Coursework Report - Group Cover Page

Coursework: Serial and Parallel Robot Kinematics

Student name and number	PART I.A	PART I.B	PART II.1	PART II.2	PART III
Joe Jeffcock 21070345	Y	Y	Y	Y	Y
Sompoch Tanticharoenkarn 21070374	Y	Y	Y	Y	Y

Note: Students must tick the sections that they have been undertaking or helped with.

UFMF4X-15-M: Robotic Fundamentals
Coursework: Serial and Parallel Robot Kinematics

21070345 Joe Jeffcock, 21070374 Sompoch Tanticharoenkarn

January 12, 2022

Contents

1 Part 1: Serial Robot

1.1 Kinematics

1.1.1 DH Table and Forward Kinematics

The end effector pose of an actuated robot arm is a function of angles at each of its joints, termed its Forward Kinematics (FK). In this section, we investigate FK in a model of the Lynxmotion AL5D robot arm described in Figure 1. Compound transformations obtained by Denavit–Hartenberg (DH) parameters are used to calculate the transformation matrix ${}^0T_5$ of the end effector from the base of the AL5D from joint values $(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5)$ via FK expressed in Equation 1. An example of compound transformations is shown in Equation 2.

$$ {}^0T_5 = FK(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5) \tag{1} $$

$$ {}^bT_{ee} = {}^b\,T_1^1T_{ee} \tag{2} $$

where:
${}^bT_{ee}$ is the transformation matrix of the pose of the end effector with respect to the base
${}^bT_1$ is the transformation matrix of the pose of the first joint with respect to the base
${}^1T_{ee}$ is the transformation matrix of the pose of the end effector with respect to the first joint

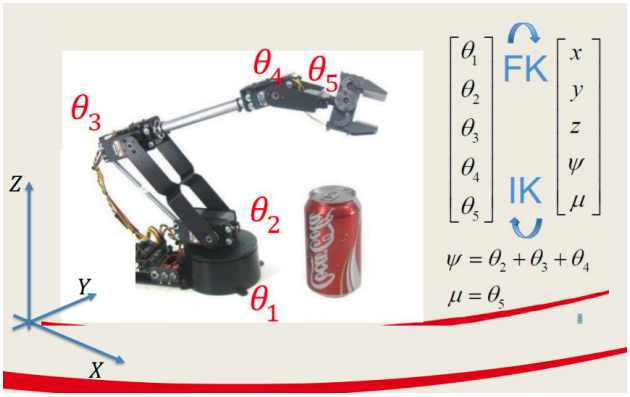


Figure 1: Description of the AL5D (Jafari 2021)

This report uses the Distal convention of DH parameters to describe frames between each joint, where the transformation matrix from joint $n-1$ to n obtained by the Distal convention is expressed by Equation 3. Fixed frames for the Lynxmotion AL5D robot arm are shown in Figure 2, and its DH parameters to create transformation matrices are shown in Table 1.

$$ DH(n) = {}^{n-1}\,T_n = \begin{bmatrix} cos(\theta_n) & -sin(\theta_n)cos(\alpha_n) & sin(\alpha_n)sin(\theta_n) & a_n cos(\theta_n) \\ sin(\theta_n) & cos(\alpha_n)cos(\theta_n) & -cos(\theta_n)sin(\alpha_n) & a_n sin(\theta_n) \\ 0 & sin(\alpha_n) & cos(\alpha_n) & d_n \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3} $$

where:
a_n is distance along x_n, from Z_{n-1} to Z_n.
α_n is rotation about x_n, from Z_{n-1} to Z_n.
d_n is distance along z_{n-1}, from x_{n-1} to x_n.
θ_n is rotation about z_{n-1}, from x_{n-1} to x_n

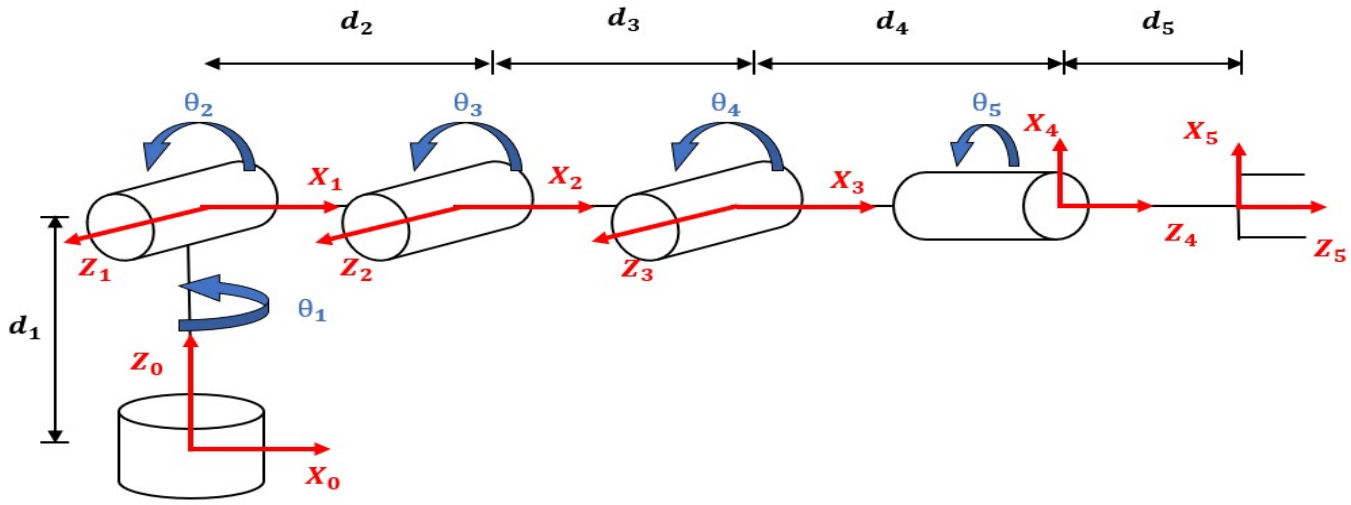


Figure 2: Frame Assignment in the Lynxmotion AL5D

Table 1: Distal Convention Parameters

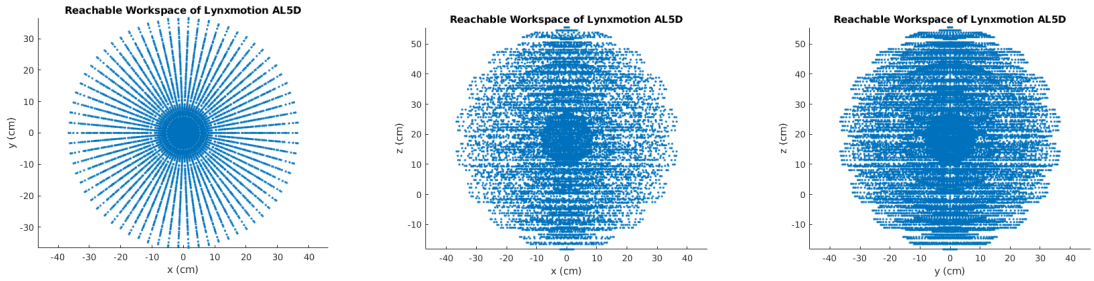
n	a_n	α_n	d_n	θ_n
1	0	90°	d_1	θ_1
2	d_2	0	0	θ_2
3	d_3	0	0	θ_3
4	d_4	90°	0	θ_4
5	0	0	d_5	θ_5

Using transformation matrices (Equation 3) obtained from the DH parameters in Table 1, we compute the FK transformation matrix of the AL5D by equation 4:

$$FK(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5) = \Pi_{n=1}^{5} DH(a_n, \alpha_n, d_n, \theta_n) = {}^0T_1^1T_2^2T_3^3T_4^4T_5 = {}^0T_5 \tag{4}$$

The (x, y, z, ψ, μ) pose from Figure 1 can be obtained from ${}^0T_5$ in Equation 4 by taking rows 1,2 and 3 of column 4 to be x, y and z respectively. $\psi = \theta_2, \theta_3, \theta_4$ and $\mu = \theta_5$ can be computed without FK.

1.1.2 Workspace



(a) XY Plot (b) XZ Plot (c) YZ Plot

Figure 3: 2D Plots of AL5D Reachable Workspace

For the purpose of this section, we assume each joint in the AL5D has a range of $-180°$ to $180°$. We compute an approximation of the workspace of this robot by discretising the joint space and plotting the

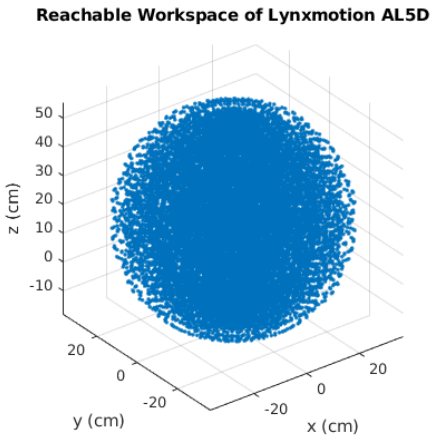


Figure 4: 3D Plot of AL5D Reachable Workspace

end-effector positions computed by FK at all possible angles.

The range θ_1 is discretised over 36 linearly spaced samples and the ranges of $\theta_2 - \theta_5$ are descritised over 10 linearly spaced samples. All possible values of this discretisation are plot on a graph to approximate the AL5D reachable workspace, presented in 2D and 3D by Figures 3 and 4, respectively. A higher density of samples is used for θ_1 as it is the only joint that effects rotation about the z-axis.

From Figures 3 and 4, we observe the approximated workspace forms a sphere-like structure. Hence, we infer that the reachable workspace is a sphere of radius approximately 37.55cm, centered at approximately (0cm,0cm,18.5cm), corresponding with the length of $d_2 + d_3 + d_4$ and the point at ${}^0T_1$, respectively.

1.1.3 Inverse Kinematics

Inverse Kinematics (IK) refers to the computation of joint values in a robot given a pose. IK of the AL5D is given by Equation 5 for joint values $\theta_1 - \theta_5$, using the (x, y, z, ψ, μ) definition of the end-effector pose illustrated by Figure 1.

Analytical and numerical solutions are two widely used methods for computing IK. In this section, we derive the analytical solution of IK in the AL5D robot. The analytical solution of IK in the AL5D can be derived geometrically to obtain a closed form solution given by Equation 5. The benefits of the closed form solution are that it exhibits minimal error while being fast to compute and easy to understand. The geometry used to compute IK in the AL5D is shown in Figure 5 where the R-axis accounts for rotation by θ_1 along Z.

$$\theta_1, \theta_2, \theta_3, \theta_4, \theta_5 = IK(x_{ee}, y_{ee}, z_{ee}, \psi, \mu) \quad s.t. \quad \psi = \theta_2 + \theta_3 + \theta_4, \mu = \theta_5 \tag{5}$$

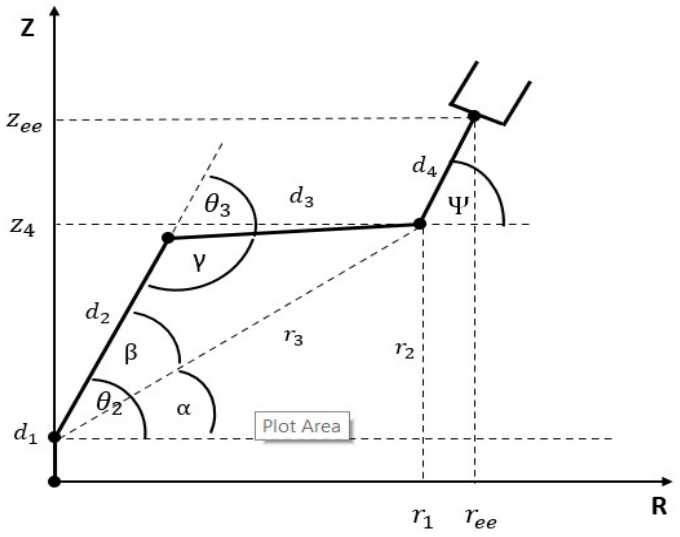


Figure 5: Diagram for solving inverse kinematics of the AL5D

The following equations are used to derived the inverse kinematics of AL5D.

$$\begin{aligned} r_{ee} &= \sqrt{x_{ee}^2 + y_{ee}^2} \\ r_1 &= r_{ee} - d_4 cos(\psi) \\ r_2 &= z_{ee} - d_4 - sin(\psi) - d_1 \\ r_3 &= \sqrt{r_1^2 + r_2^2} \end{aligned} \tag{6}$$

From equation 6, α can be calculated using simple cotangent equation, and β and γ can be calculated by using the law of cosine.

$$\begin{aligned} \alpha &= tan^{-1}(r_2/r_1) \\ \beta &= cos^{-1}(d_3^2 - d_2^2 - r_3^2)/(-2d_2r_3) \\ \gamma &= cos^{-1}(r_3^2 - d_2^2 - d_3^2)/(-2d_2d_3) \end{aligned} \tag{7}$$

Finally, from equations 7, all angles of the robot can be computed as followed:

$$\begin{aligned} \theta_1 &= tan^{-1}(y_{ee}/x_{ee}) \\ \theta_2 &= \alpha \pm \beta \\ \theta_3 &= \pi - \gamma \\ \theta_4 &= \psi - \theta_2 - \theta 3 + \theta_{4,0} \\ \theta_5 &= \mu + \theta_{5,0} \end{aligned} \tag{8}$$

where $\theta_{4,0}$ and $\theta_{4,0}$ are initial constant value defined by initial configuration of the robot.

