UFMF4X-15-M - Robotic Fundamentals

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

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.

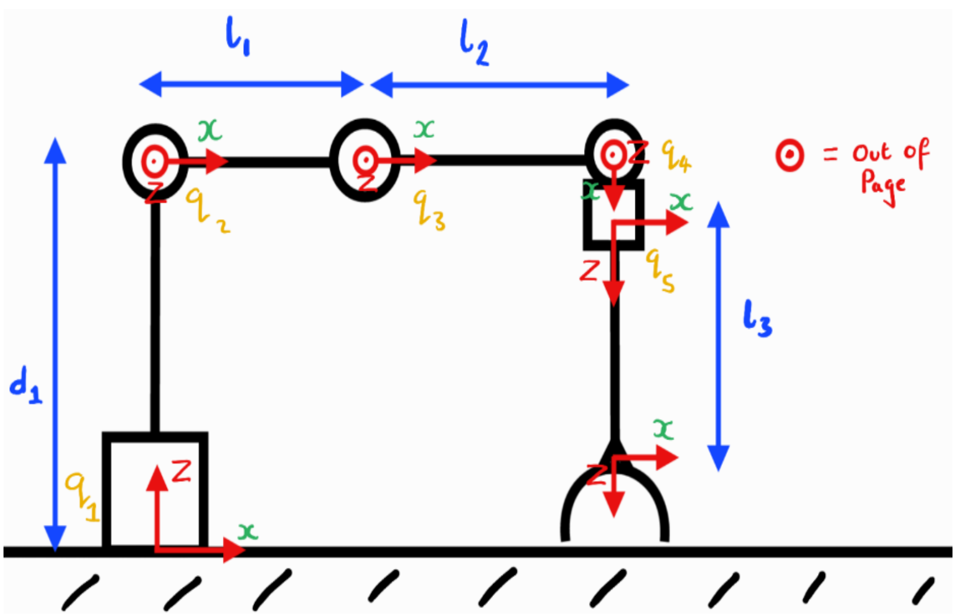
The serial robot simulated in this project is the LynxMotion ALD5 Robot Arm (*lynxmotion* n.d.). This robot has five degrees of freedom (DoF), counting the final rotating prismatic joint which does not affect the spatial position of the robot's end effector. The parallel robot simulated is in the style 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	q1
2	l_1	0	0	q2
3	l_2	0	0	q3
4	0	90	0	q4
5	0	0	l_3	q5

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

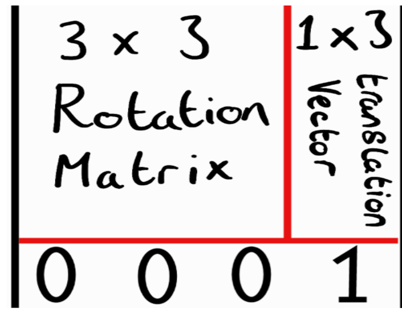
FIG. 1. (a) LynxMotion robot arm annotated with the z-axis of rotation at each joint in the $x-z$ plane of the base. The robot is drawn in it's zero point configuration, meaning that each angle $q1-5=0$. From this Diagram, table (b) is generated and substituted into the transformation matrix given in 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



(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 $q1, q3, q5 = 0$ and $q2, q4 = \pi/2$, 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.

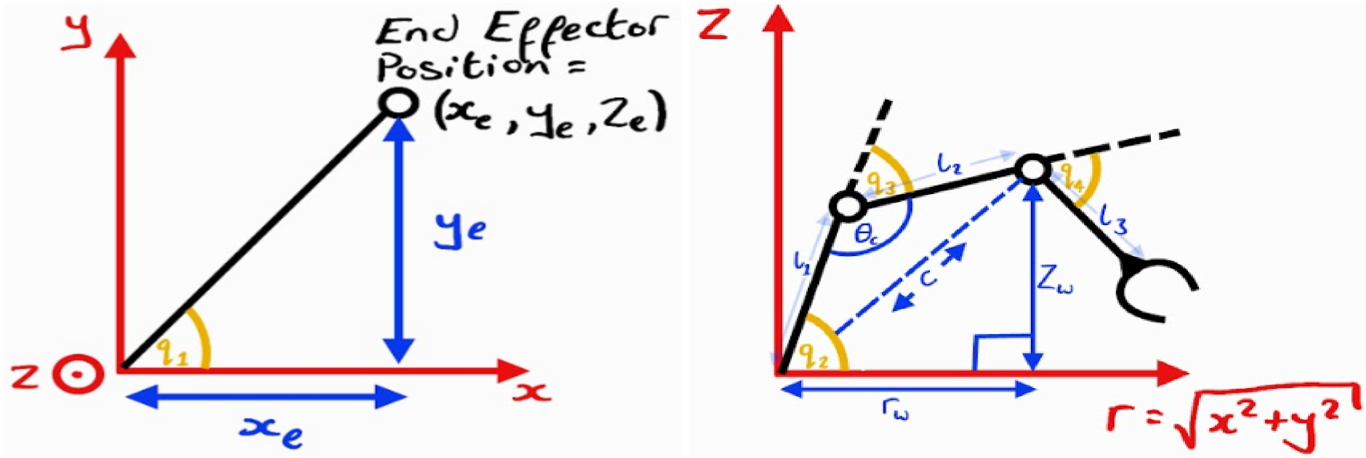
B. WorkSpace

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 ($q1 - q5$). This can be done entirely using trigonometric analysis of the robot; this is illustrated in FIG. 3.

$q1$ is derived using FIG. 3 (a). $q5$ 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.



(a) Top-Down View

(b) Side-On View

FIG. 3. The top down view allows one to find $q1$, the side on view allows for one to directly find $q2$ and $q3$. $q4$ is then found using the equation for ψ.

The remaining angles, $q2, q3, q4$ 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

$$\psi = q2 + q3 + q4$$

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 II D. 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 II D 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 it's base is taken as the origin in that arm's frame, and the position of the end effector is shifted to it's coordinate in that arm's frame.

The angle of the arm to the end effector is the third joint angle, $q3$ and is fixed based on the end effector angle, a. For the first arm, $q3 = a + \frac{\pi}{6}$ 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 it's edge and the vector to it's 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.

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

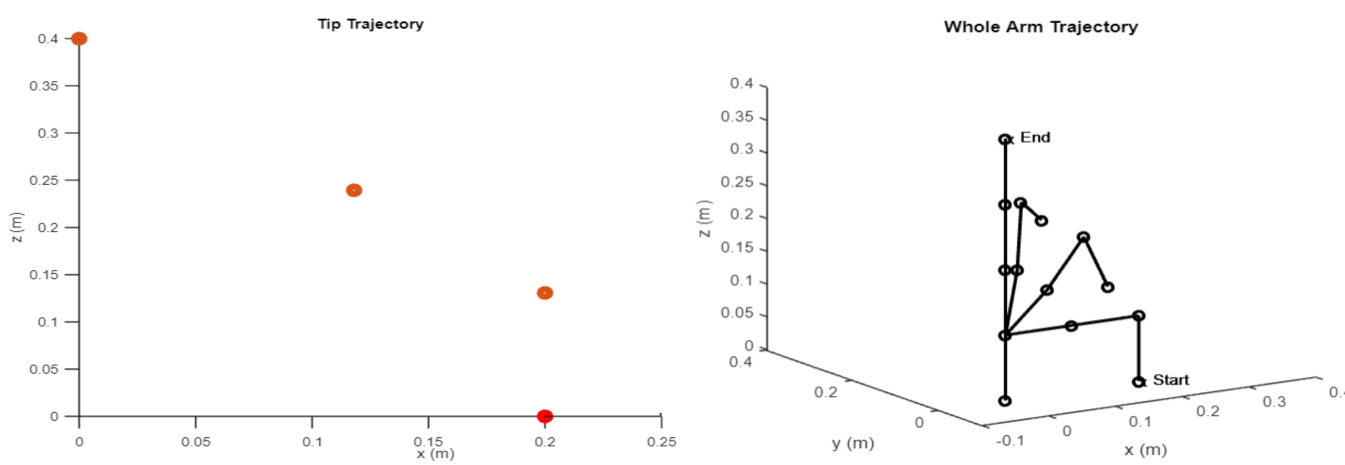
(a) T_e for $q1 = q2 = q3 = q4 = q5 = 0$

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

(b) T_e for $q1 = 0, q2 = 90, q3 = 0, q4 = 90, q5 = 0$

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:



(a) Movement of the End Effector Position in the $x - z$ plane

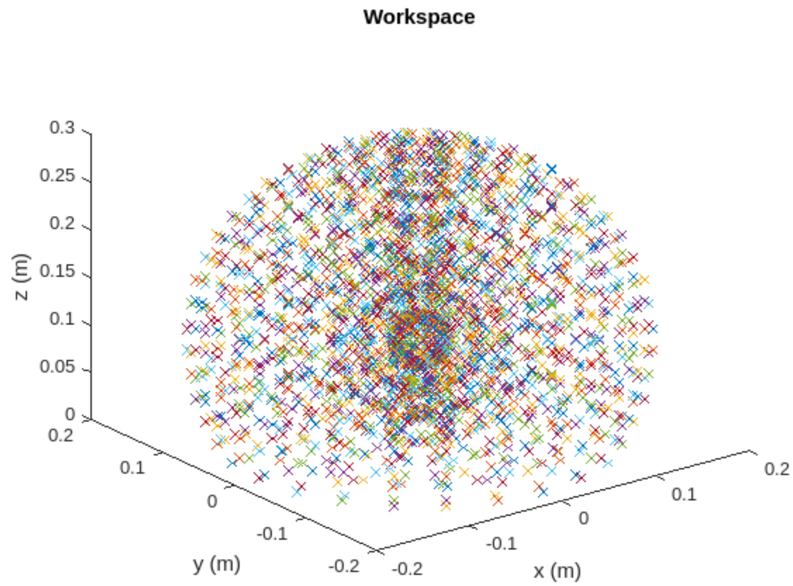
(b) Movement of the Whole Arm In Three Dimensions

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.

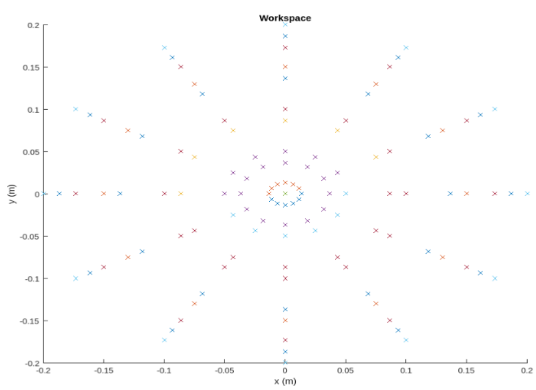
FIG. 5 shows the robot beginning at the configuration given by the matrix shown in FIG. 4 (a) and ending at the configuration given by the matrix in FIG. 4 (b)

B. WorkSpace

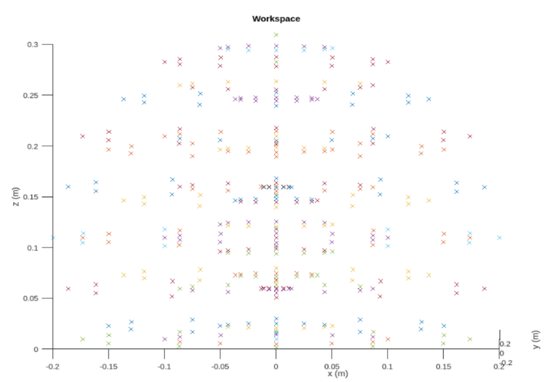
FIG. 6 Shows the comprehensive workspace of the LynxMotion arm, operating on a flat surface. The workspace is hemispherical since the robot cannot pass through the ground. For the main figure, FIG. 6 (a), each joint is incremented in steps of 15 degrees, steps of 30 degrees are used for (b) and (c); 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.



(a) 3D WorkSpace View



(b) Top Down View ($x - y$ plane)



(c) Side On View ($x - z$ plane)

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:

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

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

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

$q3$ 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:

```
solution 1:
     0     0     0     0     0
```

(a) $(x, y, z) = (0.2, 0, 0)$

```
solution 1:
     0    90     0    90     0
```

(b) $(x, y, z) = (0, 0, 0.4)$

```
solution 1:
   45.0000   25.2300   25.4500    9.3100         0
```

(c) $(x, y, z) = (0.17, 0.17, 0.17)$

```
solution 1:
   63.4300    4.1300   63.9900   -8.1200         0
```

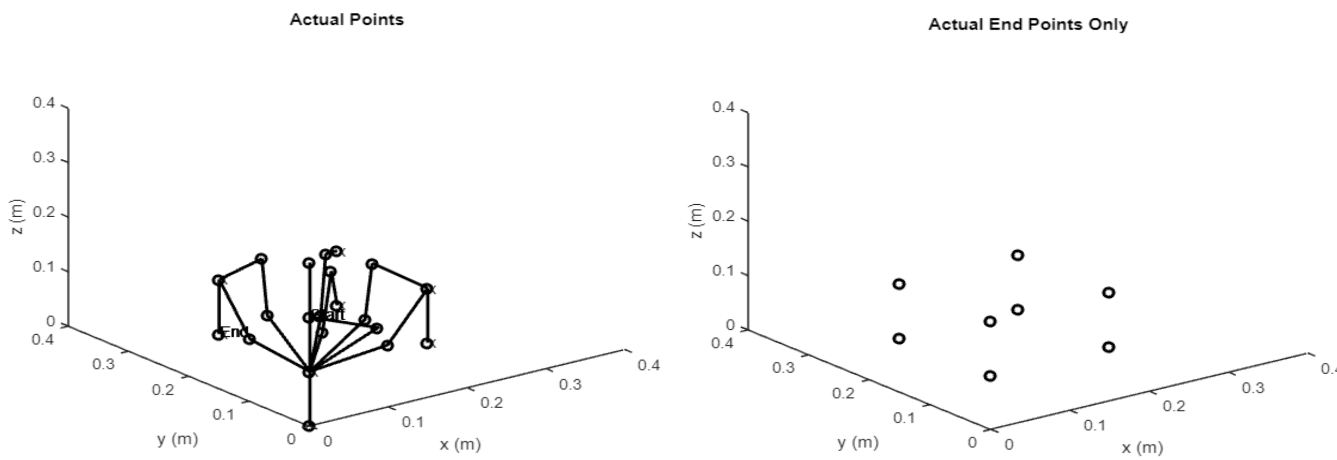
(d) $(x, y, z) = (0.1, 0.2, 0.15)$

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.



(a) Full LynxMotion Arm

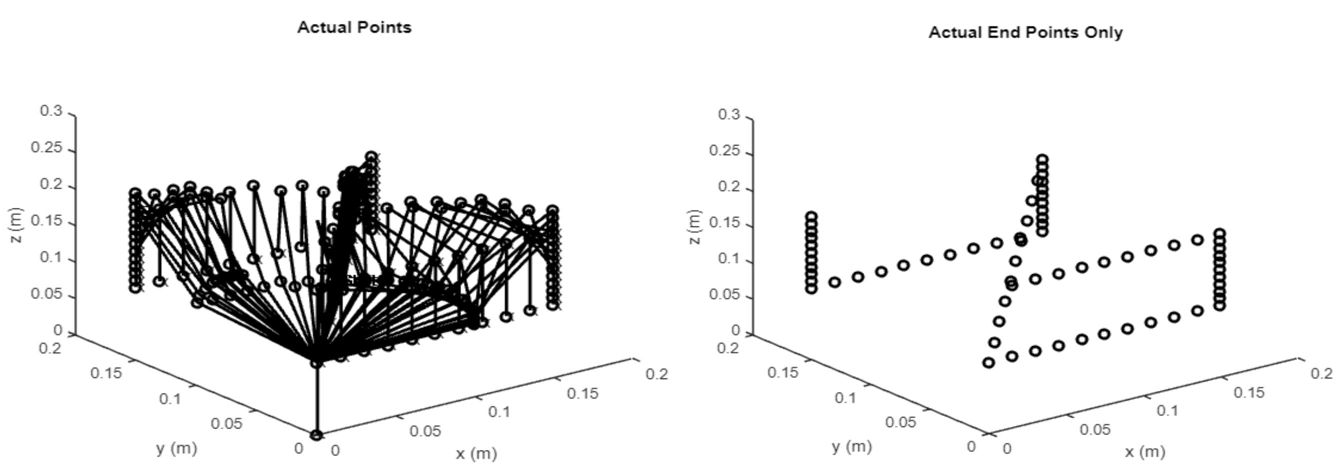
(b) End Effector Position Only

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



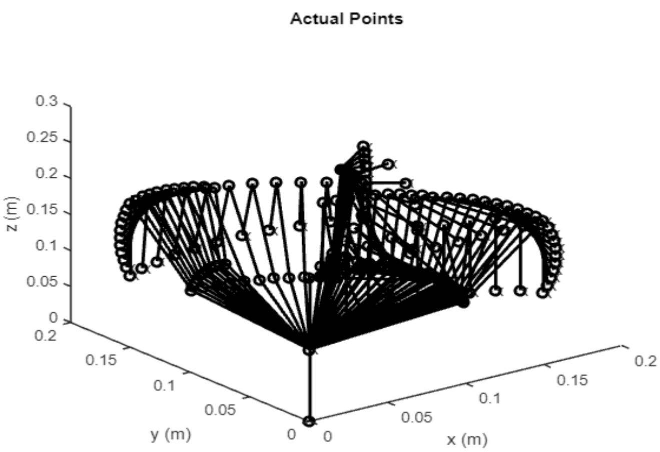
(a) Full LynxMotion Arm

(b) End Effector Position Only

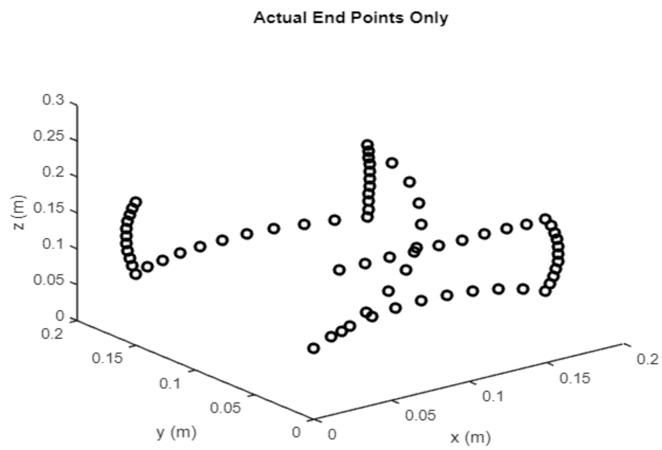
FIG. 9. The robot simulated moving in a linear path between each required vertex point. (a) shows the full arm, while (b) confirms the endpoint position only. The robot begins at $(x, y, z) = (0.0, 0.0, 0.2)$ and finishes at $(x, y, z) = (0.0, 0.15, 0.2)$.

