ROBOTIC FUNDAMENTALS UFMF4X-15-M

SERIAL AND PARALLEL ROBOT KINEMATICS

December 14, 2017

Aidan Scannell

University of Bristol

INTRODUCTION

This report outlines the design of serial kinematics for a Lynxmotion arm and parallel kinematics for a planar parallel robot used in surgical procedures. The first section will discuss the design of the serial robot manipulator and the second section will discuss the design of the parallel robot manipulator.

SERIAL ROBOT KINEMATICS - LYNXMOTION ARM

Forward Kinematics

This section outlines the design of the forward kinematics for the Lynxmotion arm. Forward kinematics is the use of kinematic equations to relate the position of the end-effector with the joint parameters.

Kinematic Structure

The first step in the derivation of forward kinematics is the generation of a suitable diagram that shows the joint and link structure of the robotic arm, this can be seen in Figure 1. There are 5 joints labeled $J_1 - J_5$ which are connected by 6 links, some of which have zero length. Each pair of successive joints has a common perpendicular distance that has been marked on the diagram.

Coordinate frames are then assigned to the joints that attach the links, such that a single transformation is associated with each joint. Using the Denavit-Hartenberg convention for assigning frames allows the position and orientation of the end-effector to be given by Equation 1 (**?**),

$$H = {}^0T_N = {}^0T_1 {}^1T_2 {}^2T_3 ... {}^{N-1}T_N \tag{1}$$

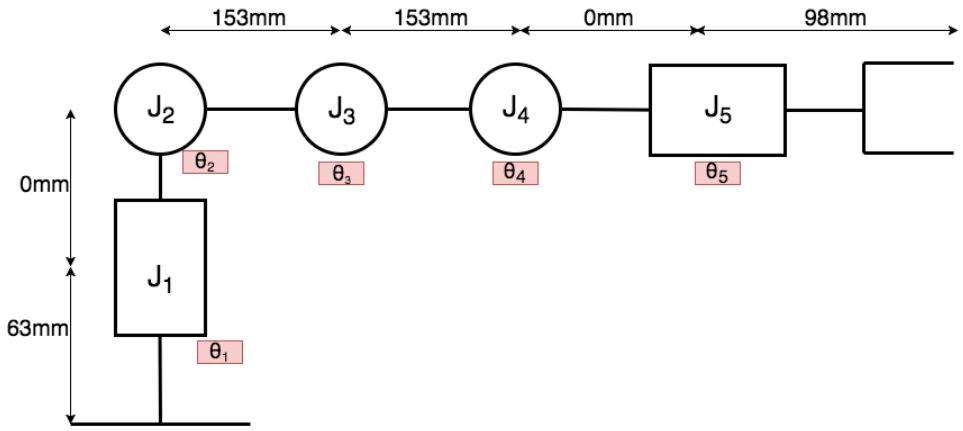


Figure 1: Joint and link structure of Lynxmotion arm

$$ {}^{i-1}T_i = \begin{bmatrix} {}^{i}R_{i-1} & {}^{i}d_{i-1} \\ 0 & 1 \end{bmatrix} \tag{2} $$

where ${}^{i}R_{i-1}$ is the rotation matrix and ${}^{i}d_{i-1}$ is the displacement between links i and $i-1$. There are two conventions that can be used when assigning frames of reference, proximal and distal, the proximal convention was selected for this project. Coordinate frames are assigned as follows for the proximal convention:

- The z-axis is chosen to lie along the joint axis.
- The x-axis is chosen to lie perpendicular to the current joint (i) and the next joint ($i+1$).
- The y-axis is chosen to complete the right hand rule for coordinate frames.

The resulting coordinate frames for the Lynxmotion arm can be seen in Figure 2. In Denavit-Hatenberg each joint is represented as 2 translations (a and d) and 2 angles (α and θ). The DH parameters for the Lynxmotion arm represented in Figure 2 are shown in Table 1.

When using the proximal DH method each joint's transformation matrix is given by Equation 3; this matrix describes the transformation between two coordinate frames and

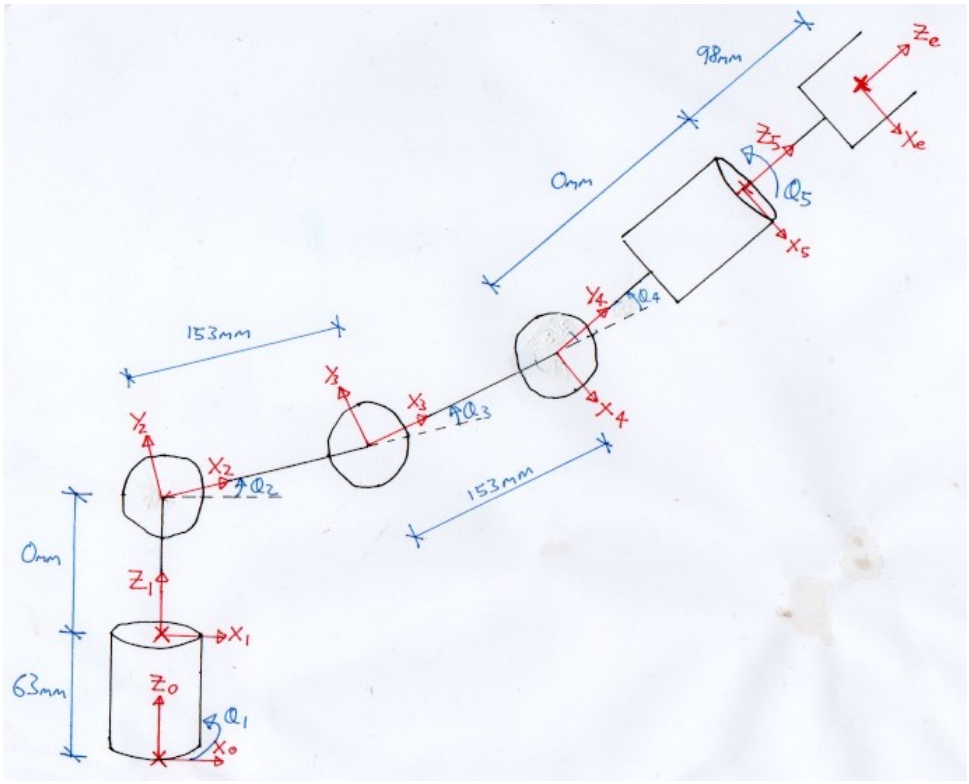


Figure 2: Proximal axis structure used for DH

Table 1: Denavit-Hartenberg parameters

n	a_{n-1}	α_{n-1}	d_n	θ_n
1	0	0	63	θ_1
2	0	$\frac{\pi}{2}$	0	θ_2
3	153	0	0	θ_3
4	153	0	0	$\theta_4 - \frac{\pi}{2}$
5	0	$-\frac{\pi}{2}$	0	θ_5
e	0	0	98	0

consists of the rotation and translation matrices.

$$T = \left(\begin{array}{ccc|c} & R & & T \\ \hline 0 & 0 & 0 & 1 \end{array}\right) = \left(\begin{array}{ccc|c} cos(\theta) & -sin(\theta) & 0 & a_{n-1} \\ sin(\theta)cos(\alpha_{n-1}) & cos(\theta)cos(\alpha_{n-1}) & -sin(\alpha_{n-1}) & -sin(\alpha_{n-1})d \\ sin(\theta)sin(\alpha_{n-1}) & cos(\theta)sin(\alpha_{n-1}) & cos(\alpha_{n-1}) & cos(\alpha_{n-1})d \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{3}$$

Each joint's transformation matrices are determined by subbing in the values from the DH table to obtain the matrices shown in Equations 4-9.

$$
{}^{0}T_{1} = \left(\begin{array}{ccc|c} cos(\theta_1) & -sin(\theta_1) & 0 & 0 \\ sin(\theta_1) & cos(\theta_1) & 0 & 0 \\ 0 & 0 & 1 & l_1 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{4}
$$

$$
{}^{1}T_{2} = \left(\begin{array}{ccc|c} cos(\theta_2) & -sin(\theta_2) & 0 & 0 \\ 0 & 0 & -1 & 0 \\ sin(\theta_2) & cos(\theta_2) & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{5}
$$

$$
{}^{2}T_{3} = \left(\begin{array}{ccc|c} cos(\theta_3) & -sin(\theta_3) & 0 & l_2 \\ sin(\theta_3) & cos(\theta_3) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{6}
$$

$$
{}^{3}T_{4} = \left(\begin{array}{ccc|c} cos(\theta_4) & -sin(\theta_4) & 0 & l_3 \\ sin(\theta_4) & cos(\theta_4) & 0 & 0 \\ 0 & 0 & 1 & l_1 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{7}
$$

$$
{}^{4}T_{5} = \left(\begin{array}{ccc|c} cos(\theta_5) & -sin(\theta_5) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -sin(\theta_5) & -cos(\theta_5) & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{8}
$$

$$ {}^{5}T_{E} = \left(\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & l_5 \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{9} $$

Once these have been calculated the compound transformation matrix for each joint can be calculated using Equation 1 (where N is the joint number). Theses matrices relate the position of each joint to the reference coordinate frame. Figure 3 shows the initial configuration of the manipulator with all joint angles set to zero. The joint angles (°) are marked next to each joint.