1.2 Trajectories

1.2.1 Task Definition

In this section, we task the AL5D with replacing a light bulb. Using (x, y, z, ψ, μ) poses, this involves:

1. starting parked at $(10, 0, 18.5, -90°, 0°)$
2. going to bulb at $(0, 20, 45, 90°, 360°)$
3. unscrewing bulb to $(0, 20, 45, 90°, 0°)$

4. disposing of bulb at $(7.5, 10, -10, -90°, 0°)$
5. approaching new bulb at $(-15, 25, 15, 0°, 90°)$
6. grasping new bulb at $(-18, 30, 15, 0°, 90°)$
7. placing new bulb at $(0, 20, 45, 90°, 360°)$
8. screwing in bulb to $(0, 20, 45, 90°, 0°)$
9. returning to park at $(10, 0, 18.5, -90°, 0°)$

1.2.2 Visualisation of Task

With reference to the task-specific poses outlined in Section 1.2.1, the IK for each pose is computed to obtain joint space coordinates of the AL5D. The resulting configurations of the AL5D are illustrated for each pose in Figure 6 along with corresponding step numbers. Figure 7 plots the path of the task, represented by line segments between XYZ coordinates of adjacent poses.

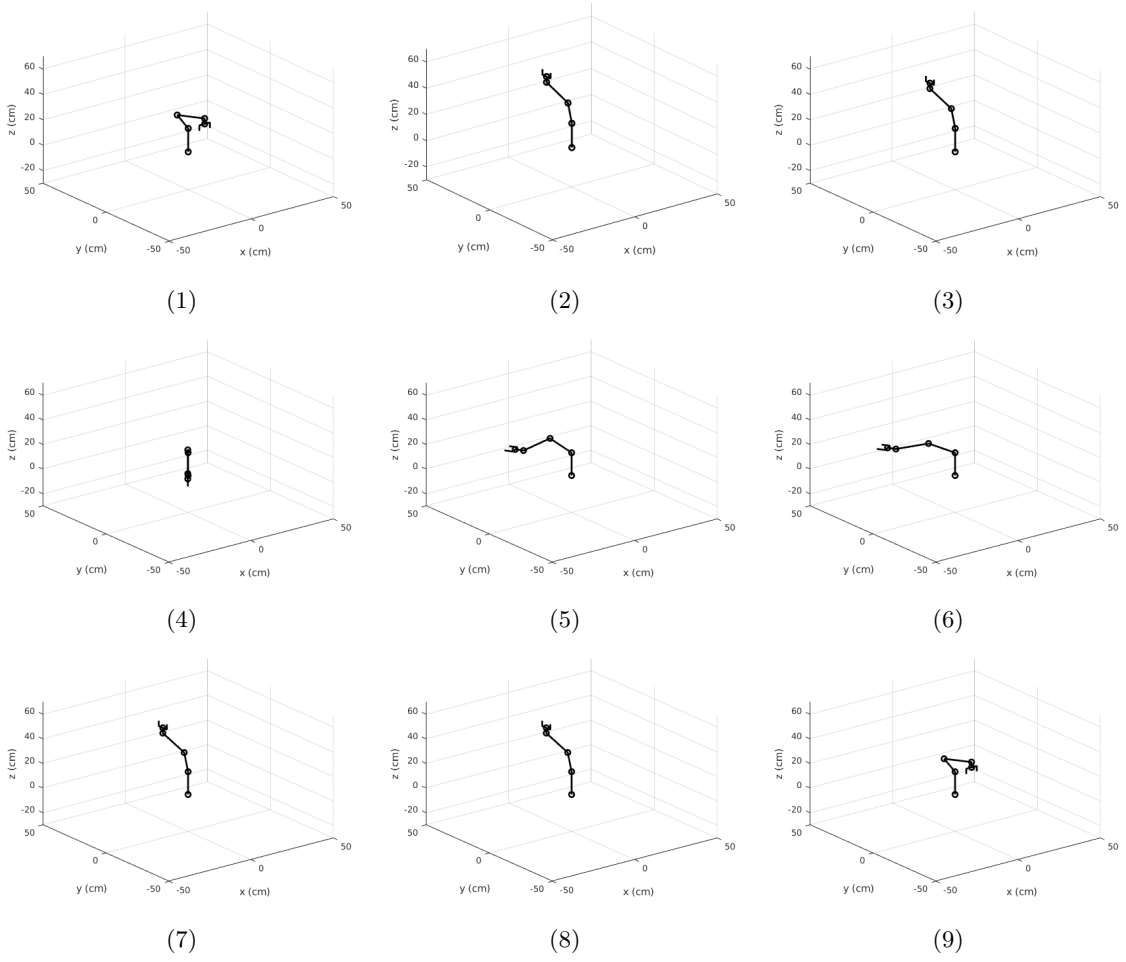


Figure 6: AL5D IK Computed for each Step of the Task in Section 1.2.1

1.2.3 Joint Space Trajectory

Joint space trajectories can be defined using polynomial equations to effect motion in a robot arm. The benefits of using polynomial equation is that it can keep the 'jerk' or change in acceleration low, which can

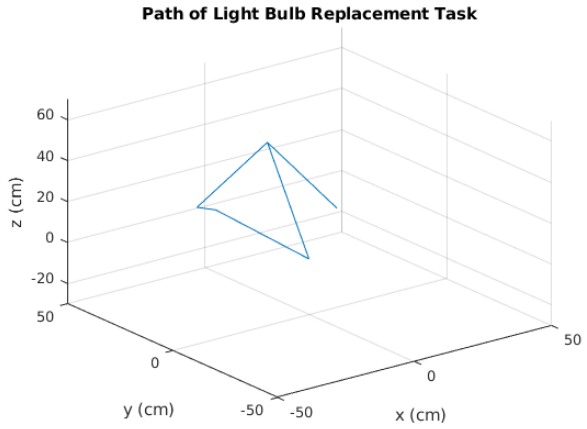


Figure 7: Path of Task in Section 1.2.1

prevent damage to the robot. In this report, we use cubic polynomial equation to set up the value of each joint of the robot respect to time. The cubic polynomial joint trajectory equation is shown in equation 9.

$$\theta = a_0 + a_1 t + a_2 t^2 + a_3 t^3 \tag{9}$$

we substitute the initial angles and time (θ_i, t_i), and final angle and time (θ_f, t_f) into the equation separately, we can have 2 equations as followed:

$$\theta_i = a_0 + a_1 t_i + a_2 t_i^2 + a_3 t_i^3 \tag{10}$$

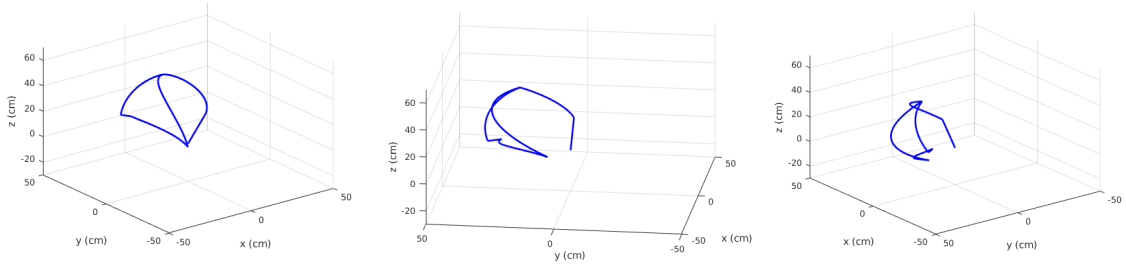
$$\theta_f = a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3 \tag{11}$$

Differentiate equation 10 and 11 we can have 2 following equations:

$$d\theta_i/dt = v_i = a_1 + 2a_2 t_i + a_3 t_i \tag{12}$$

$$d\theta_f/dt = v_f = a_1 + 2a_2 t_f + a_3 t_f \tag{13}$$

We set initial and final velocity (v_i and v_f) equal to zero, and we can solve 4 unknowns (a_0, a_1, a_2, and a_3), using 4 equations (equation 10 to 13). Results are shown in Figure 8 and a video is included in Section A.1.



(a) Plot rotated by 0° (b) Plot rotated by 45° (c) Plot rotated by 90°

Figure 8: Plot of Task Execution by Joint Space Trajectories

1.2.4 Cartesian Space Trajectory

Cartesian space trajectories are computed by five steps: (1) obtaining a function for the path; (2) sampling the function to obtain discrete points; (3) applying inverse kinematics to sampled points; (4) fitting a function to the points in joint space coordinates; (5) sampling the fit function. The five steps are discussed in detail in this section.

We derive two methods to sample the Cartesian path for Step 1, the first being linear equations traditionally used to obtain the path. We alternatively propose the vector representation of a line which is more robust and quicker to implement.

Function for Path by Linear Equations

$$\begin{aligned} y &= m_1 x + b_1 \\ y &= m_2 z + b_2 \end{aligned} \tag{14}$$

The equations of the line between the start and end points is given by Equation 14, where (x, y, z) corresponds to a point in Cartesian space, m_1, m_2 are slopes, and b_1, b_2 are constants. m_1, m_2, b_1, and b_2 can be solved by substituting known initial and final positions into equation 14.

By differentiating equation 14 with time, we obtain the equations outlined in Equation 15:

$$\begin{aligned} \dot{y} &= m_1 \dot{x} \\ \dot{y} &= m_2 \dot{z} \end{aligned} \tag{15}$$

From equation 15, we are able to obtain $\dot{z}$ in terms of $\dot{x}$.

$$\dot{z} = \frac{m_1}{m_2}\dot{x} \tag{16}$$

The velocity of the end effector $||u||$ can be expressed by Equation 17:

$$\begin{aligned} \vec{u} &= \dot{x}\vec{i} + \dot{y}\vec{j} + \dot{z}\vec{k} \\ ||u|| &= \sqrt{\dot{x}^2 + \dot{y}^2 + \dot{z}^2} = \sqrt{\dot{x}^2 + (m_1\dot{x})^2 + (\frac{m_1}{m_2}\dot{x})^2} \end{aligned} \tag{17}$$

From equation 17, we can expressed $\dot{x}$, $\dot{y}$, $\dot{z}$ separately in term of $||u||$ by Equation 18:

$$\begin{aligned} \dot{x} &= \frac{||u||}{\sqrt{1 + (m_1)^2 + (\frac{m_1}{m_2})^2}} \\ \dot{y} &= m_1 \frac{||u||}{\sqrt{1 + (m_1)^2 + (\frac{m_1}{m_2})^2}} \\ \dot{z} &= \frac{m_1}{m_2} \frac{||u||}{\sqrt{1 + (m_1)^2 + (\frac{m_1}{m_2})^2}} \end{aligned} \tag{18}$$

Finally, we can achieve the continuous equations for the path of the Cartesian trajectory by integrating Equation 18 with time, resulting in Equation 19.

$$\begin{aligned} x(t) &= \frac{u(t)}{\sqrt{1 + (m_1)^2 + (\frac{m_1}{m_2})^2}} + x_0 \\ y(t) &= m_1 \frac{u(t)}{\sqrt{1 + (m_1)^2 + (\frac{m_1}{m_2})^2}} + y_0 \\ z(t) &= \frac{m_1}{m_2} \frac{u(t)}{\sqrt{1 + (m_1)^2 + (\frac{m_1}{m_2})^2}} + z_0 \end{aligned} \tag{19}$$

The functions of *psi* and *mu* can be obtained by a similar sequence of differentiation and integration in Equations 14-19 using the initial equation $\psi = m_3\mu + b_3$.

Function for Path by Vector Representation of a Line However, equation 19 cannot be used to plan the trajectory of the robot in scenarios where the gradients m_1 and m_1 from equation 14 are of an incorrect sign or cannot be solved, such as when z is constant causing m_2 to be infinity. To overcome this, we implement the vector representation of a line between points $P1$ and $P2$ to create a function for the Cartesian path in our experiments. The vector alternative to Equation 19 is given by Equation 20.

$$\begin{bmatrix} x(t) \\ y(t) \\ z(t) \\ \psi(t) \\ \mu(t) \end{bmatrix} = \begin{bmatrix} x_{P1} \\ y_{P1} \\ z_{P1} \\ \psi_{P1} \\ \mu_{P1} \end{bmatrix} + t \begin{bmatrix} x_{P2} - x_{P1} \\ y_{P2} - y_{P1} \\ z_{P2} - z_{P1} \\ \psi_{P2} - \psi_{P1} \\ \mu_{P2} - \mu_{P1} \end{bmatrix}, 0 \leq t \leq 1 \tag{20}$$

Sampling Function to Obtain Discrete Points To sample points along the equations of the line between points $P1$ and $P2$, a sample size M is chosen. Let $distance(P_i, P_j)$ be the Euclidean distance between points P_i and P_j, and u be the desired speed.