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



(a) Full LynxMotion Arm

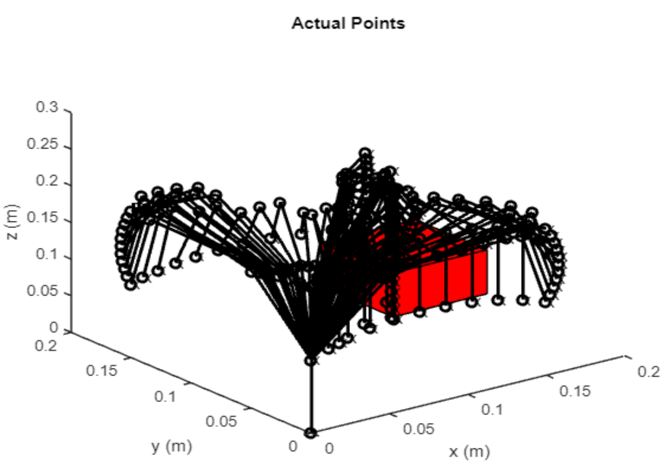


(b) End Effector Position Only

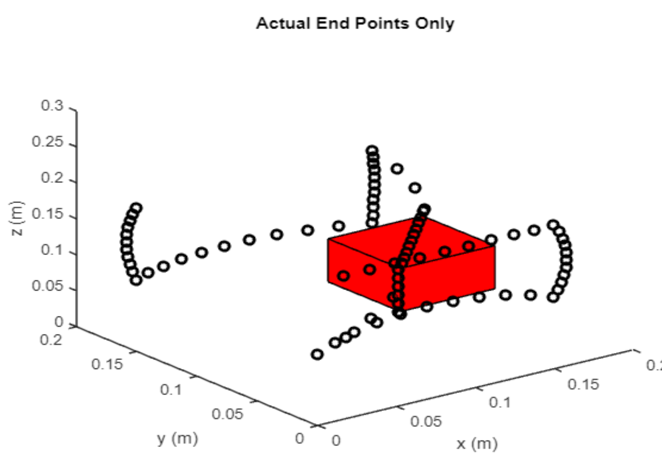
FIG. 10. The robot simulated moving in an arcing path between each required vertex point. (a) shows the full arm, while (b) confirms the endpoint position only. The robot begins at $(x, y, z) = (0.0, 0.0, 0.2)$ and finishes at $(x, y, z) = (0.0, 0.15, 0.2)$.

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

3. Obstacle Avoidance



(a) Full LynxMotion Arm



(b) End Effector Position Only

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 it's path to move around the red cube.

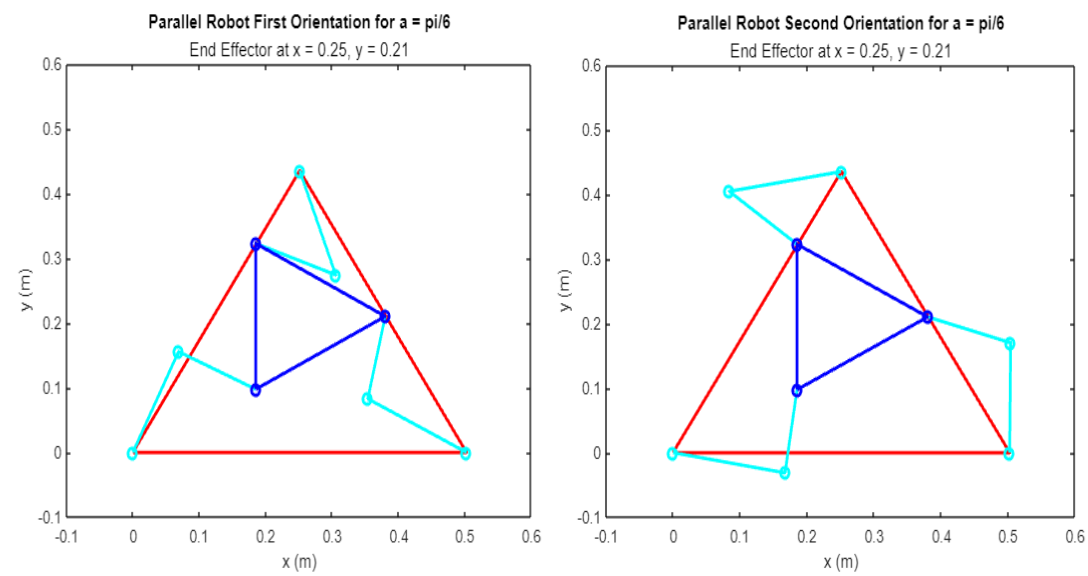
The red cuboid interrupts the arc of the robot. Once it reaches the obstacle, the robot alters it's path and traverses the obstacle surface. The solution is general, the obstacle position may be moved and the robot will still traverse across it. While inefficient, it is necessary to re-run the entire 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.



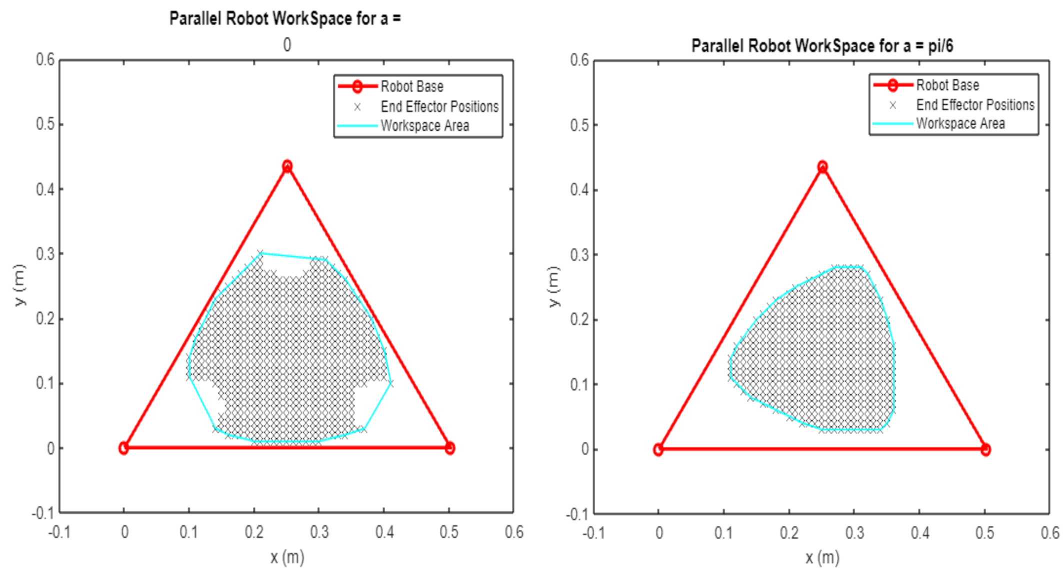
(a) First Configuration

(b) Second Configuration

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 it's 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.



(a) $a = 0$ (b) $a = \frac{\pi}{6}$

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.

```
% Define our set of angles for each joint
q1_set = [ 0 20 40 45 ]'*pi/180 ; % base can spin full 360 degrees
q2_set = [ 0 30 60 90]'*pi/180 ; % first p joint can move 0 to 180
q3_set = [ 0 10 20 30 ]'*pi/180 ; % second p joint can move -180 to 180
q4_set = [ 0 -10 -20 -30]'*pi/180 ; % third p joint can move -180 to 180
q5_set = [ 0 90 180 30 ]'*pi/180 ; % claw can spin full 360 degrees

% Robot hits straight arm singularity only if q2 = q4 = 90 and q3 = 0,
% as d1 link must always point straight up.

for i = 1:length(q2_set)
  if q2_set(i) == 90 && q4_set(i) == 90 && q3_set == 0
    q2_set(i) = 95
    q4_set(i) = 85
    q2_set(i) = 5
  end
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 $Det(J) = 0$, an example is given in FIG. 15. The code in FIG.s 14 & 15 are developed from the Appendix A simulation.

```
% Check each end effector point to ensure it is not at a singularity. Throw an error if it is.
% End effector position and orientation is defined by T_e transformation matrix.

% Use matlab's native jacobian computation.
Jacobian_of_arm = jacobian([T_e],[q1,q2,q3,q4,q5])
% Calculate the general determinant
Det_J = det(Jacobian_of_arm)

% check each value of T_e_specified to see if Det(Jacobian_of_arm) == 0
for i = 1:length(q1_set(i)
  Det_J_Specified = 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)])

  % If it is, stop the simulation and throw an error message.
  if Det_J_Specified == 0
    error('One of your robot orientation angle sets (position %s) results in a singularity! ...
        Please change it and try again.', i)
  end
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.

```
T_e =

[sin(q1)*sin(q5) + cos(q5)*(cos(q4)*(cos(q1)*cos(q2)*cos(q3) -
cos(q1)*sin(q2)*sin(q3)) - sin(q4)*(cos(q1)*cos(q2)*sin(q3) +
cos(q1)*cos(q3)*sin(q2))),
cos(q5)*sin(q1) - sin(q5)*(cos(q4)*
(cos(q1)*cos(q2)*cos(q3) - cos(q1)*sin(q2)*sin(q3)) - sin(q4)*
(cos(q1)*cos(q2)*sin(q3) + cos(q1)*cos(q3)*sin(q2))),
cos(q4)* (cos(q1)*cos(q2)*sin(q3) + cos(q1)*cos(q3)*sin(q2)) + sin(q4)*
(cos(q1)*cos(q2)*cos(q3) - cos(q1)*sin(q2)*sin(q3)),
l3*(cos(q4)*(cos(q1)*cos(q2)*sin(q3) + cos(q1)*cos(q3)*sin(q2)) + sin(q4)*
(cos(q1)*cos(q2)*cos(q3) - cos(q1)*sin(q2)*sin(q3))) + l1*cos(q1)*cos(q2) +
l2*cos(q1)*cos(q2)*cos(q3) - l2*cos(q1)*sin(q2)*sin(q3)]

[- cos(q1)*sin(q5) - cos(q5)*(cos(q4)*(sin(q1)*sin(q2)*sin(q3) -
cos(q2)*cos(q3)*sin(q1)) + sin(q4)*(cos(q2)*sin(q1)*sin(q3) +
cos(q3)*sin(q1)*sin(q2))),
sin(q5)*(cos(q4)*(sin(q1)*sin(q2)*sin(q3) -
cos(q2)*cos(q3)*sin(q1)) + sin(q4)*(cos(q2)*sin(q1)*sin(q3) +
cos(q3)*sin(q1)*sin(q2))) - cos(q1)*cos(q5),
cos(q4)*(cos(q2)*sin(q1)*sin(q3) + cos(q3)*sin(q1)*sin(q2)) - sin(q4)*
(sin(q1)*sin(q2)*sin(q3) - cos(q2)*cos(q3)*sin(q1)),
l3*(cos(q4)*(cos(q2)*sin(q1)*sin(q3) + cos(q3)*sin(q1)*sin(q2)) - sin(q4)*
(sin(q1)*sin(q2)*sin(q3) - cos(q2)*cos(q3)*sin(q1))) + l1*cos(q2)*sin(q1) +
l2*cos(q2)*cos(q3)*sin(q1) - l2*sin(q1)*sin(q2)*sin(q3)]

[cos(q5)*(cos(q4)*(cos(q2)*sin(q3) + cos(q3)*sin(q2)) + sin(q4)*(cos(q2)*cos(q3) - sin(q2)*sin(q3))),
-sin(q5)*(cos(q4)*(cos(q2)*sin(q3) + cos(q3)*sin(q2)) + sin(q4)*(cos(q2)*cos(q3) - sin(q2)*sin(q3))),
sin(q4)*(cos(q2)*sin(q3) + cos(q3)*sin(q2)) - cos(q4)*(cos(q2)*cos(q3) - sin(q2)*sin(q3)),
d1 + l1*sin(q2) - l3*(cos(q4)*(cos(q2)*cos(q3) - sin(q2)*sin(q3)) - sin(q4)*
(cos(q2)*sin(q3) + cos(q3)*sin(q2))) + l2*cos(q2)*sin(q3) + l2*cos(q3)*sin(q2)]

[ 0, 0, 0, 1 ]
```

The full forward kinematics file is given here:

```
%% lynxmotion arm has 5 dof!
% NOTE: Defined angles start from x = 0, rotating +ve or -ve direction, about the z
% axis always!!
%#ok<*NOPTS>  do not pester me about semi colons
%#ok<*SAGROW> do not pester me about lists
% click line numbers to enforce a pause at that line during run.
% remember, cos(x) is radians, cosd(x) would be degrees.
% remember, arrays count from 1 not 0 in matlab.
% clear the output
clear all %#ok<*CLALL>
close all
clc

%% DH TABLE
% a    alpha   d   theta
% 0    90      d1  q1
% l1   0       0   q2
% l2   0       0   q3
% 0    90      0   q4
```

```
% 0    0       l3  q5
%% Set our variables
syms d1 l1 l2 l3; % length variables
syms q1 q2 q3 q4 q5; % angle variables
%% Show our 5 tranformation matrices
T_01 =[
[cos(q1), 0,  sin(q1),  0]
[sin(q1), 0, -cos(q1),  0]
[      0, 1,        0, d1]
[      0, 0,        0,  1]];

T_12 =[
[cos(q2), -sin(q2), 0, l1*cos(q2)]
[sin(q2),  cos(q2), 0, l1*sin(q2)]
[      0,        0, 1,          0]
[      0,        0, 0,          1]];

T_23 = [
[cos(q3), -sin(q3), 0, l2*cos(q3)]
[sin(q3),  cos(q3), 0, l2*sin(q3)]
[      0,        0, 1,          0]
[      0,        0, 0,          1]];

T_34 = [
[cos(q4), 0,  sin(q4),          0]
[sin(q4), 0, -cos(q4),          0]
[      0, 1,        0,          0]
[      0, 0,        0,          1]];

T_45 = [
[cos(q5), -sin(q5), 0,  0]
[sin(q5),  cos(q5), 0,  0]
[      0,        0, 1,  l3]
[      0,        0, 0,  1]];

%% Multiply them in the required order (right to left)
% You can show or comment out the components.

% uncomment to see step by step multiplication of T matrices.
% T_34*T_45
% T_23*T_34*T_45;
% T_12*T_23*T_34*T_45;
T_e = T_01*T_12*T_23*T_34*T_45;

% Show our final transformation matrix
T_e

% now get a specified result, for example:

T_e_Specified = subs(T_e, [q1, q2, q3, q4, q5, l1, l2, l3, d1], ...
    [0*(pi/180),90*(pi/180),0*(pi/180), 0*(pi/180), 0*(pi/180), 1, 1, 1, 1])

%% The following code simulates the forward kinematics of a the lynxmotion
% arm using my forward kinematics calculated above.
% Figure 2 shows its movement from start to end
% position, Figure 1 shows the location of its end effector at points of
% its trajectory and Figure 3 shows the maximum potential workspace of the
% arm's end effector.

disp('The following code simulates the forward kinematics of a simple 2DOF')
disp('serial manipulator. Figure 2 shows ts movement from start to end position,')
disp('Figure 1 shows the location of its end effector at points of its trajectory')
disp('and Figure 3 shows the maximum potential workspace of its end effector')

%% A series of joint angles
% The following variables are defined in the form of column-vectors with
% 4 rows each. Each row represents a different position (angle) of the joint.
% e.g. initiatlly we hae 60 degrees for q1 and -30 for q2.
% USING _set TO MARK WHEN WE HAVE DEFINED OUR SYMBOLIC VARIABLES.
q1_set = [ 0 20 40 45 ]'*pi/180 ; % base can spin full 360 degrees
```

```
q2_set = [ 0 30 60 90]'*pi/180 ; % first p joint can move 0 to 180
q3_set = [ 0 10 20 30 ]'*pi/180 ; % second p joint can move -180 to 180
q4_set = [ 0 -10 -20 -30]'*pi/180 ; % third p joint can move -180 to 180
q5_set = [ 0 90 180 30 ]'*pi/180 ; % claw can spin full 360 degrees


%% Links Lengths
d1_set = 0.1 ; % example lengths, start with all as 10cm
l1_set = 0.1 ;
l2_set = 0.1 ;
l3_set = 0.1 ;

%% Trigonometric abbreviations
% not required at the moment but see orginal uwe script if you want to add
% them.


%% Tip position
% These equations are taken from our general T_e matrix above.

% CHECK: I think these are going to be the x,y,z translation values from
% T_E and ignore the rot values for orientation of the end effector.
xt_element = T_e(1,4); % the x transformation is the 4th element of the 1st row of our general ...
    transformation matrix, T_e.
yt_element = T_e(2,4); % the y transformation is the 4th element of the 2nd row of our general ...
    transformation matrix, T_e.
zt_element = T_e(3,4); % the z transformation is the 4th element of the 3rd row of our general ...
    transformation matrix, T_e.

% to add the angles and lengths, we will use a quick loop.
% lengths are just one value, but angles have an array, so we'll loop
% through the angle arrays.
xt = [];
for i = 1:length(q1_set)
    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.
    xt(i,1) = xt_value ;
end



% repeat for y
yt = [];
for i = 1:length(q1_set)
    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.
    yt(i,1) = yt_value ;
end



% repeat for z
zt = [];
for i = 1:length(q1_set)
    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.
    zt(i,1) = zt_value ;
end

% pt is a 4 by 3 double as required for the formatting below.
pt = [ xt yt zt ] ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Plot the trajectory of the end-effector
figure (1)
%line below just dictates where the image appears on the screen in matlab
%desktop
set(1,'position',[680 558 560 420])