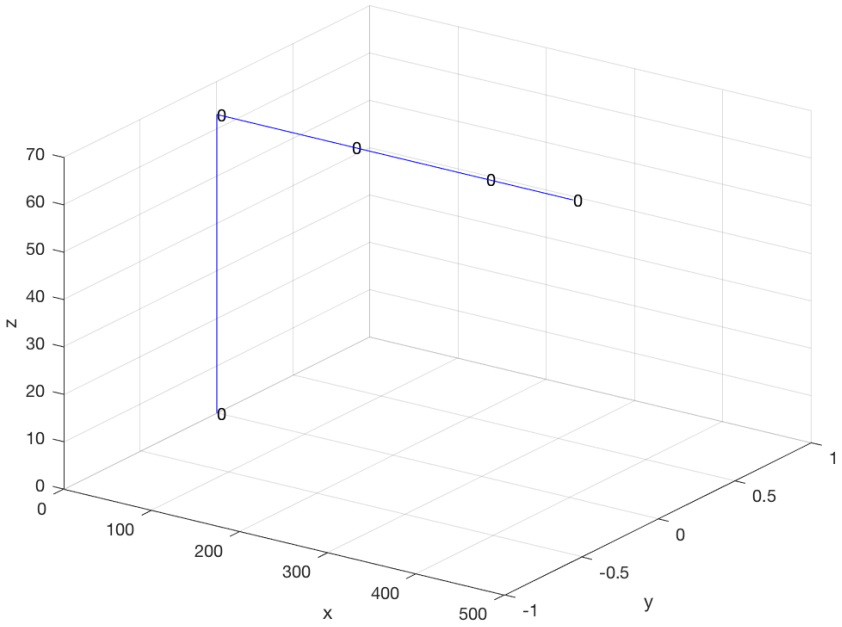
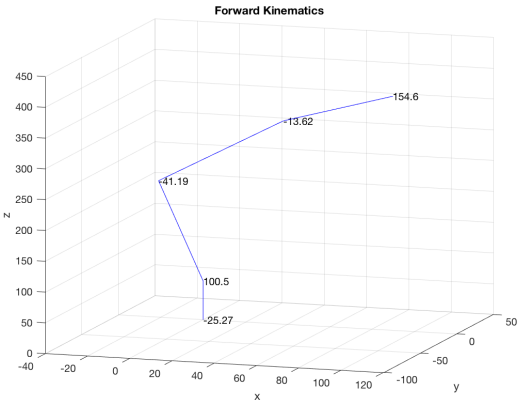
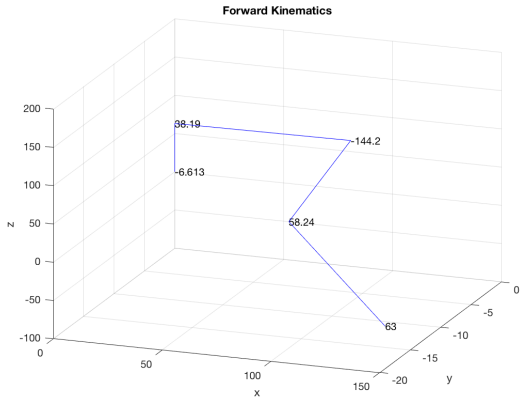


Figure 3: Initial configuration of manipulator with all joint angles set to 0° (3D view)

The manipulator was simulated and tested in MATLAB by randomly selecting joint angles (within joint ranges) and plotting them. The results of two forward kinematics tests are shown in Figure 4.



(a) $J_1 = -25.27°$, $J_2 = 100.5°$, $J_3 = -41.19°$, $J_4 = -13.62°$, $J_5 = 154.6°$. Resulting end-effector position: x = 107.31, y = -50.66, z = 415.13, pitch = 45.69



(b) $J_1 = -6.61°$, $J_2 = 38.19°$, $J_3 = -144.24°$, $J_4 = 58.24°$, $J_5 = 63.00°$. Resulting end-effector position: x= 142.83, y = -16.56, z = -62.05, pitch = -47.80

Figure 4: Forward kinematics results with joint angles

Workspace

The workspace of a manipulator is defined as the set of points reachable by the end-effector (**?**). The performance of a manipulator is related to it's position and orientation and are an important criterion for comparing two manipulators. In practice it is desirable to not only have a large workspace but also a high quality workspace. Many different types of workspaces have been defined but two of the simplest and most useful are dexterous and reachable workspaces:

- **Reachable workspace** is the volume of space that the center of the end-effector can reach for a given orientation.
- **Dexterous workspace** is defined as the volume within, which every point can be reached by the end-effector in any desired orientation.

Analytical methods are not practical as they are limited to certain applications. Numerical methods are relatively simple and more general but only provide an approximate boundary. Calculating the workspace numerically involves first determining each joints range of motion.

Table 2: Joint ranges for Lynxmotion arm (BRL number 10)

	Joint				
	1	2	3	4	5
Range (°)	(-90, 90)	(0, 135)	(-145, 0)	(-90, 90)	(0,360)

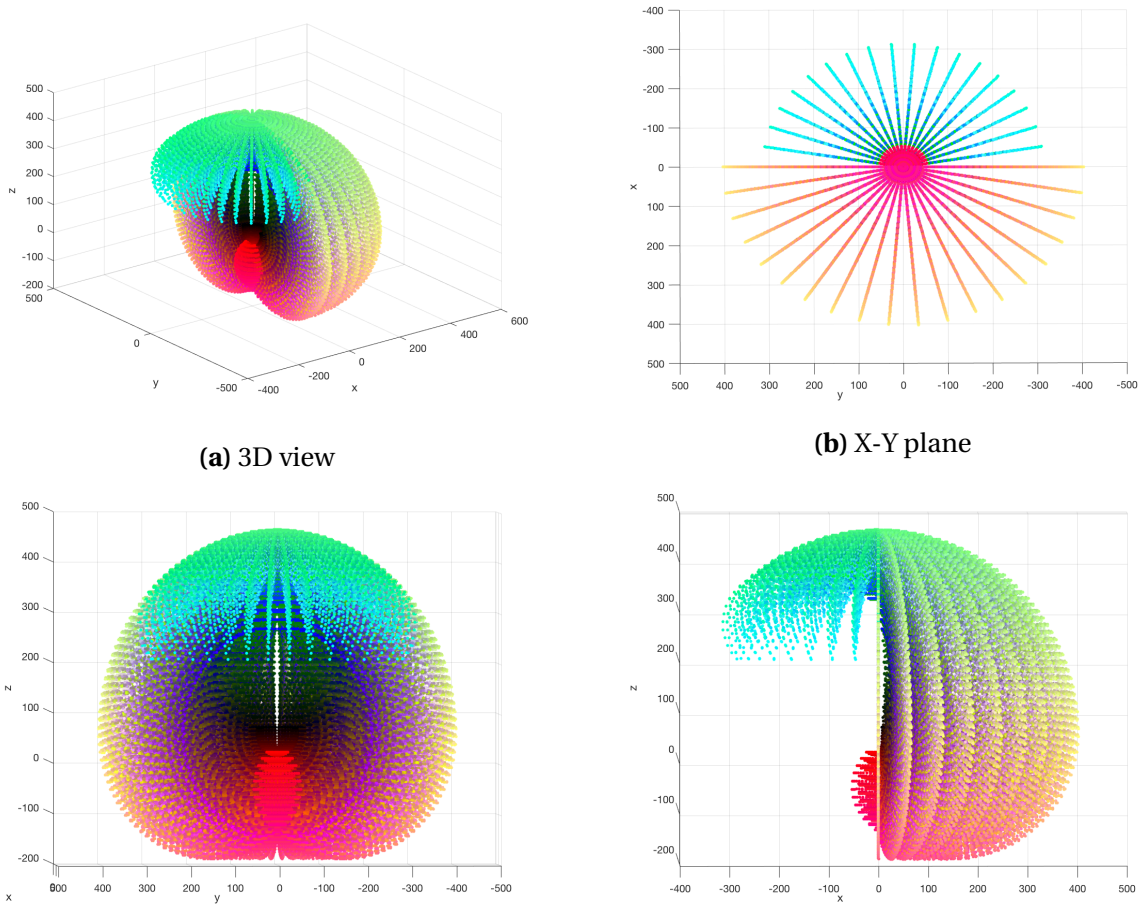
Table 2 shows the range of motion for each joint. The dexterous workspace was determined numerically by iterating through each joints range and using the forward kinematics to calculate the position of the end-effector. Figure 6 shows the dexterous workspace for the simulated Lynxmotion arm in multiple views. θ_5 was not varied for the workspace as it does not effect the position of the end effector.

In many applications the workspace is used to ensure that a given trajectory does not pass through points which are not reachable. It is therefore also important to consider holes and voids in the workspace that were not considered in this project. These may be due to collisions between links, the end-effector passing through the base of the manipulator etc.

Inverse Kinematics

Inverse kinematics involves the mapping of the end-effectors position and orientation to the required joint angles (arm configuration). There are two main ways for deriving the inverse kinematics analytically, the geometric approach and the algebraic approach. A geometric approach was used in this project and is detailed below.

The first step in deriving the inverse kinematics involves decomposing the the spatial geometry into several plane geometry problems. The solution is broken down into two separate problems, one considering the X-Y plane to solve for θ_1 and one considering the X-Z plane to solve for θ_2, θ_3 and θ_4. Figure 6a shows the X-Y plane that is used to solve for θ_1 and Figure 6b shows the X-Z plane that is used to solve for θ_2, θ_3 and θ_4.



(a) 3D view

(b) X-Y plane

(c) Y-Z plane

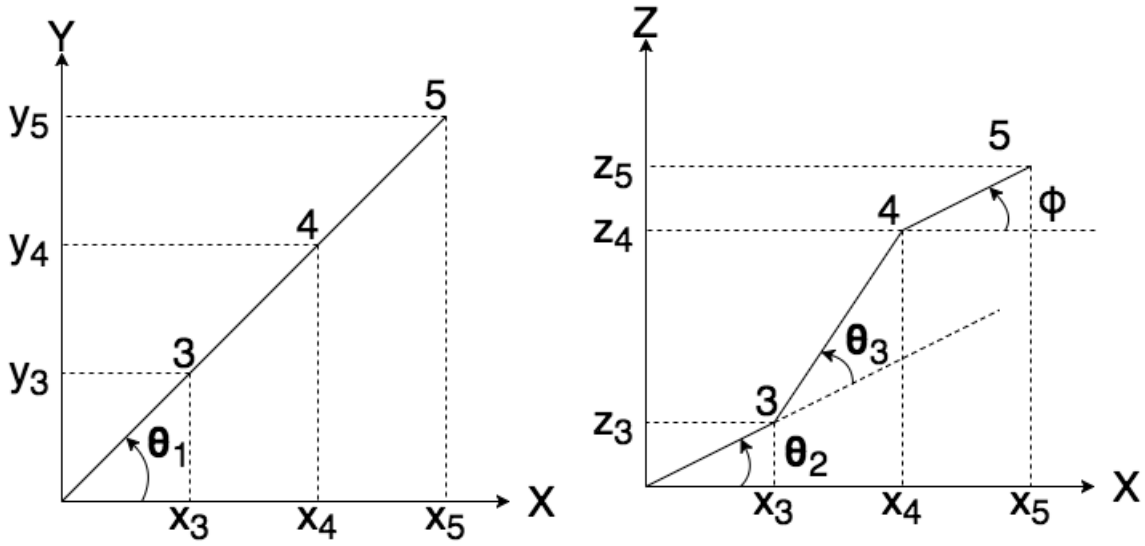
(d) X-Z plane

Figure 5: Figures showing different views of the workspace.