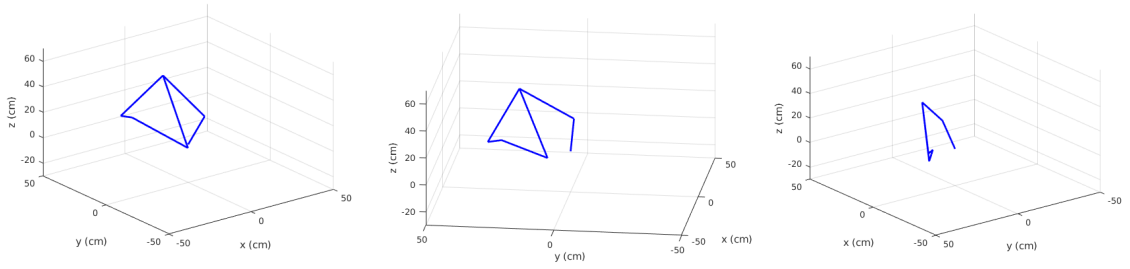
A tuple of linearly spaced samples in the x dimension $(x(t_i)|i = 1...M)$ is obtained via linear equations (Equation 19) given total time $t_{total} = \frac{distance(P1,P2)}{u}$ and time steps $t_i = i\frac{t_{total}}{M-1}$. This can similarly be applied to obtain samples in the y, z, ψ and μ dimensions

A tuple of linearly spaced samples in the x dimension $(x(t_i)|i = 1...M)$ is obtained via the vector representation of a line (Equation 20) given total time $t_{total} = \frac{distance(P1,P2)}{u}$ and time steps $t_i = \frac{i(\frac{t_{total}}{M-1})}{t_{total}}$, normalised to the range (0,1). This can similarly be applied to obtain samples in the y, z, ψ and μ dimensions.

Compute Inverse Kinematics on Sampled Points Inverse Kinematics is applied to the sampled points to obtain their coordinates in the AL5D joint space. Section 1.1.3 describes the Inverse Kinematics of the AL5D in further detail.

Fitting a Function to Points in the Joint Space For each joint in the samples obtained by Inverse Kinematics on Cartesian points, we fit a function across points over time to obtain an approximation of the Cartesian motion in joint space coordinates. This is implemented using Ordinary Least Squares (OLS) polynomial regression, where error is minimised to fit an N-polynomial to the points for $N \in \mathbb{N}$.

Sampling Joint Space Points from the Fitted Function Finally, the Cartesian Space Trajectory is obtained by sampling joint coordinates from the the polynomial function described above. This creates an approximation of Cartesian motion in the AL5D despite the execution of a trajectory in the joint space. Results are shown in Figure 9 and a video is included in Section A.1.



(a) Plot rotated by 0° (b) Plot rotated by 45° (c) Plot rotated by 90°

Figure 9: Plot of Task Execution by Cartesian Space Trajectories

1.2.5 Obstacle Avoidance Trajectory

In this section, we propose an adaptation of the Bug 1 algorithm for obstacle avoidance to operate on 3D bounding boxes. Following this, we show that the adaptation enables the AL5D to navigate around an obstacle present in the light bulb task.

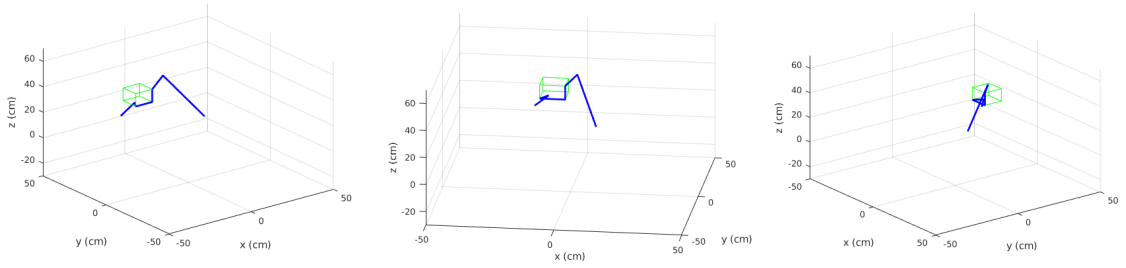
Firstly, we pose obstacle avoidance as a 2D problem to implement Bug 1. We assume the robot is equipped with a sensor able to localise obstacles in 2D bounding boxes output as polygons, defined as a counter-clockwise ordered list of vertices.

Let $distance(P_i, P_j)$ be the euclidean distance between points P_i, P_j. To generate a path between points P_1 and P_2 using the Bug 1 algorithm, a ray is traced from $P1$ to $P2$ using a sufficiently small step size - we choose $\frac{1}{5 \times distance(P1,P2)}$. For any step n along the ray, if the point P_n is not in an obstacle but P_{n+1} is, the algorithm begins to circle the obstacle's 2D bounding box $box2d$.

By visiting each point in the bounding box, we find the entry point $P^{entry}_{box2d} = min(distance(P^m_{box2d}, P1) | P^m_{box2d} \in box2d)$ at index $entry$ of the polygon, and exit point $P^{exit}_{box2d} = min(distance(P^m_{box2d}, P2) \,|\, P^m_{box2d} \in box2d)$ at index $exit$ of the polygon. The robot traverses the polygon by visiting each point between the entry and exit points, where the path from $P1$ to P^{exit}_{box2d} output by Bug 1 is thus $(P1, P_n, P^{entry}_{box2d}, P^{entry+1}_{box2d}, ..., P^{exit-1}_{box2d}, P^{exit}_{box2d})$. An obstacle avoidance path can thus be obtained from $P1$ to $P2$ by recursively calling Bug 1 from the the exit point of any obstacle traversed to the goal point $P2$.

Secondly, to adapt Bug 1 to 3D bounding boxes, we choose an entry point that is incident to a 2D plane on which Bug 1 can be solved.

Let $box3d$ be the 3D bounding box of an obstacle, consisting of 6 planes. Let P_n be a point along the ray from $P1$ to $P2$ that is not in $box3d$, and $P_n + 1$ be the next point along the ray that is in $box3d$. Let $plane_{entry}$ be the plane through which the ray enters the bounding box. We choose the entry point $P^{entry}_{box3d} = min(distance(P_n, P^m_{box3d}) \,|\, P^m_{box3d} \in box3d)$. Since $box3d$ is cuboidal, P^{entry}_{box3d} will always be incident to $plane_{entry}$ given a sufficiently small step size, meaning (P_n, P^{entry}_{box3d}) will not travel through $box3d$. The 2D Bug 1 algorithm described above is then performed on any plane incident to P^{entry}_{box3d}, where the point closest to $P2$ will be chosen as the exit point. We implement a simplified version of this algorithm that performs Bug 1 only on one of the adjacent planes, executing Cartesian Space Trajectories between each point in the output path. The results are illustrated by Figures 10 and 11 with paths drawn in blue and obstacles in green. A video is included in Section A.1.



(a) Plot rotated by 0° (b) Plot rotated by 45° (c) Plot rotated by 90°

Figure 10: Plot of Sub-task Execution by Object Avoidance Trajectories

1.2.6 Conclusion on Trajectories

Quantitatively, we note that Joint Space trajectories took a longer average time of approximately 0.334s to compute, while Cartesian space trajectories took an average of 0.00177s. This is due to high time complexity of solving linear equations in Joint Space Trajectories.

Qualitatively, Figures 8 and 9 show that Joint Space Trajectories are less efficient than Cartesian Space trajectories as they generate longer paths between poses for the same task. However, from videos in Section A.1, we feel that Joint Space Trajectories appear to move more predictably than Cartesian Space Trajectories, making them more user-friendly for human-robot interaction.

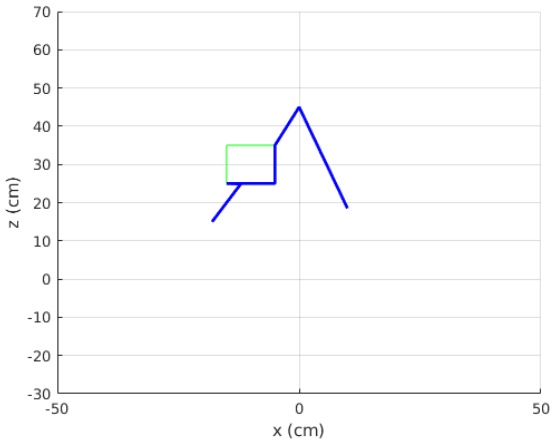
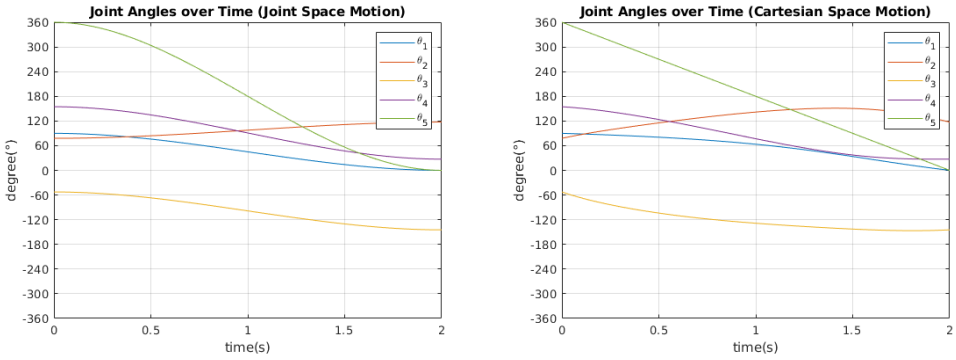


Figure 11: Plot of 2D Plane on which Bug 1 Executed for the Task

Figure 12 compares joint angles over time in the AL5D during "returning to park" of the task. We observe that the Joint Space Trajectory forms cubic curves for each joint, while the Cartesian Space Trajectory has a mix of polynomials.



(a) Joint space trajectory (Free motion) (b) Cartesian space trajectory (Linear motion)

Figure 12: Plot of each joint angle over time for "returning to park"

From the Obstacle Avoidance Trajectory video (Section A.1), we observe the gripper fingers colliding with the object. This can be mitigated by inflating bounding boxes of obstacles by the length of the gripper. Further work can be undertaken to prevent collision of the arm links with obstacles.

2 Part 2: Planar Parallel Robot

2.1 Inverse Kinematics

By given end effector position and its orientation (x_C, y_C, α), the diagram of a single link of the planar parallel robot can be drawn as illustrated in figure 13.

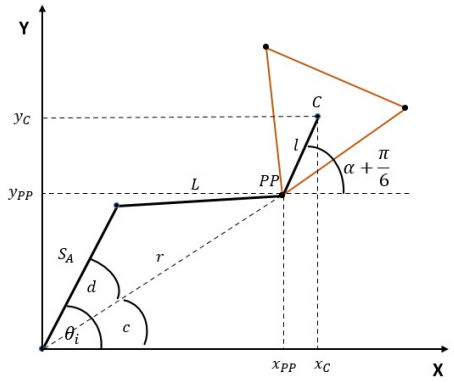


Figure 13: Diagram of a single link of planar parallel robot

and the angles of θ_i can be calculated using following closed form expression:

$$x_{PP} = x_C - l \cdot cos(\alpha + \pi/6) \tag{21}$$

$$y_{PP} = y_C - l \cdot sin(\alpha + \pi/6) \tag{22}$$

where l is the distance between PP_i and c

$$r = \sqrt{x_{PP}^2 + y_{PP}^2} \tag{23}$$

$$c = tan^{-1}(y_{PP}/x_{PP}) \tag{24}$$

$$d = cos^{-1}((S_A^2 + r^2 - L^2)/(2S_A r)) \tag{25}$$

$$\theta_i = c \pm d \tag{26}$$

To get other angle θ of other origin, since other links have the same principle for calculating the angle θ, we use transformation matrix to convert the end effector position and its orientation to the new X and Y coordinate, where 2^{nd} and 3^{rd} origin rotate around z-axis for 120° and 240° respectively, the transformed origins are illustrated in figure 14. The transformation matrix from 1^{st} origin to 2^{nd} and 3^{rd} origin can be achieved using DH convention method.

$$T12 = \begin{bmatrix} 1 & 0 & 0 & R \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} cos(120°) & -cos(120°)sin(120°) & 0 & 0 \\ sin(120°) & cos(120°) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{27}$$

$$T13 = \begin{bmatrix} cos(240°) & -cos(240°)sin(240°) & 0 & 0 \\ sin(240°) & cos(240°) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & R \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{28}$$

The position of the end effector respected to 2^{nd} origin and 3^{rd} origin can be calculated by using transformation matrix.

