D(3) corresponds to q3(5 & 6)
153 % we can see D(i) corresponds to values
154 % q3(2*i -1) and q3(2*i)
155 % we need to ensure this is *consistant* across all q lists below.
156 %% Find q2
157
158 % two options based on q3
159 for i = 1:length(D_real)
160     % for each r_w,z_w value, find first q2 value using first q3 value.
161     % Switching order here corrects angles... why?
162     q2(end+1) = atan2(z_w_real(i),r_w_real(i)) + atan2(l2*sin(q3(2*i)), l1+l2*cos(q3(2*i))) ;
163     % add +1 to
164     q2(end+1) = atan2(z_w_real(i),r_w_real(i)) + atan2(l2*sin(q3(2*i-1)), ...
165         l1+l2*cos(q3(2*i-1))) ;
166 end
167 % disp("q2")
168 % disp(q2)
169 %% Find q4
170
171 % two options based on two sets of q2,q3
172 % use psi = q2 + q3 + q4
173 for i = 1:length(D_real)
174     q4(end+1) = psi_real(i) - q2(2*i -1) - q3(2*i -1) +pi/2 ;
175     q4(end+1) = psi_real(i) - q2(2*i) - q3(2*i) +pi/2 ;
176 end
177
178 % disp("q4")
179 % disp(q4)
180 %% Remove duplicate values
181 % make a matrix out of the lists where one row is one solution
182
183 % we know matrix will have initial size: no. of items in q2,3 or 4 x 5
184 % angles
185 Solution_Matrix = zeros(length(q2), 5);
186 for i = 1:length(q2)
187     Solution_Matrix(i,1) = q1 ;
188     Solution_Matrix(i,2) = q2(i) ;
189     Solution_Matrix(i,3) = q3(i) ;
190     Solution_Matrix(i,4) = q4(i) ;
191     Solution_Matrix(i,5) = q5 ;
192 end
193 % disp(Solution_Matrix)
194 % remove duplicate rows
195 Unique_Solutions = unique(Solution_Matrix,"rows","stable") ;

```

```

196     % stable prevents order being changed.
197
198
199     %% we also know, q4 can't be more than 360 degrees. filter these out.
200     validSolutions = [] ;
201     for i = 1:size(Unique.Solutions,1)
202         if abs(Unique.Solutions(i,4)) < 2*pi
203             validSolutions = [validSolutions; Unique.Solutions(i,:)] ;
204         end
205     end
206     % stable prevents order being changed.
207     for i = 1:5
208         IK.OUTPUT(j,i) = validSolutions(1,i) ;
209     end
210 end
211
212 % disp("Using IK: Array of q1-5 values:")
213 % disp(IK.OUTPUT);
214
215
216 %% Expand angles to include motion between points
217
218 % define points per line and new list
219 points_per_line = 10 ;
220 expanded_angles = zeros((size(points.to_plot,1) - 1)*points_per_line, 5) ;
221
222 % populate new list
223 for i = 1:(size(points.to_plot,1) - 1)
224     for j = 1:points_per_line
225         expanded_angles((i-1)*10+j, 1) = IK.OUTPUT(i,1) + ( IK.OUTPUT(i+1,1) - IK.OUTPUT(i,1) ) ...
226             * j/points_per_line ;%q3
227         expanded_angles((i-1)*10+j, 2) = IK.OUTPUT(i,2) + ( IK.OUTPUT(i+1,2) - IK.OUTPUT(i,2) ) ...
228             * j/points_per_line ;%q3
229         expanded_angles((i-1)*10+j, 3) = IK.OUTPUT(i,3) + ( IK.OUTPUT(i+1,3) - IK.OUTPUT(i,3) ) ...
230             * j/points_per_line ;%q3
231         expanded_angles((i-1)*10+j, 4) = IK.OUTPUT(i,4) + ( IK.OUTPUT(i+1,4) - IK.OUTPUT(i,4) ) ...
232             * j/points_per_line ;%q3
233         expanded_angles((i-1)*10+j, 5) = IK.OUTPUT(i,5) + ( IK.OUTPUT(i+1,5) - IK.OUTPUT(i,5) ) ...
234             * j/points_per_line ;%q3
235     end
236 end
237
238 % disp(expanded_angles)
239
240 %% 3. use FK to plot the arm trajectory in 3d space and save the end effector points to display ...
241     the smiley face.
242
243 %IK.OUTPUT = [[45 30 30 30 0]*pi/180]
244 syms q1 q2 q3 q4 q5; % angle variables
245
246 % Show our 5 tranformation matrices
247
248 T_01 = [
249     cos(q1), 0,   sin(q1), 0]
250     [sin(q1), 0, -cos(q1), 0]
251     [      0, 1,      0, d1]
252     [      0, 0,      0, 1]];
253
254 T_12 = [
255     cos(q2), -sin(q2), 0, 11*cos(q2)]
256     [sin(q2),  cos(q2), 0, 11*sin(q2)]
257     [      0,      0, 1,      0]
258     [      0,      0, 0,      1]];
259
260 T_23 = [
261     cos(q3), -sin(q3), 0, 12*cos(q3)]
262     [sin(q3),  cos(q3), 0, 12*sin(q3)]
263     [      0,      0, 1,      0]
264     [      0,      0, 0,      1]];
265
266 T_34 = [
267     cos(q4), 0,   sin(q4),      0]
268     [sin(q4), 0, -cos(q4),      0]

```



```

260 [      0, 1,      0,      0]
261 [      0, 0,      0,      1]];
262
263 T_45 = [
264 [cos(q5), -sin(q5), 0, 0]
265 [sin(q5), cos(q5), 0, 0]
266 [      0,      0, 1, 13]
267 [      0,      0, 0, 1]];
268
269 % Get overall forward kinematics
270 T_e = T_01*T_12*T_23*T_34*T_45;
271
272 % extract the x,y,z elements of these
273 xt_element = T_e(1,4); % the x transformation is the 4th element of the 1st row of our general ...
    transformation matrix, T_e.
274 yt_element = T_e(2,4); % the y transformation is the 4th element of the 2nd row of our general ...
    transformation matrix, T_e.
275 zt_element = T_e(3,4); % the z transformation is the 4th element of the 3rd row of our general ...
    transformation matrix, T_e.
276
277 % Define all our angles in their own arrays
278 q1_set = [] ;
279 q2_set = [] ;
280 q3_set = [] ;
281 q4_set = [] ;
282 q5_set = [] ;
283 for i = 1:size(expanded_angles,1) % i in range number of rows
284     % number of columns always 5 for q1-5
285     q1_set(i) = expanded_angles(i,1) ;
286     q2_set(i) = expanded_angles(i,2) ;
287     q3_set(i) = expanded_angles(i,3) ;
288     q4_set(i) = expanded_angles(i,4) ;
289     q5_set(i) = expanded_angles(i,5) ;
290 end
291
292 % final joint given by T_01T_12T_23T_34T_45 translations - movement from base to joint 4
293 % note there's no translation in this joint so no change from above.
294 % make our general elements for each array
295 movement = T_01*T_12*T_23*T_34*T_45 ;
296 xt_element = movement(1,4) ;
297 yt_element = movement(2,4) ;
298 zt_element = movement(3,4) ;
299 % fill the joint 2 4x1 arrays
300 EE = [] ; % End Effector Positions list
301 for i = 1:length(q1_set)
302     xt_value = subs(xt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
303     yt_value = subs(yt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
304     zt_value = subs(zt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
305     EE(i,1) = xt_value ; % x values in column 1
306     EE(i,2) = yt_value ; % y values in column 2
307     EE(i,3) = zt_value ; % z values in column 3
308 end
309
310 % Check if end effector positions match points to plot.
311 % disp("EE values: ")
312 % disp(EE)
313 % disp("do they match? 0 = no, 1 = yes: ")
314 % isequal(round(points_to_plot,4,"decimals"),round(EE,4,"decimals"))
315
316 %% Check when obstacle in way
317 %object
318 % object is a smaller cuboid, detail x,y,z dimensions
319 % i = initial, f = final
320 % ob_x_i = 0.06 ;
321 % ob_x_f = 0.12 ;
322 %
323 % ob_y_i = 0.01 ;
324 % ob_y_f = 0.07 ;
325 %
326 % ob_z_i = 0.12 ;

```