```

```

% IN 3D...
plot3(pt(1,1), pt(1,2), pt(1,3), 'ro')       % plot the first position of the robot's end effector
hold on
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
title('Tip Trajectory') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');


%% Plot the robotic arm, in 4 different positions
% this currently just moves q1, the base. Do we want to rotate more than
% one joint? We defo don't want to rotate just the base.
figure (2)
%line below just dictates where the image appears on the screen in matlab
%desktop
set(2,'position',[116 190 560 420])

%base is at origin
base = 0 ;


% next joint given by T01 translations (movement from base to joint 1)
% need to make into a 4 by 3 array.
Tj1_x = zeros(4,1) ; % x translation is zero
Tj1_y = zeros(4,1) ; % y translation is zero
Tj1_z = subs(zeros(4,1), 0, d1_set);  % z translation is d1

% next joint given by T_01T_12 translations - movement from base to joint 2
% make our general elements for each array
movement = T_01*T_12 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
Tj2_x = [] ;
Tj2_y = [] ;
Tj2_z = [] ;
for i = 1:length(q1_set)
    xt_value = subs(xt_element, [l1 d1 q1 q2], [l1_set d1_set q1_set(i) q2_set(i)]) ;
    yt_value = subs(yt_element, [l1 d1 q1 q2], [l1_set d1_set q1_set(i) q2_set(i)]) ;
    zt_value = subs(zt_element, [l1 d1 q1 q2], [l1_set d1_set q1_set(i) q2_set(i)]) ;
    Tj2_x(i,1) = xt_value ;
    Tj2_y(i,1) = yt_value ;
    Tj2_z(i,1) = zt_value ;
end
% above worked, now continue for the rest.


% next joint given by T_01T_12T_23 translations - movement from base to joint 3
% make our general elements for each array
movement = T_01*T_12*T_23 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
Tj3_x = [] ;
Tj3_y = [] ;
Tj3_z = [] ;
for i = 1:length(q1_set)
    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)]) ;
    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)]) ;
    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)]) ;
    Tj3_x(i,1) = xt_value ;
    Tj3_y(i,1) = yt_value ;
    Tj3_z(i,1) = zt_value ;
end

```

```
% next joint given by T_01T_12T_23T_34 translations - movement from base to joint 4
% note there's no translation in this joint so no change from above.
% make our general elements for each array
movement = T_01*T_12*T_23*T_34 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
Tj4_x = [] ;
Tj4_y = [] ;
Tj4_z = [] ;
for i = 1:length(q1_set)
    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)]) ;
    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)]) ;
    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)]) ;
    Tj4_x(i,1) = xt_value ;
    Tj4_y(i,1) = yt_value ;
    Tj4_z(i,1) = zt_value ;
end

% We already calculated our end effector positions - that's the pt array.





    % generate the graph. Note: only 4 joints will be visible as there is
    % no spatial distinction between joins three and 4.
for i = 1:4 % zeros is the base position, doesn't change.
    xx = [base; Tj1_x(i); Tj2_x(i); Tj3_x(i); Tj4_x(i); pt(i,1) ] ;
    yy = [base; Tj1_y(i); Tj2_y(i); Tj3_y(i); Tj4_y(i); pt(i,2) ] ;
    zz = [base; Tj1_z(i); Tj2_z(i); Tj3_z(i); Tj4_z(i); pt(i,3) ] ;

    pause(1) % pause long enough that we see the first position
    plot3(xx,yy,zz,'ko-','Linewidth',2)
    axis equal
    hold on

    % label axes, start point, end point.
    xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)')
    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') ;
    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') ;
    axis([ -0.1 0.4 -0.1 0.4 0 0.4 ])
    hold off % CHANGE TO HOLD ON TO SEE ALL LINES AT ONCE.
    pause(2)
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Workspace - once working, set to finer increments to get workspace plot.

% Here we must extend the range of the set angles to encompass all possible
% angles.

% SETTING ARRAYS: start:interval:end. x = 1:2:7 -> x = [1,3,5,7]
% to be able to plot, keep intervals consistant.
q1_set = (0:30:360)*pi/180  ;% base angle can go 0 to 360, using intervals of 5 atm.
q2_set = (-90:30:90)*pi/180 ; % joint has range -90 to 90
q3_set = (-180:30:180)*pi/180 ; % joint has range -180 to 180
q4_set = (-180:30:180)*pi/180 ; % joint has range -180 to 180
q5_set = 0; % rot of end effector has range 0 to 360 but this doesn't
% affect workspace.
% we don't care about q5 as workspace is about how far the robot
% can reach, the final orientation of the end effector doesn't matter
% therefore, set q5 as a constant.
```

```

%% Angles Full Range of motion
% Not realistic but worth knowing

% q1_set = 0:30:360 ; % base angle can go 0 to 360, using intervals of 5 atm.
% q2_set = -90:30:90 ; % joint has range -90 to 90
% q3_set = -180:30:180 ; % joint has range -180 to 180
% q4_set = -180:30:180 ; % joint has range -180 to 180
% q5_set = 0; % rot of end effector has range 0 to 360 but this doesn't
%% Plot the workspace of the robot
figure (3)
%line below just dictates where the image appears on the screen in matlab
%desktop
set(3,'position',[1243 190 560 420])


for i = 1:length(q1_set)    % for q1
    for j = 1:length(q2_set)  % for q2
        for k = 1:length(q3_set) % for q3
            for l = 1:length(q4_set) % for q4
                % define our x y z values for the given q 1-4 values
                xwork = subs(xt_element,[d1 l1 l2 l3 q1 q2 q3 q4 q5], [d1_set l1_set l2_set l3_set ...
                    q1_set(1,i) q2_set(1,j) q3_set(1,k) q4_set(1,l) q5_set]); % substitute in the ...
                    set values.
                ywork = subs(yt_element,[d1 l1 l2 l3 q1 q2 q3 q4 q5], [d1_set l1_set l2_set l3_set ...
                    q1_set(1,i) q2_set(1,j) q3_set(1,k) q4_set(1,l) q5_set]); % we still want our ...
                    end effector positions
                zwork = subs(zt_element,[d1 l1 l2 l3 q1 q2 q3 q4 q5], [d1_set l1_set l2_set l3_set ...
                    q1_set(1,i) q2_set(1,j) q3_set(1,k) q4_set(1,l) q5_set]); % given by subbing ...
                    into the element eqns.
                % plot this point!
                if zwork < 0 || abs( q2_set(1,j) + q3_set(1,k) + q4_set(1,l) ) > 2*pi % skip ...
                    plotting this point if the z value is less than zero, e.g arm goes into the ...
                    table. or if psi greater than 360
                   continue
                end
%                 disp("position set: " + num2str(i)  + ", point : " + num2str(j) + num2str(k) + ...
    num2str(l) )
%                 disp("x: " + double(xwork) + ", y: " +double(ywork) + ", z: " + double(zwork) + ...
    ", psi: " + round((q2_set(1,j) + q3_set(1,k) + q4_set(1,l)),2,"decimals") + ", mu: " + ...
    q5_set*180/pi)
                plot3(xwork,ywork,zwork,'x');
                hold on
            end
        end
    end
        % this can take some time, so display current percentage
        % done while you wait.
        percent_done = (i*j*k*l)/(length(q1_set)*length(q2_set)*length(q3_set)*length(q4_set))*100
end

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

```
   %% Inverse Kinematics Calculator for LynxMotion Arm
% key is to maintain list order across lists of each angle.
clear all %#ok<*CLALL> %#ok<*SAGROW>
close all
clc

%% Links Length
d1 = 0.1 ;
l1 = 0.1 ;
l2 = 0.1 ;
l3 = 0.1 ;

%% Desired position of end-effector - SET ME

% Cartesian Coords for x,y,z values.
px = 0.17;
py = 0.17 ;
pz = 0.17;

% mu and psi angles for orientation
mu = 0; % mu is rotation of end effector relative to "wrist"

% list of all possible psi values. they will be narrowed down later to only
% real values. technically could go negative angle but it just ends up
% repeating. e.g +90 is same as -270 degrees logically.
psi = 0:pi/6:2*pi; % RADIANS
psi = psi - pi/2; % angular correction to line up with zero point of FK.

% psi is angle between end effector and negative-z-axis, based on FK.
% psi = q2 + q3 + q4
% End effector desired position and orientation given by array End_Effector
End_Effector = [ px py pz ]' ;
% disp('Desired position =')
% disp("x,y,z")
% disp(End_Effector)
% disp("mu")
% disp(mu)
if norm(End_Effector) > l1+l2+l3+d1  % workspace is above ground
    error('desired position is out of the workspace')
end


%% %%%%%%%%%%%% Inverse Kinematics of LynxMotion Arm %%%%%%%%%%%%%%%%
% For details on calculations, view notes in ReadMe.
% We must find values for the 5 joint angles below.
sym q1 ;  % single value
q2 = [] ; % potentially many values
q3 = [] ; % potentially many values
q4 = [] ; % potentially many values
sym q5 ;  % single value

%% Simple angles to find:

% Find q1
q1 = atan2(py,px) ;
% Find q5
q5 = mu ;

%% Define extra variables to help us find q2,3,4
```

```
% define (r,z) plane
r = sqrt(px^2 + py^2) ; % r is hypotenues in x-y plane.

% For each value of psi, there is a value of rw, zw and D
r_w = (1:length(psi)) ;
z_w = (1:length(psi)) ;
D = (1:length(psi)) ;
for i = 1:length(psi)
% define position of wrist in (r,z) plane
    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

    z_w(i) = pz - d1 -l3*sin(psi(i)) ;

    % define D - a placeholder variable for a large combination we derived in
    % notes

    D(i) = - (r_w(i)^2 + z_w(i)^2 - l1^2 - l2^2) / (2*l1*l2) ;
    D(i) = round(D(i), 7) ; % rounding D avoids fake imaginary numbers due to rounding
                    % errors.
end
%% 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.
%% get only real values

psi_real = [] ;
r_w_real = [] ;
z_w_real = [] ;
D_real = [] ;
for i = 1:length(psi)
    if imag(sqrt(1-D(i)^2)) == 0  % for real values, append to real lists.
        psi_real(end + 1) = round(psi(i),7) ; % round each list to get zeros rather than e^-10 or ...
            something.
        r_w_real(end + 1) = round(r_w(i),7) ;
        z_w_real(end + 1) = round(z_w(i),7) ;
        D_real(end + 1)= round(D(i),7) ;
    end
end

% we can see these four lists MUST have same length.
% disp(psi_real)
% disp(D_real)
% disp(z_w_real)
% disp(r_w_real)

%% Find q3 possibilities
% there are two posibilities for each value of D depending on
for i = 1:length(D_real)

    q3(end+1) = atan2( sqrt(1-D_real(i)^2), -D_real(i)) ;
    q3(end+1) = atan2(-sqrt(1-D_real(i)^2), -D_real(i)) ;
end

%  disp("q3")
%  disp(q3)
%% Explaining list order.
% matlab indexes from 1 not 0.
% if D has 3 values, we can see q3 will have 6, where
% D(1) corresponds to q3(1 & 2)
% D(2) corresponds to q3(3 & 4)
% D(3) corresponds to q3(5 & 6)
% we can see D(i) corresponds to values
% q3(2*i -1) and q3(2*i)
% we need to ensure this is *consistant* across all q lists below.
%% Find q2

% two options based on q3
for i = 1:length(D_real)
    % for each r_w,z_w value, find first q2 value using first q3 value
```

```
    q2(end+1) = atan2(z_w_real(i),r_w_real(i)) + atan2(l2*sin(q3(2*i)), l1+l2*cos(q3(2*i))) ;
    % add +1 to
    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))) ;
end
%  disp("q2")
%  disp(q2)
%% Find q4

% two options based on two sets of q2,q3
% use psi = q2 + q3 + q4
% +pi/2 is the geometry correction for zero point of q4
for i = 1:length(D_real) % round to remove zero discrepancies.
    q4(end+1) = psi_real(i) - q2(2*i -1) - q3(2*i -1) +pi/2 ;
    q4(end+1) = psi_real(i) - q2(2*i) - q3(2*i) +pi/2 ;
end


%  disp("q4")
%  disp(q4)
%% Remove duplicate values
% make a matrix out of the lists where one row is one solution

% we know matrix will have initial size: no. of items in q2, 3 or 4 x 5
% angles
Solution_Matrix = zeros(length(q2), 5);
for i = 1:length(q2)
    Solution_Matrix(i,1) = q1 ;
    Solution_Matrix(i,2) = q2(i) ;
    Solution_Matrix(i,3) = q3(i) ;
    Solution_Matrix(i,4) = q4(i) ;
    Solution_Matrix(i,5) = q5 ;
end
% disp(Solution_Matrix)
% remove duplicate rows
Unique_Solutions = unique(Solution_Matrix,"rows","stable");
% stable prevents order being changed.


%% we also know, q4 can't be more than 360 degrees. filter these out.
valid_Solutions = [] ;
for i = 1:size(Unique_Solutions,1)
    if abs(Unique_Solutions(i,4)) < 2*pi
        valid_Solutions = [valid_Solutions; Unique_Solutions(i,:)] ;
    end
end
%% Display Solutions

for i = 1:size(valid_Solutions,1) % for no. of rows aka no. of solutions

    disp("solution " + num2str(i) + ": ")
    disp(round(valid_Solutions(i,:)*180/pi,2,"decimals"))

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

```
  %% LynxMotion Arm Task
% This file uses inverse and forward kinematics to plan and show a task for
% the lynxmotion arm in matlab.

% the chosen task is to draw the vertices of a cuboid in 3d space.

% ***Task Steps***
% 1. define x,y,z positions for 5 or more points that would make a cuboid
% in 3d space.
% 2. use IK to find the q1-5 angle values for the arm at a each point.
% 3. use FK to plot the arm trajectory in 3d space and save the end
% effector points to display the smiley face.
clear all %#ok<*CLALL> %#ok<*SAGROW>
close all
clc

%% 1. define x,y,z positions that make a smiley face in 3d space.


% TASK: plot a cuboid in 3D space.
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]; ...
    [0.15 0.15 0.2]; [0.0 0.15 0.2]; [0.0 0.15 0.1]] ;
% check the plotted points.
figure (1)

for i = 1:size(points_to_plot,1)
    plot3(points_to_plot(i,1), points_to_plot(i,2), points_to_plot(i,3), 'ko-','Linewidth',2)    % ...
        plot the first position of the robot's end effector
    hold on
end
axis([ 0 0.4 0 0.4 0 0.4 ])
title('Desired Points') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');



%% 2. use IK to find the q1-5 angle values for the arm at a each point.
% Inverse Kinematics Calculator for LynxMotion Arm
% key is to maintain list order across lists of each angle.
% Links Length
d1 = 0.1 ;
l1 = 0.1 ;
l2 = 0.1 ;
l3 = 0.1 ;

% Run through the list of point
IK_OUTPUT = zeros(size(points_to_plot,1),5); % holding array for output q angles.
for j = 1:size(points_to_plot,1)
    % Cartesian Coords for x,y,z values.
    px = points_to_plot(j,1) ;
    py = points_to_plot(j,2) ;
    pz = points_to_plot(j,3) ;

% mu and psi angles for orientation
mu = 0; % mu is rotation of end effector relative to "wrist"

% list of all possible psi values. they will be narrowed down later to only
% real values. technically could go negative angle but it just ends up
% repeating. e.g +90 is same as -270 degrees logically.
psi = (0:pi/20:2*pi) ;
```

```
for i = 1: length(psi)
    psi(i) = psi(i) - pi/2;
end


% psi is angle between end effector and negative-z-axis, based on FK.
% psi = q2 + q3 + q4
% End effector desired position and orientation given by array End_Effector
End_Effector = [ px py pz ]' ;
% disp('Desired position =')
% disp("x,y,z")
% disp(End_Effector)
% disp("mu")
% disp(mu)
if norm(End_Effector) > l1+l2+l3+d1  % workspace is above ground
    error('desired position is out of the workspace')
end


%% %%%%%%%%%%%% Inverse Kinematics of LynxMotion Arm %%%%%%%%%%%%%%%%
% For details on calculations, view notes in ReadMe.
% We must find values for the 5 joint angles below.
sym q1 ;  % single value
q2 = [] ; % potentially many values
q3 = [] ; % potentially many values
q4 = [] ; % potentially many values
sym q5 ;  % single value

%% Simple angles to find:

% Find q1
q1 = atan2(py,px) ;
% Find q5
q5 = mu ;

%% Define extra variables to help us find q2,3,4

% define (r,z) plane
r = sqrt(px^2 + py^2) ; % r is hypotenues in x-y plane.

% For each value of psi, there is a value of rw, zw and D
r_w = (1:length(psi)) ;
z_w = (1:length(psi)) ;
D = (1:length(psi)) ;
for i = 1:length(psi)
% define position of wrist in (r,z) plane
    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

    z_w(i) = pz - d1 -l3*sin(psi(i)) ;

    % define D - a placeholder variable for a large combination we derived in
    % notes

    D(i) = - (r_w(i)^2 + z_w(i)^2 - l1^2 - l2^2) / (2*l1*l2) ;
    D(i) = round(D(i), 7)  ;% rounding D avoids fake imaginary numbers due to rounding
                    % errors.
end
%% get only real values

psi_real = [] ;
r_w_real = [] ;
z_w_real = [] ;
D_real = [] ;
for i = 1:length(psi)
    if imag(sqrt(1-D(i)^2)) == 0  % for real values, append to real lists.
        psi_real(end + 1) = psi(i);%round(psi(i),7) ; % round each list to get zeros rather than e^-10 or something.
        r_w_real(end + 1) = r_w(i);%round(r_w(i),7) ;
        z_w_real(end + 1) = z_w(i);%round(z_w(i),7) ;
```

```
        D_real(end + 1)= D(i);%round(D(i),7) ;
    end
end

% we can see these four lists MUST have same length.
%  disp(psi_real)
%  disp(D_real)
%  disp(z_w_real)
%  disp(r_w_real)

%% Find q3 possibilities
% there are two posibilities for each value of D depending on
for i = 1:length(D_real)

    q3(end+1) = atan2( sqrt(1-D_real(i)^2), -D_real(i))  ;
    q3(end+1) = atan2(-sqrt(1-D_real(i)^2), -D_real(i)) ;
end

%  disp("q3")
%  disp(q3)
%% Explaining list order.
% matlab indexes from 1 not 0.
% if D has 3 values, we can see q3 will have 6, where
% D(1) corresponds to q3(1 & 2)
% D(2) corresponds to q3(3 & 4)
% D(3) corresponds to q3(5 & 6)
% we can see D(i) corresponds to values
% q3(2*i -1) and q3(2*i)
% we need to ensure this is *consistant* across all q lists below.
%% Find q2

% two options based on q3
for i = 1:length(D_real)
    % for each r_w,z_w value, find first q2 value using first q3 value.
    % Switching order here corrects angles... why?
    q2(end+1) = atan2(z_w_real(i),r_w_real(i)) + atan2(l2*sin(q3(2*i)), l1+l2*cos(q3(2*i))) ;
    % add +1 to
    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))) ;
end
%  disp("q2")
%  disp(q2)
%% Find q4

% two options based on two sets of q2,q3
% use psi = q2 + q3 + q4
for i = 1:length(D_real)
    q4(end+1) = psi_real(i) - q2(2*i -1) - q3(2*i -1) +pi/2 ;
    q4(end+1) = psi_real(i) - q2(2*i) - q3(2*i) +pi/2 ;
end

%  disp("q4")
%  disp(q4)
%% Remove duplicate values
% make a matrix out of the lists where one row is one solution

% we know matrix will have initial size: no. of items in q2,3 or 4 x 5
% angles
Solution_Matrix = zeros(length(q2), 5);
for i = 1:length(q2)
    Solution_Matrix(i,1) = q1 ;
    Solution_Matrix(i,2) = q2(i) ;
    Solution_Matrix(i,3) = q3(i) ;
    Solution_Matrix(i,4) = q4(i) ;
    Solution_Matrix(i,5) = q5 ;
end
% disp(Solution_Matrix)
% remove duplicate rows
Unique_Solutions = unique(Solution_Matrix,"rows","stable");
% stable prevents order being changed.