Deriving θ_1

Figure 6a can be used to determine θ_1 using the trigonometric function atan2, as seen in Equation 10. In order to avoid quadrant errors it is best to use the atan2 function in MATLAB which considers the sign of it's inputs to determine which quadrant the solution lies in.

$$\theta_1 = atan2(y_5, x_5) \tag{10}$$



(a) X-Y plane

(b) X-Z plane

Figure 6: Manipulator geometry decomposed into planes

Deriving θ_3

Figure 6b can be used to determine θ_3. The first step involves calculating the position of joint 4 in the X-Z plane using the end-effector position and pitch (ϕ).

$$x_4 = x_5 - l_4 c_\phi \tag{11}$$

$$z_4 = z_5 - l_4 s_\phi \tag{12}$$

Next, the position of joint 4 in terms θ_2, θ_3 and the manipulators link lengths needs to be determined, as shown in Equations 13 and 14,

$$x_4 = l_2 c_2 + l_3 c_{23} \tag{13}$$

$$z_4 = l_2 s_2 + l_3 s_{23} \tag{14}$$

where $c_i = cos(\theta_i)$, $s_i = sin(\theta_i)$, $c_{ij} = cos(\theta_i + \theta_j)$ and $s_{ij} = sin(\theta_i + \theta_j)$. Equations 13 and 14 are then squared and added together to give Equation 15. This equation can be simplified using trigonometric functions to give Equation 16.

$$x_4^2 + z_4^2 = l_2^2(c_2^2 + s_2^2) + l_3^2(c_{23}^2 + s_{23}^2) + 2l_2l_3(c_2c_{23} + s_2s_{23}) \tag{15}$$

$$x_4^2 + z_4^2 = l_2^2 + l_3^2 + 2l_2l_3c_3 \tag{16}$$

Equation 16 can then be rearranged for c_3 as shown in Equation 18. In order to use the atan2 function in MATLAB, s_3 has to be calculated using the basic trigonometric function $s^2 + c^2 = 1$.

$$c_3 = \frac{x_4^2 + z_4^2 - l_2^2 - l_3^2}{2l_2l_3} \tag{17}$$

$$s_3 = \pm\sqrt{1 - \left(\frac{x_4^2 + z_4^2 - l_2^2 - l_3^2}{2l_2l_3}\right)^2} \tag{18}$$

This provided a solution for θ_3 as shown in Equation 19. It should be noted that there are two solutions for θ_3 due to the square root.

$$\theta_3 = atan2(\pm\sqrt{1 - \left(\frac{x_4^2 + z_4^2 - l_2^2 - l_3^2}{2l_2l_3}\right)^2}, \frac{x_4^2 + z_4^2 - l_2^2 - l_3^2}{2l_2l_3}) \tag{19}$$

Deriving θ_2

In order to compute θ_2, each side of Equation 13 needs to be multiplied by c_2 and each side of Equation 14 by s_2 and the resulting equations summed.

$$x_4c_2 = l_2c_2^2 + l_3c_{23}c_2 \tag{20}$$

$$z_4s_2 = l_2s_2^2 + l_3s_{23}s_2 \tag{21}$$

$$x_4c_2 + z_4s_2 = l_2 + l_3(\frac{1}{2}(c_3 - c(2\theta_2 + \theta_3))) + \frac{1}{2}(c(2\theta_2 + \theta_3) + c_3) \tag{22}$$

The simplified equation is given below.

$$x_4 c_2 + z_4 s_2 = l_2 + l_3 c_3 \tag{23}$$

Next, both sides of Equation 13 need to be multiplied by $-s_2$ and each side of Equation 14 by c_2 and the resulting equations summed.

$$-x_4 s_2 = -l_2 c_2 s_2 - l_3 c_{23} s_2 \tag{24}$$

$$z_4 c_2 = l_2 s_2 c_2 + l_3 s_{23} c_2 \tag{25}$$

$$z_4 c_2 - x_4 s_2 = l_3 (s_{23} c_2 - c_{23} s_2) \tag{26}$$

The simplified equation is given below.

$$z_4 c_2 - x_4 s_2 = l_3 s_3 \tag{27}$$

Now Equation 23 needs to be multiplied by x_4 and Equation 27 by z_4 and the resulting equations summed.

$$c_2 x_4^2 + z_4 x_4 s_2 = x_4 (l_2 + l_3 c_3) \tag{28}$$

$$c_2 z_4^2 - z_4 x_4 s_2 = z_4 l_3 c_3 \tag{29}$$

$$c_2 (z_4^2 + x_4^2) = x_4 (l_2 + l_3 c_3) z_4 l_3 s_3 \tag{30}$$

Equation 30 can then be rearranged to obtain c_3 and s_3 as in the previous section.

$$c_2 = \frac{x_4 (l_2 + l_3 c_3) + z_4 l_3 s_3}{x_4^2 + z_4^2} \tag{31}$$

$$s_2 = \pm \sqrt{1 - \left(\frac{x_4 (l_2 + l_3 c_3) + z_4 l_3 s_3}{x_4^2 + z_4^2} \right)^2} \tag{32}$$

Table 3: Inverse kinematic solutions

	Joint Angle (°)	
	θ_2	θ_3
Solution 1	$atan2\left(+\sqrt{1-\left(\frac{x_4(l_2+l_3c_3)+z_4l_3s_3}{x_4^2+z_4^2}\right)^2}, \frac{x_4(l_2+l_3c_3)+z_4l_3s_3}{x_4^2+z_4^2}\right)$	$atan2\left(+\sqrt{1-\left(\frac{x_4^2+z_4^2-l_2^2-l_3^2}{2l_2l_3}\right)^2}, \frac{x_4^2+z_4^2-l_2^2-l_3^2}{2l_2l_3}\right)$
Solution 2	$atan2\left(-\sqrt{1-\left(\frac{x_4(l_2+l_3c_3)+z_4l_3s_3}{x_4^2+z_4^2}\right)^2}, \frac{x_4(l_2+l_3c_3)+z_4l_3s_3}{x_4^2+z_4^2}\right)$	$atan2\left(+\sqrt{1-\left(\frac{x_4^2+z_4^2-l_2^2-l_3^2}{2l_2l_3}\right)^2}, \frac{x_4^2+z_4^2-l_2^2-l_3^2}{2l_2l_3}\right)$
Solution 3	$atan2\left(+\sqrt{1-\left(\frac{x_4(l_2+l_3c_3)+z_4l_3s_3}{x_4^2+z_4^2}\right)^2}, \frac{x_4(l_2+l_3c_3)+z_4l_3s_3}{x_4^2+z_4^2}\right)$	$atan2\left(-\sqrt{1-\left(\frac{x_4^2+z_4^2-l_2^2-l_3^2}{2l_2l_3}\right)^2}, \frac{x_4^2+z_4^2-l_2^2-l_3^2}{2l_2l_3}\right)$
Solution 4	$atan2\left(-\sqrt{1-\left(\frac{x_4(l_2+l_3c_3)+z_4l_3s_3}{x_4^2+z_4^2}\right)^2}, \frac{x_4(l_2+l_3c_3)+z_4l_3s_3}{x_4^2+z_4^2}\right)$	$atan2\left(-\sqrt{1-\left(\frac{x_4^2+z_4^2-l_2^2-l_3^2}{2l_2l_3}\right)^2}, \frac{x_4^2+z_4^2-l_2^2-l_3^2}{2l_2l_3}\right)$

The solution for θ_2 is given in Equation 33, where it should be noted that s_2 depends on θ_3, which can take two values. This means that there are four possible solution for θ_2.

$$\theta_2 = atan2(\pm\sqrt{1-\left(\frac{x_4(l_2+l_3c_3)+z_4l_3s_3}{x_4^2+z_4^2}\right)^2}, \frac{x_4(l_2+l_3c_3)+z_4l_3s_3}{x_4^2+z_4^2}) \tag{33}$$

Deriving θ_4

Now that θ_2 and θ_3 have been calculated, θ_4 can be determined as follows.

$$\theta_4 = \phi - \theta_2 - \theta_3 \tag{34}$$

There will be four possible solutions due to the fact that s_3 and s_2 can take two values. The possible solutions are shown in Table 3.

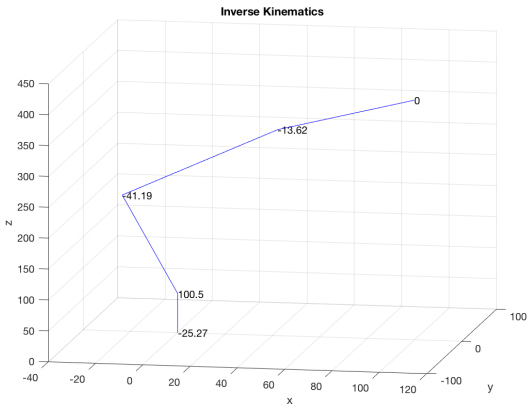
In order to select the correct solutions for a given target position the joint angles are passed through the forward kinematics to ensure that they result in the correct target position. The joint angles are also checked to ensure that they lie within their specific range. Only solutions 1 and 3 give the desired position and orientation of the end-effector and due to the joint constraints only solution 3 is valid for this manipulator.

In order to test the inverse kinematics random joint angles within their specific ranges are fed into the forward kinematics. The resulting end-effector position and pitch is then fed into

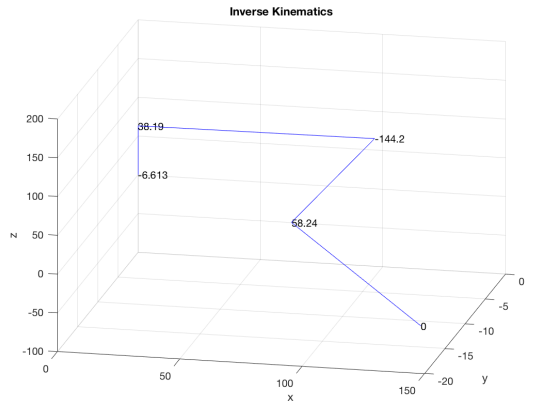
the inverse kinematics where it is then possible to check that the solutions have the correct end-effector position and pitch.

If the joint solutions are not real then this is an indicator that the target position does not lie within the workspace. This provides an easy tool for detecting if the target position lies within the workspace (in terms of singularities). In real-world applications the workspace can be used alongside the inverse kinematics to ensure that all target positions lie within the workspace.

Figure 7 shows the results obtained for the inverse kinematics after setting the target positions and orientations to those obtained from the forward kinematics in Figure 4. θ_5 does not effect the end effector pitch or orientation so was not included in the inverse kinematics. It is clear from Figure 7 that the inverse kinematics is correct.