```

327 % ob_z_f = 0.18 ;
328
329 EE_off_limits = [] ;
330 for i = 1:(size(EE,1))
331     if ( ob_x_i ≤ EE(i,1) ) && ( EE(i,1) ≤ ob_x_f ) && ( ob_y_i ≤ EE(i,2) ) && ( EE(i,2) ≤ ob_y_f ...
332         ) && ( ob_z_i ≤ EE(i,3) ) && ( EE(i,3) ≤ ob_z_f )
333         disp("obstacle hit at point ")
334         EE(i,:)
335         EE_off_limits = [EE_off_limits, i] ;
336     end
337 end
338 % disp("off limit rows: " + EE_off_limits)
339 disp(EE_off_limits)
340 length(EE_off_limits)
341 % Function to replot in this interval - go one back from the off limits to
342 % find EE values before the robot hits the object.
343 avoid_obstacle = bug2(EE_off_limits(1)-1, EE_off_limits(end)+1, EE, object) ;
344
345 % size(EE,1)
346 % remove off limit rows from EE
347 for i = 1:length(EE_off_limits)
348     EE(EE_off_limits(i), :) = [] ; % remove the first value each time to get correct one.
349 end
350 % size(EE,1)
351
352 % add in bug values to show navigating the obstacle.
353 rerouted_EE = zeros((size(EE,1)+size(avoid_obstacle,1)),3);
354 % length(rerouted_EE)
355 for i = 1:size(rerouted_EE,1)
356     if i < EE_off_limits(1)
357         % disp("add EE 1")
358         % disp(i)
359         rerouted_EE(i, :) = EE(i, :) ;
360     elseif i < (EE_off_limits(1)+size(avoid_obstacle,1))
361         % disp("add ob avoid")
362         % disp(i)
363         rerouted_EE(i, :) = avoid_obstacle((i-(EE_off_limits(1)-1)) , :) ;
364     else
365         % disp("add EE 2")
366         % disp(i)
367         rerouted_EE(i, :) = EE(i-size(avoid_obstacle,1), :) ;
368     end
369 end
370 EE = rerouted_EE ;
371
372 %% Run IK and FK again to get the new trajectory with obstacle avoidance.
373 for j = 1:size(EE,1)
374     px = EE(j,1) ;
375     py = EE(j,2) ;
376     pz = EE(j,3) ;
377
378     % mu and psi angles for orientation
379     mu = 0; % mu is rotation of end effector relative to "wrist"
380     psi = (0:pi/20:2*pi) ;
381     for i = 1: length(psi)
382         psi(i) = psi(i) - pi/2;
383     end
384     EndEffector = [ px py pz ]' ;
385     if norm(EndEffector) > l1+l2+l3+d1 % workspace is above ground
386         error('desired position is out of the workspace')
387     end
388
389
390     % % % % % % % % % % Inverse Kinematics of LynxMotion Arm % % % % % % % % % %
391     % For details on calculations, view notes in ReadMe.
392     % We must find values for the 5 joint angles below.
393     sym q1 ; % single value
394     q2 = [] ; % potentially many values
395     q3 = [] ; % potentially many values

```

```

396 q4 = [] ; % potentially many values
397 sym q5 ; % single value
398
399 %% Simple angles to find:
400
401 % Find q1
402 q1 = atan2(py,px) ;
403 % Find q5
404 q5 = mu ;
405
406 %% Define extra variables to help us find q2,3,4
407
408 % define (r,z) plane
409 r = sqrt(px^2 + py^2) ; % r is hypotenues in x-y plane.
410
411 % For each value of psi, there is a value of rw, zw and D
412 r_w = (1:length(psi)) ;
413 z_w = (1:length(psi)) ;
414 D = (1:length(psi)) ;
415 for i = 1:length(psi)
416 % define position of wrist in (r,z) plane
417 r_w(i) = r - l3*cos(psi(i)) ; %r_w and z_w will be real as cos or sin of a real number is ...
    a real number
418
419 z_w(i) = pz - d1 -l3*sin(psi(i)) ;
420
421 % define D - a placeholder variable for a large combination we derived in
422 % notes
423
424 D(i) = - (r_w(i)^2 + z_w(i)^2 - l1^2 - l2^2) / (2*l1*l2) ;
425 D(i) = round(D(i), 7) ;% rounding D avoids fake imaginary numbers due to rounding
    % errors.
426
427 end
428 %% get only real values
429
430 psi_real = [] ;
431 r_w_real = [] ;
432 z_w_real = [] ;
433 D_real = [] ;
434 for i = 1:length(psi)
435 if imag(sqrt(1-D(i)^2)) == 0 % for real values, append to real lists.
436 psi_real(end + 1) = psi(i);%round(psi(i),7) ; % round each list to get zeros rather ...
    than e^-10 or something.
437 r_w_real(end + 1) = r_w(i);%round(r_w(i),7) ;
438 z_w_real(end + 1) = z_w(i);%round(z_w(i),7) ;
439 D_real(end + 1) = D(i);%round(D(i),7) ;
440 end
441 end
442
443 % we can see these four lists MUST have same length.
444 % disp(psi_real)
445 % disp(D_real)
446 % disp(z_w_real)
447 % disp(r_w_real)
448
449 %% Find q3 possibilities
450 % there are two possibilities for each value of D depending on
451 for i = 1:length(D_real)
452
453 q3(end+1) = atan2( sqrt(1-D_real(i)^2), -D_real(i)) ;
454 q3(end+1) = atan2(-sqrt(1-D_real(i)^2), -D_real(i)) ;
455 end
456
457 % disp("q3")
458 % disp(q3)
459 %% Explaining list order.
460 % matlab indexes from 1 not 0.
461 % if D has 3 values, we can see q3 will have 6, where
462 % D(1) corresponds to q3(1 & 2)
463 % D(2) corresponds to q3(3 & 4)

```