$$P2 = T21 \cdot P1 \Rightarrow P2 = T12^{-1} \cdot P1 \tag{29}$$

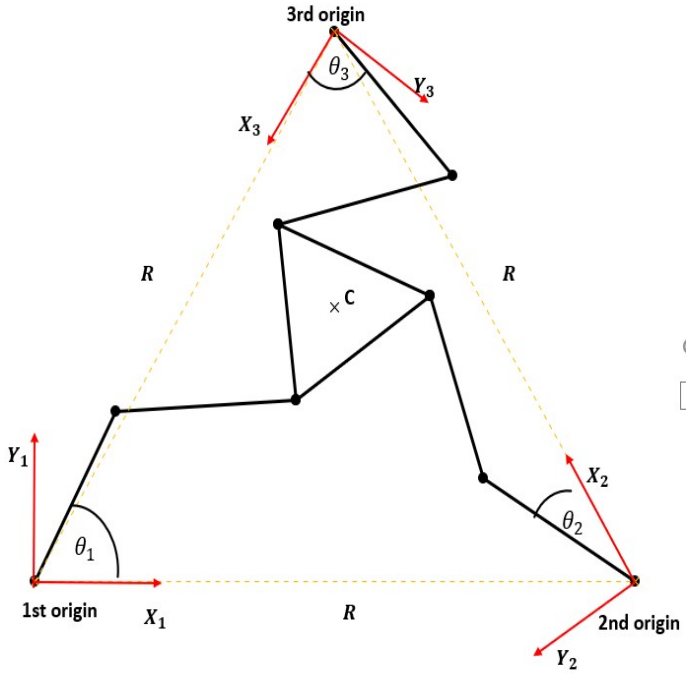


Figure 14: Diagram of a single link of planar parallel robot

$$P3 = T31 \cdot P1 \Rightarrow P3 = T13^{-1} \cdot P1 \tag{30}$$

where $P1$ is the position of the center plate respected to first origin in form of $\begin{bmatrix} x_C \\ y_C \\ 0 \\ 1 \end{bmatrix}$

Finally θ_2 and θ_3 can be calculated using transformed end effector positions with equation (21) to (26), where α remains the same. For example, by using this method, the inverse kinematics of $x_C = 0$, $y_C = 0$, and $\alpha = 0$ we can achieve $\theta_1 = 76.97°$, $\theta_2 = 48.32°$, and $\theta_3 = 92.93°$, the configuration is shown in figure 15

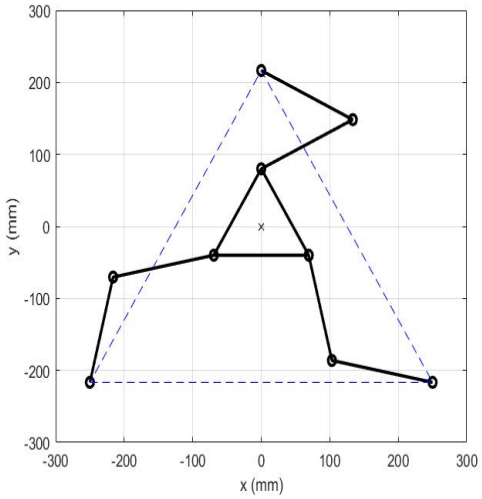


Figure 15: Configuration of planar parallel robot when $x_C = 0$, $y_C = 0$, and $\alpha = 0$

2.2 Workspace

In this report, the work space of the planar parallel robot is demonstrated by iterating over all possible values of the end effector position, x_C, y_C, and orientation, α, in a discretised space. The inverse kinematics is used to calculate θ_1, θ_2, and θ_3, if all angles can be solved for the selected configuration in real number plane, the configuration is in the work space of the planar parallel robot. In this experiment, x_C, y_C each range from (-200mm, 200mm) over 200 linearly spaced samples, and α ranges from (-45°, 45°) over 10 linearly spaced samples. The work space of this planar parallel robot is illustrated in figure 16.

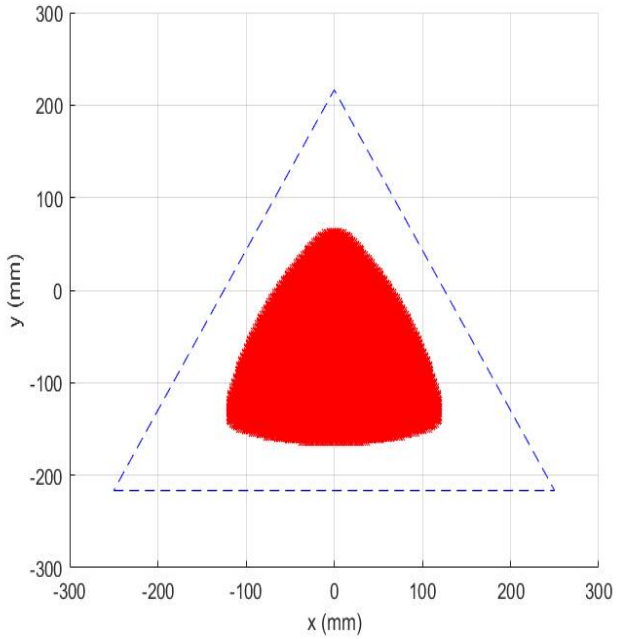


Figure 16: Work space of planar parallel robot

Learning Rate	Mean Loss
1e-05	72.315
1.67e-05	33.515
2.78e-05	17.06
4.64e-05	6.935
7.74e-05	6.615
1.29e-04	6.604
2.15e-04	inf
3.59e-04	NaN
5.99e-04	NaN
1e-03	NaN

Table 2: Mean Loss over 1000 optimisations at 100 iterations

3 Part 3: Gradient Descent for the Inverse Kinematics of a Serial Robot

3.1 Introduction

Gradient Descent is an iterative optimisation algorithm used to converge to local minima in differentiable functions. In this section, we investigate Gradient Descent for the optimisation of Inverse Kinematics (IK) in a serial robot arm through an experiment on a model of the Lynxmotion AL5D (AL5D). The experiment aims to measure the success rate and computation time of Gradient Descent for IK in the AL5D, and assess the effects of random restarts on the success rate.

3.2 Implementation

Forward Kinematics of the AL5D is a differentiable function that can be used for Gradient Descent. We use PyTorch to re-implement FK detailed in Section 1.1.1, leveraging the `torch.autograd` differentiation engine to abstract the calculation of gradients required during the optimisation step.

Given a set of joint values to optimise, FK represents the current pose of the robot. We calculate a loss value for optimisation by the sum of squared errors between each dimension of the current and target poses.

Backpropagation is achieved using the `backward()` function enabled by `torch.autograd`, computing the gradient at each joint with respect to the loss by repeated application of the chain rule (Fei-Fei Li 2021).

Finally, the optimisation step subtracts from each joint a velocity scaled by the algorithm's learning rate, where velocities are integrated from the computed gradient at each joint to form a momentum update. This is performed using PyTorch's implementation of Stochastic Gradient Descent (SGD), given a user-defined learning rate and momentum coefficient. Repeated small steps in the direction of steepest descent of the loss at each joint minimises the error, yielding joint values that approximate IK for a given target pose.

3.3 Preliminary Work

To find an optimal learning rate, we sample 10 learning rates logarithmically spaced between `lr=1e-05` and `lr=1e-03`. For each value of `lr`, 1000 optimisations are performed between random current and target poses of the AL5D, measuring the loss after 100 iterations with a recommended momentum of 0.9 (Fei-Fei Li 2021).

Table 2 shows the preliminary results. Higher learning rates are preferable, however the solution diverges for learning rates greater than 2.5e-04. 1.29e-04 converged to the least loss on average, and was thus chosen for the optimal learning rate.

3.4 Methodology

To test Gradient Descent for IK, we fix the learning rate at 1.29e-04 and momentum at 0.9, and define an arbitrary success criteria of 150 iterations – approximately 6Hz. A solution is found when error falls below

Position Tolerance (cm)	Orientation Tolerance (degree)	Mean Time (s)	SD of Time (s)	Success Rate (%)
2.0	**2.0**	0.0785	0.0175	93.8
1.0	**1.0**	0.0925	0.0177	89.9
0.5	**0.5**	0.1057	0.0174	83.7
0.25	**0.25**	0.1200	0.0157	78.0
0.125	**0.125**	0.1318	0.0143	72.2

Table 3: Results on different tolerances, < 150 iterations success criteria

Position Tolerance (cm)	Orientation Tolerance (degree)	Success Rate (%)			
		0 RRs	**1 RR**	**2 RRs**	**3 RRs**
2.0	**2.0**	94.2	97.4	98.5	98.8
1.0	**1.0**	90.0	96.7	98.3	98.9
0.5	**0.5**	83.7	91.9	95.2	96.9
0.25	**0.25**	78.3	87.4	91.1	93.3
0.125	**0.125**	70.7	81.1	85.1	88.6

Table 4: Results with Random Restarts (RRs), < 150 iterations success criteria

a given tolerance, and we vary the positional and orientational tolerances logarithmically from 2.0 to 0.125, in cm and degrees, respectively.

For each test case, 1000 optimisations are performed between random current and target poses of the AL5D obtained by FK on random joint values, running until a solution is found or the success criteria fails. All joints are assumed to range from $-180°$ to $180°$ in the generation of random poses. The success rate (%) and times taken to find successful solutions are recorded.

3.5 Results

3.5.1 Performance and Success Rate

The quantitative results of the experiment outlined in 3.4 are shown in Table 3. Qualitative results are shown in Figure 17.

Gradient Descent is shown to achieve a 93.8% success rate for IK with a mean time of 0.0785s for tolerances of 2cm and $2°$, where success rates decrease and mean time taken increases as tolerances are lowered. This occurs as gradients decrease when the error becomes small, causing convergence by Gradient Descent to slow down. Standard deviation of time taken decreases with lower tolerances, however this is likely due to the omission of unsuccessful data points that take longer times.

Qualitatively, we observe the (x, y, z, ψ, μ) definition of pose used (Figure 1) exhibits a one-to-many relationship between end-effector poses in the AL5D frame and the world frame, shown in Figures 17a and 17d where end-effectors appear flipped in successful IK solutions.

3.5.2 Success Rate on Random Restarts

To test the performance of Random Restarts (RRs), we re-run the experiment outlined in 3.4, however restarting from a random pose if optimisation to a given target fails. This is repeated up to 3 times, and the success rates are recorded.

As shown in Table 4, RRs significantly increase the success rates across all tolerances, where a higher number of restarts yields higher success rates.

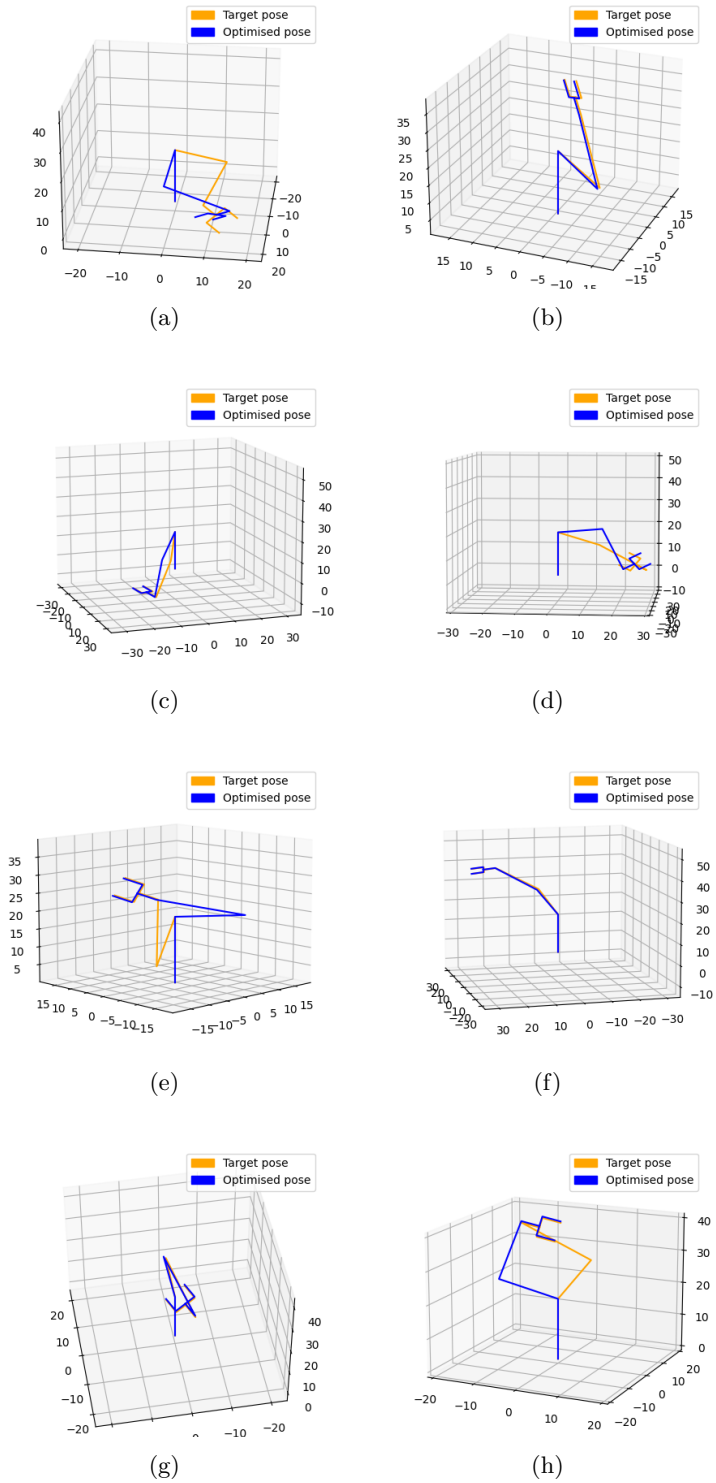


Figure 17: Examples of Successful Optimisation of IK by Gradient Descent for a Positional Tolerance of 0.5cm and Rotational Tolerance of 1°

3.6 Conclusion

We conclude that IK can successfully be approximated using Gradient Descent at a speed of 6Hz, and further show that Random Restarts can be used to increase the success rate of finding solutions. This experiment is limited in that IK solutions do not account for self-collision, where further work can be undertaken to enforce constraints during optimisation.