(a) [(142.83, -16.56, -62.05, -47.80), $(-6.61°, 38.19°, -144.24°, 58.24°)$]



(b) [(107.31, -50.66, 415.13, 45.69), $(-25.27°, 100.5°, -41.19°, -13.62°)$]

Figure 7: Inverse kinematics results. [Target end-effector position: (x, y, z, pitch), Resulting joint configuration: $(\theta_1, \theta_2, \theta_3, \theta_4)$]

Pick and Place Task

This section of the report outlines an example of the manipulator completing a task by moving through 5 positions with specified end-effector Cartesian coordinates and orientation. Table 4 shows the coordinates and orientations of each position in the pick and place task. It also

shows the joint configurations (Cartesian coordinates and joint angle) for each position in the pick and place task. The joint angles were obtained via the inverse kinematics and the joint positions were obtained via the forward kinematics. The transformation matrices of each joint relative to the origin were calculated as follows,

$$ {}^{0}T_{i} = {}^{0}T_{1}\,{}^{1}T_{2}\ldots{}^{i-1}T_{i} = \begin{bmatrix} {}^{0}R_{i} & {}^{0}d_{i} \\ 0 & 1 \end{bmatrix} $$

where ${}^{0}R_{i}$ represents the rotation matrix and ${}^{0}d_{i}$ represents the translation from the origin to joint i (Cartesian coordinates of joint i).

Table 4: Joint configurations for each position in the task.

	End-effector coordinates and pitch (x, y, z, ψ)				
	Position 1 (0, 300, 0, -90)	Position 2 (0,250,300,0)	Position 3 (250,0,300,0)	Position 4 (0,-250,300,0)	Position 5 (0,-300,0,-90)
Joint 1 (x, y, z, θ)	(0.00, -0.00, 63.00, 90.00)	(0.00, -0.00, 63.00, 90.00)	(0.00, -0.00, 63.00, 0.00)	(0.00, -0.00, 63.00, -90.00)	(0.00, -0.00, 63.00, -90.00)
Joint 2 (x, y, z, θ)	(0.00, 0.00, 63.00, 5.89)	(0.00, 0.00, 63.00, 80.38)	(0.00, 0.00, 63.00, 80.38)	(0.00, 0.00, 63.00, 80.38)	(0.00, 0.00, 63.00, 15.89)
Joint 3 (x, y, z, θ)	(0.00, 147.16, 104.89, -18.47)	(0.00, 25.56, 213.85, -46.11)	(25.56, 0.00, 213.85, -46.11)	(0.00, -25.56, 213.85, -46.11)	(0.00, -147.16, 104.89, -18.47)
Joint 4 (x, y, z, θ)	(0.00, 300.00, 98.00, -177.42)	(0.00, 152.00, 300.00, -124.27)	(152.00, 0.00, 300.00, -124.27)	(0.00, -152.00, 300.00, -124.27)	(0.00, -300.00, 98.00, -177.42)
Joint 5 (x, y, z, θ)	(0.00, 300.00, -0.00, 0.00)	(0.00, 250.00, 300.00, 0.00)	(250.00, 0.00, 300.00, 0.00)	(0.00, -250.00, 300.00, 0.00)	(0.00, -300.00, -0.00, 0.00)

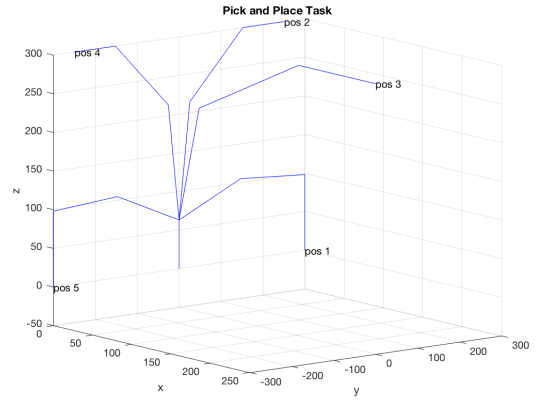
Figure 8 shows the manipulator moving between the 5 positions defining the pick and place task.

Trajectories

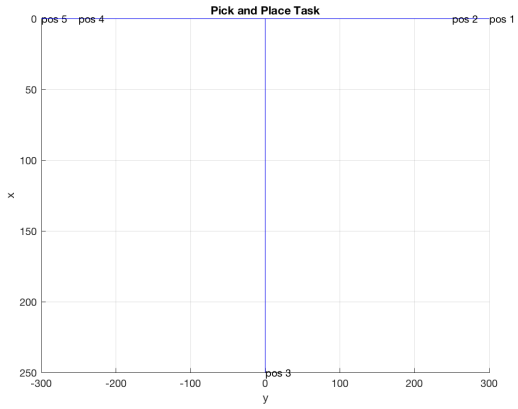
In order to perform the pick and place task a trajectory must be determined that can produce inputs that can feed a motion control system. Trajectory planning can be performed in both cartesian space and joint space. As a task containing 5 points has been defined the trajectories are defined between a finite sequence of points along the path, as apposed to point-to-point control.

There are a number of requirements that must be met for the success of trajectories:

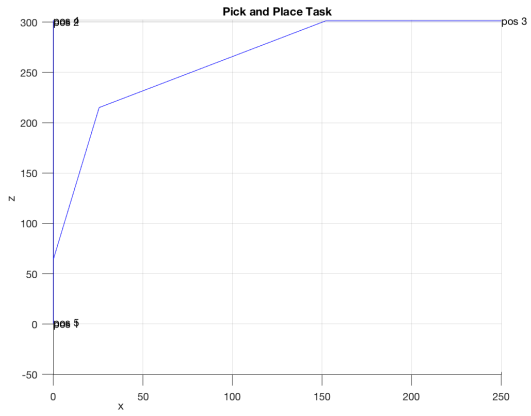
- They should move the end-effector from initial to final posture.
- The motion law cannot violate the saturation limits of the joint drives.



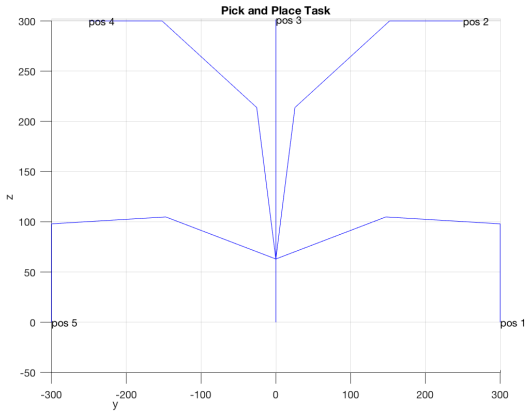
(a) 3D view



(b) X-Y plane



(c) X-Z plane



(d) Y-Z plane

Figure 8: Figures showing different views of the manipulator performing the pick and place task

- They should not excite resonant modes of the mechanical structure.
- They should be smooth to prevent jerking.

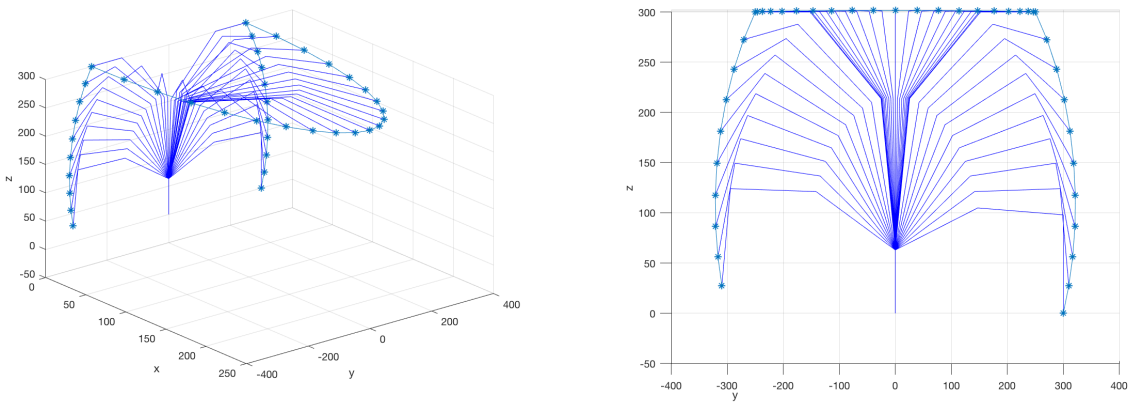
Cartesian space trajectories have motion defined in the catesian space so the motion between points is always known and is controllable. This makes it easy to visualise the trajectory but can lead to issues with singularities: manipulator running into itself and sudden jumps between joints angles. Cartesian trajectories have the advantage that they can guarantee a collision free path, unlike joint space trajectories. However, they require the inverse kinematics to be computed at every point so are computationally expensive. It is also

unknown how to set the time duration.

Joint space trajectories have motion defined by their joint values, which results in unpredictable movement between points. Joint space trajectories have the advantage that the inverse kinematics is only computed once and also that joint/velocity constraints can easily be accounted for.

Free Motion Trajectory

Free motion trajectories are joint space trajectories. The joint angles for each location in the task are calculated using the inverse kinematics. The most basic free motion trajectory involves incrementing the joint angles by a fixed amount for a given number of steps between each task location. Figure 9 shows the manipulator performing the task using this type of free motion. It is clear from the figure that no control is achieved over the end-effector position.