```

464 % D(3) corresponds to q3(5 & 6)
465 % we can see D(i) corresponds to values
466 % q3(2*i -1) and q3(2*i)
467 % we need to ensure this is *consistant* across all q lists below.
468 %% Find q2
469
470 % two options based on q3
471 for i = 1:length(D.real)
472     % for each r_w,z_w value, find first q2 value using first q3 value.
473     % Switching order here corrects angles... why?
474     q2(end+1) = atan2(z_w.real(i),r_w.real(i)) + atan2(l2*sin(q3(2*i)), l1+l2*cos(q3(2*i))) ;
475     % add +1 to
476     q2(end+1) = atan2(z_w.real(i),r_w.real(i)) + atan2(l2*sin(q3(2*i-1)), ...
477         l1+l2*cos(q3(2*i-1))) ;
478
479 % disp("q2")
480 % disp(q2)
481 %% Find q4
482
483 % two options based on two sets of q2,q3
484 % use psi = q2 + q3 + q4
485 for i = 1:length(D.real)
486     q4(end+1) = psi.real(i) - q2(2*i -1) - q3(2*i -1) +pi/2 ;
487     q4(end+1) = psi.real(i) - q2(2*i) - q3(2*i) +pi/2 ;
488 end
489
490 % disp("q4")
491 % disp(q4)
492 %% Remove duplicate values
493 % make a matrix out of the lists where one row is one solution
494
495 % we know matrix will have initial size: no. of items in q2,3 or 4 x 5
496 % angles
497 Solution_Matrix = zeros(length(q2), 5);
498 for i = 1:length(q2)
499     Solution_Matrix(i,1) = q1 ;
500     Solution_Matrix(i,2) = q2(i) ;
501     Solution_Matrix(i,3) = q3(i) ;
502     Solution_Matrix(i,4) = q4(i) ;
503     Solution_Matrix(i,5) = q5 ;
504 end
505 % disp(Solution_Matrix)
506 % remove duplicate rows
507 Unique_Solutions = unique(Solution_Matrix,"rows","stable") ;
508 % stable prevents order being changed.
509
510 %% we also know, q4 can't be more than 360 degrees. filter these out.
511 valid_Solutions = [] ;
512 for i = 1:size(Unique_Solutions,1)
513     % condition for a valid solution: q4 is within 0 - 2pi, q2 is more
514     % than 45% to avoid arm hitting the box
515     if abs(Unique_Solutions(i,4)) < 2*pi && Unique_Solutions(i,2) > pi/3
516         valid_Solutions = [valid_Solutions; Unique_Solutions(i,:)] ;
517     end
518 end
519 % stable prevents order being changed.
520 for i = 1:5
521     IK_OUTPUT(j,i) = valid_Solutions(1,i) ;
522 end
523 end
524
525
526
527 %% Plot the arm trajectory in 3d space
528 syms q1 q2 q3 q4 q5; % angle variables
529
530 % Show our 5 tranformation matrices
531 T_01 =[
532 cos(q1), 0, sin(q1), 0]

```

```

533 [sin(q1), 0, -cos(q1), 0]
534 [      0, 1,      0, d1]
535 [      0, 0,      0, 1]];
536
537 T_12 = [
538 [cos(q2), -sin(q2), 0, l1*cos(q2)]
539 [sin(q2),  cos(q2), 0, l1*sin(q2)]
540 [      0,      0, 1,      0]
541 [      0,      0, 0,      1]];
542
543 T_23 = [
544 [cos(q3), -sin(q3), 0, l2*cos(q3)]
545 [sin(q3),  cos(q3), 0, l2*sin(q3)]
546 [      0,      0, 1,      0]
547 [      0,      0, 0,      1]];
548
549 T_34 = [
550 [cos(q4), 0,  sin(q4),      0]
551 [sin(q4), 0, -cos(q4),      0]
552 [      0, 1,      0,      0]
553 [      0, 0,      0,      1]];
554
555 T_45 = [
556 [cos(q5), -sin(q5), 0, 0]
557 [sin(q5),  cos(q5), 0, 0]
558 [      0,      0, 1, 13]
559 [      0,      0, 0, 1]];
560
561 % Get overall forward kinematics
562 T_e = T_01*T_12*T_23*T_34*T_45;
563
564 % extract the x,y,z elements of these
565 xt_element = T_e(1,4); % the x transformation is the 4th element of the 1st row of our general ...
    transformation matrix, T_e.
566 yt_element = T_e(2,4); % the y transformation is the 4th element of the 2nd row of our general ...
    transformation matrix, T_e.
567 zt_element = T_e(3,4); % the z transformation is the 4th element of the 3rd row of our general ...
    transformation matrix, T_e.
568
569 % Define all our angles in their own arrays
570 q1.set = [] ;
571 q2.set = [] ;
572 q3.set = [] ;
573 q4.set = [] ;
574 q5.set = [] ;
575 for i = 1:size(IK_OUTPUT,1) % i in range number of rows
576     % number of columns always 5 for q1-5
577     q1.set(i) = IK_OUTPUT(i,1) ;
578     q2.set(i) = IK_OUTPUT(i,2) ;
579     q3.set(i) = IK_OUTPUT(i,3) ;
580     q4.set(i) = IK_OUTPUT(i,4) ;
581     q5.set(i) = IK_OUTPUT(i,5) ;
582 end
583
584
585 %
586 %Now simply plot using our FK
587
588
589 % define variable for number of points
590 NoP = size(IK_OUTPUT,1) ; % same as number of sets of q values.
591
592 %base is at origin
593 base = 0 ;
594
595
596 % next joint given by T01 translations (movement from base to joint 1)
597 % need to make into a 4 by 3 array.
598 Tj1_x = zeros(NoP,1) ; % x translation is zero
599 Tj1_y = zeros(NoP,1) ; % y translation is zero

```

```

600 Tj1_z = subs(zeros(NoP,1), 0, d1) ; % z translation is d1
601
602 % next joint given by T_01T_12 translations - movement from base to joint 2
603 % make our general elements for each array
604 movement = T_01*T_12 ;
605 xt_element = movement(1,4) ;
606 yt_element = movement(2,4) ;
607 zt_element = movement(3,4) ;
608 % fill the joint 2 4x1 arrays
609 Tj2_x = [] ;
610 Tj2_y = [] ;
611 Tj2_z = [] ;
612 for i = 1:length(q1.set)
613     xt_value = subs(xt_element, [ q1 q2], [q1.set(i) q2.set(i)]) ;
614     yt_value = subs(yt_element, [ q1 q2], [q1.set(i) q2.set(i)]) ;
615     zt_value = subs(zt_element, [ q1 q2], [q1.set(i) q2.set(i)]) ;
616     Tj2_x(i,1) = xt_value ;
617     Tj2_y(i,1) = yt_value ;
618     Tj2_z(i,1) = zt_value ;
619 end
620 % above worked, now continue for the rest.
621
622
623 % next joint given by T_01T_12T_23 translations - movement from base to joint 3
624 % make our general elements for each array
625 movement = T_01*T_12*T_23 ;
626 xt_element = movement(1,4) ;
627 yt_element = movement(2,4) ;
628 zt_element = movement(3,4) ;
629 % fill the joint 2 4x1 arrays
630 Tj3_x = [] ;
631 Tj3_y = [] ;
632 Tj3_z = [] ;
633 for i = 1:length(q1.set)
634     xt_value = subs(xt_element,[q1 q2 q3], [q1.set(i) q2.set(i) q3.set(i)]) ;
635     yt_value = subs(yt_element,[q1 q2 q3], [q1.set(i) q2.set(i) q3.set(i)]) ;
636     zt_value = subs(zt_element,[q1 q2 q3], [q1.set(i) q2.set(i) q3.set(i)]) ;
637     Tj3_x(i,1) = xt_value ;
638     Tj3_y(i,1) = yt_value ;
639     Tj3_z(i,1) = zt_value ;
640 end
641
642
643 % next joint given by T_01T_12T_23T_34 translations - movement from base to joint 4
644 % note there's no translation in this joint so no change from above.
645 % make our general elements for each array
646 movement = T_01*T_12*T_23*T_34 ;
647 xt_element = movement(1,4) ;
648 yt_element = movement(2,4) ;
649 zt_element = movement(3,4) ;
650 % fill the joint 2 4x1 arrays
651 Tj4_x = [] ;
652 Tj4_y = [] ;
653 Tj4_z = [] ;
654 for i = 1:length(q1.set)
655     xt_value = subs(xt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
656     yt_value = subs(yt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
657     zt_value = subs(zt_element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
658     Tj4_x(i,1) = xt_value ;
659     Tj4_y(i,1) = yt_value ;
660     Tj4_z(i,1) = zt_value ;
661 end
662
663 % final joint given by T_01T_12T_23T_34T_45 translations - movement from base to joint 4
664 % note there's no translation in this joint so no change from above.
665 % make our general elements for each array
666 movement = T_01*T_12*T_23*T_34*T_45 ;
667 xt_element = movement(1,4) ;
668 yt_element = movement(2,4) ;
669 zt_element = movement(3,4) ;

```