References

Fei-Fei Li, Ranjay Krishna, D. X. (2021), 'Cs231n convolutional neural networks for visual recognition'. Stanford University.
URL: *http://cs231n.stanford.edu/*

Jafari, A. (2021), 'Robotic fundamentals'. University of the West of England, Bristol.

Ostertagová, E. (2012), 'Modelling using polynomial regression', *Procedia Engineering* **48**, 500–506. Modelling of Mechanical and Mechatronics Systems.
URL: *https://www.sciencedirect.com/science/article/pii/S1877705812046085*

Appendices

A Videos and Resources

A.1 Links to Videos

- Joint Space Trajectory: `https://youtu.be/s36cvkFZ8No`
- Cartesian Space Trajectory: `https://youtu.be/GydlTabq0iE`
- Obstacle Avoidance Trajectory: `https://youtu.be/Le-Bzpz2gT8`
- Gradient Descent for IK: `https://youtu.be/nilcjl7FEQs`

A.2 Link to GitHub Repository

Project code is available at: `https://github.com/joejeffcock-pg/ufmf4x_cw1`

B README and Code Listing

1. README.md
2. requirements.txt
3. Part 1
4. Part 2
5. Part 3

Serial and Parallel Robot Kinematics

System Requirements

Matlab R2021b:

- Curve Fitting Toolbox (version 3.6)
- Symbolic Math Toolbox (version 9.0)

Python 3.8 (included in `requirements.txt`):

```
cycler==0.11.0
fonttools==4.28.5
kiwisolver==1.3.2
matplotlib==3.5.1
numpy==1.22.0
packaging==21.3
Pillow==9.0.0
pyparsing==3.0.6
python-dateutil==2.8.2
six==1.16.0
torch==1.10.1
typing_extensions==4.0.1
```

Scripts

Part 1: Serial Robot

Matlab scripts for plotting:

- `workspace.m`
- `plot_task.m`
- `free_motion.m`
- `cartesian_motion.m`
- `avoidance_trajectory.m`

Part 2: Parallel Robot

Matlab scripts for plotting:

- `final_course_work_parallel.m`

Part 3: Gradient Descent for Inverse Kinematics

Visualisation of optimisation:

```
usage: lynxmotion_grad.py [-h] [--display] [--animate] [--verbose]
```

```
optional arguments:
  -h, --help  show this help message and exit
  --display   show strat/end plots of IK optimisation
  --animate   show animated plot of IK optimisation
  --verbose   verbose output during IK optimisation
```

Python scripts for testing and evaluation:

- test_learning_rate.py
- test_tolerance.py
- test_restarts.py
- eval_tolerance.py
- eval_learning_rate.py
- eval_restarts.py

```
cycler==0.11.0
fonttools==4.28.5
kiwisolver==1.3.2
matplotlib==3.5.1
numpy==1.22.0
packaging==21.3
Pillow==9.0.0
pyparsing==3.0.6
python-dateutil==2.8.2
six==1.16.0
torch==1.10.1
typing_extensions==4.0.1
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_1/avoidance_trajectory.m

```
arm = LynxmotionAL5D();
obstacle = Cuboid([-10 27 30], [5 5 5]);

poses = [
%     33.25, 0 , 14.2, -pi/2, 0; % parked
%     10, 0 , 18.5, -pi/2, 0; % parked
%     0    , 20, 45,  pi/2, 2*pi; % remove bulb 1
%     0    , 20, 45,  pi/2, 0; % remove bulb 2
%     7.5   , 10, -10  , -pi/2, 0; % dispose of bulb
%     -15   , 25, 15  , 0, pi/2; % pick new bulb 1
    -18   , 30, 15  , 0, pi/2; % pick new bulb 2
    0    , 20, 45,  pi/2, 0; % replace bulb 1
    0    , 20, 45,  pi/2, 2*pi; % replace bulb 2
    10, 0 , 18.5, -pi/2, 0; % parked
];

c_samples = 50; % cartesian samples between each pose
j_samples = 50; % joint samples to form our trajectory
poly = 10;
points = zeros(0, 3); % store points to plot trajectory
az = -37.5; % initial azimuth angle for view

for i = 0:size(poses,1)-2
    p1 = poses(i+1,:);
    p2 = poses(i+2,:);

    % compute joint trajectories
    path = bug_algorithm(p1, p2, obstacle);
    for j = 1:size(path,1)-1
        bugp1 = path(j+0,:);
        bugp2 = path(j+1,:);
        trajectory = cartesian_space_trajectory(arm, bugp1, bugp2, c_samples, j_samples,
poly);

        for k = 1:c_samples
            j1 = trajectory(k,:);

            % store points to plot trajectory
            eef = arm.forward_kinematics(j1(1),j1(2),j1(3),j1(4),j1(5));
            points = [points;[eef(1,4) eef(2,4) eef(3,4)]];

            % animated joint trajectory
            figure(1);
            arm.draw(j1(1),j1(2),j1(3),j1(4),j1(5));
            hold on;
            obstacle.draw()

plot3(points(1:size(points,1),1),points(1:size(points,1),2),points(1:size(points,1),3),'-
','Linewidth',2,'Color',[0 0 1 0.5]);

            axis([-50 50 -50 50 -30 70]);
            view([az, 30]);
            az = az - 0.2;
            xlabel('x (cm)');
            ylabel('y (cm)');
            zlabel('z (cm)');
            grid on;
            pause(0.01);
            hold off;
        end
    end
end
```

```
% plot trajectory
figure(2);
obstacle.draw()
hold on
plot3(points(:,1),points(:,2),points(:,3),'b-','Linewidth',2);
hold off
axis([-50 50 -50 50 -30 70])
xlabel('x (cm)');
ylabel('y (cm)');
zlabel('z (cm)');
grid on;

figure(3);
obstacle.draw()
hold on
plot3(points(:,1),points(:,2),points(:,3),'b-','Linewidth',2);
hold off
axis([-50 50 -50 50 -30 70])
xlabel('x (cm)');
ylabel('y (cm)');
zlabel('z (cm)');
grid on;
view([-37.5 + 45, 30]);

figure(4);
obstacle.draw()
hold on
plot3(points(:,1),points(:,2),points(:,3),'b-','Linewidth',2);
hold off
axis([-50 50 -50 50 -30 70])
xlabel('x (cm)');
ylabel('y (cm)');
zlabel('z (cm)');
grid on;
view([-37.5 + 90, 30]);

figure(5);
obstacle.draw()
hold on
plot3(points(:,1),points(:,2),points(:,3),'b-','Linewidth',2);
hold off
axis([-50 50 -50 50 -30 70])
xlabel('x (cm)');
ylabel('y (cm)');
zlabel('z (cm)');
grid on;
view([0, 0]);



function path = get_closest_plane(obj, start, goal)
    % choose the closest plane and entry point
    min_dist = inf;
    entry_args = [1 1]; % default: plane 1 point 1
    for i=1:size(obj.skeleton,1)
        for j=1:5
            point = zeros(1,3);
            point(:) = obj.skeleton(i,j,:);
            dist = norm(start - point);
            if dist < min_dist
                min_dist = dist;
                entry_args = [i j];
            end
        end
    end
```

```
    % perform bug1 on 2D plane
    % exit point is point closest to goal
    min_dist = inf;
    exit_args = [1 1]; % default: plane 1 point 1
    for j=1:5
        point = zeros(1,3);
        point(:) = obj.skeleton(entry_args(1),j,:);
        dist = norm(goal - point);
        if dist < min_dist
            min_dist = dist;
            exit_args = [entry_args(1) j];
        end
    end

    % return points along plane from entry point to exit point
    if entry_args(2) <= exit_args(2)
        path = obj.skeleton(entry_args(1), entry_args(2):exit_args(2), :);
    else
        path = [obj.skeleton(entry_args(1), entry_args(2):4, :)
obj.skeleton(entry_args(1), 1:exit_args(2), :)];
    end
end

function path = bug_algorithm(start, goal, obstacle)
    dist = norm(start(1:3) - goal(1:3));
    steps = round(dist) * 5;
    poses = zeros(steps, 5);
    for j = 1:5
        poses(:,j) = linspace(start(j), goal(j), steps);
    end

    path = [start; goal];
    for i = 1:steps - 1
        pose = poses(i,:);
        next = poses(i+1,:);
        if obstacle.point_inside(next(1:3)) == 1
            % get closest plane to point in pose
            plane = get_closest_plane(obstacle,next(1:3), goal(1:3));
            plane = reshape(plane,size(plane,2), 3);

            % add psi,mu dims to plane to create a path
            bug_path = zeros(size(plane,1),5);
            bug_path(:,1:3) = plane;
            bug_path(:,4) = pose(4);
            bug_path(:,5) = pose(5);

            % create bug solution
            path = [start; pose; bug_path; goal];
            break
        end
    end
end
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_1/cartesian_motion.m

```
clc
clear all
close all

arm = LynxmotionAL5D();

poses = [
%     33.25, 0 , 14.2, -pi/2, 0; % parked
    10, 0 , 18.5, -pi/2, 0; % parked
    0    , 20, 45,  pi/2, 2*pi; % remove bulb 1
    0    , 20, 45,  pi/2, 0; % remove bulb 2
    7.5   , 10, -10  , -pi/2, 0; % dispose of bulb
    -15   , 25, 15  , 0, pi/2; % pick new bulb 1
    -18   , 30, 15  , 0, pi/2; % pick new bulb 2
    0    , 20, 45,  pi/2, 0; % replace bulb 1
    0    , 20, 45,  pi/2, 2*pi; % replace bulb 2
    10, 0 , 18.5, -pi/2, 0; % parked
];

t_i = 0; % time at pose i
t_f = 2; % time at pose f = i + 1
c_samples = 50; % cartesian samples between each pose
j_samples = 50; % joint samples to form our trajectory
poly = 10;
points = zeros(size(poses,1) * c_samples, 3); % store points to plot trajectory
az = -37.5; % initial azimuth angle for view

q1_array = zeros(1,c_samples);
q2_array = zeros(1,c_samples);
q3_array = zeros(1,c_samples);
q4_array = zeros(1,c_samples);
q5_array = zeros(1,c_samples);

for i = 0:size(poses,1)-2
    p1 = poses(i+1,:);
    p2 = poses(i+2,:);

    % compute joint trajectories
    trajectory = cartesian_space_trajectory(arm, p1, p2, c_samples, j_samples, poly);

    for j = 1:c_samples
        j1 = trajectory(j,:);

        % store points to plot trajectory
        eef = arm.forward_kinematics(j1(1),j1(2),j1(3),j1(4),j1(5));
        points(i*c_samples+j,1) = eef(1,4);
        points(i*c_samples+j,2) = eef(2,4);
        points(i*c_samples+j,3) = eef(3,4);

        % animated joint trajectory
        figure(1);
        arm.draw(j1(1),j1(2),j1(3),j1(4),j1(5));
        hold on;

plot3(points(1:i*c_samples+j,1),points(1:i*c_samples+j,2),points(1:i*c_samples+j,3),'-
','Linewidth',2,'Color',[0 0 1 0.5]);

        axis([-50 50 -50 50 -30 70]);
        view([az, 30]);
        az = az - 0.2;
        xlabel('x (cm)');
        ylabel('y (cm)');
        zlabel('z (cm)');
```

```
        grid on;
        pause(0.01);
        hold off;

        q1_array(1,j) = j1(1);
        q2_array(1,j) = j1(2);
        q3_array(1,j) = j1(3);
        q4_array(1,j) = j1(4);
        q5_array(1,j) = j1(5);
    end

    % advance time
    t_i = t_i + 2;
    t_f = t_f + 2;
end

% plot trajectory
figure(2);
plot3(points(:,1),points(:,2),points(:,3),'b-','Linewidth',2);
axis([-50 50 -50 50 -30 70])
xlabel('x (cm)');
ylabel('y (cm)');
zlabel('z (cm)');
grid on;

figure(3);
plot3(points(:,1),points(:,2),points(:,3),'b-','Linewidth',2);
axis([-50 50 -50 50 -30 70])
xlabel('x (cm)');
ylabel('y (cm)');
zlabel('z (cm)');
grid on;
view([-37.5 - 45, 30]);

figure(4);
plot3(points(:,1),points(:,2),points(:,3),'b-','Linewidth',2);
axis([-50 50 -50 50 -30 70])
xlabel('x (cm)');
ylabel('y (cm)');
zlabel('z (cm)');
grid on;
view([-37.5 - 90, 30]);

% plot last motion angle vs time
time_steps = linspace(0,2,c_samples);

figure(5);

plot(time_steps,q1_array*180/pi)
hold on
plot(time_steps,q2_array*180/pi)
plot(time_steps,q3_array*180/pi)
plot(time_steps,q4_array*180/pi)
plot(time_steps,q5_array*180/pi)
legend('\theta_1','\theta_2','\theta_3','\theta_4','\theta_5')
xlabel('time(s)')
ylabel('degree(°)')
xlim([0,2])
ylim([-360,360])
yticks(-360:60:360);
grid on
title('Joint Angles over Time (Cartesian Space Motion)')
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_1/cartesian_space_trajectory.m