```

```
%% we also know, q4 can't be more than 360 degrees. filter these out.
valid_Solutions = [] ;
for i = 1:size(Unique_Solutions,1)
    if abs(Unique_Solutions(i,4)) < 2*pi
        valid_Solutions = [valid_Solutions; Unique_Solutions(i,:)] ;
    end
end

% stable prevents order being changed.
    for i = 1:5
        IK_OUTPUT(j,i) = valid_Solutions(1,i) ;

    end
end
disp("Using IK: Array of q1-5 values:")
disp(IK_OUTPUT)



%% 3. use FK to plot the arm trajectory in 3d space
syms q1 q2 q3 q4 q5; % angle variables

% Show our 5 tranformation matrices
T_01 =[
[cos(q1), 0,  sin(q1),  0]
[sin(q1), 0, -cos(q1),  0]
[      0, 1,        0, d1]
[      0, 0,        0,  1]];

T_12 =[
[cos(q2), -sin(q2), 0, l1*cos(q2)]
[sin(q2),  cos(q2), 0, l1*sin(q2)]
[      0,        0, 1,          0]
[      0,        0, 0,          1]];

T_23 = [
[cos(q3), -sin(q3), 0, l2*cos(q3)]
[sin(q3),  cos(q3), 0, l2*sin(q3)]
[      0,        0, 1,          0]
[      0,        0, 0,          1]];

T_34 = [
[cos(q4), 0,  sin(q4),          0]
[sin(q4), 0, -cos(q4),          0]
[      0, 1,        0,          0]
[      0, 0,        0,          1]];

T_45 = [
[cos(q5), -sin(q5), 0,  0]
[sin(q5),  cos(q5), 0,  0]
[      0,        0, 1, l3]
[      0,        0, 0,  1]];

% Get overall forward kinematics
T_e = T_01*T_12*T_23*T_34*T_45;

% extract the x,y,z elements of these
xt_element = T_e(1,4); % the x transformation is the 4th element of the 1st row of our general ...
    transformation matrix, T_e.
yt_element = T_e(2,4); % the y transformation is the 4th element of the 2nd row of our general ...
    transformation matrix, T_e.
zt_element = T_e(3,4); % the z transformation is the 4th element of the 3rd row of our general ...
    transformation matrix, T_e.

% Define all our angles in their own arrays
q1_set = [] ;
q2_set = [] ;
q3_set = [] ;
q4_set = [] ;
```

```
q5_set = [] ;
for i = 1:size(IK_OUTPUT,1) % i in range number of rows
    % number of columns always 5 for q1-5
    q1_set(i) = IK_OUTPUT(i,1) ;
    q2_set(i) = IK_OUTPUT(i,2) ;
    q3_set(i) = IK_OUTPUT(i,3) ;
    q4_set(i) = IK_OUTPUT(i,4) ;
    q5_set(i) = IK_OUTPUT(i,5) ;
end



% Now simply plot using our FK


% define variable for number of points
NoP = size(IK_OUTPUT,1) ; % same as number of sets of q values.

%base is at origin
base = 0 ;


% next joint given by T01 translations (movement from base to joint 1)
% need to make into a 4 by 3 array.
Tj1_x = zeros(NoP,1) ; % x translation is zero
Tj1_y = zeros(NoP,1) ; % y translation is zero
Tj1_z = subs(zeros(NoP,1), 0, d1) ;  % z translation is d1

% next joint given by T_01T_12 translations - movement from base to joint 2
% make our general elements for each array
movement = T_01*T_12 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
Tj2_x = [] ;
Tj2_y = [] ;
Tj2_z = [] ;
for i = 1:length(q1_set)
    xt_value = subs(xt_element, [ q1 q2], [q1_set(i) q2_set(i)]) ;
    yt_value = subs(yt_element, [ q1 q2], [q1_set(i) q2_set(i)]) ;
    zt_value = subs(zt_element, [ q1 q2], [q1_set(i) q2_set(i)]) ;
    Tj2_x(i,1) = xt_value ;
    Tj2_y(i,1) = yt_value ;
    Tj2_z(i,1) = zt_value ;
end
% above worked, now continue for the rest.


% next joint given by T_01T_12T_23 translations - movement from base to joint 3
% make our general elements for each array
movement = T_01*T_12*T_23 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
Tj3_x = [] ;
Tj3_y = [] ;
Tj3_z = [] ;
for i = 1:length(q1_set)
    xt_value = subs(xt_element,[q1 q2 q3], [q1_set(i) q2_set(i) q3_set(i)]) ;
    yt_value = subs(yt_element,[q1 q2 q3], [q1_set(i) q2_set(i) q3_set(i)]) ;
    zt_value = subs(zt_element,[q1 q2 q3], [q1_set(i) q2_set(i) q3_set(i)]) ;
    Tj3_x(i,1) = xt_value ;
    Tj3_y(i,1) = yt_value ;
    Tj3_z(i,1) = zt_value ;
end


% next joint given by T_01T_12T_23T_34 translations - movement from base to joint 4
```

```
% note there's no translation in this joint so no change from above.
% make our general elements for each array
movement = T_01*T_12*T_23*T_34 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
Tj4_x = [] ;
Tj4_y = [] ;
Tj4_z = [] ;
for i = 1:length(q1_set)
    xt_value = subs(xt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    yt_value = subs(yt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    zt_value = subs(zt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    Tj4_x(i,1) = xt_value ;
    Tj4_y(i,1) = yt_value ;
    Tj4_z(i,1) = zt_value ;
end

% final joint given by T_01T_12T_23T_34T_45 translations - movement from base to joint 4
% note there's no translation in this joint so no change from above.
% make our general elements for each array
movement = T_01*T_12*T_23*T_34*T_45 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
EE = [] ; % End Effector Positions list
for i = 1:length(q1_set)
    xt_value = subs(xt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    yt_value = subs(yt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    zt_value = subs(zt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    EE(i,1) = xt_value ; % x values in column 1
    EE(i,2) = yt_value ; % y values in column 2
    EE(i,3) = zt_value ; % z values in column 3
end

% Check if end effector positions match points to plot.
disp("EE values (should match points to plot): ")
disp(EE)
disp("do they match? 0 = no, 1 = yes: ")
isequal(round(points_to_plot,2,"decimals"),round(EE,2,"decimals"))

figure (2)

    % generate the graph. Note: only 4 joints will be visible as there is
    % no spatial distinction between joints three and 4.
for i = 1:NoP % zeros is the base position, doesn't change.
    xx = [base; Tj1_x(i); Tj2_x(i); Tj3_x(i); Tj4_x(i); EE(i,1) ] ;
    yy = [base; Tj1_y(i); Tj2_y(i); Tj3_y(i); Tj4_y(i); EE(i,2) ] ;
    zz = [base; Tj1_z(i); Tj2_z(i); Tj3_z(i); Tj4_z(i); EE(i,3) ] ;
    axis([ 0 0.4 0 0.4 0 0.4 ])
    plot3(xx,yy,zz,'ko-','Linewidth',2)
    hold on % CHANGE TO HOLD ON TO SEE ALL LINES AT ONCE.
    pause(2)

    % label axes, start point, end point.
    text(EE(1,1) + 0.002,EE(1,2) + 0.002,EE(1,3) + 0.002,'Start') ;
    text(EE(end,1) + 0.002,EE(end,2) + 0.002,EE(end,3) + 0.002,'End') ;

    % label end effector points
    text(EE(i,1),EE(i,2),EE(i,3), 'x')
end
title('Actual Points') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');


figure (3) % end points only
for i = 1:NoP
    plot3(EE(i,1),EE(i,2),EE(i,3),'ko-','Linewidth',2)
    hold on
```

```
end
title('Actual End Points Only') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');
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

```
%% linear motion (straight lines)
clear all %#ok<*CLALL> %#ok<*SAGROW>
close all
clc
% link lengths
d1 = 0.1 ;
l1 = 0.1 ;
l2 = 0.1 ;
l3 = 0.1 ;

% assuming 10 HZ robot, performing each line in 1 second
%% TRYING LINEAR FIRST
%1. get start, end point and line eqn between.
%2. sample function for 10 points between each start and end point
%3. apply IK and FK

%% 1 & 2 -> find eqn for each line and add 10 points to the trajectory.
% function at bottom to do this.

points_per_line = 10 ;
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]] ;
trajectories = zeros((size(points_to_plot,1) - 1)*points_per_line, 3) ;

% Use our function to get the gradient and constant, filter for lines along
% each axis.
for i = 1:(size(points_to_plot,1) - 1)
    [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_
        ;
    for j = 1:points_per_line

        if r_dir == 10
            if i == 1 && j == 1
                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)  ;
                continue
            end
            trajectories(( (i-1)*10+j) ,1) = trajectories(( (i-1)*10+j-1) ,1);
            trajectories(( (i-1)*10+j) ,2) = trajectories(( (i-1)*10+j-1) ,2);
            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)  ;
            continue
        end
        x = points_to_plot(i,1) + r*cos(r_dir) * (j/points_per_line) ;
        y = points_to_plot(i,2) + r*sin(r_dir) * (j/points_per_line) ;
        r_temp = sqrt(x^2 + y^2) ;
        z =  grad*r_temp + const ;
        trajectories(( (i-1)*10+j) ,1) = x ;
        trajectories(( (i-1)*10+j) ,2) = y ;
        trajectories(( (i-1)*10+j) ,3) = z ;

    end
end
disp(trajectories)



%% 3
```

```
for j = 1:size(trajectories,1)
    px = trajectories(j,1) ;
    py = trajectories(j,2) ;
    pz = trajectories(j,3) ;

    % mu and psi angles for orientation
    mu = 0; % mu is rotation of end effector relative to "wrist"
    psi = (0:pi/20:2*pi) ;
    for i = 1: length(psi)
        psi(i) = psi(i) - pi/2;
    end
    End_Effector = [ px py pz ]' ;
    if norm(End_Effector) > l1+l2+l3+d1  % workspace is above ground
        error('desired position is out of the workspace')
    end


    %% %%%%%%%%%%%% Inverse Kinematics of LynxMotion Arm %%%%%%%%%%%%%%%%
    % For details on calculations, view notes in ReadMe.
    % We must find values for the 5 joint angles below.
    sym q1 ;  % single value
    q2 = [] ; % potentially many values
    q3 = [] ; % potentially many values
    q4 = [] ; % potentially many values
    sym q5 ;  % single value

    %% Simple angles to find:

    % Find q1
    q1 = atan2(py,px) ;
    % Find q5
    q5 = mu ;

    %% Define extra variables to help us find q2,3,4

    % define (r,z) plane
    r = sqrt(px^2 + py^2) ; % r is hypotenues in x-y plane.

    % For each value of psi, there is a value of rw, zw and D
    r_w = (1:length(psi)) ;
    z_w = (1:length(psi)) ;
    D = (1:length(psi)) ;
    for i = 1:length(psi)
    % define position of wrist in (r,z) plane
        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

        z_w(i) = pz - d1 -l3*sin(psi(i)) ;

        % define D - a placeholder variable for a large combination we derived in
        % notes

        D(i) = - (r_w(i)^2 + z_w(i)^2 - l1^2 - l2^2) / (2*l1*l2) ;
        D(i) = round(D(i), 7)  ;% rounding D avoids fake imaginary numbers due to rounding
                        % errors.
    end
    %% get only real values

    psi_real = [] ;
    r_w_real = [] ;
    z_w_real = [] ;
    D_real = [] ;
    for i = 1:length(psi)
        if imag(sqrt(1-D(i)^2)) == 0  % for real values, append to real lists.
            psi_real(end + 1) = psi(i);%round(psi(i),7) ; % round each list to get zeros rather ...
                than e^-10 or something.
            r_w_real(end + 1) = r_w(i);%round(r_w(i),7) ;
            z_w_real(end + 1) = z_w(i);%round(z_w(i),7) ;
            D_real(end + 1)= D(i);%round(D(i),7) ;
        end
```

```
end

% we can see these four lists MUST have same length.
%  disp(psi_real)
%  disp(D_real)
%  disp(z_w_real)
%  disp(r_w_real)

%% Find q3 possibilities
% there are two posibilities for each value of D depending on
for i = 1:length(D_real)

    q3(end+1) = atan2( sqrt(1-D_real(i)^2), -D_real(i)) ;
    q3(end+1) = atan2(-sqrt(1-D_real(i)^2), -D_real(i)) ;
end

%  disp("q3")
%  disp(q3)
%% Explaining list order.
% matlab indexes from 1 not 0.
% if D has 3 values, we can see q3 will have 6, where
% D(1) corresponds to q3(1 & 2)
% D(2) corresponds to q3(3 & 4)
% D(3) corresponds to q3(5 & 6)
% we can see D(i) corresponds to values
% q3(2*i -1) and q3(2*i)
% we need to ensure this is *consistant* across all q lists below.
%% Find q2

% two options based on q3
for i = 1:length(D_real)
    % for each r_w,z_w value, find first q2 value using first q3 value.
    % Switching order here corrects angles... why?
    q2(end+1) = atan2(z_w_real(i),r_w_real(i)) + atan2(l2*sin(q3(2*i)), l1+l2*cos(q3(2*i))) ;
    % add +1 to
    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))) ;
end
%  disp("q2")
%  disp(q2)
%% Find q4

% two options based on two sets of q2,q3
% use psi = q2 + q3 + q4
for i = 1:length(D_real)
    q4(end+1) = psi_real(i) - q2(2*i -1) - q3(2*i -1) +pi/2 ;
    q4(end+1) = psi_real(i) - q2(2*i) - q3(2*i) +pi/2 ;
end

%  disp("q4")
%  disp(q4)
%% Remove duplicate values
% make a matrix out of the lists where one row is one solution

% we know matrix will have initial size: no. of items in q2,3 or 4 x 5
% angles
Solution_Matrix = zeros(length(q2), 5);
for i = 1:length(q2)
    Solution_Matrix(i,1) = q1 ;
    Solution_Matrix(i,2) = q2(i) ;
    Solution_Matrix(i,3) = q3(i) ;
    Solution_Matrix(i,4) = q4(i) ;
    Solution_Matrix(i,5) = q5 ;
end
% disp(Solution_Matrix)
% remove duplicate rows
Unique_Solutions = unique(Solution_Matrix,"rows","stable") ;
% stable prevents order being changed.
```

```
    %% we also know, q4 can't be more than 360 degrees. filter these out.
    valid_Solutions = [] ;
    for i = 1:size(Unique_Solutions,1)
        if abs(Unique_Solutions(i,4)) < 2*pi
            valid_Solutions = [valid_Solutions; Unique_Solutions(i,:)] ;
        end
    end
    % stable prevents order being changed.
        for i = 1:5
            IK_OUTPUT(j,i) = valid_Solutions(1,i) ;

        end
end
disp("Using IK: Array of q1-5 values:")
%disp(IK_OUTPUT);



%% 3. use FK to plot the arm trajectory in 3d space
syms q1 q2 q3 q4 q5; % angle variables

% Show our 5 tranformation matrices
T_01 =[
[cos(q1), 0,  sin(q1),  0]
[sin(q1), 0, -cos(q1),  0]
[      0, 1,        0, d1]
[      0, 0,        0,  1]];

T_12 =[
[cos(q2), -sin(q2), 0, l1*cos(q2)]
[sin(q2),  cos(q2), 0, l1*sin(q2)]
[      0,        0, 1,          0]
[      0,        0, 0,          1]];

T_23 = [
[cos(q3), -sin(q3), 0, l2*cos(q3)]
[sin(q3),  cos(q3), 0, l2*sin(q3)]
[      0,        0, 1,          0]
[      0,        0, 0,          1]];

T_34 = [
[cos(q4), 0,  sin(q4),          0]
[sin(q4), 0, -cos(q4),          0]
[      0, 1,        0,          0]
[      0, 0,        0,          1]];

T_45 = [
[cos(q5), -sin(q5), 0,  0]
[sin(q5),  cos(q5), 0,  0]
[      0,        0, 1,  l3]
[      0,        0, 0,  1]];

% Get overall forward kinematics
T_e = T_01*T_12*T_23*T_34*T_45;

% extract the x,y,z elements of these
xt_element = T_e(1,4); % the x transformation is the 4th element of the 1st row of our general ...
    transformation matrix, T_e.
yt_element = T_e(2,4); % the y transformation is the 4th element of the 2nd row of our general ...
    transformation matrix, T_e.
zt_element = T_e(3,4); % the z transformation is the 4th element of the 3rd row of our general ...
    transformation matrix, T_e.