```

670 % fill the joint 2 4x1 arrays
671 EE = [] ; % End Effector Positions list
672 for i = 1:length(q1.set)
673     xt.value = subs(xt.element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
674     yt.value = subs(yt.element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
675     zt.value = subs(zt.element,[q1 q2 q3 q4], [q1.set(i) q2.set(i) q3.set(i) q4.set(i)]) ;
676     EE(i,1) = xt.value ; % x values in column 1
677     EE(i,2) = yt.value ; % y values in column 2
678     EE(i,3) = zt.value ; % z values in column 3
679 end
680 %% Plot the graphs
681 figure (2)
682 % plot the object in the way
683 % object has 6 faces, 8 corners.
684 ob_x = [ob_x-i, ob_x-i, ob_x-i, ob_x-i; ob_x-i, ob_x-f, ob_x-f, ob_x-i; ob_x-i, ob_x-f, ob_x-f, ...
        ob_x-i; ob_x-f, ob_x-f, ob_x-f, ob_x-i; ob_x-i, ob_x-i, ob_x-f, ob_x-f; ob_x-i, ob_x-i, ...
        ob_x-f, ob_x-f] ;
685 ob_y = [ob_y-i, ob_y-f, ob_y-f, ob_y-i; ob_y-i, ob_y-i, ob_y-i, ob_y-i; ob_y-f, ob_y-f, ob_y-f, ...
        ob_y-f; ob_y-f, ob_y-i, ob_y-i, ob_y-f; ob_y-i, ob_y-f, ob_y-f, ob_y-i; ob_y-i, ob_y-f, ...
        ob_y-f, ob_y-i] ;
686 ob_z = [ob_z-i, ob_z-i, ob_z-f, ob_z-f; ob_z-i, ob_z-i, ob_z-f, ob_z-f; ob_z-f, ob_z-f, ob_z-i, ...
        ob_z-i; ob_z-f, ob_z-f, ob_z-i, ob_z-i; ob_z-f, ob_z-f, ob_z-f, ob_z-f; ob_z-i, ob_z-i, ...
        ob_z-i, ob_z-i] ;
687 fill3(ob_x(1,:),ob_y(1,:),ob_z(1:,:), 'r')
688 hold on
689 fill3(ob_x(2,:),ob_y(2,:),ob_z(2:,:), 'r')
690 hold on
691 fill3(ob_x(3,:),ob_y(3,:),ob_z(3:,:), 'r')
692 hold on
693 fill3(ob_x(4,:),ob_y(4,:),ob_z(4:,:), 'r')
694 hold on
695 fill3(ob_x(5,:),ob_y(5,:),ob_z(5:,:), 'r')
696 hold on
697 fill3(ob_x(6,:),ob_y(6,:),ob_z(6:,:), 'r')
698 hold on
699 % generate the graph. Note: only 4 joints will be visible as there is
700 % no spatial distinction between joints three and 4.
701 for i = 1:NoP % zeros is the base position, doesn't change.
702     xx = [base; Tj1_x(i); Tj2_x(i); Tj3_x(i); Tj4_x(i); EE(i,1) ] ;
703     yy = [base; Tj1_y(i); Tj2_y(i); Tj3_y(i); Tj4_y(i); EE(i,2) ] ;
704     zz = [base; Tj1_z(i); Tj2_z(i); Tj3_z(i); Tj4_z(i); EE(i,3) ] ;
705     axis([ 0 0.2 0 0.2 0 0.3 ])
706     plot3(xx,yy,zz,'ko-', 'Linewidth',2)
707     hold on % CHANGE TO HOLD ON TO SEE ALL LINES AT ONCE.
708     pause(0.3)
709
710 % label axes, start point, end point.
711 text(EE(1,1) + 0.002,EE(1,2) + 0.002,EE(1,3) + 0.002,'Start') ;
712 text(EE(end,1) + 0.002,EE(end,2) + 0.002,EE(end,3) + 0.002,'End') ;
713
714 % label end effector points
715 text(EE(i,1),EE(i,2),EE(i,3), 'x')
716 end
717 title('Actual Points') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');
718
719
720 figure (3) % end points only
721
722 % plot the object in the way
723 % object has 6 faces, 8 corners
724 ob_x = [ob_x-i, ob_x-i, ob_x-i, ob_x-i; ob_x-i, ob_x-f, ob_x-f, ob_x-i; ob_x-i, ob_x-f, ob_x-f, ...
        ob_x-i; ob_x-f, ob_x-f, ob_x-f, ob_x-i; ob_x-i, ob_x-i, ob_x-f, ob_x-f; ob_x-i, ob_x-i, ...
        ob_x-f, ob_x-f] ;
725 ob_y = [ob_y-i, ob_y-f, ob_y-f, ob_y-i; ob_y-i, ob_y-i, ob_y-i, ob_y-i; ob_y-f, ob_y-f, ob_y-f, ...
        ob_y-f; ob_y-f, ob_y-i, ob_y-i, ob_y-f; ob_y-i, ob_y-f, ob_y-f, ob_y-i; ob_y-i, ob_y-f, ...
        ob_y-f, ob_y-i] ;
726 ob_z = [ob_z-i, ob_z-i, ob_z-f, ob_z-f; ob_z-i, ob_z-i, ob_z-f, ob_z-f; ob_z-f, ob_z-f, ob_z-i, ...
        ob_z-i; ob_z-f, ob_z-f, ob_z-i, ob_z-i; ob_z-f, ob_z-f, ob_z-f, ob_z-f; ob_z-i, ob_z-i, ...
        ob_z-i, ob_z-i] ;
727 fill3(ob_x(1,:),ob_y(1,:),ob_z(1:,:), 'r')

```