```
function trajectory = cartesian_space_trajectory(arm,p1,p2,c_samples,j_samples,poly)
    % interpolate between P1 and P2 to sample poses
    % equivalent to f(t) = P1(1-t) + P2(t) for 0<=t<=1
    sample_poses = zeros(c_samples, length(p1));
    for j = 1:length(p1)
        sample_poses(:,j) = linspace(p1(j), p2(j), c_samples);
    end

    % compute IK for sampled poses in cartesian space
    % to obtain c_samples in joint space coordinates
    sample_joints = zeros(c_samples,5);
    for i = 1:size(sample_poses,1)
        ps = sample_poses(i,:);
        sample_joints(i,:) = arm.inverse_kinematics(ps(1),ps(2),ps(3),ps(4),ps(5));
    end

    % for each joint
    trajectory = zeros(j_samples,5);
    for i = 1:5
        % fit a polynomial to the joint space coordinates over time
        yy1 = sample_joints(:,i);
        xx1 = linspace(0,1,c_samples); % increments of 1 unit
        p = polyfit(xx1,yy1,poly);

        % resample from the fit polynomial
        % to obtain cartesian trajectory in joint space
        xx2 = linspace(0,1,j_samples); % increments of 1 unit
        trajectory(:,i) = polyval(p,xx2,poly);
    end
end
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_1/Cuboid.m

```
classdef Cuboid
    properties
        x = [-1 1]
        y = [-1 1]
        z = [-1 1]
        skeleton
        offset
    end
    methods
        function obj = Cuboid(offset, scale)
            % scale box
            obj.x = obj.x * scale(1);
            obj.y = obj.y * scale(2);
            obj.z = obj.z * scale(3);

            % define cross sections
            cross = zeros(2,5,3);
            cross(1,:,:) = [obj.x(1) 0 obj.x(1); obj.x(2) 0 obj.x(1); obj.x(2) 0
obj.x(2); obj.x(1) 0 obj.x(2); obj.x(1) 0 obj.x(1)];
            cross(2,:,:) = [0 obj.y(1) obj.x(1); 0 obj.y(2) obj.x(1); 0 obj.y(2)
obj.x(2); 0 obj.y(1) obj.x(2); 0 obj.y(1) obj.x(1)];

            % apply offsets
            obj.offset = offset;
            cross(:,:,1) = cross(:,:,1) + offset(1);
            cross(:,:,2) = cross(:,:,2) + offset(2);
            cross(:,:,3) = cross(:,:,3) + offset(3);

            % create skeleton from cross sections
            obj.skeleton = zeros(4,5,3);
            obj.skeleton(1,:,:) = cross(1,:,:) + reshape([0 obj.y(1) 0],1,1,3);
            obj.skeleton(2,:,:) = cross(1,:,:) + reshape([0 obj.y(2) 0],1,1,3);
            obj.skeleton(3,:,:) = cross(2,:,:) + reshape([obj.x(1) 0 0],1,1,3);
            obj.skeleton(4,:,:) = cross(2,:,:) + reshape([obj.x(2) 0 0],1,1,3);
        end

        function draw(obj)
            hold_start = ishold;
            % draw XZ sides
            for i=1:size(obj.skeleton,1)
                plot3(obj.skeleton(i,:,1),obj.skeleton(i,:,2),obj.skeleton(i,:,3),'g-')
                if hold_start == 0
                    hold on
                end
            end
            if hold_start == 0
                hold off
            end
        end

        function inside = point_inside(obj, point)
            inside = 0;
            px = point(1);
            py = point(2);
            pz = point(3);

            ox = obj.x + obj.offset(1);
            oy = obj.y + obj.offset(2);
            oz = obj.z + obj.offset(3);
            if px > ox(1) && px < ox(2) && py > oy(1) && py < oy(2) && pz > oz(1) && pz
< oz(2)
                inside = 1;
            end
```

```
            end
        end
    end
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_1/free_motion.m

```
arm = LynxmotionAL5D();

poses = [
%     33.25, 0 , 14.2, -pi/2, 0; % parked
    10, 0 , 18.5, -pi/2, 0; % parked
    0    , 20, 45,  pi/2, 2*pi; % remove bulb 1
    0    , 20, 45,  pi/2, 0; % remove bulb 2
    7.5   , 10, -10  , -pi/2, 0; % dispose of bulb
    -15   , 25, 15  , 0, pi/2; % pick new bulb 1
    -18   , 30, 15  , 0, pi/2; % pick new bulb 2
    0    , 20, 45,  pi/2, 0; % replace bulb 1
    0    , 20, 45,  pi/2, 2*pi; % replace bulb 2
    10, 0 , 18.5, -pi/2, 0; % parked
];

t_i = 0; % time at pose i
t_f = 2; % time at pose f = i + 1
samples = 50; % samples between each pose
points = zeros(size(poses,1) * samples, 3); % store points to plot trajectory
az = -37.5; % initial azimuth angle for view

for i = 0:size(poses,1)-2
    p1 = poses(i+1,:);
    p2 = poses(i+2,:);

    % compute IK for pose i and i + 1
    j1 = arm.inverse_kinematics(p1(1), p1(2), p1(3), p1(4), p1(5));
    j2 = arm.inverse_kinematics(p2(1), p2(2), p2(3), p2(4), p2(5));

    % compute joint trajectories
    q1_spj = joint_space_trajectory(j1(1),j2(1),0,0,t_i,t_f,samples);
    q2_spj = joint_space_trajectory(j1(2),j2(2),0,0,t_i,t_f,samples);
    q3_spj = joint_space_trajectory(j1(3),j2(3),0,0,t_i,t_f,samples);
    q4_spj = joint_space_trajectory(j1(4),j2(4),0,0,t_i,t_f,samples);
    q5_spj = joint_space_trajectory(j1(5),j2(5),0,0,t_i,t_f,samples);

    for j = 1:samples
        % store points to plot trajectory
        eef =
arm.forward_kinematics(q1_spj(j),q2_spj(j),q3_spj(j),q4_spj(j),q5_spj(j));
        points(i*samples+j,1) = eef(1,4);
        points(i*samples+j,2) = eef(2,4);
        points(i*samples+j,3) = eef(3,4);

        % animated joint trajectory
        figure(1);
        arm.draw(q1_spj(j),q2_spj(j),q3_spj(j),q4_spj(j),q5_spj(j));
        hold on;

plot3(points(1:i*samples+j,1),points(1:i*samples+j,2),points(1:i*samples+j,3),'-
','Linewidth',2,'Color',[0 0 1 0.5]);

        axis([-50 50 -50 50 -30 70]);
        view([az, 30]);
        az = az - 0.2;
        grid on;
        pause(0.01);
        hold off;
    end

    % advance time
    t_i = t_i + 2;
    t_f = t_f + 2;
```

```
end

% plot trajectory
figure(2);
plot3(points(:,1),points(:,2),points(:,3),'b-','Linewidth',2);
axis([-50 50 -50 50 -30 70])
grid on;

% plot last motion angle vs time
time_steps = linspace(0,2,samples);

figure(3);
plot(time_steps,q1_spj*180/pi)
hold on
plot(time_steps,q2_spj*180/pi)
plot(time_steps,q3_spj*180/pi)
plot(time_steps,q4_spj*180/pi)
plot(time_steps,q5_spj*180/pi)
legend('\theta_1','\theta_2','\theta_3','\theta_4','\theta_5')
xlabel('time(s)')
ylabel('degree(°)')
xlim([0,2])
ylim([-360,360])
yticks(-360:60:360);
grid on
title('Joint Angles over Time (Joint Space Motion)')
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_1/joint_space_trajectory.m

```
function trajectory = joint_space_trajectory(theta_i, theta_f, velocity_i, velocity_f,
t_i, t_f, samples)
    syms a0 a1 a2 a3;

    % define system of equations
    eq1 = 0 == a0 + a1*t_i + a2*t_i^2 + a3*t_i^3 - theta_i;
    eq2 = 0 == a0 + a1*t_f + a2*t_f^2 + a3*t_f^3 - theta_f;
    eq3 = velocity_i == a1 + 2*a2*t_i + 3*a3*t_i^2;
    eq4 = velocity_f == a1 + 2*a2*t_f + 3*a3*t_f^2;

    % compute polynomial solution (degree 3)
    [a0, a1, a2, a3] = vpasolve(eq1,eq2,eq3,eq4,a0,a1,a2,a3);
    p = cast([a3 a2 a1 a0], "double");

    % sample points along trajectory
    time_step = linspace(t_i,t_f,samples);
    trajectory = polyval(p,time_step,3);

end
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_1/LynxmotionAL5D.m

```
classdef LynxmotionAL5D
    properties (Constant)
        % link lengths (cm)
        d1 = 18.5
        d2 = 14.5
        d3 = 18.75
        d4 = 4.3
        % gripper finger lengths
        f1 = 3.2
        f2 = 4.3
    end
    properties
        eef_pose
    end
    methods
        function T = forward_kinematics(obj, q1, q2, q3, q4, q5)
            % transforms of lynxmotion
            T01  = tf_from_distal(0, deg2rad(90), obj.d1, q1);
            T12  = tf_from_distal(obj.d2, 0, 0, q2);
            T23  = tf_from_distal(obj.d3, 0, 0, q3);
            T33p = tf_from_distal(0, deg2rad(90), 0, q4);
            T3p4 = tf_from_distal(0, 0, obj.d4, 0);
            T45  = tf_from_distal(0, 0, 0, q5);

            % forward kinematics
            T = T01 * T12 * T23 * T33p * T3p4 * T45;
        end

        function joints = inverse_kinematics(obj, x, y, z, psi, mu)
            q1 = atan2(y,x);
            q5 = mu;

            % compute wrist position
            % base->wrist is a 3DoF problem
            vec_z = obj.d4 * sin(psi);
            D1 = obj.d4 * cos(psi);
            vec_x = D1 * cos(q1);
            vec_y = D1 * sin(q1);
            xw = x - vec_x;
            yw = y - vec_y;
            zw = z - vec_z;

            % solve for q2 from wrist
            r1 = sqrt(xw^2 + yw^2);
            r2 = zw - obj.d1;
            r3 = sqrt(r1^2 + r2^2);
            phi2 = atan2(r2,r1);
            phi1 = acos((obj.d3^2 - obj.d2^2 - r3^2)/(-2 * obj.d2 * r3));
            q2 = phi2 + phi1;

            % solve for q3 from wrist
            phi3 = acos((r3^2 - obj.d2^2 - obj.d3^2)/(-2*obj.d2*obj.d3));
            q3 = -(pi - phi3);

            % add pi/2 in psi to obtain rotation in robot frame
            q4 = (psi + pi/2) - q2 - q3;
            joints = [q1 q2 q3 q4 q5];
        end

        function draw(obj, q1, q2, q3, q4, q5)
            % transforms of lynxmotion
            T = zeros(7,4,4);
            T(1,:,:) = eye(4);
```

```
            T(2,:,:) = [tf_from_distal(0, deg2rad(90), obj.d1, q1)];
            T(3,:,:) = [tf_from_distal(obj.d2, 0, 0, q2)];
            T(4,:,:) = [tf_from_distal(obj.d3, 0, 0, q3)];
            T(5,:,:) = [tf_from_distal(0, deg2rad(90), 0, q4)];
            T(6,:,:) = [tf_from_distal(0, 0, obj.d4, 0)];
            T(7,:,:) = [tf_from_distal(0, 0, 0, q5)];

            % joint positions from FK
            T0eef = eye(4);
            points = zeros(7,3);
            for i = 1:7
                T0eef = T0eef * squeeze(T(i,:,:));
                points(i,1) = T0eef(1,4);
                points(i,2) = T0eef(2,4);
                points(i,3) = T0eef(3,4);
            end

            % gripper points
            T0g1f1 = T0eef  * [tf_from_distal( obj.f1, 0, 0     , 0)];
            T0g1f2 = T0g1f1 * [tf_from_distal( 0     , 0, obj.f2, 0)];
            T0g2f1 = T0eef  * [tf_from_distal(-obj.f1, 0, 0     , 0)];
            T0g2f2 = T0g2f1 * [tf_from_distal( 0     , 0, obj.f2, 0)];
            g1 = [T0eef(1:3, 4) T0g1f1(1:3, 4) T0g1f2(1:3, 4)];
            g2 = [T0eef(1:3, 4) T0g2f1(1:3, 4) T0g2f2(1:3, 4)];

            % plot points
            plot3(points(:,1), points(:,2), points(:,3), 'ko-', 'Linewidth', 2);
            hold on
            plot3(g1(1,:), g1(2,:), g1(3,:), 'k-', 'Linewidth', 2);
            plot3(g2(1,:), g2(2,:), g2(3,:), 'k-', 'Linewidth', 2);
            hold off
        end
    end
end