% Define all our angles in their own arrays
q1_set = [] ;
q2_set = [] ;
q3_set = [] ;
q4_set = [] ;
q5_set = [] ;
for i = 1:size(IK_OUTPUT,1) % i in range number of rows
```

```
    % number of columns always 5 for q1-5
    q1_set(i) = IK_OUTPUT(i,1) ;
    q2_set(i) = IK_OUTPUT(i,2) ;
    q3_set(i) = IK_OUTPUT(i,3) ;
    q4_set(i) = IK_OUTPUT(i,4) ;
    q5_set(i) = IK_OUTPUT(i,5) ;
end


%
%Now simply plot using our FK


% define variable for number of points
NoP = size(IK_OUTPUT,1) ; % same as number of sets of q values.

%base is at origin
base = 0 ;


% next joint given by T01 translations (movement from base to joint 1)
% need to make into a 4 by 3 array.
Tj1_x = zeros(NoP,1) ; % x translation is zero
Tj1_y = zeros(NoP,1) ; % y translation is zero
Tj1_z = subs(zeros(NoP,1), 0, d1) ;  % z translation is d1

% next joint given by T_01T_12 translations - movement from base to joint 2
% make our general elements for each array
movement = T_01*T_12 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
Tj2_x = [] ;
Tj2_y = [] ;
Tj2_z = [] ;
for i = 1:length(q1_set)
    xt_value = subs(xt_element, [ q1 q2], [q1_set(i) q2_set(i)]) ;
    yt_value = subs(yt_element, [ q1 q2], [q1_set(i) q2_set(i)]) ;
    zt_value = subs(zt_element, [ q1 q2], [q1_set(i) q2_set(i)]) ;
    Tj2_x(i,1) = xt_value ;
    Tj2_y(i,1) = yt_value ;
    Tj2_z(i,1) = zt_value ;
end
% above worked, now continue for the rest.


% next joint given by T_01T_12T_23 translations - movement from base to joint 3
% make our general elements for each array
movement = T_01*T_12*T_23 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
Tj3_x = [] ;
Tj3_y = [] ;
Tj3_z = [] ;
for i = 1:length(q1_set)
    xt_value = subs(xt_element,[q1 q2 q3], [q1_set(i) q2_set(i) q3_set(i)]) ;
    yt_value = subs(yt_element,[q1 q2 q3], [q1_set(i) q2_set(i) q3_set(i)]) ;
    zt_value = subs(zt_element,[q1 q2 q3], [q1_set(i) q2_set(i) q3_set(i)]) ;
    Tj3_x(i,1) = xt_value ;
    Tj3_y(i,1) = yt_value ;
    Tj3_z(i,1) = zt_value ;
end


% next joint given by T_01T_12T_23T_34 translations - movement from base to joint 4
% note there's no translation in this joint so no change from above.
% make our general elements for each array
```

```
movement = T_01*T_12*T_23*T_34 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
Tj4_x = [] ;
Tj4_y = [] ;
Tj4_z = [] ;
for i = 1:length(q1_set)
    xt_value = subs(xt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    yt_value = subs(yt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    zt_value = subs(zt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    Tj4_x(i,1) = xt_value ;
    Tj4_y(i,1) = yt_value ;
    Tj4_z(i,1) = zt_value ;
end

% final joint given by T_01T_12T_23T_34T_45 translations - movement from base to joint 4
% note there's no translation in this joint so no change from above.
% make our general elements for each array
movement = T_01*T_12*T_23*T_34*T_45 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
EE = [] ; % End Effector Positions list
for i = 1:length(q1_set)
    xt_value = subs(xt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    yt_value = subs(yt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    zt_value = subs(zt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    EE(i,1) = xt_value ; % x values in column 1
    EE(i,2) = yt_value ; % y values in column 2
    EE(i,3) = zt_value ; % z values in column 3
end

% Check if end effector positions match points to plot.
disp("EE values (should match points to plot): ")
disp(EE)
disp("do they match? 0 = no, 1 = yes: ")
isequal(round(trajectories,4,"decimals"),round(EE,4,"decimals"))

figure (2)

    % generate the graph. Note: only 4 joints will be visible as there is
    % no spatial distinction between joints three and 4.
for i = 1:NoP % zeros is the base position, doesn't change.
    xx = [base; Tj1_x(i); Tj2_x(i); Tj3_x(i); Tj4_x(i); EE(i,1) ] ;
    yy = [base; Tj1_y(i); Tj2_y(i); Tj3_y(i); Tj4_y(i); EE(i,2) ] ;
    zz = [base; Tj1_z(i); Tj2_z(i); Tj3_z(i); Tj4_z(i); EE(i,3) ] ;
    axis([ 0 0.2 0 0.2 0 0.3 ])
    plot3(xx,yy,zz,'ko-','Linewidth',2)
    hold on % CHANGE TO HOLD ON TO SEE ALL LINES AT ONCE.
    pause(0.3)

    % label axes, start point, end point.
    text(EE(1,1) + 0.002,EE(1,2) + 0.002,EE(1,3) + 0.002,'Start') ;
    text(EE(end,1) + 0.002,EE(end,2) + 0.002,EE(end,3) + 0.002,'End') ;

    % label end effector points
    text(EE(i,1),EE(i,2),EE(i,3), 'x')
end
title('Actual Points') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');


figure (3) % end points only
for i = 1:NoP
    plot3(EE(i,1),EE(i,2),EE(i,3),'ko-','Linewidth',2)
    hold on
end
title('Actual End Points Only') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');
```

```
axis([ 0 0.2 0 0.2 0 0.3 ])




%% Function to find gradient and constant of line equation
function [grad, const, r_dir, r] = find_line_eqn(x1, x2, y1, y2, z1, z2)

    % find which vector r = in x,y plane magnitude and direction
    % r dir is angle from x axis along r, at the point of x1,y1 in the x,y
    % plane.

    % check for lines along  each axis first
    % line along z
    if (x2 - x1) == 0 && (y2 - y1) == 0
        grad = "null" ;
        const = "null" ;
        r_dir = 10 ; % max value of atan2 is < pi/2 therefore < 10
        r2 = 0 ;
        r1 = 0 ;
        r = 0 ;
        return
    % line along y
    elseif (x2 - x1) == 0
        r1 = y1 ;
        r2 = y2 ;
        r = r2 -r1 ;
        r_dir = pi/2 ;
    % line along x
    elseif (y2 - y1) == 0
        r1 = x1 ;
        r2 = x2 ;
        r = r2 -r1 ;
        r_dir = 0 ;
    else
        r1 = sqrt(x1^2 + y1^2) ;
        r2 = sqrt(x2^2 + y2^2) ;
        r = sqrt((y2-y1)^2 + (x2-x1)^2) ;
        r_dir = atan2(y2-y1, x2-x1) ;
        if r_dir < 0
            r_dir = pi - r_dir;
        end
    end

    grad = (z2-z1)/(r) ;
    const = (z1*r2 - z2*r1)/ (r) ;
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

```
%% Free Trajectories (Arcing motion)

clear all %#ok<*CLALL> %#ok<*SAGROW>
close all
clc
% link lengths
d1 = 0.1 ;
l1 = 0.1 ;
l2 = 0.1 ;
l3 = 0.1 ;

% assuming 10 HZ robot, performing each line in 1 second
%% FREE MOTION
%1. do IK to find angles to reach each point
%2. sample angle intervals to show trajectories
%3. get expanded angles list
%4. run FK.

%% Define points to plot
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]] ;

%% Run IK
for j = 1:size(points_to_plot,1)
    px = points_to_plot(j,1) ;
    py = points_to_plot(j,2) ;
    pz = points_to_plot(j,3) ;

    % mu and psi angles for orientation
    mu = 0; % mu is rotation of end effector relative to "wrist"
    psi = (0:pi/20:2*pi) ;
    for i = 1: length(psi)
        psi(i) = psi(i) - pi/2;
    end
    End_Effector = [ px py pz ]' ;
    if norm(End_Effector) > l1+l2+l3+d1  % workspace is above ground
        error('desired position is out of the workspace')
    end


    %% %%%%%%%%%%%% Inverse Kinematics of LynxMotion Arm %%%%%%%%%%%%%%%%%
    % For details on calculations, view notes in ReadMe.
    % We must find values for the 5 joint angles below.
    sym q1 ;  % single value
    q2 = [] ; % potentially many values
    q3 = [] ; % potentially many values
    q4 = [] ; % potentially many values
    sym q5 ;  % single value

    %% Simple angles to find:

    % Find q1
    q1 = atan2(py,px) ;
    % Find q5
    q5 = mu ;

    %% Define extra variables to help us find q2,3,4

    % define (r,z) plane
```

```
    r = sqrt(px^2 + py^2) ; % r is hypotenues in x-y plane.

    % For each value of psi, there is a value of rw, zw and D
    r_w = (1:length(psi)) ;
    z_w = (1:length(psi)) ;
    D = (1:length(psi)) ;
    for i = 1:length(psi)
    % define position of wrist in (r,z) plane
        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

        z_w(i) = pz - d1 -l3*sin(psi(i)) ;

        % define D - a placeholder variable for a large combination we derived in
        % notes

        D(i) = - (r_w(i)^2 + z_w(i)^2 - l1^2 - l2^2) / (2*l1*l2) ;
        D(i) = round(D(i), 7)  ;% rounding D avoids fake imaginary numbers due to rounding
                        % errors.
    end
    %% get only real values

    psi_real = [] ;
    r_w_real = [] ;
    z_w_real = [] ;
    D_real = [] ;
    for i = 1:length(psi)
        if imag(sqrt(1-D(i)^2)) == 0  % for real values, append to real lists.
            psi_real(end + 1) = psi(i);%round(psi(i),7) ; % round each list to get zeros rather ...
                than e^-10 or something.
            r_w_real(end + 1) = r_w(i);%round(r_w(i),7) ;
            z_w_real(end + 1) = z_w(i);%round(z_w(i),7) ;
            D_real(end + 1)= D(i);%round(D(i),7) ;
        end
    end

    % we can see these four lists MUST have same length.
    %  disp(psi_real)
    %  disp(D_real)
    %  disp(z_w_real)
    %  disp(r_w_real)

    %% Find q3 possibilities
    % there are two posibilities for each value of D depending on
    for i = 1:length(D_real)

        q3(end+1) = atan2( sqrt(1-D_real(i)^2), -D_real(i)) ;
        q3(end+1) = atan2(-sqrt(1-D_real(i)^2), -D_real(i)) ;
    end

    %  disp("q3")
    %  disp(q3)
    %% Explaining list order.
    % matlab indexes from 1 not 0.
    % if D has 3 values, we can see q3 will have 6, where
    % D(1) corresponds to q3(1 & 2)
    % D(2) corresponds to q3(3 & 4)
    % D(3) corresponds to q3(5 & 6)
    % we can see D(i) corresponds to values
    % q3(2*i -1) and q3(2*i)
    % we need to ensure this is *consistant* across all q lists below.
    %% Find q2

    % two options based on q3
    for i = 1:length(D_real)
        % for each r_w,z_w value, find first q2 value using first q3 value.
        % Switching order here corrects angles... why?
        q2(end+1) = atan2(z_w_real(i),r_w_real(i)) + atan2(l2*sin(q3(2*i)), l1+l2*cos(q3(2*i))) ;
        % add +1 to
        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))) ;
    end
    %  disp("q2")
    %  disp(q2)
    %% Find q4

    % two options based on two sets of q2,q3
    % use psi = q2 + q3 + q4
    for i = 1:length(D_real)
        q4(end+1) = psi_real(i) - q2(2*i -1) - q3(2*i -1) +pi/2 ;
        q4(end+1) = psi_real(i) - q2(2*i) - q3(2*i) +pi/2 ;
    end

    %  disp("q4")
    %  disp(q4)
    %% Remove duplicate values
    % make a matrix out of the lists where one row is one solution

    % we know matrix will have initial size: no. of items in q2,3 or 4 x 5
    % angles
    Solution_Matrix = zeros(length(q2), 5);
    for i = 1:length(q2)
        Solution_Matrix(i,1) = q1 ;
        Solution_Matrix(i,2) = q2(i) ;
        Solution_Matrix(i,3) = q3(i) ;
        Solution_Matrix(i,4) = q4(i) ;
        Solution_Matrix(i,5) = q5 ;
    end
    % disp(Solution_Matrix)
    % remove duplicate rows
    Unique_Solutions = unique(Solution_Matrix,"rows","stable") ;
    % stable prevents order being changed.


    %% we also know, q4 can't be more than 360 degrees. filter these out.
    valid_Solutions = [] ;
    for i = 1:size(Unique_Solutions,1)
        if abs(Unique_Solutions(i,4)) < 2*pi
            valid_Solutions = [valid_Solutions; Unique_Solutions(i,:)] ;
        end
    end
    % stable prevents order being changed.
        for i = 1:5
            IK_OUTPUT(j,i) = valid_Solutions(1,i) ;

        end
end
disp("Using IK: Array of q1-5 values:")
disp(IK_OUTPUT);


%% Expand angles to include motion between points

% define points per line and new list
points_per_line = 10 ;
expanded_angles = zeros((size(points_to_plot,1) - 1)*points_per_line, 5) ;

% populate new list
for i = 1:(size(points_to_plot,1) - 1)
    for j = 1:points_per_line
       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
       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
       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
       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
       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
```

```

    end
end
disp(" Array of Expanded Angles: ")
disp(expanded_angles)
%% 3. use FK to plot the arm trajectory in 3d space
syms q1 q2 q3 q4 q5; % angle variables

% Show our 5 tranformation matrices
T_01 =[
[cos(q1), 0,  sin(q1),  0]
[sin(q1), 0, -cos(q1),  0]
[      0, 1,        0, d1]
[      0, 0,        0,  1]];

T_12 =[
[cos(q2), -sin(q2), 0, l1*cos(q2)]
[sin(q2),  cos(q2), 0, l1*sin(q2)]
[      0,        0, 1,          0]
[      0,        0, 0,          1]];

T_23 = [
[cos(q3), -sin(q3), 0, l2*cos(q3)]
[sin(q3),  cos(q3), 0, l2*sin(q3)]
[      0,        0, 1,          0]
[      0,        0, 0,          1]];

T_34 = [
[cos(q4), 0,  sin(q4),          0]
[sin(q4), 0, -cos(q4),          0]
[      0, 1,        0,          0]
[      0, 0,        0,          1]];

T_45 = [
[cos(q5), -sin(q5), 0,  0]
[sin(q5),  cos(q5), 0,  0]
[      0,        0, 1,  l3]
[      0,        0, 0,  1]];

% Get overall forward kinematics
T_e = T_01*T_12*T_23*T_34*T_45;

% extract the x,y,z elements of these
xt_element = T_e(1,4); % the x transformation is the 4th element of the 1st row of our general ...
    transformation matrix, T_e.
yt_element = T_e(2,4); % the y transformation is the 4th element of the 2nd row of our general ...
    transformation matrix, T_e.
zt_element = T_e(3,4); % the z transformation is the 4th element of the 3rd row of our general ...
    transformation matrix, T_e.

% Define all our angles in their own arrays
q1_set = [] ;
q2_set = [] ;
q3_set = [] ;
q4_set = [] ;
q5_set = [] ;
for i = 1:size(expanded_angles,1) % i in range number of rows
    % number of columns always 5 for q1-5
    q1_set(i) = expanded_angles(i,1) ;
    q2_set(i) = expanded_angles(i,2) ;
    q3_set(i) = expanded_angles(i,3) ;
    q4_set(i) = expanded_angles(i,4) ;
    q5_set(i) = expanded_angles(i,5) ;
end


%
%Now simply plot using our FK