```

728 hold on
729 fill3(ob_x(2,:),ob_y(2,:),ob_z(2:),'r')
730 hold on
731 fill3(ob_x(3,:),ob_y(3,:),ob_z(3:),'r')
732 hold on
733 fill3(ob_x(4,:),ob_y(4,:),ob_z(4:),'r')
734 hold on
735 fill3(ob_x(5,:),ob_y(5,:),ob_z(5:),'r')
736 hold on
737 fill3(ob_x(6,:),ob_y(6,:),ob_z(6:),'r')
738 hold on
739
740 for i = 1:NoP
741     plot3(Ee(i,1),Ee(i,2),Ee(i,3),'ko-','Linewidth',2)
742     hold on
743 end
744 title('Actual End Points Only') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');
745 axis([ 0 0.2 0 0.2 0 0.3 ])
746
747
748
749
750 %% Function to Find Where trajectory impacts the object and correct it.
751 function [Obstacle_Navigation] = bug2(first_off_lim_point, final_off_lim_point, EE_list, object)
752
753 % find entry point, correct this to be on object surface
754 % dsearch in allows us to search object surface array for the nearest value
755 % to the entry values we give
756 Entry = EE_list(first_off_lim_point -1, :); % start bug from point before the robot enters the object
757 New_Entry = zeros(1,3) ;
758 New_Entry(1) = object(dsearchn(object(:,1), Entry(1)), 1) ;
759 New_Entry(2) = object(dsearchn(object(:,2), Entry(2)), 2) ;
760 New_Entry(3) = object(dsearchn(object(:,3), Entry(3)), 3) ;
761
762 % find exit point, correct this to be on object surface
763 Exit = EE_list(final_off_lim_point +1, :); % end bug from point after the robot enters the object
764 New_Exit = zeros(1,3) ;
765 New_Exit(1) = object(dsearchn(object(:,1), Exit(1)), 1) ;
766 New_Exit(2) = object(dsearchn(object(:,2), Exit(2)), 2) ;
767 New_Exit(3) = object(dsearchn(object(:,3), Exit(3)), 3) ;
768
769 disp("Meet and leave object surface at: ")
770 disp(New_Entry)
771 disp(New_Exit)
772
773 % offset entry and exit points by a tiny amount so points are not on the
774 % surface but just outside.
775 New_Entry = New_Entry - 0.003 ;
776 New_Exit = New_Exit + 0.003 ;
777
778 % Get 10 points between entry and exit points on the surface of our object.
779 % Define whole object using an array
780 % define points per line for each axis and object array
781 zMotion_points_on_object = 5 ;
782 xyMotion_points_on_object = 10 ;
783 Obstacle_Navigation = zeros(zMotion_points_on_object+xyMotion_points_on_object+1,3) ;
784
785 % populate object array
786 % first go up to top of object (z motion)
787 for i = 1:(zMotion_points_on_object + 1)
788     Obstacle_Navigation(i,1) = New_Entry(1) ;
789     Obstacle_Navigation(i,2) = New_Entry(2) ;
790     Obstacle_Navigation(i,3) = New_Entry(3) + (i-1)*(New_Exit(3) - ...
791         New_Entry(3))/zMotion_points_on_object ;
792 end
793
794
795 %then go across the top of the object (x-y motion)
796 for i = 1:(xyMotion_points_on_object + 1)

```



```
797     Obstacle_Navigation(i+zMotion_points_on_object+1,1) = New_Entry(1) + (i-1)*(New_Exit(1) - ...  
        New_Entry(1))/xyMotion_points_on_object ;  
798     Obstacle_Navigation(i+zMotion_points_on_object+1,2) = New_Entry(2) + (i-1)*(New_Exit(2) - ...  
        New_Entry(2))/xyMotion_points_on_object ;  
799     Obstacle_Navigation(i+zMotion_points_on_object+1,3) = New_Exit(3) ;  
800 end  
801  
802  
803 end
```

Appendix G: Parallel Robot Inverse Kinematics

Code available at: https://github.com/Hinlo/MatLab-Simulation-of-Serial-and-Parallel-Robots/blob/main/FINN_IK_PARALLEL_BOT.m

FINN_IK_PARALLEL_BOT.m

```

1      %% Inverse Kinematics Calculator for LynxMotion Arm
2  clear all %#ok<*CLALL> %#ok<*SAGROW>
3  close all
4  clc
5
6
7  %% STEPS - FULL NOTES ON DRIVE
8  % 1. SPECIFY ROBOT PARAMETERS
9  % 2. SPECIFY DESIRED END POSITION (x_c, y_c, a)
10 % 3. SOLVE IK -> find (j1, j2, j3) for each arm.
11 % 4. PRINT THE 8 POSSIBLE ORIENTATIONS, DISPLAY 2 OF THESE IN FIGURES ONE
12 % AND TWO.
13 % using q for arm 1 angles.
14 % using s for arm 2 angles (skipping r to avoid confusing with radius
15 % values.
16 % using t for arm 3 angles.
17 %% 1. SPECIFY ROBOT PARAMETERS
18 % these values are taken from coursework appendix
19 S.A = 0.17 ; % j1 to j2 link length
20 L = 0.13 ; % j2 to j3 link length
21 r_e = 0.13 ; % radius of circle formed by end effector joints
22 r_b = 0.29 ; % radius of circle formed by base joints
23
24
25
26 %% 2. SPECIFY DESIRED END POSITION (x_c, y_c, a)
27 x_c = 0.25 ;
28 y_c = 0.21 ;
29 a = pi/6 ;
30
31
32 %% arm 1 - joint angles given by q
33 q3 = a + pi/6 ;
34 % two possibilities for q1, q2 as they form an "elbow"
35 q1 = [] ;
36 q2 = [] ;
37 [q1(1), q1(2), q2(1), q2(2)] = find_line_eqn(S.A, L, r_e, x_c, y_c, q3) ;
38
39
40 %% arm 2 - joint angles given by s
41
42 % Find relative position of end effector in this arms frame.
43 x_c2 = x_c - sqrt(3)*r_b ;
44 y_c2 = y_c ;
45
46 % Find angles s1, s2, s3
47 s3 = a + 5*pi/6 ; % q3 + 120 degrees
48 % two possibilities for s1, s2 as they form an "elbow"
49 s1 = [] ;
50 s2 = [] ;
51 [s1(1), s1(2), s2(1), s2(2)] = find_line_eqn(S.A, L, r_e, x_c2, y_c2, s3) ;
52
53
54 %% arm 3 - joint angles given by t
55
56
57 % Find relative position of end effector in this arms frame.
58 x_c3 = x_c - sqrt(3)*r_b/2 ;
59 y_c3 = y_c - (3/2)*r_b ;

```