% Transform from DH Parameters (Distal)
function T = tf_from_distal(a, alpha, d, theta)
    T = [cos(theta) -cos(alpha)*sin(theta)  sin(alpha)*sin(theta) a*cos(theta);
         sin(theta)  cos(alpha)*cos(theta) -sin(alpha)*cos(theta) a*sin(theta);
         0           sin(alpha)             cos(alpha)            d           ;
         0           0                      0                     1            ];
end
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_1/plot_task.m

```
arm = LynxmotionAL5D();

poses = [
%     33.25, 0 , 14.2, -pi/2, 0; % parked
    10, 0 , 18.5, -pi/2, 0; % parked
    0    , 20, 45,  pi/2, 2*pi; % remove bulb 1
    0    , 20, 45,  pi/2, 0; % remove bulb 2
    7.5   , 10, -10  , -pi/2, 0; % dispose of bulb
    -15   , 25, 15  , 0, pi/2; % pick new bulb 1
    -18   , 30, 15  , 0, pi/2; % pick new bulb 2
    0    , 20, 45,  pi/2, 0; % replace bulb 1
    0    , 20, 45,  pi/2, 2*pi; % replace bulb 2
    10, 0 , 18.5, -pi/2, 0; % parked
];

for i = 1:size(poses,1)
    p1 = poses(i,:);
    joints = arm.inverse_kinematics(p1(1), p1(2), p1(3), p1(4), p1(5));

    figure(i)
    arm.draw(joints(1), joints(2), joints(3), joints(4), joints(5));
    axis([-50 50 -50 50 -30 70])
    xlabel("x (cm)");
    ylabel("y (cm)");
    zlabel("z (cm)");
    grid on;
end

figure(size(poses,1) + 1)
xx = poses(:,1);
yy = poses(:,2);
zz = poses(:,3);
plot3(xx,yy,zz)
axis([-50 50 -50 50 -30 70])
xlabel("x (cm)");
ylabel("y (cm)");
zlabel("z (cm)");
title("Path of Light Bulb Replacement Task")
grid on;
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_1/workspace.m

```
arm = LynxmotionAL5D();

% try lots of angles
t1 = linspace(deg2rad(-180),deg2rad(180),36);
t2 = linspace(deg2rad(-180),deg2rad(180),10);
t3 = linspace(deg2rad(-180),deg2rad(180),10);
t4 = linspace(deg2rad(-180),deg2rad(180),10);
t5 = linspace(deg2rad(-180),deg2rad(180),10);

% memory for samples
xwork = zeros(length(t1),length(t2),length(t3),length(t4),length(t5));
ywork = zeros(length(t1),length(t2),length(t3),length(t4),length(t5));
zwork = zeros(length(t1),length(t2),length(t3),length(t4),length(t5));

for i = 1:length(t1)
    for j = 1:length(t2)
        for k = 1:length(t3)
            for l = 1:length(t4)
                for m = 1:length(t5)
                    eef = arm.forward_kinematics(t1(i), t2(j), t3(k), t4(l), t5(m));
                    xwork(i,j,k,l,m) = eef(1,4);
                    ywork(i,j,k,l,m) = eef(2,4);
                    zwork(i,j,k,l,m) = eef(3,4);
                end
            end
        end
    end
end

% plot workspace in 3D
figure(1)
scatter3(xwork(:),ywork(:),zwork(:),'.')
xlabel("x (cm)");
ylabel("y (cm)");
zlabel("z (cm)");
title("Reachable Workspace of Lynxmotion AL5D")
axis equal

figure(2)
scatter(xwork(:),ywork(:),'.')
xlabel("x (cm)");
ylabel("y (cm)");
title("Reachable Workspace of Lynxmotion AL5D")
axis equal

figure(3)
scatter(xwork(:),zwork(:),'.')
xlabel("x (cm)");
ylabel("z (cm)");
title("Reachable Workspace of Lynxmotion AL5D")
axis equal

figure(4)
scatter(ywork(:),zwork(:),'.')
xlabel("y (cm)");
ylabel("z (cm)");
title("Reachable Workspace of Lynxmotion AL5D")
axis equal
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_2/final_course_work_parallel.m

```
clear
close all
clc

% create robot
robot = planarRobot();

% parameter for IK
px = 0;
py = 0;
phi_1 = pi/6+0;

% draw robot
draw_ik(robot, px, py, phi_1)

% draw work space of the robot
draw_ws(robot);

function draw_ik(robot, px, py, phi_1)
    %% This function takes in the position of ee and angle alpha repected to the first
origin
    %% output is the plot of the robot

    ik_angle = robot.IK(px, py, phi_1);
    theta_1 = ik_angle(1,1);
    theta_2 = ik_angle(2,1);
    theta_3 = ik_angle(3,1);

    % displace angles
    disp(theta_1*180/pi)
    disp(theta_2*180/pi)
    disp(theta_3*180/pi)

    % draw robot
    figure(1)
    robot.draw(px, py, phi_1, theta_1, theta_2, theta_3);

end

function draw_ws(robot)
    %% This function will sample the ee position and orientation
    %% and check if that configuration can be solved by IK or not
    %% if yes, that config is in work space.

    % set min max
    xmin = -200;
    xmax = 200;
    ymin = -200;
    ymax = 200;

    % set sampling rate
    samp_rate = 200;
    phi_samp_rate =20;

    % create 0 array for putting in the positions in work space
    x_result = zeros(1,samp_rate*samp_rate*phi_samp_rate);
    y_result = zeros(1,samp_rate*samp_rate*phi_samp_rate);

    % create line space
    x = linspace(xmin, xmax, samp_rate);
    y = linspace(ymin, ymax, samp_rate);
    phi = linspace(pi/6-pi/4 ,pi/6+pi/4,phi_samp_rate);
```

```
    % loop for testing the point is in the work space or not
    for i = 1:phi_samp_rate
        for j = 1:samp_rate
            for k = 1:samp_rate
                px = x(k);
                py = y(j);
                angle = phi(i);

                % do the IK fo all angles
                ik_angle = robot.IK(px, py, angle);
                theta_1 = ik_angle(1,1);
                theta_2 = ik_angle(2,1);
                theta_3 = ik_angle(3,1);

                % if all angles are real, register x and y in work space
                if  and(and(isreal(theta_1),isreal(theta_2)),isreal(theta_3))
                    x_result(k + (j-1)*samp_rate + (i-1)*samp_rate*samp_rate) = px;
                    y_result(k + (j-1)*samp_rate + (i-1)*samp_rate*samp_rate) = py;
                end
            end
        end
    end

    % plotting work space
    figure(2)
    scatter(x_result,y_result,'rx')
    hold on

    % plotting the frame
    x_offset = 500/2;
    y_offset = 500*sin(60*pi/180)/2;
    frame_x = [-x_offset x_offset 0 -x_offset];
    frame_y = [-y_offset -y_offset y_offset -y_offset];
    xlim([-300,300])
    ylim([-300,300])
    xlabel('x (mm)')
    ylabel('y (mm)')
    grid on
    plot(frame_x,frame_y,"LineStyle","--","Color",'blue');

end
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_2/planarRobot.m

```
classdef planarRobot
    properties
        Sa = 150;   % first link
        L = 150;    % second link
        R = 500;    % outer size
        r = 80;     % center plate size
        x_offset = 500/2;
        y_offset = 500*sin(60*pi/180)/2;
    end

    methods

        % function for calculating angle for each origin
        function angle = IK(obj, px, py, phi_1)

            px1 = px + obj.x_offset;
            py1 = py + obj.y_offset;

            % Transformation matrix from point 1 to 2
            T12_prime = DH_transform(obj.R,0,0,0);
            T12 = DH_transform(0,0,0, 2*pi/3);
            T12 = T12_prime*T12;
            T21 = T12^(-1);

            % Transformation matrix from point 1 to 3
            T13_prime = DH_transform(0,0,0,4*pi/3);
            T13 = DH_transform(-obj.R,0,0,0);
            T13 = T13_prime*T13;
            T31 = T13^(-1);

            %P2x = T21 * P1x
            P1 = [px1 py1 0 1]';
            P2 = T21*P1;
            P3 = T31*P1;

            % extract x y position for each pose
            px2 = P2(1,1);
            py2 = P2(2,1);
            px3 = P3(1,1);
            py3 = P3(2,1);


            % calculation each theta
            theta_1 = CalTheta(obj.Sa, obj.L, obj.r, phi_1, px1, py1);
            theta_2 = CalTheta(obj.Sa, obj.L, obj.r, phi_1, px2, py2);
            theta_3 = CalTheta(obj.Sa, obj.L, obj.r, phi_1, px3, py3);

            % angle for each origin [c+d c-d], use c+d
            angle = [theta_1
                     theta_2
                     theta_3];
        end


        function draw(obj, px, py, phi_1, theta_1, theta_2, theta_3)
            %% function for plotting the robot configuration

            px1 = px + obj.x_offset;
            py1 = py + obj.y_offset;

            % transformation matrix
            T12_prime = DH_transform(obj.R,0,0,0);
            T12 = DH_transform(0,0,0, 2*pi/3);
```

```
T12 = T12_prime*T12;
T21 = T12^(-1);

% Transformation matrix from point 1 to 3
T13_prime = DH_transform(0,0,0,4*pi/3);
T13 = DH_transform(-obj.R,0,0,0);
T13 = T13_prime*T13;
T31 = T13^(-1);

% center point viewd from different origin
P1 = [px1 py1 0 1]';
P2 = T21*P1;
P3 = T31*P1;

px2 = P2(1,1);
py2 = P2(2,1);
px3 = P3(1,1);
py3 = P3(2,1);

P = [px1 py1;
     px2 py2;
     px3 py3];

ANGLE = [theta_1;
         theta_2;
         theta_3];

% XY = 3 origins x 2 dimensions x 3 points
% 3 points = each joints of one link
XY = zeros(3,2,3);
for i =1:3
    XY(i,1,2) = obj.Sa*cos(ANGLE(i,1));
    XY(i,2,2) = obj.Sa*sin(ANGLE(i,1));
    XY(i,1,3) = P(i,1) - obj.r*cos(phi_1);
    XY(i,2,3) = P(i,2) - obj.r*sin(phi_1);
end

% change P2xy to P1xy by P1xy = T12 x P2xy
for i = 1:3
    x2 = squeeze(XY(2,1,i));
    y2 = squeeze(XY(2,2,i));
    p2 = [x2 y2 0 1]';
    p1 = T12*p2;
    XY(2,1,i) = p1(1,1);
    XY(2,2,i) = p1(2,1);
end

% change P3xy to P1xy by P1xy = T13 x P3xy
for i = 1:3
    x3 = squeeze(XY(3,1,i));
    y3 = squeeze(XY(3,2,i));
    p3 = [x3 y3 0 1]';
    p1 = T13*p3;
    XY(3,1,i) = p1(1,1);
    XY(3,2,i) = p1(2,1);
end

% prepare for drawing
x1 = squeeze(XY(1,1,:) - obj.x_offset);
y1 = squeeze(XY(1,2,:) - obj.y_offset);
x2 = squeeze(XY(2,1,:) - obj.x_offset);
y2 = squeeze(XY(2,2,:) - obj.y_offset);
x3 = squeeze(XY(3,1,:) - obj.x_offset);
y3 = squeeze(XY(3,2,:) - obj.y_offset);
```

```
            % draw center plate
            planar_x = [x1(3) x2(3) x3(3) x1(3)];
            planar_y = [y1(3) y2(3) y3(3) y1(3)];

            % draw frame
            frame_x = [x1(1) x2(1) x3(1) x1(1)];
            frame_y = [y1(1) y2(1) y3(1) y1(1)];

            % draw center point
            plot(px,py,'kx')
            hold on

            % draw each link
            plot(x1,y1,'ko-', 'Linewidth', 2)
            plot(x2,y2,'ko-', 'Linewidth', 2)
            plot(x3,y3,'ko-', 'Linewidth', 2)
            plot(planar_x, planar_y, 'ko-', 'Linewidth', 2)
            plot(frame_x,frame_y,"LineStyle","--","Color",'blue');
            xlim([-300,300])
            ylim([-300,300])
            xlabel('x (mm)')
            ylabel('y (mm)')
            hold on
            grid on
        end

    end

end



function T = DH_transform(a,alpha,d,theta)
    T = [cos(theta) -cos(alpha)*sin(theta)   sin(alpha)*sin(theta)   a*cos(theta);
         sin(theta)  cos(alpha)*cos(theta)  -sin(alpha)*cos(theta)   a*sin(theta);
         0           sin(alpha)              cos(alpha)              d           ;
         0           0                       0                       1           ;];
end


function theta = CalTheta(Sa, L, r, phi, px, py)
    %% function for calculating theta
    x = px - r*cos(phi);
    y = py - r*sin(phi);
    r = sqrt(x^2 + y^2);
    c = atan2(y,x);
    d = acos((Sa^2 + r^2 - L^2)/(2 * Sa *r));
    theta = [c+d c-d];
end
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_3/eval_learning_rate.py

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

results = {}
with open('learning_rate_loss.csv','r') as f:
    f.readline()
    for line in f:
        test = [float(v) for v in line.split(',')]
        lr = '{:.3g}'.format(test[0])
        losses = test[1:]
        results[lr] = losses

df = pd.DataFrame(data=results)
mean = df.mean()
std = df.std()
print("Mean:")
print(mean)
print("SD:")
print(std)

df_plot = df.replace(float('inf'), float('nan'))
df_plot = df.dropna(axis='columns')
sns.boxplot(data=df_plot)
plt.show()
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_3/eval_restarts.py

```python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

results = {}
with open('restart_success.csv','r') as f:
    f.readline()
    for line in f:
        test = [float(v) for v in line.split(',')]
        label = '{:.3g},{:.3g}'.format(test[0], test[1])
        restarts = int(test[2])
        successes = test[3]

        if label not in results:
            results[label] = {}
        results[label][restarts] = successes/1000 * 100

df = pd.DataFrame(data=results)
df = df.transpose()
print(df)

# success = df.ge(0)
# print(success)
# print(success.sum()/1000 * 100)

# df = df.replace(-1, float('nan'))
# mean = df.mean()
# std = df.std()
# print("Mean:")
# print(mean)
# print("SD:")
# print(std)

# sns.boxplot(data=df)
# plt.show()
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_3/eval_tolerance.py