```

```
% define variable for number of points
NoP = size(expanded_angles,1) ; % same as number of sets of q values.

%base is at origin
base = 0 ;


% next joint given by T01 translations (movement from base to joint 1)
% need to make into a 4 by 3 array.
Tj1_x = zeros(NoP,1) ; % x translation is zero
Tj1_y = zeros(NoP,1) ; % y translation is zero
Tj1_z = subs(zeros(NoP,1), 0, d1) ;  % z translation is d1

% next joint given by T_01T_12 translations - movement from base to joint 2
% make our general elements for each array
movement = T_01*T_12 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
Tj2_x = [] ;
Tj2_y = [] ;
Tj2_z = [] ;
for i = 1:length(q1_set)
    xt_value = subs(xt_element, [ q1 q2], [q1_set(i) q2_set(i)]) ;
    yt_value = subs(yt_element, [ q1 q2], [q1_set(i) q2_set(i)]) ;
    zt_value = subs(zt_element, [ q1 q2], [q1_set(i) q2_set(i)]) ;
    Tj2_x(i,1) = xt_value ;
    Tj2_y(i,1) = yt_value ;
    Tj2_z(i,1) = zt_value ;
end
% above worked, now continue for the rest.


% next joint given by T_01T_12T_23 translations - movement from base to joint 3
% make our general elements for each array
movement = T_01*T_12*T_23 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
Tj3_x = [] ;
Tj3_y = [] ;
Tj3_z = [] ;
for i = 1:length(q1_set)
    xt_value = subs(xt_element,[q1 q2 q3], [q1_set(i) q2_set(i) q3_set(i)]) ;
    yt_value = subs(yt_element,[q1 q2 q3], [q1_set(i) q2_set(i) q3_set(i)]) ;
    zt_value = subs(zt_element,[q1 q2 q3], [q1_set(i) q2_set(i) q3_set(i)]) ;
    Tj3_x(i,1) = xt_value ;
    Tj3_y(i,1) = yt_value ;
    Tj3_z(i,1) = zt_value ;
end


% next joint given by T_01T_12T_23T_34 translations - movement from base to joint 4
% note there's no translation in this joint so no change from above.
% make our general elements for each array
movement = T_01*T_12*T_23*T_34 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
Tj4_x = [] ;
Tj4_y = [] ;
Tj4_z = [] ;
for i = 1:length(q1_set)
    xt_value = subs(xt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    yt_value = subs(yt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    zt_value = subs(zt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    Tj4_x(i,1) = xt_value ;
```

```
    Tj4_y(i,1) = yt_value ;
    Tj4_z(i,1) = zt_value ;
end

% final joint given by T_01T_12T_23T_34T_45 translations - movement from base to joint 4
% note there's no translation in this joint so no change from above.
% make our general elements for each array
movement = T_01*T_12*T_23*T_34*T_45 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
EE = [] ; % End Effector Positions list
for i = 1:length(q1_set)
    xt_value = subs(xt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    yt_value = subs(yt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    zt_value = subs(zt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    EE(i,1) = xt_value ; % x values in column 1
    EE(i,2) = yt_value ; % y values in column 2
    EE(i,3) = zt_value ; % z values in column 3
end

% Check if end effector positions match points to plot.
% disp("EE values (should match points to plot): ")
% disp(EE)
% disp("do they match? 0 = no, 1 = yes: ")
% isequal(round(points_to_plot,4,"decimals"),round(EE,4,"decimals"))

figure (2)

    % generate the graph. Note: only 4 joints will be visible as there is
    % no spatial distinction between joints three and 4.
for i = 1:NoP % zeros is the base position, doesn't change.
    xx = [base; Tj1_x(i); Tj2_x(i); Tj3_x(i); Tj4_x(i); EE(i,1) ] ;
    yy = [base; Tj1_y(i); Tj2_y(i); Tj3_y(i); Tj4_y(i); EE(i,2) ] ;
    zz = [base; Tj1_z(i); Tj2_z(i); Tj3_z(i); Tj4_z(i); EE(i,3) ] ;
    axis([ 0 0.2 0 0.2 0 0.3 ])
    plot3(xx,yy,zz,'ko-','Linewidth',2)
    hold on % CHANGE TO HOLD ON TO SEE ALL LINES AT ONCE.
    pause(0.3)

    % label axes, start point, end point.
    text(EE(1,1) + 0.002,EE(1,2) + 0.002,EE(1,3) + 0.002,'Start') ;
    text(EE(end,1) + 0.002,EE(end,2) + 0.002,EE(end,3) + 0.002,'End') ;

    % label end effector points
    text(EE(i,1),EE(i,2),EE(i,3), 'x')
end
title('Actual Points') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');


figure (3) % end points only
for i = 1:NoP
    plot3(EE(i,1),EE(i,2),EE(i,3),'ko-','Linewidth',2)
    hold on
end
title('Actual End Points Only') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');
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

```
    %% Bug 2 Obstacle Avoidance
% linear motion (straight lines)
clear all %#ok<*CLALL> %#ok<*SAGROW>
close all
clc
% link lengths
d1 = 0.1 ;
l1 = 0.1 ;
l2 = 0.1 ;
l3 = 0.1 ;

% assuming 10 HZ robot, performing each line in 1 second
% Using free motion rather than linear so as to track curved objects as
% well as straight line ones.

%% Object avoidance
% this file works in the following steps
% 1. object boundaries defined, currently a cube directly in the object
% path. Object task is still to plot the corners of a larger cube as in the
% previous free motion task.
% 2. Ik and Fk are performed for the task, the end effector positions are
% then sampled to see where they first impact and leave the object.
% 3. The end effector path is corrected using the obstacle avoidance function at the end
% to the bug 2 algorithm moving along the edge of the object.
% 4. IK and FK are re run with the new final path to ensure correct q1-5
% values and maintain link lengths.
% 5. Robot motion is plotted to show it still hits the desired points and
% avoids the object.


%% Define the object

% object is a smaller cuboid, detail x,y,z dimensions
% i = initial, f = final
ob_x_i = 0.06 ;
ob_x_f = 0.12 ;

ob_y_i = 0.01 ;
ob_y_f = 0.07 ;

ob_z_i = 0.12 ;
ob_z_f = 0.18 ;

% Define whole object using an array
% define points per line for each axis and object array
points_per_line = 10 ;
object = zeros(points_per_line+1,3) ;

% populate object array
for i = 1:(points_per_line + 1)
    object(i,1) = ob_x_i + (i-1)*(ob_x_f - ob_x_i)/points_per_line ;
    object(i,2) = ob_y_i + (i-1)*(ob_y_f - ob_y_i)/points_per_line ;
    object(i,3) = ob_z_i + (i-1)*(ob_z_f - ob_z_i)/points_per_line ;
end
% disp(object)
%% Define points to plot
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]] ;

```

```

%% Run IK
for j = 1:size(points_to_plot,1)
    px = points_to_plot(j,1) ;
    py = points_to_plot(j,2) ;
    pz = points_to_plot(j,3) ;

    % mu and psi angles for orientation
    mu = 0; % mu is rotation of end effector relative to "wrist"
    psi = (0:pi/20:2*pi) ;
    for i = 1: length(psi)
        psi(i) = psi(i) - pi/2;
    end
    End_Effector = [ px py pz ]' ;
    if norm(End_Effector) > l1+l2+l3+d1  % workspace is above ground
        error('desired position is out of the workspace')
    end


    %% %%%%%%%%%%%% Inverse Kinematics of LynxMotion Arm %%%%%%%%%%%%%%%%
    % For details on calculations, view notes in ReadMe.
    % We must find values for the 5 joint angles below.
    sym q1 ;  % single value
    q2 = [] ; % potentially many values
    q3 = [] ; % potentially many values
    q4 = [] ; % potentially many values
    sym q5 ;  % single value

    %% Simple angles to find:

    % Find q1
    q1 = atan2(py,px) ;
    % Find q5
    q5 = mu ;

    %% Define extra variables to help us find q2,3,4

    % define (r,z) plane
    r = sqrt(px^2 + py^2) ; % r is hypotenues in x-y plane.

    % For each value of psi, there is a value of rw, zw and D
    r_w = (1:length(psi)) ;
    z_w = (1:length(psi)) ;
    D = (1:length(psi)) ;
    for i = 1:length(psi)
    % define position of wrist in (r,z) plane
        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

        z_w(i) = pz - d1 -l3*sin(psi(i)) ;

        % define D - a placeholder variable for a large combination we derived in
        % notes

        D(i) = - (r_w(i)^2 + z_w(i)^2 - l1^2 - l2^2) / (2*l1*l2) ;
        D(i) = round(D(i), 7)  ;% rounding D avoids fake imaginary numbers due to rounding
                        % errors.
    end
    %% get only real values

    psi_real = [] ;
    r_w_real = [] ;
    z_w_real = [] ;
    D_real = [] ;
    for i = 1:length(psi)
        if imag(sqrt(1-D(i)^2)) == 0  % for real values, append to real lists.
            psi_real(end + 1) = psi(i);%round(psi(i),7) ; % round each list to get zeros rather ...
                than e^-10 or something.
            r_w_real(end + 1) = r_w(i);%round(r_w(i),7) ;
            z_w_real(end + 1) = z_w(i);%round(z_w(i),7) ;
```

```
            D_real(end + 1)= D(i);%round(D(i),7) ;
        end
    end

    % we can see these four lists MUST have same length.
    %  disp(psi_real)
    %  disp(D_real)
    %  disp(z_w_real)
    %  disp(r_w_real)

    %% Find q3 possibilities
    % there are two posibilities for each value of D depending on
    for i = 1:length(D_real)

        q3(end+1) = atan2( sqrt(1-D_real(i)^2), -D_real(i)) ;
        q3(end+1) = atan2(-sqrt(1-D_real(i)^2), -D_real(i)) ;
    end

    %  disp("q3")
    %  disp(q3)
    %% Explaining list order.
    % matlab indexes from 1 not 0.
    % if D has 3 values, we can see q3 will have 6, where
    % D(1) corresponds to q3(1 & 2)
    % D(2) corresponds to q3(3 & 4)
    % D(3) corresponds to q3(5 & 6)
    % we can see D(i) corresponds to values
    % q3(2*i -1) and q3(2*i)
    % we need to ensure this is *consistant* across all q lists below.
    %% Find q2

    % two options based on q3
    for i = 1:length(D_real)
        % for each r_w,z_w value, find first q2 value using first q3 value.
        % Switching order here corrects angles... why?
        q2(end+1) = atan2(z_w_real(i),r_w_real(i)) + atan2(l2*sin(q3(2*i)), l1+l2*cos(q3(2*i))) ;
        % add +1 to
        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))) ;
    end
    %  disp("q2")
    %  disp(q2)
    %% Find q4

    % two options based on two sets of q2,q3
    % use psi = q2 + q3 + q4
    for i = 1:length(D_real)
        q4(end+1) = psi_real(i) - q2(2*i -1) - q3(2*i -1) +pi/2 ;
        q4(end+1) = psi_real(i) - q2(2*i) - q3(2*i) +pi/2 ;
    end


    %  disp("q4")
    %  disp(q4)
    %% Remove duplicate values
    % make a matrix out of the lists where one row is one solution

    % we know matrix will have initial size: no. of items in q2,3 or 4 x 5
    % angles
    Solution_Matrix = zeros(length(q2), 5);
    for i = 1:length(q2)
        Solution_Matrix(i,1) = q1 ;
        Solution_Matrix(i,2) = q2(i) ;
        Solution_Matrix(i,3) = q3(i) ;
        Solution_Matrix(i,4) = q4(i) ;
        Solution_Matrix(i,5) = q5 ;
    end
    % disp(Solution_Matrix)
    % remove duplicate rows
    Unique_Solutions = unique(Solution_Matrix,"rows","stable") ;
```

```
    % stable prevents order being changed.


    %% we also know, q4 can't be more than 360 degrees. filter these out.
    valid_Solutions = [] ;
    for i = 1:size(Unique_Solutions,1)
        if abs(Unique_Solutions(i,4)) < 2*pi
            valid_Solutions = [valid_Solutions; Unique_Solutions(i,:)] ;
        end
    end
    % stable prevents order being changed.
        for i = 1:5
            IK_OUTPUT(j,i) = valid_Solutions(1,i) ;

        end
end
% disp("Using IK: Array of q1-5 values:")
% disp(IK_OUTPUT);


%% Expand angles to include motion between points

% define points per line and new list
points_per_line = 10 ;
expanded_angles = zeros((size(points_to_plot,1) - 1)*points_per_line, 5) ;

% populate new list
for i = 1:(size(points_to_plot,1) - 1)
    for j = 1:points_per_line
       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
       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
       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
       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
       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

    end
end
% disp(expanded_angles)
%% 3. use FK to plot the arm trajectory in 3d space and save the end effector points to display ...
    the smiley face.
%IK_OUTPUT = [[45 30 30 30 0]*pi/180]
syms q1 q2 q3 q4 q5; % angle variables

% Show our 5 tranformation matrices
T_01 =[
[cos(q1), 0,  sin(q1),  0]
[sin(q1), 0, -cos(q1),  0]
[      0, 1,        0, d1]
[      0, 0,        0,  1]];

T_12 =[
[cos(q2), -sin(q2), 0, l1*cos(q2)]
[sin(q2),  cos(q2), 0, l1*sin(q2)]
[      0,        0, 1,          0]
[      0,        0, 0,          1]];

T_23 = [
[cos(q3), -sin(q3), 0, l2*cos(q3)]
[sin(q3),  cos(q3), 0, l2*sin(q3)]
[      0,        0, 1,          0]
[      0,        0, 0,          1]];

T_34 = [
[cos(q4), 0,  sin(q4),          0]
[sin(q4), 0, -cos(q4),          0]
```

```
[      0, 1,        0,          0]
[      0, 0,        0,          1]];

T_45 = [
[cos(q5), -sin(q5), 0,  0]
[sin(q5),  cos(q5), 0,  0]
[      0,        0, 1,  13]
[      0,        0, 0,  1]];

% Get overall forward kinematics
T_e = T_01*T_12*T_23*T_34*T_45;

% extract the x,y,z elements of these
xt_element = T_e(1,4); % the x transformation is the 4th element of the 1st row of our general ... transformation matrix, T_e.
yt_element = T_e(2,4); % the y transformation is the 4th element of the 2nd row of our general ... transformation matrix, T_e.
zt_element = T_e(3,4); % the z transformation is the 4th element of the 3rd row of our general ... transformation matrix, T_e.

% Define all our angles in their own arrays
q1_set = [] ;
q2_set = [] ;
q3_set = [] ;
q4_set = [] ;
q5_set = [] ;
for i = 1:size(expanded_angles,1) % i in range number of rows
    % number of columns always 5 for q1-5
    q1_set(i) = expanded_angles(i,1) ;
    q2_set(i) = expanded_angles(i,2) ;
    q3_set(i) = expanded_angles(i,3) ;
    q4_set(i) = expanded_angles(i,4) ;
    q5_set(i) = expanded_angles(i,5) ;
end

% final joint given by T_01T_12T_23T_34T_45 translations - movement from base to joint 4
% note there's no translation in this joint so no change from above.
% make our general elements for each array
movement = T_01*T_12*T_23*T_34*T_45 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
EE = [] ; % End Effector Positions list
for i = 1:length(q1_set)
    xt_value = subs(xt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    yt_value = subs(yt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    zt_value = subs(zt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    EE(i,1) = xt_value ; % x values in column 1
    EE(i,2) = yt_value ; % y values in column 2
    EE(i,3) = zt_value ; % z values in column 3
end

% Check if end effector positions match points to plot.
% disp("EE values: ")
% disp(EE)
% disp("do they match? 0 = no, 1 = yes: ")
% isequal(round(points_to_plot,4,"decimals"),round(EE,4,"decimals"))

 %% Check when obstacle in way
%object
% object is a smaller cuboid, detail x,y,z dimensions
% i = initial, f = final
% ob_x_i = 0.06 ;
% ob_x_f = 0.12 ;
%
% ob_y_i = 0.01 ;
% ob_y_f = 0.07 ;
%
% ob_z_i = 0.12 ;
```

```
% ob_z_f = 0.18 ;

EE_off_limits = [] ;
for i = 1:(size(EE,1))
    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 ...
        ) && ( ob_z_i ≤ EE(i,3) ) && ( EE(i,3) ≤ ob_z_f )
       disp("obstacle hit at point ")
       EE(i,:)
       EE_off_limits = [EE_off_limits, i] ;
    end
end
% disp("off limit rows: " + EE_off_limits)
disp(EE_off_limits)
length(EE_off_limits)
% Function to replot in this interval - go one back from the off limits to
% find EE values before the robot hits the object.
avoid_obstacle = bug2(EE_off_limits(1)-1, EE_off_limits(end)+1, EE, object) ;

% size(EE,1)
% remove off limit rows from EE
for i = 1:length(EE_off_limits)
    EE(EE_off_limits(1), :) = [] ; % remove the first value each time to get correct one.
end
% size(EE,1)

% add in bug values to show navigating the obstacle.
rerouted_EE = zeros((size(EE,1)+size(avoid_obstacle,1)),3);
% length(rerouted_EE)
for i = 1:size(rerouted_EE,1)
    if i < EE_off_limits(1)
%         disp("add EE 1")
%         disp(i)
        rerouted_EE(i, :) = EE(i, :) ;
    elseif i < (EE_off_limits(1)+size(avoid_obstacle,1))
%         disp("add ob avoid")
%         disp(i)
        rerouted_EE(i, :) = avoid_obstacle((i-(EE_off_limits(1)-1)) , :) ;
    else
%         disp("add EE 2")
%         disp(i)
        rerouted_EE(i, :) = EE(i-size(avoid_obstacle,1), :) ;
    end
end
EE = rerouted_EE ;


%% Run IK and FK again to get the new trajectory with obstacle avoidance.
for j = 1:size(EE,1)
    px = EE(j,1) ;
    py = EE(j,2) ;
    pz = EE(j,3) ;

    % mu and psi angles for orientation
    mu = 0; % mu is rotation of end effector relative to "wrist"
    psi = (0:pi/20:2*pi) ;
    for i = 1: length(psi)
        psi(i) = psi(i) - pi/2;
    end
    End_Effector = [ px py pz ]' ;
    if norm(End_Effector) > l1+l2+l3+d1  % workspace is above ground
        error('desired position is out of the workspace')
    end


    %% %%%%%%%%%%%% Inverse Kinematics of LynxMotion Arm %%%%%%%%%%%%%%%%
    % For details on calculations, view notes in ReadMe.
    % We must find values for the 5 joint angles below.
    sym q1 ;  % single value
    q2 = [] ; % potentially many values
    q3 = [] ; % potentially many values
```

```
    q4 = [] ; % potentially many values
    sym q5 ;  % single value

    %% Simple angles to find:

    % Find q1
    q1 = atan2(py,px) ;
    % Find q5
    q5 = mu ;

    %% Define extra variables to help us find q2,3,4

    % define (r,z) plane
    r = sqrt(px^2 + py^2) ; % r is hypotenues in x-y plane.