```

60
61 % Find angles t1, t2, t3
62 t3 = a + 3*pi/2 ; % s3 + 120 degrees, q3 + 240 degrees
63 % two possibilities for t1, t2 as they form an "elbow"
64 t1 = [] ;
65 t2 = [] ;
66 [t1(1), t1(2), t2(1), t2(2)] = find_line_eqn(S_A, L, r_e, x_c3, y_c3, t3) ;
67
68
69
70 %% Check that solutions are real
71 in_workspace = false ;
72 angles_array = [q1(1), q2(1), q1(2), q2(2), q3, s1(1), s2(1), s1(2), s2(2), s3, t1(1), t2(1), ...
    t1(2), t2(2), t3] ;
73
74 if imag(angles_array) == zeros(1,15)
75     in_workspace = true ;
76 end
77 %% Display Solutions
78 % there are 2 solutions for each of the 3 arms
79 % therefore 2^3 total orientations = 8 total
80 if in_workspace == true
81     disp(" For a given position, there are 8 possible orientations: ")
82     disp(" ")
83     %     pause(2)
84
85     %Referring as arms being in orientation 1 or 2, we have:
86     disp("solution 1: ")
87     disp("orientation: 111")
88     disp("*****")
89     disp("Arm 1 (q1, q2, q3): ")
90     disp( q1(1) *180/pi+ " , " + q2(1) *180/pi + " , " + q3 *180/pi)
91     disp(" ")
92     disp("Arm 2 (s1, s2, s3): ")
93     disp( s1(1) *180/pi+ " , " + s2(1) *180/pi + " , " + s3 *180/pi)
94     disp(" ")
95     disp("Arm 3 (t1, t2, t3): ")
96     disp( t1(1) *180/pi+ " , " + t2(1) *180/pi + " , " + t3 *180/pi)
97     disp("*****")
98     disp(" ")
99     %     pause(2)
100
101
102     disp("solution 2: ")
103     disp("orientation: 222")
104     disp("*****")
105     disp("Arm 1 (q1, q2, q3): ")
106     disp( q1(2) *180/pi+ " , " + q2(2) *180/pi + " , " + q3 *180/pi)
107     disp(" ")
108     disp("Arm 2 (s1, s2, s3): ")
109     disp( s1(2) *180/pi+ " , " + s2(2) *180/pi + " , " + s3 *180/pi)
110     disp(" ")
111     disp("Arm 3 (t1, t2, t3): ")
112     disp( t1(2) *180/pi+ " , " + t2(2) *180/pi + " , " + t3 *180/pi)
113     disp("*****")
114     disp(" ")
115     %     pause(2)
116
117
118     disp("solution 3: ")
119     disp("orientation: 121")
120     disp("*****")
121     disp("Arm 1 (q1, q2, q3): ")
122     disp( q1(1) *180/pi+ " , " + q2(1) *180/pi + " , " + q3 *180/pi)
123     disp(" ")
124     disp("Arm 2 (s1, s2, s3): ")
125     disp( s1(2) *180/pi+ " , " + s2(2) *180/pi + " , " + s3 *180/pi)
126     disp(" ")
127     disp("Arm 3 (t1, t2, t3): ")
128     disp( t1(1) *180/pi+ " , " + t2(1) *180/pi + " , " + t3 *180/pi)

```

```

129     disp("*****")
130     disp(" ")
131     %     pause(2)
132
133     disp("solution 4: ")
134     disp("orientation: 112")
135     disp("*****")
136     disp("Arm 1 (q1, q2, q3): ")
137     disp( q1(1) *180/pi+ " " + q2(1) *180/pi + " " + q3 *180/pi)
138     disp(" ")
139     disp("Arm 2 (s1, s2, s3): ")
140     disp( s1(1) *180/pi+ " " + s2(1) *180/pi + " " + s3 *180/pi)
141     disp(" ")
142     disp("Arm 3 (t1, t2, t3): ")
143     disp( t1(2) *180/pi+ " " + t2(2) *180/pi + " " + t3 *180/pi)
144     disp("*****")
145     disp(" ")
146     %     pause(2)
147
148     disp("solution 5: ")
149     disp("orientation: 211")
150     disp("*****")
151     disp("Arm 1 (q1, q2, q3): ")
152     disp( q1(2) *180/pi+ " " + q2(2) *180/pi + " " + q3 *180/pi)
153     disp(" ")
154     disp("Arm 2 (s1, s2, s3): ")
155     disp( s1(1) *180/pi+ " " + s2(1) *180/pi + " " + s3 *180/pi)
156     disp(" ")
157     disp("Arm 3 (t1, t2, t3): ")
158     disp( t1(1) *180/pi+ " " + t2(1) *180/pi + " " + t3 *180/pi)
159     disp("*****")
160     disp(" ")
161     %     pause(2)
162
163     disp("solution 6: ")
164     disp("orientation: 221")
165     disp("*****")
166     disp("Arm 1 (q1, q2, q3): ")
167     disp( q1(2) *180/pi+ " " + q2(2) *180/pi + " " + q3 *180/pi)
168     disp(" ")
169     disp("Arm 2 (s1, s2, s3): ")
170     disp( s1(2) *180/pi+ " " + s2(2) *180/pi + " " + s3 *180/pi)
171     disp(" ")
172     disp("Arm 3 (t1, t2, t3): ")
173     disp( t1(1) *180/pi+ " " + t2(1) *180/pi + " " + t3 *180/pi)
174     disp("*****")
175     disp(" ")
176     %     pause(2)
177
178     disp("solution 7: ")
179     disp("orientation: 212")
180     disp("*****")
181     disp("Arm 1 (q1, q2, q3): ")
182     disp( q1(2) *180/pi+ " " + q2(2) *180/pi + " " + q3 *180/pi)
183     disp(" ")
184     disp("Arm 2 (s1, s2, s3): ")
185     disp( s1(1) *180/pi+ " " + s2(1) *180/pi + " " + s3 *180/pi)
186     disp(" ")
187     disp("Arm 3 (t1, t2, t3): ")
188     disp( t1(2) *180/pi+ " " + t2(2) *180/pi + " " + t3 *180/pi)
189     disp("*****")
190     disp(" ")
191     %     pause(2)
192
193     disp("solution 8: ")
194     disp("orientation: 122")
195     disp("*****")
196     disp("Arm 1 (q1, q2, q3): ")
197     disp( q1(1) *180/pi+ " " + q2(1) *180/pi + " " + q3 *180/pi)
198     disp(" ")

```

```

199     disp("Arm 2 (s1, s2, s3): ")
200     disp( s1(2) *180/pi+ " , " + s2(2) *180/pi + " , " + s3 *180/pi)
201     disp(" ")
202     disp("Arm 3 (t1, t2, t3): ")
203     disp( t1(2) *180/pi+ " , " + t2(2) *180/pi + " , " + t3 *180/pi)
204     disp("*****")
205     disp(" ")
206
207 else
208     disp("Chosen Position is outside of robot workspace.")
209 end
210
211
212 %% Plot the position of the robot
213
214 figure(1) % orientation 1
215 % plot the outer triangle
216 % bottom left, bottom right, top
217 tri_x = [0, sqrt(3)*r_b, (1/2)*sqrt(3)*r_b, 0] ;
218 tri_y = [0, 0, (3/2)*r_b, 0] ;
219 plot(tri_x,tri_y,'ro-','Linewidth',2)
220 hold on
221
222
223 % Plot arm 1
224 xj1 = 0 ;
225 xj2 = xj1 + S_A*cos(q1(1)) ;
226 xj3 = xj2 + L*cos(q2(1)) ;
227 arm1_x = [xj1, xj2, xj3] ;
228
229 yj1 = 0 ;
230 yj2 = yj1 + S_A*sin(q1(1)) ;
231 yj3 = yj2 + L*sin(q2(1)) ;
232 arm1_y = [yj1, yj2, yj3] ;
233
234 plot(arm1_x, arm1_y, 'co-','Linewidth',2)
235 hold on
236
237
238 % Plot arm 2
239 xj1 = sqrt(3)*r_b ;
240 xj2 = xj1 + S_A*cos(s1(1)) ;
241 xj3 = xj2 + L*cos(s2(1)) ;
242 arm2_x = [xj1, xj2, xj3] ;
243
244 yj1 = 0 ;
245 yj2 = yj1 + S_A*sin(s1(1)) ;
246 yj3 = yj2 + L*sin(s2(1)) ;
247 arm2_y = [yj1, yj2, yj3] ;
248
249 plot(arm2_x, arm2_y, 'co-','Linewidth',2)
250 hold on
251
252 % Plot arm 3
253 xj1 = (1/2)*sqrt(3)*r_b ;
254 xj2 = xj1 + S_A*cos(t1(1)) ;
255 xj3 = xj2 + L*cos(t2(1)) ;
256 arm3_x = [xj1, xj2, xj3] ;
257
258 yj1 = (3/2)*r_b ;
259 yj2 = yj1 + S_A*sin(t1(1)) ;
260 yj3 = yj2 + L*sin(t2(1)) ;
261 arm3_y = [yj1, yj2, yj3] ;
262
263 plot(arm3_x, arm3_y, 'co-','Linewidth',2)
264 hold on
265
266 %plot inner triangle
267 inner.triangle_x = [arm1_x(3), arm2_x(3), arm3_x(3), arm1_x(3)] ;
268 inner.triangle_y = [arm1_y(3), arm2_y(3), arm3_y(3), arm1_y(3)] ;

```