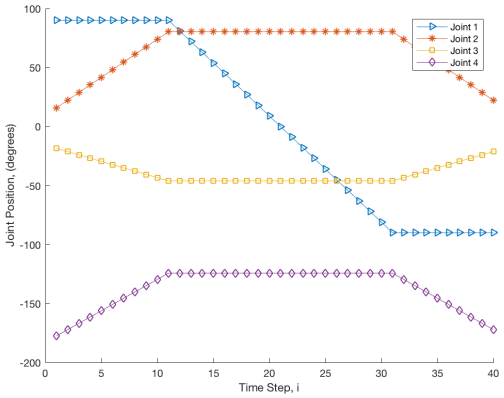


(a) 3D view **(b)** Y-Z plane

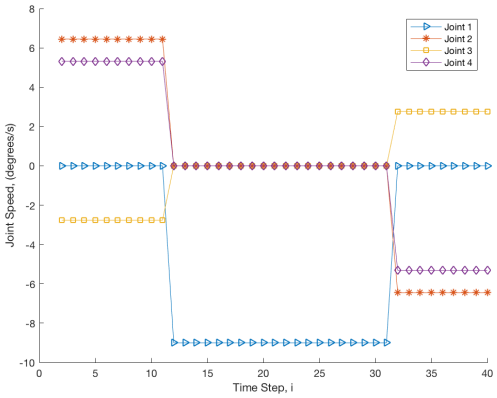
Figure 9: Figures showing different views of the manipulator performing the pick and place task with a free motion trajectory

Figure 10 shows the joint position, velocity and acceleration throughout the task. It is clear that the joint position has continuous motion as expected. However, the velocity jumps between values with extremely high accelerations which may not be achieveable by the joint drives. Ensuring that the joint drives operate under maximum velocity and acceleration is

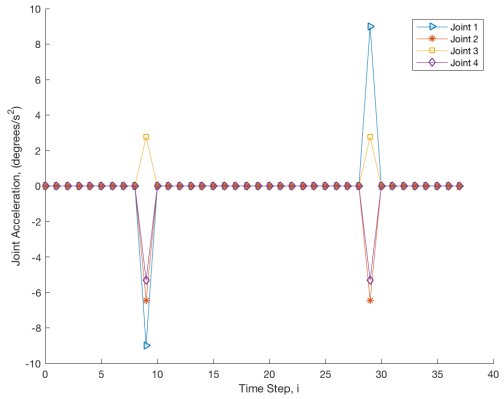
extremely hard for basic free motion. For this reason better trajectories in joint space, such as quintic polynomials are normally used.



(a) Joint Position



(b) Joint Velocity



(c) Joint Acceleration

Figure 10: Figures showing joints position, velocity and acceleration throughout the task with free motion trajectory

Polynomial Trajectories

Cubic polynomial trajectories are joint space trajectories that enable the start and end velocities to be specified as well as the joint positions. This is because there are four constraints to satisfy and cubics provide four polynomial coefficients. The function representing the joint angle in terms of time is set to a cubic polynomial with coefficients a_0, a_1, a_2, a_3, as seen in Equation 35. The derivatives can be taken to obtain functions for velocity and acceleration.

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 \tag{35}$$

$$\dot{q}(t) = a_1 + 2a_2 t + 3a_3 t^2 \tag{36}$$

$$\ddot{q}(t) = 2a_2 + 6a_3 t \tag{37}$$

The initial and final conditions for position and velocity can be used to generate four simulations equations.

$$q_0 = a_0 + a_1 t_0 + a_2 t_0^2 + a_3 t_0^3 \tag{38}$$

$$v_0 = a_1 + 2a_2 t_0 + 3a_3 t_0^2 \tag{39}$$

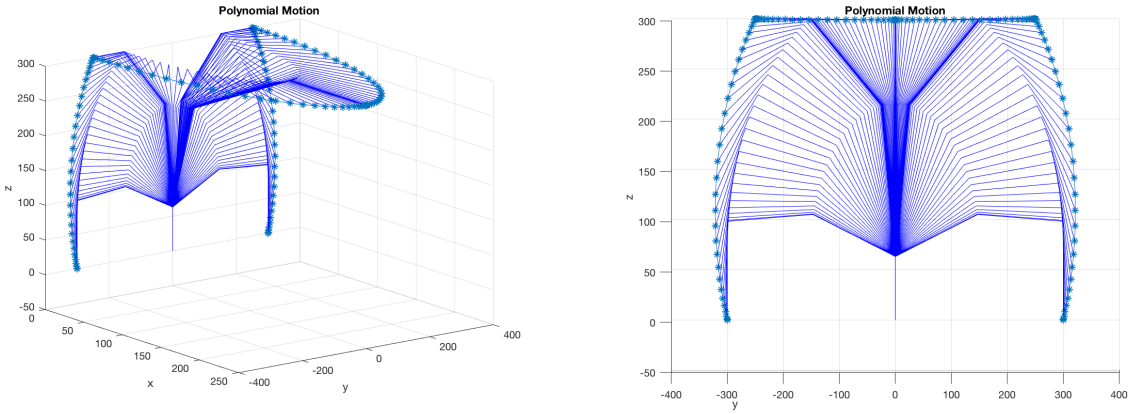
$$q_f = a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3 \tag{40}$$

$$v_f = a_1 + 2a_2 t_f + 3a_3 t_f^2 \tag{41}$$

They can be represented in matrix form as seen in Equation 42.

$$\begin{pmatrix} 1 & t_0 & t_0^2 & t_0^3 \\ 0 & 1 & 2t_0 & 3t_0^3 \\ 1 & t_f & t_f^2 & t_f^3 \\ 0 & 1 & 2t_f & 3t_f^2 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} q_0 \\ v_0 \\ q_f \\ v_f \end{pmatrix} \tag{42}$$

This enables the coefficients to be determined for given initial and final joint positions and velocities. Figure 11 shows the cubic polynomial trajectory implemented between each task location where the initial and final velocities were set to zero.



(a) 3D view **(b)** Y-Z plane

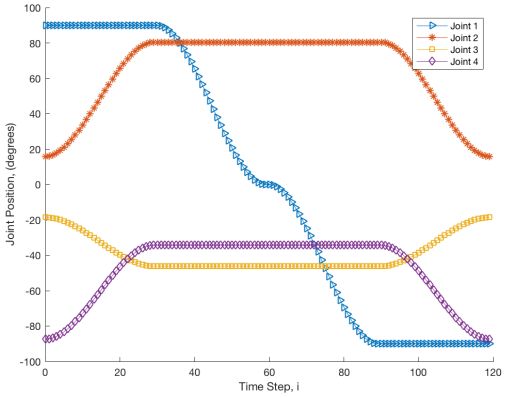
Figure 11: Figures showing different views of the manipulator performing the pick and place task with a cubic polynomial trajectory

Figure 12 shows the resulting joint position, velocity and acceleration throughout the task. It is clear that the cubic polynomial is capable of providing smooth joint velocity in comparison to the straight line and basic free motion trajectories. The derivative of acceleration is known as jerk and it should be noted that there are discontinuities in the acceleration, which will lead to impulsive jerk. This may excite vibration modes of the mechanical structure and should therefore be avoided.

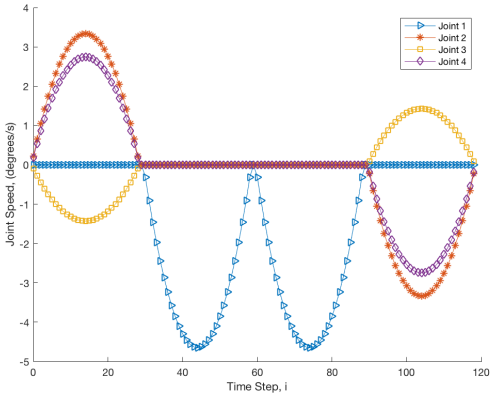
For this reason, it can be more beneficial to model constraints on the initial and final joint accelerations as well as the velocity and positions. There are now 6 unknowns so 6 polynomial coefficients are required. Therefore a quintic polynomial is required, as shown in Equation 43

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5 \tag{43}$$

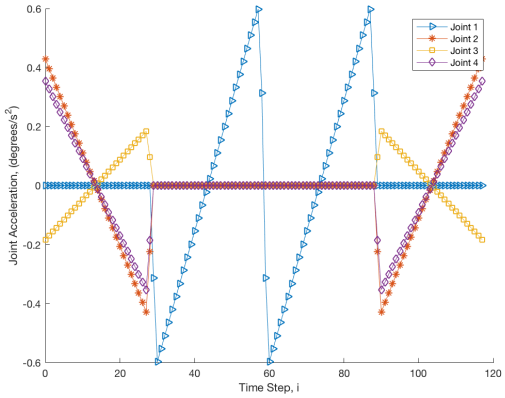
$$\dot{q}(t) = a_1 + 2a_2 t + 3a_3 t^2 + 4a_4 t^3 + 5a_5 t^4 \tag{44}$$



(a) Joint Position



(b) Joint Velocity



(c) Joint Acceleration

Figure 12: Joints position, velocity and acceleration throughout the task with a cubic polynomial trajectory

$$\ddot{q}(t) = 2a_2 + 6a_3 t + 12a_4 t^2 + 20a_5 t^3 \tag{45}$$

Differentiating Equation 43 gives the velocity and acceleration of the joint with respect to time. Six simultaneous equations can be generated based on the initial and final conditions.

$$q_0 = a_0 + a_1 t_0 + a_2 t_0^2 + a_3 t_0^3 + a_4 t_0^4 + a_5 t_0^5 \tag{46}$$

$$v_0 = a_1 + 2a_2 t_0 + 3a_3 t_0^2 + 4a_4 t_0^3 + 5a_5 t_0^4 \tag{47}$$

$$a_0 = 2a_2 + 6a_3 t_0 + 12a_4 t_0^2 + 20a_5 t_0^3 \tag{48}$$

$$q_f = a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3 + a_4 t_f^4 + a_5 t_f^5 \tag{49}$$

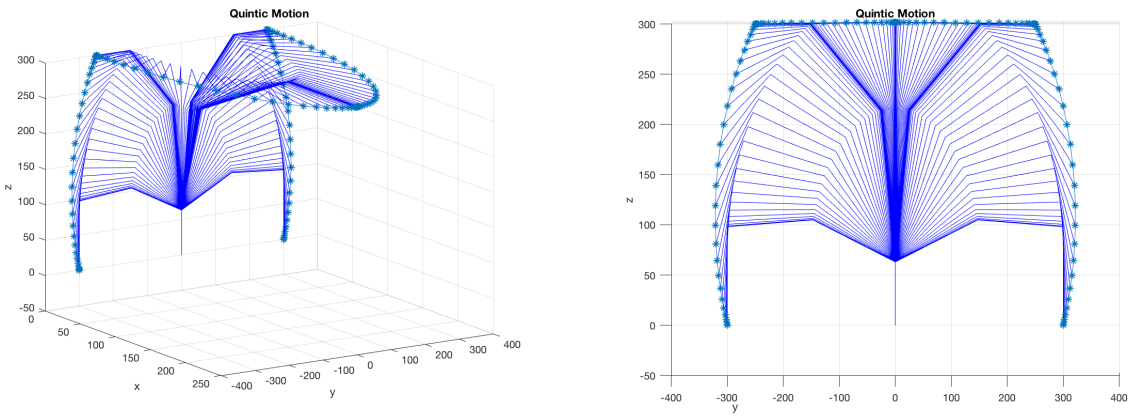
$$v_f = a_1 + 2a_2 t_f + 3a_3 t_f^2 + 4a_4 t_f^3 + 5a_5 t_f^4 \tag{50}$$

$$a_f = 2a_2 + 6a_3 t_f + 12a_4 t_f^2 + 20a_5 t_f^3 \tag{51}$$

Once again, these can be represented in matrix form as shown in Equation 52.

$$\begin{pmatrix} 1 & t_0 & t_0^2 & t_0^3 & t_0^4 & t_0^5 \\ 0 & 1 & 2t_0 & 3t_0^3 & 4t_0^3 & 5t_0^4 \\ 0 & 0 & 2 & 6t_0 & 12t_0^2 & 20t_0^3 \\ 1 & t_f & t_f^2 & t_f^3 & t_f^4 & t_f^5 \\ 0 & 1 & 2t_f & 3t_f^2 & 4t_f^3 & 5t_f^4 \\ 0 & 0 & 0 & 6t_f & 12t_f^2 & 20t_f^3 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{pmatrix} = \begin{pmatrix} q_0 \\ v_0 \\ a_0 \\ q_f \\ v_f \\ a_f \end{pmatrix} \tag{52}$$

Specifying initial and final conditions of zero velocity and acceleration enables the six coefficients to be determined. Figure 13 shows the resulting motion obtained from the quintic polynomial trajectory.