    % For each value of psi, there is a value of rw, zw and D
    r_w = (1:length(psi)) ;
    z_w = (1:length(psi)) ;
    D = (1:length(psi)) ;
    for i = 1:length(psi)
    % define position of wrist in (r,z) plane
        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

        z_w(i) = pz - d1 -l3*sin(psi(i)) ;

        % define D - a placeholder variable for a large combination we derived in
        % notes

        D(i) = - (r_w(i)^2 + z_w(i)^2 - l1^2 - l2^2) / (2*l1*l2) ;
        D(i) = round(D(i), 7)  ;% rounding D avoids fake imaginary numbers due to rounding
                        % errors.
    end
    %% get only real values

    psi_real = [] ;
    r_w_real = [] ;
    z_w_real = [] ;
    D_real = [] ;
    for i = 1:length(psi)
        if imag(sqrt(1-D(i)^2)) == 0  % for real values, append to real lists.
            psi_real(end + 1) = psi(i);%round(psi(i),7) ; % round each list to get zeros rather ... than e^-10 or something.
            r_w_real(end + 1) = r_w(i);%round(r_w(i),7) ;
            z_w_real(end + 1) = z_w(i);%round(z_w(i),7) ;
            D_real(end + 1)= D(i);%round(D(i),7) ;
        end
    end

    % we can see these four lists MUST have same length.
    %  disp(psi_real)
    %  disp(D_real)
    %  disp(z_w_real)
    %  disp(r_w_real)

    %% Find q3 possibilities
    % there are two posibilities for each value of D depending on
    for i = 1:length(D_real)

        q3(end+1) = atan2( sqrt(1-D_real(i)^2), -D_real(i)) ;
        q3(end+1) = atan2(-sqrt(1-D_real(i)^2), -D_real(i)) ;
    end

    %  disp("q3")
    %  disp(q3)
    %% Explaining list order.
    % matlab indexes from 1 not 0.
    % if D has 3 values, we can see q3 will have 6, where
    % D(1) corresponds to q3(1 & 2)
    % D(2) corresponds to q3(3 & 4)
```

```
    % D(3) corresponds to q3(5 & 6)
    % we can see D(i) corresponds to values
    % q3(2*i -1) and q3(2*i)
    % we need to ensure this is *consistant* across all q lists below.
    %% Find q2

    % two options based on q3
    for i = 1:length(D_real)
        % for each r_w,z_w value, find first q2 value using first q3 value.
        % Switching order here corrects angles... why?
        q2(end+1) = atan2(z_w_real(i),r_w_real(i)) + atan2(l2*sin(q3(2*i)), l1+l2*cos(q3(2*i))) ;
        % add +1 to
        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))) ;
    end
    %  disp("q2")
    %  disp(q2)
    %% Find q4

    % two options based on two sets of q2,q3
    % use psi = q2 + q3 + q4
    for i = 1:length(D_real)
        q4(end+1) = psi_real(i) - q2(2*i -1) - q3(2*i -1) +pi/2 ;
        q4(end+1) = psi_real(i) - q2(2*i) - q3(2*i) +pi/2 ;
    end

    %  disp("q4")
    %  disp(q4)
    %% Remove duplicate values
    % make a matrix out of the lists where one row is one solution

    % we know matrix will have initial size: no. of items in q2,3 or 4 x 5
    % angles
    Solution_Matrix = zeros(length(q2), 5);
    for i = 1:length(q2)
        Solution_Matrix(i,1) = q1 ;
        Solution_Matrix(i,2) = q2(i) ;
        Solution_Matrix(i,3) = q3(i) ;
        Solution_Matrix(i,4) = q4(i) ;
        Solution_Matrix(i,5) = q5 ;
    end
    % disp(Solution_Matrix)
    % remove duplicate rows
    Unique_Solutions = unique(Solution_Matrix,"rows","stable") ;
    % stable prevents order being changed.


    %% we also know, q4 can't be more than 360 degrees. filter these out.
    valid_Solutions = [] ;
    for i = 1:size(Unique_Solutions,1)
        % condition for a valid solution: q4 is within 0 - 2pi, q2 is more
        % than 45% to avoid arm hitting the box
        if abs(Unique_Solutions(i,4)) < 2*pi && Unique_Solutions(i,2) > pi/3
            valid_Solutions = [valid_Solutions; Unique_Solutions(i,:)] ;
        end
    end
    % stable prevents order being changed.
        for i = 1:5
            IK_OUTPUT(j,i) = valid_Solutions(1,i) ;
        end
end



%% Plot the arm trajectory in 3d space
syms q1 q2 q3 q4 q5; % angle variables

% Show our 5 tranformation matrices
T_01 =[
[cos(q1), 0,  sin(q1),  0]
```

```
[sin(q1), 0, -cos(q1),  0]
[      0, 1,        0, d1]
[      0, 0,        0,  1]];

T_12 =[
[cos(q2), -sin(q2), 0, l1*cos(q2)]
[sin(q2),  cos(q2), 0, l1*sin(q2)]
[      0,        0, 1,          0]
[      0,        0, 0,          1]];

T_23 = [
[cos(q3), -sin(q3), 0, l2*cos(q3)]
[sin(q3),  cos(q3), 0, l2*sin(q3)]
[      0,        0, 1,          0]
[      0,        0, 0,          1]];

T_34 = [
[cos(q4), 0,  sin(q4),          0]
[sin(q4), 0, -cos(q4),          0]
[      0, 1,        0,          0]
[      0, 0,        0,          1]];

T_45 = [
[cos(q5), -sin(q5), 0,  0]
[sin(q5),  cos(q5), 0,  0]
[      0,        0, 1,  l3]
[      0,        0, 0,  1]];

% Get overall forward kinematics
T_e = T_01*T_12*T_23*T_34*T_45;

% extract the x,y,z elements of these
xt_element = T_e(1,4); % the x transformation is the 4th element of the 1st row of our general ...
    transformation matrix, T_e.
yt_element = T_e(2,4); % the y transformation is the 4th element of the 2nd row of our general ...
    transformation matrix, T_e.
zt_element = T_e(3,4); % the z transformation is the 4th element of the 3rd row of our general ...
    transformation matrix, T_e.

% Define all our angles in their own arrays
q1_set = [] ;
q2_set = [] ;
q3_set = [] ;
q4_set = [] ;
q5_set = [] ;
for i = 1:size(IK_OUTPUT,1) % i in range number of rows
    % number of columns always 5 for q1-5
    q1_set(i) = IK_OUTPUT(i,1) ;
    q2_set(i) = IK_OUTPUT(i,2) ;
    q3_set(i) = IK_OUTPUT(i,3) ;
    q4_set(i) = IK_OUTPUT(i,4) ;
    q5_set(i) = IK_OUTPUT(i,5) ;
end


%
%Now simply plot using our FK


% define variable for number of points
NoP = size(IK_OUTPUT,1) ; % same as number of sets of q values.

%base is at origin
base = 0 ;


% next joint given by T01 translations (movement from base to joint 1)
% need to make into a 4 by 3 array.
Tj1_x = zeros(NoP,1) ; % x translation is zero
Tj1_y = zeros(NoP,1) ; % y translation is zero
```

```
Tj1_z = subs(zeros(NoP,1), 0, d1) ;  % z translation is d1

% next joint given by T_01T_12 translations - movement from base to joint 2
% make our general elements for each array
movement = T_01*T_12 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
Tj2_x = [] ;
Tj2_y = [] ;
Tj2_z = [] ;
for i = 1:length(q1_set)
    xt_value = subs(xt_element, [ q1 q2], [q1_set(i) q2_set(i)]) ;
    yt_value = subs(yt_element, [ q1 q2], [q1_set(i) q2_set(i)]) ;
    zt_value = subs(zt_element, [ q1 q2], [q1_set(i) q2_set(i)]) ;
    Tj2_x(i,1) = xt_value ;
    Tj2_y(i,1) = yt_value ;
    Tj2_z(i,1) = zt_value ;
end
% above worked, now continue for the rest.


% next joint given by T_01T_12T_23 translations - movement from base to joint 3
% make our general elements for each array
movement = T_01*T_12*T_23 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
Tj3_x = [] ;
Tj3_y = [] ;
Tj3_z = [] ;
for i = 1:length(q1_set)
    xt_value = subs(xt_element,[q1 q2 q3], [q1_set(i) q2_set(i) q3_set(i)]) ;
    yt_value = subs(yt_element,[q1 q2 q3], [q1_set(i) q2_set(i) q3_set(i)]) ;
    zt_value = subs(zt_element,[q1 q2 q3], [q1_set(i) q2_set(i) q3_set(i)]) ;
    Tj3_x(i,1) = xt_value ;
    Tj3_y(i,1) = yt_value ;
    Tj3_z(i,1) = zt_value ;
end


% next joint given by T_01T_12T_23T_34 translations - movement from base to joint 4
% note there's no translation in this joint so no change from above.
% make our general elements for each array
movement = T_01*T_12*T_23*T_34 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
% fill the joint 2 4x1 arrays
Tj4_x = [] ;
Tj4_y = [] ;
Tj4_z = [] ;
for i = 1:length(q1_set)
    xt_value = subs(xt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    yt_value = subs(yt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    zt_value = subs(zt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    Tj4_x(i,1) = xt_value ;
    Tj4_y(i,1) = yt_value ;
    Tj4_z(i,1) = zt_value ;
end

% final joint given by T_01T_12T_23T_34T_45 translations - movement from base to joint 4
% note there's no translation in this joint so no change from above.
% make our general elements for each array
movement = T_01*T_12*T_23*T_34*T_45 ;
xt_element = movement(1,4) ;
yt_element = movement(2,4) ;
zt_element = movement(3,4) ;
```

```
% fill the joint 2 4x1 arrays
EE = [] ; % End Effector Positions list
for i = 1:length(q1_set)
    xt_value = subs(xt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    yt_value = subs(yt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    zt_value = subs(zt_element,[q1 q2 q3 q4], [q1_set(i) q2_set(i) q3_set(i) q4_set(i)]) ;
    EE(i,1) = xt_value ; % x values in column 1
    EE(i,2) = yt_value ; % y values in column 2
    EE(i,3) = zt_value ; % z values in column 3
end
%% Plot the graphs
figure (2)
% plot the object in the way
% object has 6 faces, 8 corners.
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_f; 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_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] ;
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] ;
fill3(ob_x(1,:),ob_y(1,:),ob_z(1,:),'r')
hold on
fill3(ob_x(2,:),ob_y(2,:),ob_z(2,:),'r')
hold on
fill3(ob_x(3,:),ob_y(3,:),ob_z(3,:),'r')
hold on
fill3(ob_x(4,:),ob_y(4,:),ob_z(4,:),'r')
hold on
fill3(ob_x(5,:),ob_y(5,:),ob_z(5,:),'r')
hold on
fill3(ob_x(6,:),ob_y(6,:),ob_z(6,:),'r')
hold on
    % generate the graph. Note: only 4 joints will be visible as there is
    % no spatial distinction between joints three and 4.
for i = 1:NoP % zeros is the base position, doesn't change.
    xx = [base; Tj1_x(i); Tj2_x(i); Tj3_x(i); Tj4_x(i); EE(i,1) ] ;
    yy = [base; Tj1_y(i); Tj2_y(i); Tj3_y(i); Tj4_y(i); EE(i,2) ] ;
    zz = [base; Tj1_z(i); Tj2_z(i); Tj3_z(i); Tj4_z(i); EE(i,3) ] ;
    axis([ 0 0.2 0 0.2 0 0.3 ])
    plot3(xx,yy,zz,'ko-','Linewidth',2)
    hold on % CHANGE TO HOLD ON TO SEE ALL LINES AT ONCE.
    pause(0.3)

    % label axes, start point, end point.
    text(EE(1,1) + 0.002,EE(1,2) + 0.002,EE(1,3) + 0.002,'Start') ;
    text(EE(end,1) + 0.002,EE(end,2) + 0.002,EE(end,3) + 0.002,'End') ;

    % label end effector points
    text(EE(i,1),EE(i,2),EE(i,3), 'x')
end
title('Actual Points') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');


figure (3) % end points only

% plot the object in the way
% object has 6 faces, 8 corners
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_f; 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_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] ;
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] ;
fill3(ob_x(1,:),ob_y(1,:),ob_z(1,:),'r')
```

```
hold on
fill3(ob_x(2,:),ob_y(2,:),ob_z(2,:),'r')
hold on
fill3(ob_x(3,:),ob_y(3,:),ob_z(3,:),'r')
hold on
fill3(ob_x(4,:),ob_y(4,:),ob_z(4,:),'r')
hold on
fill3(ob_x(5,:),ob_y(5,:),ob_z(5,:),'r')
hold on
fill3(ob_x(6,:),ob_y(6,:),ob_z(6,:),'r')
hold on

for i = 1:NoP
    plot3(EE(i,1),EE(i,2),EE(i,3),'ko-','Linewidth',2)
    hold on
end
title('Actual End Points Only') ; xlabel('x (m)') ; ylabel('y (m)') ; zlabel('z (m)');
axis([ 0 0.2 0 0.2 0 0.3 ])




%% Function to Find Where trajectory impacts the object and correct it.
function [Obstacle_Navigation] = bug2(first_off_lim_point, final_off_lim_point, EE_list, object)

% find entry point, correct this to be on object surface
% dsearch in allows us to search object surface array for the nearest value
% to the entry values we give
Entry = EE_list(first_off_lim_point -1, :); % start bug from point before the robot enters the object
New_Entry = zeros(1,3) ;
New_Entry(1) = object(dsearchn(object(:,1), Entry(1)), 1) ;
New_Entry(2) = object(dsearchn(object(:,2), Entry(2)), 2) ;
New_Entry(3) = object(dsearchn(object(:,3), Entry(3)), 3) ;

% find exit point, correct this to be on object surface
Exit = EE_list(final_off_lim_point +1, :); % end bug from point after the robot enters the object
New_Exit = zeros(1,3) ;
New_Exit(1) = object(dsearchn(object(:,1), Exit(1)), 1) ;
New_Exit(2) = object(dsearchn(object(:,2), Exit(2)), 2) ;
New_Exit(3) = object(dsearchn(object(:,3), Exit(3)), 3) ;

disp("Meet and leave object surface at: ")
disp(New_Entry)
disp(New_Exit)

% offset entry and exit points by a tiny amount so points are not on the
% surface but just outside.
New_Entry = New_Entry - 0.003 ;
New_Exit = New_Exit + 0.003 ;

% Get 10 points between entry and exit points on the surface of our object.
% Define whole object using an array
% define points per line for each axis and object array
zMotion_points_on_object = 5 ;
xyMotion_points_on_object = 10 ;
Obstacle_Navigation = zeros(zMotion_points_on_object+xyMotion_points_on_object+1,3) ;

% populate object array
% first go up to top of object (z motion)
for i = 1:(zMotion_points_on_object + 1)
    Obstacle_Navigation(i,1) = New_Entry(1) ;
    Obstacle_Navigation(i,2) = New_Entry(2) ;
    Obstacle_Navigation(i,3) = New_Entry(3) + (i-1)*(New_Exit(3) - ...
        New_Entry(3))/zMotion_points_on_object ;
end



%then go across the top of the object (x-y motion)
for i = 1:(xyMotion_points_on_object + 1)
```

```
    Obstacle_Navigation(i+zMotion_points_on_object+1,1) = New_Entry(1) + (i-1)*(New_Exit(1) - ...
        New_Entry(1))/xyMotion_points_on_object ;
    Obstacle_Navigation(i+zMotion_points_on_object+1,2) = New_Entry(2) + (i-1)*(New_Exit(2) - ...
        New_Entry(2))/xyMotion_points_on_object ;
    Obstacle_Navigation(i+zMotion_points_on_object+1,3) = New_Exit(3) ;
end