```

269 plot(inner_triangle_x, inner_triangle_y, 'bo-', 'Linewidth', 2)
270 hold on
271
272 axis([-0.1 0.6 -0.1 0.6])
273 title('Parallel Robot First Orientation for  $a = \pi/6$ ') ; xlabel('x (m)') ; ylabel('y (m)');
274 subtitle('End Effector at  $x = 0.25$ ,  $y = 0.21$ ');
275
276
277
278 %*****
279 figure(2) % orientation 2
280 % plot the outer triangle
281 % bottom left, bottom right, top
282 tri_x = [0, sqrt(3)*r_b, (1/2)*sqrt(3)*r_b, 0] ;
283 tri_y = [0, 0, (3/2)*r_b, 0] ;
284 plot(tri_x, tri_y, 'ro-', 'Linewidth', 2)
285 hold on
286
287
288 % Plot arm 1
289 xj1 = 0 ;
290 xj2 = xj1 + S_A*cos(q1(2)) ;
291 xj3 = xj2 + L*cos(q2(2)) ;
292 arm1_x = [xj1, xj2, xj3] ;
293
294 yj1 = 0 ;
295 yj2 = yj1 + S_A*sin(q1(2)) ;
296 yj3 = yj2 + L*sin(q2(2)) ;
297 arm1_y = [yj1, yj2, yj3] ;
298
299 plot(arm1_x, arm1_y, 'co-', 'Linewidth', 2)
300 hold on
301
302
303 % Plot arm 2
304 xj1 = sqrt(3)*r_b ;
305 xj2 = xj1 + S_A*cos(s1(2)) ;
306 xj3 = xj2 + L*cos(s2(2)) ;
307 arm2_x = [xj1, xj2, xj3] ;
308
309 yj1 = 0 ;
310 yj2 = yj1 + S_A*sin(s1(2)) ;
311 yj3 = yj2 + L*sin(s2(2)) ;
312 arm2_y = [yj1, yj2, yj3] ;
313
314 plot(arm2_x, arm2_y, 'co-', 'Linewidth', 2)
315 hold on
316
317 % Plot arm 3
318 xj1 = (1/2)*sqrt(3)*r_b ;
319 xj2 = xj1 + S_A*cos(t1(2)) ;
320 xj3 = xj2 + L*cos(t2(2)) ;
321 arm3_x = [xj1, xj2, xj3] ;
322
323 yj1 = (3/2)*r_b ;
324 yj2 = yj1 + S_A*sin(t1(2)) ;
325 yj3 = yj2 + L*sin(t2(2)) ;
326 arm3_y = [yj1, yj2, yj3] ;
327
328 plot(arm3_x, arm3_y, 'co-', 'Linewidth', 2)
329 hold on
330
331 %plot inner triangle
332 inner_triangle_x = [arm1_x(3), arm2_x(3), arm3_x(3), arm1_x(3)] ;
333 inner_triangle_y = [arm1_y(3), arm2_y(3), arm3_y(3), arm1_y(3)] ;
334 plot(inner_triangle_x, inner_triangle_y, 'bo-', 'Linewidth', 2)
335 hold on
336
337 axis([-0.1 0.6 -0.1 0.6])
338 title('Parallel Robot Second Orientation for  $a = \pi/6$ ') ; xlabel('x (m)') ; ylabel('y (m)');

```

```

339 subtitle('End Effector at x = 0.25, y = 0.21');
340 %% Function to find angles 1 & 2 using angle 3 and coords of end effector
341 % arm base frame
342
343 function [angle1_a, angle1_b, angle2_a, angle2_b] = find_line_eqn(S_A, L, r_e, x_ee, y_ee, angle3 )
344
345 % find x3, y3 the position of the joint at angle3
346 x3 = x_ee - r_e*cos(angle3) ;
347 y3 = y_ee - r_e*sin(angle3) ;
348
349 % calculate both angle1 values
350 angle1_a = atan2(y3,x3) + acos((S_A^2 + x3^2 + y3^2 - L^2) / (2*S_A*sqrt(x3^2 + y3^2))) ;
351 angle1_b = atan2(y3,x3) - acos((S_A^2 + x3^2 + y3^2 - L^2) / (2*S_A*sqrt(x3^2 + y3^2))) ;
352
353 % calculate both angle2 values
354
355 % for angle1_a,
356 angle2_a = -(pi - angle1_a - acos((S_A^2 + L^2 - x3^2 - y3^2)/(2*S_A*L))) ;
357 % for angle1_b,
358 angle2_b = +(pi + angle1_b - acos((S_A^2 + L^2 - x3^2 - y3^2)/(2*S_A*L))) ;
359
360 end

```

Appendix H: Parallel Robot WorkSpace

Code available at: https://github.com/Hinlo/MatLab-Simulation-of-Serial-and-Parallel-Robots/blob/main/FINN_WS_PARALLEL_BOT.m

FINN_WS_PARALLEL_BOT.m

```

1      %% Parallel Robot WorkSpace Plotter.
2  clear all %#ok<*CLALL> %#ok<*SAGROW>
3  close all
4  clc
5
6
7  %% STEPS - FULL NOTES ON DRIVE
8  % 1. SPECIFY ROBOT PARAMETERS
9  % 2. SPECIFY RANGE OF DESIRED END POSITIONS (x_c, y_c, a)
10 % 4. SOLVE IK -> find (j1, j2, j3) for each arm AT EACH POSITION.
11 % using q for arm 1 angles.
12 % using s for arm 2 angles (skipping r to avoid confusing with radius
13 % values.
14 % using t for arm 3 angles.
15 % IGNORE IMAGINARY POSITIONS, PLOT REAL POSITIONS TO GET AN OUTLINE OF WS
16 %% 1. SPECIFY ROBOT PARAMETERS
17 % these values are taken from coursework appendix
18 S_A = 0.17 ; % j1 to j2 link length
19 L = 0.13 ; % j2 to j3 link length
20 r_e = 0.13 ; % radius of circle formed by end effector joints
21 r_b = 0.29 ; % radius of circle formed by base joints
22
23
24
25 %% 2. SPECIFY DESIRED END POSITION (x_c, y_c, a)
26 x_c = -0.1:0.01:0.6 ; % test ranges for x and y
27 y_c = -0.1:0.01:0.6 ;
28 a = pi/12; % constant orientation - interesting values for plot: 0, pi/12, pi/6, pi/2.3
29
30 % pre-make lists for the elbow angles
31 q1 = zeros(length(x_c)*length(y_c),2) ; % 2 angles at each position for elbow angles 1 and 2
32 q2 = zeros(length(x_c)*length(y_c),2) ;
33
34 s1 = zeros(length(x_c)*length(y_c),2) ; % Repeat for s angles
35 s2 = zeros(length(x_c)*length(y_c),2) ;
36
37
38 t1 = zeros(length(x_c)*length(y_c),2) ; % Repeat for t angles
39 t2 = zeros(length(x_c)*length(y_c),2) ;
40
41 %% arm 1 - joint angles given by q
42 q3 = a + pi/6 ;
43 count = 1 ;
44 % two possibilities for q1, q2 as they form an "elbow"
45 for i = 1: length(x_c)
46     for j = 1: length(y_c)
47         [q1(count,1), q1(count,2), q2(count,1), q2(count,2)] = find_line_eqn(S_A, L, r_e, x_c(i), ...
48             y_c(j), q3) ;
49         count = count + 1;
50     end
51 end
52 %% arm 2 - joint angles given by s
53
54 % Find relative position of end effector in this arms frame.
55 x_c2 = x_c - sqrt(3)*r_b ;
56 y_c2 = y_c ;
57
58 % Find angles s1, s2, s3

```