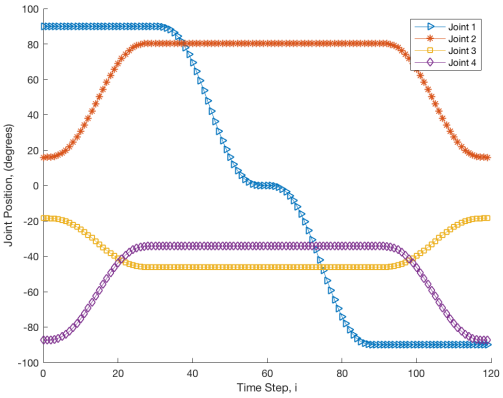


(a) 3D view **(b)** Y-Z plane

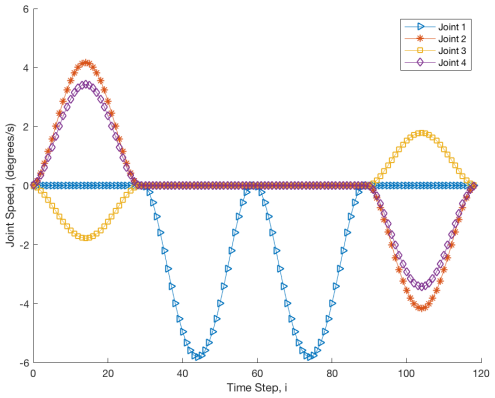
Figure 13: Figures showing different views of the manipulator performing the pick and place task with a quintic polynomial trajectory

Figure 14 shows the joint positions, velocities and accelerations throughout the task.

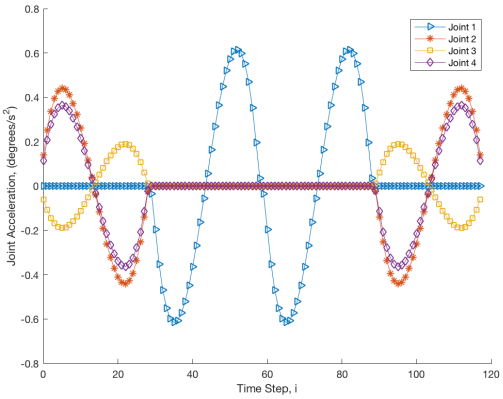
It is clear from the Figure 14c that the acceleration is much smoother as it contains no discontinuities. This is beneficial as it reduces the amount of jerk.



(a) Joint Position



(b) Joint Velocity



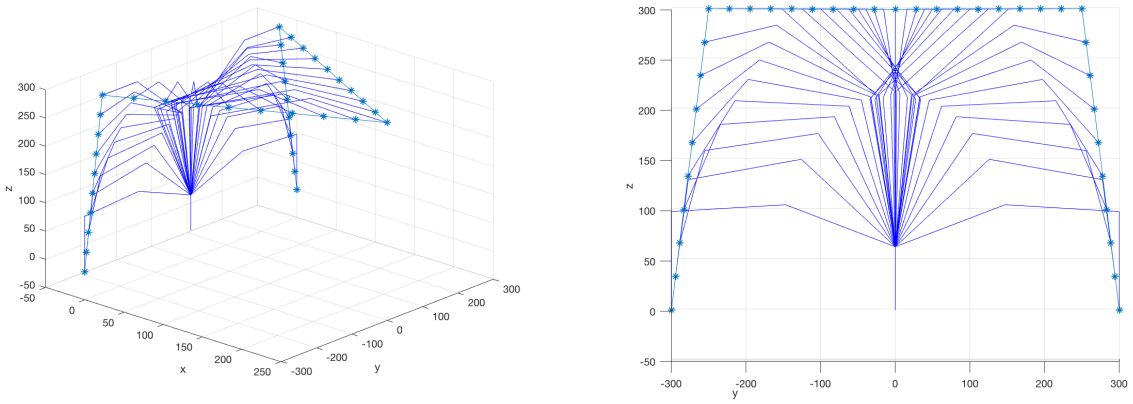
(c) Joint Acceleration

Figure 14: Joints position, velocity and acceleration throughout the task with a quintic polynomial trajectory

It should be noted that a disadvantage of using quintic polynomial trajectories is that they are susceptible to over-fitting, resulting in oscillatory behaviour. With higher order polynomials the numerical accuracy also decreases. Polynomial trajectories also require re-computation of coefficients if a single point is changed.

Straight Line Trajectory

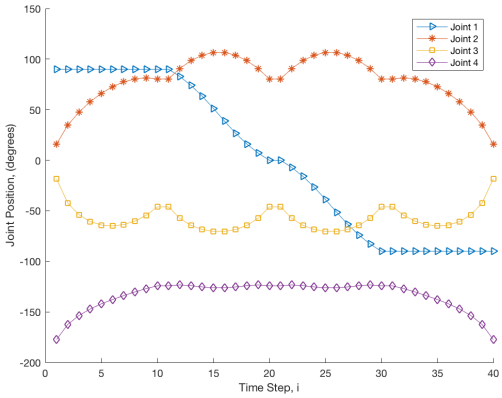
Straight line trajectories are cartesian space trajectories and are determined by sampling from a straight line between consecutive points to obtain a sequence of cartesian coordinates. Each set of coordinates can be used to calculate the required joint angles by using the manipulators inverse kinematics. Figure 15 shows the results of implementing a basic straight line trajectory.



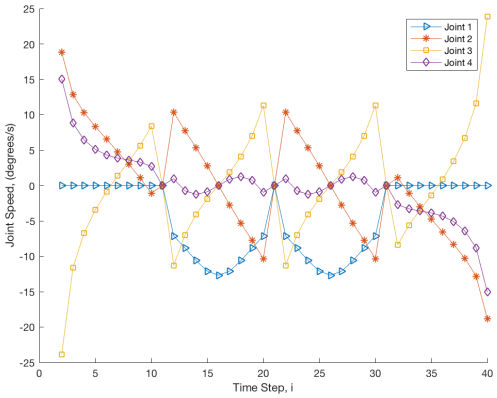
(a) 3D view **(b)** Y-Z plane

Figure 15: Figures showing different views of the manipulator performing the pick and place task with a straight line trajectory.

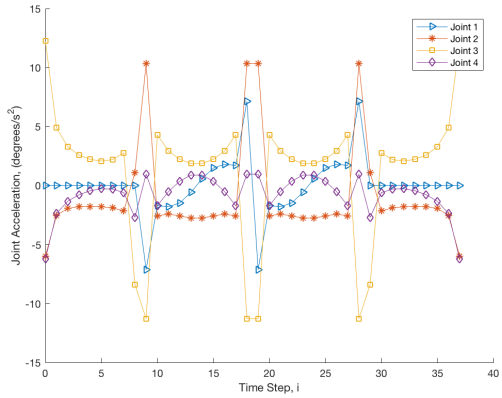
Figure 18 shows the joint positions, velocities and accelerations throughout the task. It is clear that there are some extremely high changes in velocity (accelerations), which may not be feasible by the joint drives.



(a) Joint Position



(b) Joint Velocity

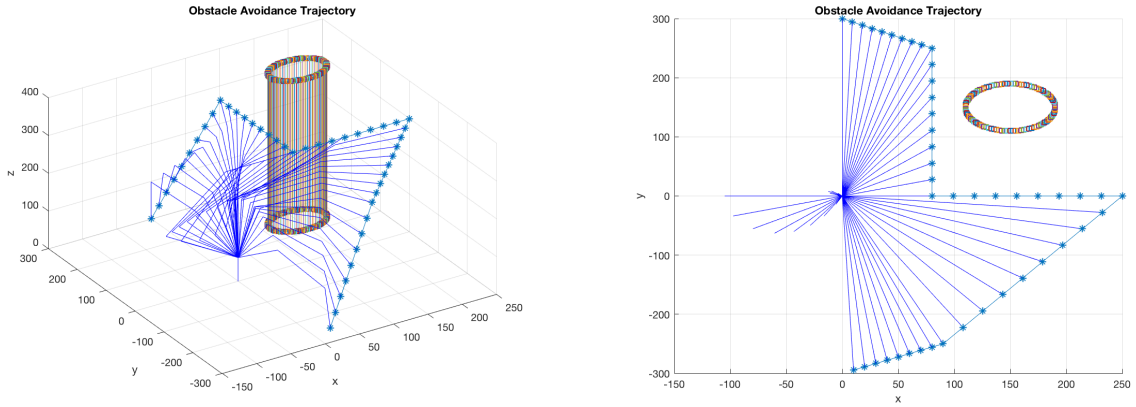


(c) Joint Acceleration

Figure 16: Joints position, velocity and acceleration throughout the task with straight line trajectory

Obstacle Avoidance

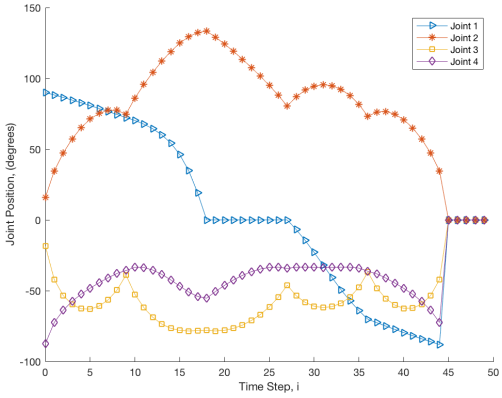
The straight line trajectory was implemented for the collision avoidance as the end-effector position is controllable. Figure 17 shows the cylindrical obstacle and the straight line trajectory avoiding it. However, this trajectory results in velocity and acceleration discontinuities and needs to be improved. Linear trajectories with parabolic blends applied to the cartesian points would be capable of providing straight line motion with more desireable velocity and acceleration profiles. This trajectory calculates the cartesian coordinates and blends the rigid connections at each task location. The manipulators inverse kinematics are used to determine the joint angles for each location. This provides continuous motion but requires correction to ensure that the end-effector passes through each target location.