```python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

results = {}
with open('tolerance_times.csv','r') as f:
    f.readline()
    for line in f:
        test = [float(v) for v in line.split(',')]
        label = '{:.3g},{:.3g}'.format(test[0], test[1])
        times = test[2:]
        results[label] = times

print(results.keys())
df = pd.DataFrame(data=results)
print(df)

success = df.ge(0)
print(success)
print(success.sum()/1000 * 100)

df = df.replace(-1, float('nan'))
mean = df.mean()
std = df.std()
print("Mean:")
print(mean)
print("SD:")
print(std)

sns.boxplot(data=df)
plt.show()
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_3/lynxmotion_grad.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch import optim

from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

import numpy as np
from math import sin, cos, radians, pi

import time
import argparse

class LynxmotionGrad(nn.Module):
    def __init__(self):
        super(LynxmotionGrad, self).__init__()
        self.q1 = nn.Parameter(torch.rand(1) * 2 * pi - pi)
        self.q2 = nn.Parameter(torch.rand(1) * 2 * pi - pi)
        self.q3 = nn.Parameter(torch.rand(1) * 2 * pi - pi)
        self.q4 = nn.Parameter(torch.rand(1) * 2 * pi - pi)
        self.q5 = nn.Parameter(torch.rand(1) * 2 * pi - pi)
        # link lengths (cm)
        self.d1 = 18.5
        self.d2 = 14.5
        self.d3 = 18.75
        self.d4 = 4.3
        # gripper finger lengths
        self.f1 = 3.2
        self.f2 = 4.3


    def forward_kinematics(self):
        pose = torch.zeros(5)
        q1 = self.q1
        q2 = self.q2
        q3 = self.q3
        q4 = self.q4
        q5 = self.q5

        pose[0] = torch.cos(q1)*(self.d3*torch.cos(q2 + q3) + self.d2*torch.cos(q2) -
self.d4*torch.sin(q2 + q3 + q4)) # x
        pose[1] = torch.sin(q1)*(self.d3*torch.cos(q2 + q3) + self.d2*torch.cos(q2) -
self.d4*torch.sin(q2 + q3 + q4)) # y
        pose[2] = self.d1 - self.d3*torch.sin(q2 + q3) - self.d2*torch.sin(q2) -
self.d4*torch.cos(q2 + q3 + q4) # z
        pose[3] = q2 + q3 + q4 + radians(90) # psi
        pose[4] = q5 # mu
        return pose

    def loss_function(self, target_pose):
        current_pose = self.forward_kinematics()
        error = torch.zeros(5)

        # positional error
        error[:3] = target_pose[:3] - current_pose[:3]

        # rotational error (degrees)
        error[3:] = (target_pose[3:] - current_pose[3:]) * (180.0/pi)

        loss = torch.sum(torch.pow(error, 2))
        return loss
```

```
    def l2_norm(self, target_pose):
        target_pose_np = target_pose.detach().numpy()
        current_pose_np = self.forward_kinematics().detach().numpy()

        pos_dist = np.linalg.norm(target_pose_np[:3] - current_pose_np[:3])
        psi_dist = np.linalg.norm(target_pose_np[3] - current_pose_np[3])
        mu_dist = np.linalg.norm(target_pose_np[4] - current_pose_np[4])

        return np.array([pos_dist, psi_dist, mu_dist])

    def inverse_kinematics_step(self, optimiser, target_pose):
        optimiser.zero_grad()
        loss = self.loss_function(target_pose)
        loss.backward()
        optimiser.step()

        errors = self.l2_norm(target_pose)
        return loss, errors

    def draw(self, ax, colour='gray'):
        q1 = self.q1.detach().numpy()
        q2 = self.q2.detach().numpy()
        q3 = self.q3.detach().numpy()
        q4 = self.q4.detach().numpy()
        q5 = self.q5.detach().numpy()

        # transforms of lynxmotion
        T = np.zeros((7,4,4))
        T[0,:,:] = np.eye(4)
        T[1,:,:] = tf_from_distal(0, radians(-90), self.d1, q1)
        T[2,:,:] = tf_from_distal(self.d2, 0, 0, q2)
        T[3,:,:] = tf_from_distal(self.d3, 0, 0, q3)
        T[4,:,:] = tf_from_distal(0, radians(-90), 0, q4)
        T[5,:,:] = tf_from_distal(0, 0, self.d4, 0)
        T[6,:,:] = tf_from_distal(0, 0, 0, q5)

        # joint positions from FK
        T0eef = np.eye(4)
        points = np.zeros((7,3))
        for i in range(7):
            T0eef = T0eef.dot(T[i,:,:])
            points[i,0] = T0eef[0,3]
            points[i,1] = T0eef[1,3]
            points[i,2] = T0eef[2,3]

        # gripper points
        T0g1f1 = T0eef.dot( tf_from_distal( self.f1, 0, 0      , 0))
        T0g1f2 = T0g1f1.dot(tf_from_distal( 0      , 0, self.f2, 0))
        T0g2f1 = T0eef.dot( tf_from_distal(-self.f1, 0, 0      , 0))
        T0g2f2 = T0g2f1.dot(tf_from_distal( 0      , 0, self.f2, 0))
        g1 = np.array([T0eef[:3, 3], T0g1f1[:3, 3], T0g1f2[:3, 3]])
        g2 = np.array([T0eef[:3, 3], T0g2f1[:3, 3], T0g2f2[:3, 3]])

        ax.plot3D(points[:,0], points[:,1], points[:,2], colour)
        ax.plot3D(g1[:,0], g1[:,1], g1[:,2], colour)
        ax.plot3D(g2[:,0], g2[:,1], g2[:,2], colour)


def tf_from_distal(a, alpha, d, theta):
    return np.array([[cos(theta), -cos(alpha)*sin(theta),  sin(alpha)*sin(theta),
a*cos(theta)],
                     [sin(theta),  cos(alpha)*cos(theta), -sin(alpha)*cos(theta),
a*sin(theta)],
                     [0          ,  sin(alpha)           ,  cos(alpha)
```

```
 , d              ],
                      [0            ,  0                          ,  0
 , 1              ]])


 def draw():
     plt.clf()
     ax = plt.axes(projection='3d')
     ax.set_xlim(-50,50)
     ax.set_ylim(-50,50)
     ax.set_zlim(-30,70)
     target.draw(ax, 'orange')
     robot.draw(ax, 'blue')
     t_patch = mpatches.Patch(color="orange", label="Target pose")
     o_patch = mpatches.Patch(color="blue", label="Optimised pose")
     plt.legend(handles=[t_patch, o_patch])


 if __name__ == "__main__":
     parser = argparse.ArgumentParser()
     parser.add_argument('--display', action='store_true', dest='display', help='show
strat/end plots of IK optimisation')
     parser.add_argument('--animate', action='store_true', dest='animate', help='show
animated plot of IK optimisation')
     parser.add_argument('--verbose', action='store_true', dest='verbose', help='verbose
output during IK optimisation')
     args = parser.parse_args()

     robot = LynxmotionGrad()
     optimiser = optim.SGD(robot.parameters(), lr=1.29e-04, momentum=0.9)

     # random target
     target = LynxmotionGrad()
     with torch.no_grad():
         target_pose = target.forward_kinematics()

     if args.display:
         draw()
         plt.show()

     tolerances = np.array([0.5, 0.0175, 0.0175]) # 0.5cm xyz; 1deg psi,mu
     loss, errors = robot.inverse_kinematics_step(optimiser, target_pose)

     if args.animate:
         plt.ion()
     iteration = 0
     time_start = time.time()

     # optimise until within tolerance
     while (errors > tolerances).any():
         loss, errors = robot.inverse_kinematics_step(optimiser, target_pose)
         iteration += 1

         if args.animate:
             draw()
             plt.show()
             plt.pause(1e-10)
         if args.verbose:
             print('{} loss: {:.3f} error: {}'.format(iteration, loss, errors))

     time_elapsed = time.time() - time_start
     if args.animate:
         plt.ioff()
```

```
    print('time taken: {:.2f} seconds'.format(time_elapsed))
    if args.display:
        draw()
        plt.show()
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_3/test_learning_rate.py

```
import torch
import numpy as np
from torch import optim
from lynxmotion_grad import LynxmotionGrad

learning_rates = np.logspace(-5, -3, num=10)
results = {}
no_tests = 1000
no_iterations = 100

for lr in learning_rates:
    print('Testing lr={}'.format(lr))
    losses = []

    for i in range(no_tests):
        robot = LynxmotionGrad()
        with torch.no_grad():
            target_pose = LynxmotionGrad().forward_kinematics()
        optimiser = optim.SGD(robot.parameters(), lr=lr, momentum=0.9)

        for j in range(no_iterations):
            loss, errors = robot.inverse_kinematics_step(optimiser, target_pose)
        losses.append(loss.item())

    results[lr] = losses

filename = 'learning_rate_loss.csv'
with open(filename, 'w') as f:
    f.write('lr,')
    f.write(','.join(str(i) for i in range(no_tests)))
    f.write('\n')
    for lr in results:
        f.write('{},'.format(lr))
        f.write(','.join(str(loss) for loss in results[lr]))
        f.write('\n')
print('Loss values written to {}'.format(filename))
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_3/test_restarts.py

```
import torch
import numpy as np
from torch import optim
from lynxmotion_grad import LynxmotionGrad
from collections import defaultdict

pos_tols = np.logspace(1,-3,5,base=2)
ori_tols = np.logspace(1,-3,5,base=2) * np.pi/180
results = defaultdict(int)
no_tests = 1000
no_iterations = 150 # approx 5Hz
lr = 1.29e-04

for i in range(pos_tols.size):
    pos_tol = float(pos_tols[i])
    ori_tol = float(ori_tols[i])
    print('Testing position tolerance={} orientation tolerance={}'.format(pos_tol,
ori_tol))
    tolerances = np.array([pos_tol, ori_tol, ori_tol])

    for i in range(no_tests):
        with torch.no_grad():
            target_pose = LynxmotionGrad().forward_kinematics()

        solved = 0
        for restarts in range(4):
            if not solved:
                robot = LynxmotionGrad()
                optimiser = optim.SGD(robot.parameters(), lr=lr, momentum=0.9)

                loss, errors = robot.inverse_kinematics_step(optimiser, target_pose)
                iteration = 0
                while (errors > tolerances).any() and iteration < no_iterations:
                    loss, errors = robot.inverse_kinematics_step(optimiser,
target_pose)
                    iteration += 1

                if iteration < no_iterations:
                    solved = 1

            results[(pos_tol, ori_tol, restarts)] += solved

filename = 'restart_success.csv'
with open(filename, 'w') as f:
    f.write('pos_tol,ori_tol,restarts,')
    f.write('successes')
    f.write('\n')
    for pos_tol, ori_tol, restarts in results:
        f.write('{},{},{},{}'.format(pos_tol, ori_tol, restarts, results[pos_tol,
ori_tol, restarts]))
        f.write('\n')
print('Restart written to {}'.format(filename))
```

/home/jeff/school/msc/fundamentals/ufmf4x_cw1/part_3/test_tolerance.py

```python
import torch
import numpy as np
from torch import optim
from lynxmotion_grad import LynxmotionGrad
import time

pos_tols = np.logspace(1,-3,5,base=2)
ori_tols = np.logspace(1,-3,5,base=2) * np.pi/180
results = {}
no_tests = 1000
no_iterations = 150 # approx 5Hz
lr = 1.29e-04

for i in range(pos_tols.size):
    pos_tol = pos_tols[i]
    ori_tol = ori_tols[i]
    print('Testing position tolerance={} orientation tolerance={}'.format(pos_tol,
ori_tol))
    tolerances = np.array([pos_tol, ori_tol, ori_tol])
    times = []

    for i in range(no_tests):
        robot = LynxmotionGrad()
        with torch.no_grad():
            target_pose = LynxmotionGrad().forward_kinematics()
        optimiser = optim.SGD(robot.parameters(), lr=lr, momentum=0.9)

        time_start = time.time()
        loss, errors = robot.inverse_kinematics_step(optimiser, target_pose)
        iteration = 0
        while (errors > tolerances).any() and iteration < no_iterations:
            loss, errors = robot.inverse_kinematics_step(optimiser, target_pose)
            iteration += 1

        time_elapsed = time.time() - time_start
        if iteration < no_iterations:
            times.append(time_elapsed)
        else:
            times.append(-1)

    results[(pos_tol, ori_tol)] = times

filename = 'tolerance_times.csv'
with open(filename, 'w') as f:
    f.write('pos_tol,ori_tol,')
    f.write(','.join(str(i) for i in range(no_tests)))
    f.write('\n')
    for pos_tol, ori_tol in results:
        f.write('{},{},'.format(pos_tol, ori_tol))
        f.write(','.join(str(time_elapsed) for time_elapsed in results[(pos_tol,
ori_tol)]))
        f.write('\n')
print('Times written to {}'.format(filename))
```