```

59 s3 = a + 5*pi/6 ; % q3 + 120 degrees
60 count = 1 ;
61 % two possibilities for s1, s2 as they form an "elbow"
62 for i = 1: length(x_c)
63     for j = 1: length(y_c)
64         [s1(count,1), s1(count,2), s2(count,1), s2(count,2)] = findlineeqn(S_A, L, r_e, x_c2(i), ...
65             y_c2(j), s3) ;
66         count = count + 1;
67     end
68 end
69 %% arm 3 - joint angles given by t
70
71 % Find relative position of end effector in this arms frame.
72 x_c3 = x_c - sqrt(3)*r_b/2 ;
73 y_c3 = y_c - 3*r_b/2 ;
74
75 % Find angles t1, t2, t3
76 t3 = a + 3*pi/2 ; % s3 + 120 degrees, q3 + 240 degrees
77 count = 1 ;
78 % two possibilities for t1, t2 as they form an "elbow"
79 for i = 1: length(x_c)
80     for j = 1: length(y_c)
81         [t1(count,1), t1(count,2), t2(count,1), t2(count,2)] = findlineeqn(S_A, L, r_e, x_c3(i), ...
82             y_c3(j), t3) ;
83         count = count + 1;
84     end
85 end
86 %% Generate array of only real solutions
87 in_workspace = [] ;
88 count = 1 ;
89 for i = 1: length(x_c)
90     for j = 1: length(y_c)
91         angles_array_elbow1 = [x_c(i), y_c(j), q1(count,1), q2(count,1), q3, s1(count,1), ...
92             s2(count,1), s3, t1(count,1), t2(count,1), t3] ;
93         angles_array_elbow2 = [x_c(i), y_c(j), q1(count,2), q2(count,2), q3, s1(count,2), ...
94             s2(count,2), s3, t1(count,2), t2(count,2), t3] ;
95         if isreal(angles_array_elbow1) % boolean check for whether array contains imaginary numbers
96             in_workspace = [in_workspace; angles_array_elbow1] ; % if passed, append the real row
97         end
98         if isreal(angles_array_elbow2) % boolean check for whether array contains imaginary numbers
99             in_workspace = [in_workspace; angles_array_elbow2] ; % if passed, append the real row
100         end
101         count = count + 1;
102     end
103 end
104 disp(in_workspace)
105
106 % print statement to let user know if no real solutions
107 if isempty(in_workspace)
108     disp("Chosen value of *a* has no real workspace!")
109 end
110
111 % get only the position values from these.
112 workspace_positions = [in_workspace(:,1) , in_workspace(:,2) ] ;
113
114 %% Display Solutions on Graph
115 figure (1)
116 % plot the outer triangle
117 % bottom left, bottom right, top
118 tri_x = [0, sqrt(3)*r_b, (1/2)*sqrt(3)*r_b, 0] ;
119 tri_y = [0, 0, (3/2)*r_b, 0] ;
120 plot(tri_x, tri_y, 'ro-', 'Linewidth', 2)
121 hold on
122
123 % Plot the individual points

```

```

125 plot(workspace_positions(:,1), workspace_positions(:,2), "kx")
126 hold on
127
128
129 % Plot the workspace outline
130 outline = convhull(workspace_positions);
131 plot(workspace_positions(outline,1), workspace_positions(outline,2), "c" )
132
133 legend("Robot Base", "End Effector Positions", "Workspace Area")
134 axis([-0.1 0.6 -0.1 0.6])
135 title('Parallel Robot WorkSpace for a = ', num2str(a)) ; xlabel('x (m)') ; ylabel('y (m)');
136
137 %% Function to find angles 1 & 2 using angle 3 and coords of end effector
138 % arm base frame
139
140 function [angle1.a, angle1.b, angle2.a, angle2.b] = findline_eqn(S.A, L, r.e, x.ee, y.ee, angle3 )
141
142 % find x3, y3 the position of the joint at angle3
143 x3 = x.ee - r.e*cos(angle3) ;
144 y3 = y.ee - r.e*sin(angle3) ;
145
146 % calculate both angle1 values
147 angle1.a = atan2(y3,x3) + acos((S.A^2 + x3^2 + y3^2 - L^2) / (2*S.A*sqrt(x3^2 + y3^2))) ;
148 angle1.b = atan2(y3,x3) - acos((S.A^2 + x3^2 + y3^2 - L^2) / (2*S.A*sqrt(x3^2 + y3^2))) ;
149
150 % calculate both angle2 values
151
152 % for angle1.a,
153 angle2.a = -(pi - angle1.a - acos((S.A^2 + L^2 - x3^2 - y3^2)/(2*S.A*L))) ;
154 % for angle1.b,
155 angle2.b = +(pi + angle1.b - acos((S.A^2 + L^2 - x3^2 - y3^2)/(2*S.A*L))) ;
156
157 end

```

REFERENCES

- Finlo Heath (Dec. 2022). *My GitHub Repository: MatLab-Simulation-of-Serial-and-Parallel-Robots*. URL: <https://github.com/Hinlo/MatLab-Simulation-of-Serial-and-Parallel-Robots>.
- Griffin, Robert J et al. (2018). “Straight-Leg Walking Through Underconstrained Whole-Body Control”. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5747–5754. DOI: 10.1109/ICRA.2018.8460751.
- lynxmotion* (n.d.). URL: <http://www.lynxmotion.com/c-130-a15d.aspx>.
- Szep, Cristian et al. (2009). “Kinematics, workspace, design and accuracy analysis of RPRPR medical parallel robot”. In: *2009 2nd Conference on Human System Interactions*, pp. 75–80. DOI: 10.1109/HSI.2009.5090957.
- Wang, Hang et al. (2014). “Research on the Relationship between Classic Denavit-Hartenberg and Modified Denavit-Hartenberg”. In: *2014 Seventh International Symposium on Computational Intelligence and Design*. Vol. 2, pp. 26–29. DOI: 10.1109/ISCID.2014.56.
- Yufka, Alpaslan and Osman Parlaktuna (2009). “Performance comparison of bug algorithms for mobile robots”. In: *Proceedings of the 5th international advanced technologies symposium, Karabuk, Turkey*, pp. 13–15.
- Zhao, Hang et al. (2021). “Singularity Analysis and Singularity Avoidance Trajectory Planning for Industrial Robots”. In: *2021 China Automation Congress (CAC)*, pp. 6164–6169. DOI: 10.1109/CAC53003.2021.9727497.