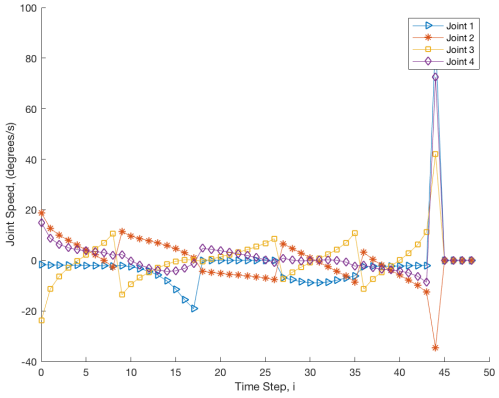
(a) 3D view **(b)** X-Y plane

Figure 17: Figures showing different views of the manipulator performing obstacle avoidance during the pick and place task with a straight line trajectory.

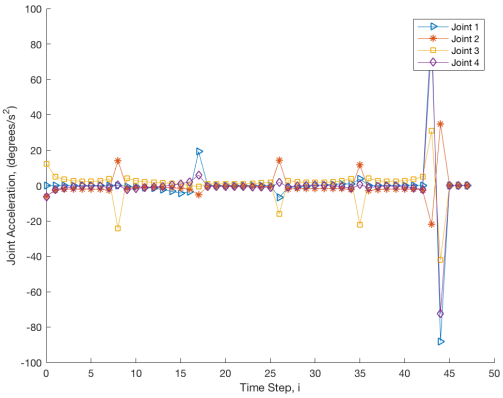
Robust collision avoidance could be implemented using a potential field model, where the trajectory will pass around the obstacle by calculating the potential at a current point and deciding on the next position based on this potential. Polynomial trajectories in joint space offer low jerk (derivative of acceleration) but no control over there end-effector position. Linear trajectories with parabolic blends in joint space have higher jerk values but as most of the movement is performed at maximum velocity they tend to have much shorter movement



(a) Joint Position



(b) Joint Velocity



(c) Joint Acceleration

Figure 18: Joints position, velocity and acceleration throughout the task performing obstacle avoidance with straight line trajectory

times.

Figure 19 shows the end-effector positions for the four trajectories performing the pick and place task. It is interesting to see that all of the joint space trajectories followed the same path, even though the cubic and quintic polynomials contained more constraints. It can also be seen that the polynomial trajectories contained more points close to specified task locations, this is due to the manipulator slowing down in order to meet the velocity and acceleration constraints. The figure also shows that the striaght line trajectory is capable of offering control of the end-effector position, unlinke the joint space trajectories.

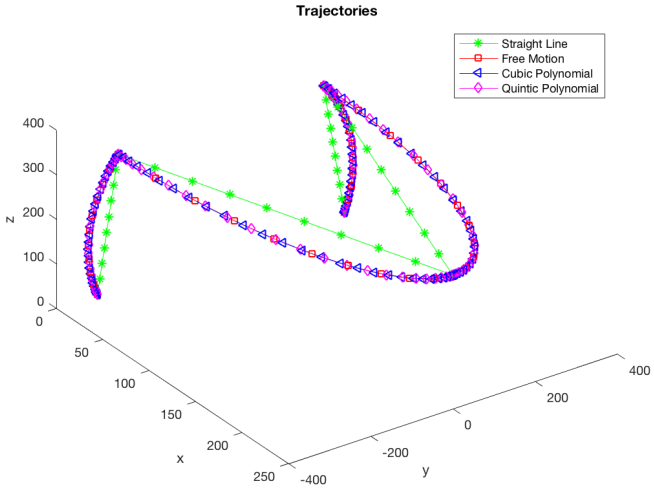


Figure 19: Comparison of trajectories

PLANAR PARALLEL MANIPULATOR

Parallel manipulators have more than one kinematic chain. The forward kinematics of parallel manipulators are computationally complicated but the inverse kinematics can be determined.

Inverse Kinematics

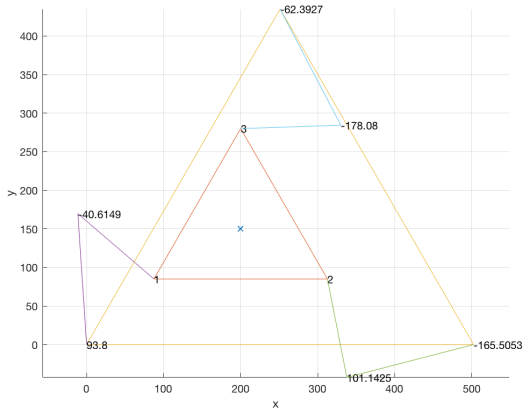
The derivation of inverse kinematics is based on vector chain analysis and from this transformation matrices can be obtained. Figure 20 shows the planar parallel robot in different orientations at three positions with the joint angles marked on each plot. This demonstrates that the inverse kinematics works for the parallel robot. The selected positions show that the angle α can be changed successfully while still achieving the target location. Figures 20d and 20e show the arm plotted at the same location with $\alpha = 0°$ and $\alpha = -40°$. The arm could not be plotted with an angle of $\alpha = 40°$ at this location as it lies outside of the workspace.

The manipulator can only rotate through approximately $80°$ when at the center location and as it does so the workspace reduces in size. It should be noted that the amount of rotation is limited, in case any operations require the end-effector needle to be rotated.

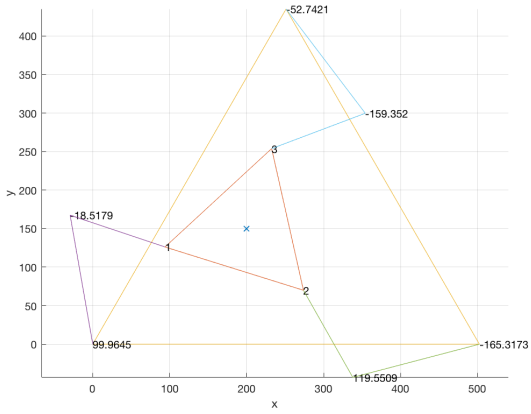
Workspace

Figure 21 shows the workspace of the planar parallel robot for different orientations (α). The workspace was determined by iterating over a range of positions and excluding all locations that resulted in imaginary numbers in the calculation of theta (arising from negative numbers in the square root). These locations correspond to singularities. When determining the workspace it is also important to consider configurations where the arm links may hit each other.

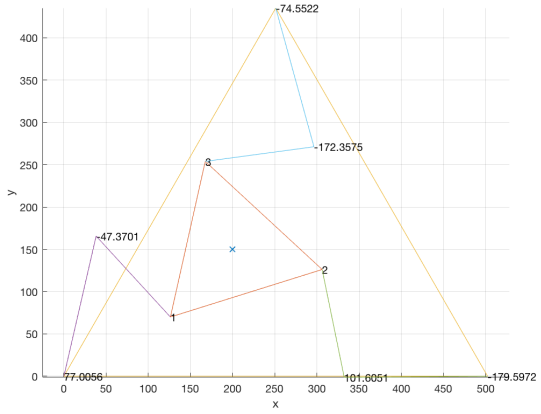
The figures on the right of Figure 21 show the end-effector positions for each arm when



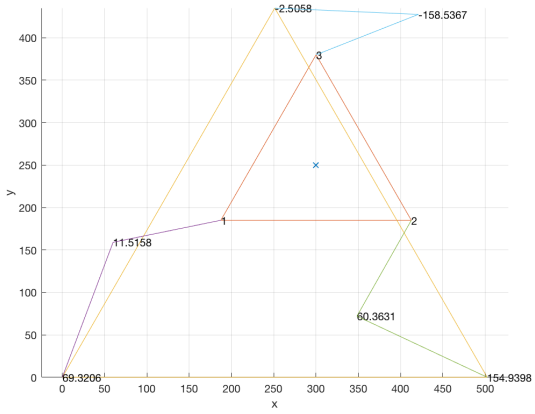
(a) $X_c = 200$, $Y_c = 150$, $a = 0°$



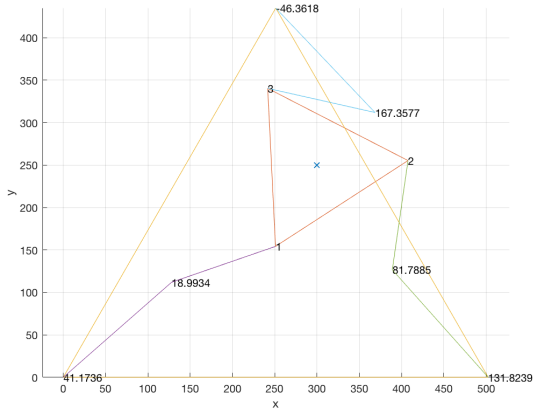
(b) $X_c = 200$, $Y_c = 150$, $a = 30°$



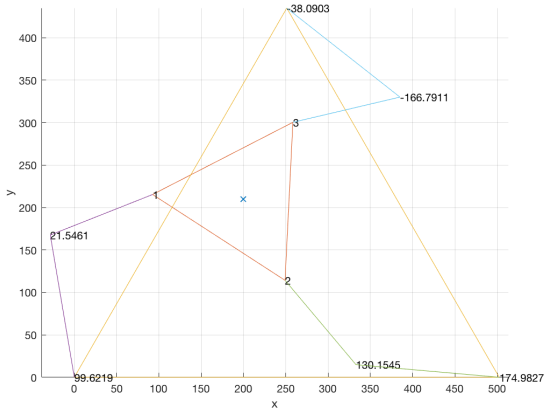
(c) $X_c = 200$, $Y_c = 150$, $a = -30°$