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

```
    %% Inverse Kinematics Calculator for LynxMotion Arm
clear all %#ok<*CLALL> %#ok<*SAGROW>
close all
clc


%% STEPS - FULL NOTES ON DRIVE
% 1. SPECIFY ROBOT PARAMETERS
% 2. SPECIFY DESIRED END POSITION (x_c, y_c, a)
% 3. SOLVE IK -> find (j1, j2, j3) for each arm.
% 4. PRINT THE 8 POSSIBLE ORIENTATIONS, DISPLAY 2 OF THESE IN FIGURES ONE
% AND TWO.
% using q for arm 1 angles.
% using s for arm 2 angles (skipping r to avoid confusing with radius
% values.
% using t for arm 3 angles.
%% 1. SPECIFY ROBOT PARAMETERS
% these values are taken from coursework appendix
S_A = 0.17 ; %  j1 to j2 link length
L = 0.13 ; % j2 to j3 link length
r_e = 0.13 ; % radius of circle formed by end effector joints
r_b = 0.29 ; % radius of circle formed by base joints



%% 2. SPECIFY DESIRED END POSITION (x_c, y_c, a)
x_c = 0.25 ;
y_c = 0.21 ;
a = pi/6 ;


%% arm 1 - joint angles given by q
q3 = a + pi/6 ;
% two possibilities for q1, q2 as they form an "elbow"
q1 = [] ;
q2 = [] ;
[q1(1), q1(2), q2(1), q2(2)] = find_line_eqn(S_A, L, r_e, x_c, y_c, q3) ;


%% arm 2 - joint angles given by s

% Find relative position of end effector in this arms frame.
x_c2 = x_c - sqrt(3)*r_b ;
y_c2 = y_c ;

% Find angles s1, s2, s3
s3 = a + 5*pi/6 ; % q3 + 120 degrees
% two possibilities for s1, s2 as they form an "elbow"
s1 = [] ;
s2 = [] ;
[s1(1), s1(2), s2(1), s2(2)] = find_line_eqn(S_A, L, r_e, x_c2, y_c2, s3) ;


%% arm 3 - joint angles given by t


% Find relative position of end effector in this arms frame.
x_c3 = x_c - sqrt(3)*r_b/2 ;
y_c3 = y_c - (3/2)*r_b ;
```

```

% Find angles t1, t2, t3
t3 = a + 3*pi/2 ; % s3 + 120 degrees, q3 + 240 degrees
% two possibilities for t1, t2 as they form an "elbow"
t1 = [] ;
t2 = [] ;
[t1(1), t1(2), t2(1), t2(2)] = find_line_eqn(S_A, L, r_e, x_c3, y_c3, t3) ;



%% Check that solutions are real
in_workspace = false ;
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] ;

if imag(angles_array) == zeros(1,15)
   in_workspace = true ;
end
%% Display Solutions
% there are 2 solutions for each of the 3 arms
% therefore 2^3 total orientations = 8 total
if in_workspace == true
    disp(" For a given position, there are 8 possible orientations: ")
    disp(" ")
%     pause(2)

    %Referring as arms being in orientation 1 or 2, we have:
    disp("solution 1: ")
    disp("orientation: 111")
    disp("***********")
    disp("Arm 1 (q1, q2, q3): ")
    disp( q1(1) *180/pi+ ", " + q2(1) *180/pi + ", " + q3 *180/pi)
    disp(" ")
    disp("Arm 2 (s1, s2, s3): ")
    disp( s1(1) *180/pi+ ", " + s2(1) *180/pi + ", " + s3 *180/pi)
    disp(" ")
    disp("Arm 3 (t1, t2, t3): ")
    disp( t1(1) *180/pi+ ", " + t2(1) *180/pi + ", " + t3 *180/pi)
    disp("***********")
    disp(" ")
%     pause(2)


    disp("solution 2: ")
    disp("orientation: 222")
    disp("***********")
    disp("Arm 1 (q1, q2, q3): ")
    disp( q1(2) *180/pi+ ", " + q2(2) *180/pi + ", " + q3 *180/pi)
    disp(" ")
    disp("Arm 2 (s1, s2, s3): ")
    disp( s1(2) *180/pi+ ", " + s2(2) *180/pi + ", " + s3 *180/pi)
    disp(" ")
    disp("Arm 3 (t1, t2, t3): ")
    disp( t1(2) *180/pi+ ", " + t2(2) *180/pi + ", " + t3 *180/pi)
    disp("***********")
    disp(" ")
%     pause(2)


    disp("solution 3: ")
    disp("orientation: 121")
    disp("***********")
    disp("Arm 1 (q1, q2, q3): ")
    disp( q1(1) *180/pi+ ", " + q2(1) *180/pi + ", " + q3 *180/pi)
    disp(" ")
    disp("Arm 2 (s1, s2, s3): ")
    disp( s1(2) *180/pi+ ", " + s2(2) *180/pi + ", " + s3 *180/pi)
    disp(" ")
    disp("Arm 3 (t1, t2, t3): ")
    disp( t1(1) *180/pi+ ", " + t2(1) *180/pi + ", " + t3 *180/pi)
```

```
    disp("***********")
    disp(" ")
%     pause(2)

    disp("solution 4: ")
    disp("orientation: 112")
    disp("***********")
    disp("Arm 1 (q1, q2, q3): ")
    disp( q1(1) *180/pi+ ", " + q2(1) *180/pi + ", " + q3 *180/pi)
    disp(" ")
    disp("Arm 2 (s1, s2, s3): ")
    disp( s1(1) *180/pi+ ", " + s2(1) *180/pi + ", " + s3 *180/pi)
    disp(" ")
    disp("Arm 3 (t1, t2, t3): ")
    disp( t1(2) *180/pi+ ", " + t2(2) *180/pi + ", " + t3 *180/pi)
    disp("***********")
    disp(" ")
%     pause(2)

    disp("solution 5: ")
    disp("orientation: 211")
    disp("***********")
    disp("Arm 1 (q1, q2, q3): ")
    disp( q1(2) *180/pi+ ", " + q2(2) *180/pi + ", " + q3 *180/pi)
    disp(" ")
    disp("Arm 2 (s1, s2, s3): ")
    disp( s1(1) *180/pi+ ", " + s2(1) *180/pi + ", " + s3 *180/pi)
    disp(" ")
    disp("Arm 3 (t1, t2, t3): ")
    disp( t1(1) *180/pi+ ", " + t2(1) *180/pi + ", " + t3 *180/pi)
    disp("***********")
    disp(" ")
%     pause(2)

    disp("solution 6: ")
    disp("orientation: 221")
    disp("***********")
    disp("Arm 1 (q1, q2, q3): ")
    disp( q1(2) *180/pi+ ", " + q2(2) *180/pi + ", " + q3 *180/pi)
    disp(" ")
    disp("Arm 2 (s1, s2, s3): ")
    disp( s1(2) *180/pi+ ", " + s2(2) *180/pi + ", " + s3 *180/pi)
    disp(" ")
    disp("Arm 3 (t1, t2, t3): ")
    disp( t1(1) *180/pi+ ", " + t2(1) *180/pi + ", " + t3 *180/pi)
    disp("***********")
    disp(" ")
%     pause(2)

    disp("solution 7: ")
    disp("orientation: 212")
    disp("***********")
    disp("Arm 1 (q1, q2, q3): ")
    disp( q1(2) *180/pi+ ", " + q2(2) *180/pi + ", " + q3 *180/pi)
    disp(" ")
    disp("Arm 2 (s1, s2, s3): ")
    disp( s1(1) *180/pi+ ", " + s2(1) *180/pi + ", " + s3 *180/pi)
    disp(" ")
    disp("Arm 3 (t1, t2, t3): ")
    disp( t1(2) *180/pi+ ", " + t2(2) *180/pi + ", " + t3 *180/pi)
    disp("***********")
    disp(" ")
%     pause(2)

    disp("solution 8: ")
    disp("orientation: 122")
    disp("***********")
    disp("Arm 1 (q1, q2, q3): ")
    disp( q1(1) *180/pi+ ", " + q2(1) *180/pi + ", " + q3 *180/pi)
    disp(" ")
```

```
    disp("Arm 2 (s1, s2, s3): ")
    disp( s1(2) *180/pi+ ", " + s2(2) *180/pi + ", " + s3 *180/pi)
    disp(" ")
    disp("Arm 3 (t1, t2, t3): ")
    disp( t1(2) *180/pi+ ", " + t2(2) *180/pi + ", " + t3 *180/pi)
    disp("**********")
    disp(" ")

else
    disp("Chosen Position is outside of robot workspace.")
end


%% Plot the position of the robot

figure(1) % orientation 1
% plot the outer triangle
% bottom left, bottom right, top
tri_x = [0, sqrt(3)*r_b, (1/2)*sqrt(3)*r_b, 0] ;
tri_y = [0, 0, (3/2)*r_b, 0] ;
plot(tri_x,tri_y,'ro-','Linewidth',2)
hold on


% Plot arm 1
xj1 = 0 ;
xj2 = xj1 + S_A*cos(q1(1)) ;
xj3 = xj2 + L*cos(q2(1)) ;
arm1_x = [xj1, xj2, xj3] ;

yj1 = 0 ;
yj2 = yj1 + S_A*sin(q1(1)) ;
yj3 = yj2 + L*sin(q2(1)) ;
arm1_y = [yj1, yj2, yj3] ;

plot(arm1_x, arm1_y, 'co-','Linewidth',2)
hold on


% Plot arm 2
xj1 = sqrt(3)*r_b ;
xj2 = xj1 + S_A*cos(s1(1)) ;
xj3 = xj2 + L*cos(s2(1)) ;
arm2_x = [xj1, xj2, xj3] ;

yj1 = 0 ;
yj2 = yj1 + S_A*sin(s1(1)) ;
yj3 = yj2 + L*sin(s2(1)) ;
arm2_y = [yj1, yj2, yj3] ;

plot(arm2_x, arm2_y, 'co-','Linewidth',2)
hold on

% Plot arm 3
xj1 = (1/2)*sqrt(3)*r_b ;
xj2 = xj1 + S_A*cos(t1(1)) ;
xj3 = xj2 + L*cos(t2(1)) ;
arm3_x = [xj1, xj2, xj3] ;

yj1 = (3/2)*r_b ;
yj2 = yj1 + S_A*sin(t1(1)) ;
yj3 = yj2 + L*sin(t2(1)) ;
arm3_y = [yj1, yj2, yj3] ;

plot(arm3_x, arm3_y, 'co-','Linewidth',2)
hold on

%plot inner triangle
inner_triangle_x = [arm1_x(3), arm2_x(3), arm3_x(3), arm1_x(3)] ;
inner_triangle_y = [arm1_y(3), arm2_y(3), arm3_y(3), arm1_y(3)] ;
```

```
plot(inner_triangle_x, inner_triangle_y, 'bo-','Linewidth',2)
hold on

axis([ -0.1 0.6 -0.1 0.6])
title('Parallel Robot First Orientation for a = pi/6') ; xlabel('x (m)') ; ylabel('y (m)');
subtitle('End Effector at x = 0.25, y = 0.21');



%**************************************************
figure(2) % orientation 2
% plot the outer triangle
% bottom left, bottom right, top
tri_x = [0, sqrt(3)*r_b, (1/2)*sqrt(3)*r_b, 0] ;
tri_y = [0, 0, (3/2)*r_b, 0] ;
plot(tri_x,tri_y,'ro-','Linewidth',2)
hold on


% Plot arm 1
xj1 = 0 ;
xj2 = xj1 + S_A*cos(q1(2)) ;
xj3 = xj2 + L*cos(q2(2)) ;
arm1_x = [xj1, xj2, xj3] ;

yj1 = 0 ;
yj2 = yj1 + S_A*sin(q1(2)) ;
yj3 = yj2 + L*sin(q2(2)) ;
arm1_y = [yj1, yj2, yj3] ;

plot(arm1_x, arm1_y, 'co-','Linewidth',2)
hold on


% Plot arm 2
xj1 = sqrt(3)*r_b ;
xj2 = xj1 + S_A*cos(s1(2)) ;
xj3 = xj2 + L*cos(s2(2)) ;
arm2_x = [xj1, xj2, xj3] ;

yj1 = 0 ;
yj2 = yj1 + S_A*sin(s1(2)) ;
yj3 = yj2 + L*sin(s2(2)) ;
arm2_y = [yj1, yj2, yj3] ;

plot(arm2_x, arm2_y, 'co-','Linewidth',2)
hold on

% Plot arm 3
xj1 = (1/2)*sqrt(3)*r_b ;
xj2 = xj1 + S_A*cos(t1(2)) ;
xj3 = xj2 + L*cos(t2(2)) ;
arm3_x = [xj1, xj2, xj3] ;

yj1 = (3/2)*r_b ;
yj2 = yj1 + S_A*sin(t1(2)) ;
yj3 = yj2 + L*sin(t2(2)) ;
arm3_y = [yj1, yj2, yj3] ;

plot(arm3_x, arm3_y, 'co-','Linewidth',2)
hold on

%plot inner triangle
inner_triangle_x = [arm1_x(3), arm2_x(3), arm3_x(3), arm1_x(3)] ;
inner_triangle_y = [arm1_y(3), arm2_y(3), arm3_y(3), arm1_y(3)] ;
plot(inner_triangle_x, inner_triangle_y, 'bo-','Linewidth',2)
hold on

axis([ -0.1 0.6 -0.1 0.6])
title('Parallel Robot Second Orientation for a = pi/6') ; xlabel('x (m)') ; ylabel('y (m)');
```

```
subtitle('End Effector at x = 0.25, y = 0.21');
%% Function to find angles 1 & 2 using angle 3 and coords of end effector
% arm base frame

function [angle1_a, angle1_b, angle2_a, angle2_b] = find_line_eqn(S_A, L, r_e, x_ee, y_ee, angle3 )

% find x3, y3 the position of the joint at angle3
x3 = x_ee - r_e*cos(angle3) ;
y3 = y_ee - r_e*sin(angle3) ;

% calculate both angle1 values
angle1_a = atan2(y3,x3) + acos((S_A^2 + x3^2 + y3^2 - L^2) / (2*S_A*sqrt(x3^2 + y3^2))) ;
angle1_b = atan2(y3,x3) - acos((S_A^2 + x3^2 + y3^2 - L^2) / (2*S_A*sqrt(x3^2 + y3^2))) ;

% calculate both angle2 values

% for angle1_a,
angle2_a = -(pi - angle1_a - acos((S_A^2 +L^2 - x3^2 -y3^2)/(2*S_A*L))) ;
% for angle1_b,
angle2_b = +(pi + angle1_b - acos((S_A^2 +L^2 - x3^2 -y3^2)/(2*S_A*L))) ;

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

```matlab
    %% Parallel Robot WorkSpace Plotter.
clear all %#ok<*CLALL> %#ok<*SAGROW>
close all
clc


%% STEPS - FULL NOTES ON DRIVE
% 1. SPECIFY ROBOT PARAMETERS
% 2. SPECIFY RANGE OF DESIRED END POSITIONS (x_c, y_c, a)
% 4. SOLVE IK -> find (j1, j2, j3) for each arm AT EACH POSITION.
% using q for arm 1 angles.
% using s for arm 2 angles (skipping r to avoid confusing with radius
% values.
% using t for arm 3 angles.
% IGNORE IMAGINARY POSITIONS, PLOT REAL POSITIONS TO GET AN OUTLINE OF WS
%% 1. SPECIFY ROBOT PARAMETERS
% these values are taken from coursework appendix
S_A = 0.17 ; %  j1 to j2 link length
L = 0.13 ; % j2 to j3 link length
r_e = 0.13 ; % radius of circle formed by end effector joints
r_b = 0.29 ; % radius of circle formed by base joints



%% 2. SPECIFY DESIRED END POSITION (x_c, y_c, a)
x_c = -0.1:0.01:0.6 ; % test ranges for x and y
y_c = -0.1:0.01:0.6 ;
a = pi/12; % constant orientation - interesting values for plot: 0, pi/12, pi/6, pi/2.3

% pre-make lists for the elbow angles
q1 = zeros(length(x_c)*length(y_c),2) ; % 2 angles at each position for elbow angles 1 and 2
q2 = zeros(length(x_c)*length(y_c),2) ;

s1 = zeros(length(x_c)*length(y_c),2) ; % Repeat for s angles
s2 = zeros(length(x_c)*length(y_c),2) ;


t1 = zeros(length(x_c)*length(y_c),2) ; % Repeat for t angles
t2 = zeros(length(x_c)*length(y_c),2) ;

%% arm 1 - joint angles given by q
q3 = a + pi/6 ;
count = 1 ;
% two possibilities for q1, q2 as they form an "elbow"
for i = 1: length(x_c)
    for j = 1: length(y_c)
        [q1(count,1), q1(count,2), q2(count,1), q2(count,2)] = find_line_eqn(S_A, L, r_e, x_c(i), ...
            y_c(j), q3) ;
        count = count + 1;
    end
end

%% arm 2 - joint angles given by s

% Find relative position of end effector in this arms frame.
x_c2 = x_c - sqrt(3)*r_b ;
y_c2 = y_c ;

% Find angles s1, s2, s3
```

```
s3 = a + 5*pi/6 ; % q3 + 120 degrees
count = 1 ;
% two possibilities for s1, s2 as they form an "elbow"
for i = 1: length(x_c)
    for j = 1: length(y_c)
        [s1(count,1), s1(count,2), s2(count,1), s2(count,2)] = find_line_eqn(S_A, L, r_e, x_c2(i), ...
            y_c2(j), s3) ;
        count = count + 1;
    end
end
%% arm 3 - joint angles given by t


% Find relative position of end effector in this arms frame.
x_c3 = x_c - sqrt(3)*r_b/2 ;
y_c3 = y_c - 3*r_b/2 ;

% Find angles t1, t2, t3
t3 = a + 3*pi/2 ; % s3 + 120 degrees, q3 + 240 degrees
count = 1 ;
% two possibilities for t1, t2 as they form an "elbow"
for i = 1: length(x_c)
    for j = 1: length(y_c)
        [t1(count,1), t1(count,2), t2(count,1), t2(count,2)] = find_line_eqn(S_A, L, r_e, x_c3(i), ...
            y_c3(j), t3) ;
        count = count + 1;
    end
end

%% Generate array of only real solutions
in_workspace = [] ;
count = 1 ;
for i = 1: length(x_c)
    for j = 1:length(y_c)
        angles_array_elbow1 = [x_c(i), y_c(j), q1(count,1), q2(count,1), q3, s1(count,1), ...
            s2(count,1), s3, t1(count,1), t2(count,1), t3] ;
        angles_array_elbow2 = [x_c(i), y_c(j), q1(count,2), q2(count,2), q3, s1(count,2), ...
            s2(count,2), s3, t1(count,2), t2(count,2), t3] ;
        if isreal(angles_array_elbow1) % boolean check for whether array contains imaginary numbers
           in_workspace = [in_workspace; angles_array_elbow1] ; % if passed, append the real row
        end
        if isreal(angles_array_elbow2)% boolean check for whether array contains imaginary numbers
           in_workspace = [in_workspace; angles_array_elbow2] ; % if passed, append the real row
        end
        count = count + 1;
    end
end

disp(in_workspace)

% print statement to let user know if no real solutions
if isempty(in_workspace)
    disp("Chosen value of *a* has no real workspace!")
end


% get only the position values from these.
workspace_positions = [in_workspace(:,1) , in_workspace(:,2) ] ;


%% Display Solutions on Graph
figure (1)
% plot the outer triangle
% bottom left, bottom right, top
tri_x = [0, sqrt(3)*r_b, (1/2)*sqrt(3)*r_b, 0] ;
tri_y = [0, 0, (3/2)*r_b, 0] ;
plot(tri_x,tri_y,'ro-','Linewidth',2)
hold on

% Plot the individual points
```

```
plot(workspace_positions(:,1), workspace_positions(:,2), "kx")
hold on


% Plot the workspace outline
outline = convhull(workspace_positions);
plot(workspace_positions(outline,1), workspace_positions(outline,2), "c" )

legend("Robot Base", "End Effector Positions", "Workspace Area")
axis([ -0.1 0.6 -0.1 0.6])
title('Parallel Robot WorkSpace for a = ', num2str(a)) ; xlabel('x (m)') ; ylabel('y (m)');

%% Function to find angles 1 & 2 using angle 3 and coords of end effector
% arm base frame

function [angle1_a, angle1_b, angle2_a, angle2_b] = find_line_eqn(S_A, L, r_e, x_ee, y_ee, angle3 )

% find x3, y3 the position of the joint at angle3
x3 = x_ee - r_e*cos(angle3) ;
y3 = y_ee - r_e*sin(angle3) ;

% calculate both angle1 values
angle1_a = atan2(y3,x3) + acos((S_A^2 + x3^2 + y3^2 - L^2) / (2*S_A*sqrt(x3^2 + y3^2))) ;
angle1_b = atan2(y3,x3) - acos((S_A^2 + x3^2 + y3^2 - L^2) / (2*S_A*sqrt(x3^2 + y3^2))) ;

% calculate both angle2 values

% for angle1_a,
angle2_a = -(pi - angle1_a - acos((S_A^2 +L^2 - x3^2 -y3^2)/(2*S_A*L))) ;
% for angle1_b,
angle2_b = +(pi + angle1_b - acos((S_A^2 +L^2 - x3^2 -y3^2)/(2*S_A*L))) ;

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-al5d.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.