(d) $X_c = 300$, $Y_c = 250$, $a = 0°$



(e) $X_c = 300$, $Y_c = 250$, $a = -40°$

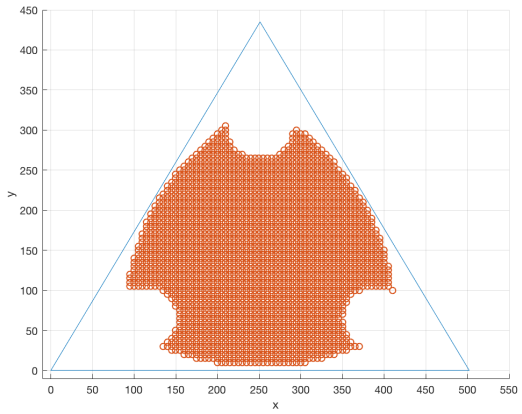


(f) $X_c = 200$, $Y_c = 210$, $a = 40°$

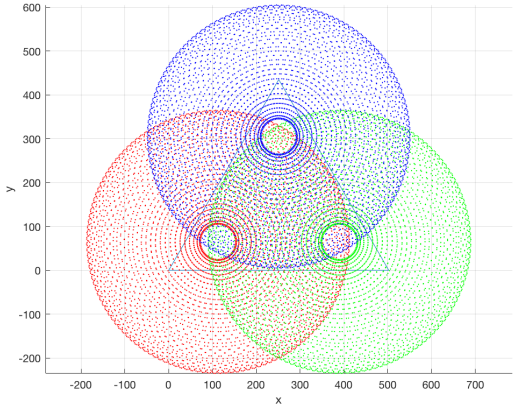
Figure 20: Inverse kinematics for the planar parallel robot in two positions.

separated and moved by themselves. The overlap of the three circles represents the workspace area.

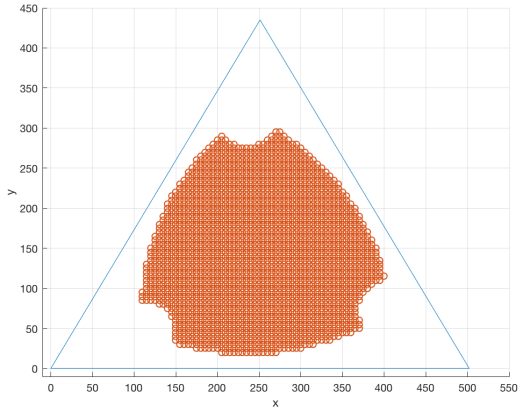
The workspace for low α values have three corner sections which cannot be reached. These should be noted as they will result in non-smooth motion if the needle is required to move across them.



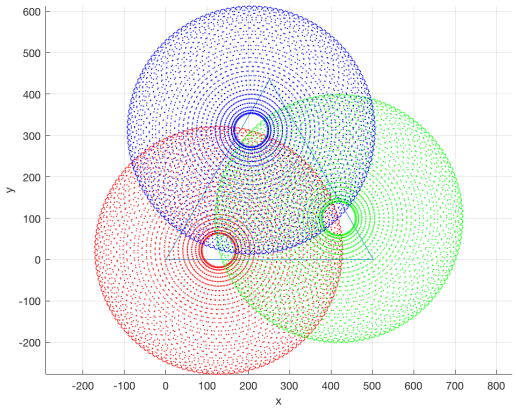
(a) $\alpha = 0°$



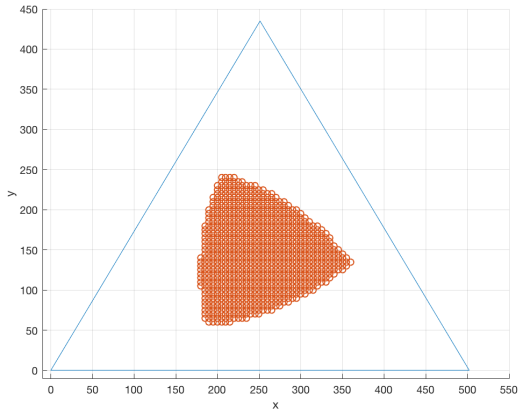
(b) $\alpha = 0°$



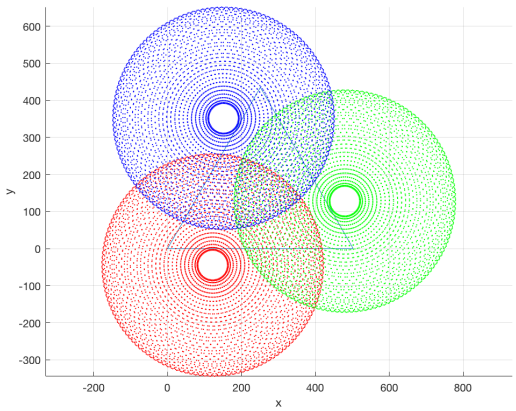
(c) $\alpha = 20°$



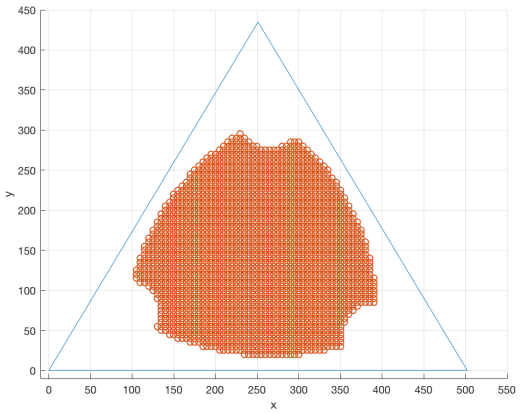
(d) $\alpha = 20°$



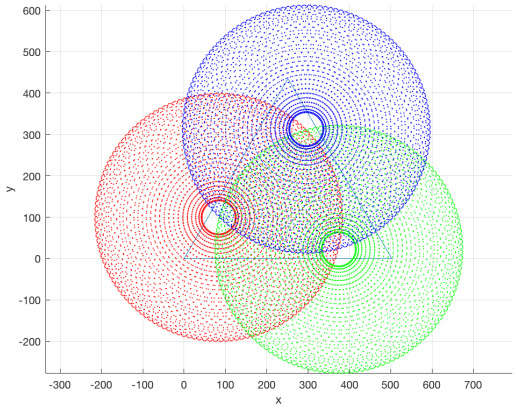
(e) $\alpha = 50°$



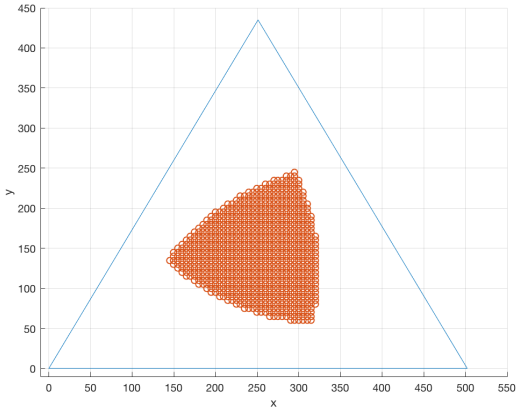
(f) $\alpha = 50°$



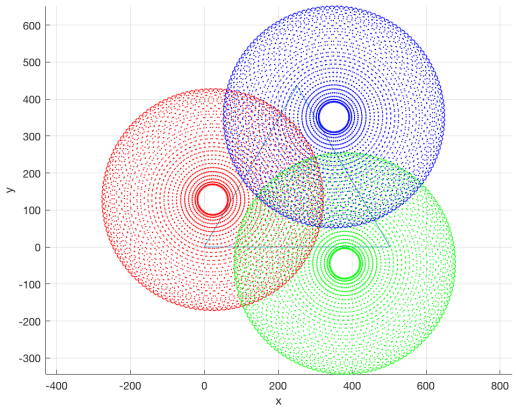
(g) $\alpha = -20°$



(h) $\alpha = -20°$



(i) $\alpha = -50°$



(j) $\alpha = -50°$

Figure 21: Workspace for planar parallel robot for different orientations (α).

Appendices

SERIAL KINEMATICS

serialKinematics.m

```
1  %  Serial Kinematics for the lnyxmotion arm.
2  %  Robotic Fundamentals UFMF4X-15-M
3  %  serialKinamtics.m
4  %
5  %  Instructions for running this code:
6  %               1. set flags in input section ( 1 = run, 0 = don't run)
7  %               2. set variables in input section
8  %               3. if forward kinematics is run then the resulting ...
       end-effector position
9  %               will be fed as an input to the inverse kinematics ...
       and override the values
10 %               in the vector xyzpIK.
11 %
12 %  Created by AidanScannell.
13 %  Copyright l' 2017 AidanScannell. All rights reserved.
14 clear all
15 clc
16 close all
17
18 %% Input
19 % set flags
20 FK_flag = 1;
21 workspace_flag = 1;
22 IK_flag = 1;
23 task_flag = 1;
24 free_motion_flag = 1;
25 straight_line_motion_flag = 1;
```

```
26 poly_motion_flag = 1;
27 quintic_poly_motion_flag = 1;
28
29 lpbb_motion_flag = 0;
30 obstacle_avoidance_flag = 1;
31
32 % set end-effector position/orientation for inverse kinematics
33 xyzpIK = [100,0,200,-90]; % [x, y, z, pitch]
34
35 % set number of steps within each joints range
36 N_steps = 10;
37
38 % set number of steps between task points for polynomial trajectories
39 time_steps = 30;
40
41 % setup variables for task
42 xyzp = ...
       [0,300,0,-90;0,250,300,0;250,0,300,0;0,-250,300,0;0,-300,0,-90]; ...
       % positions/orientations
43 N = 10; % number of steps between each position for trajectories
44
45 %% Setup
46 armLink; % import arm link class
47
48 global l1 l2 l3 l4
49 global range1 range2 range3 range4 range5
50
51 % set up lynxmotion arm parameters
52 l1 = 63; l2 = 153; l3 = 153; l4 = 98; % set arm lengths
53 range1 = [-90,90]; range2 = [0,135]; range3 = [-145,0]; range4 = ...
       [-90,90]; range5 = [0,360]; % set joint ranges
54
```

```
55 % initialise arrays for trajectory end-effector positions
56 posEEfree = [];
57 posEElin = [];
58 posEEpoly = [];
59 posEEquint = [];
60
61 %% Forward Kinematics
62 if FK_flag == 1
63     % Initialise arm link agnles (select random joint angles within ...
          joint ranges)
64     q1 =  range1(1) + (range1(2)-range1(1)).*rand(1);
65     q2 =  range2(1) + (range2(2)-range2(1)).*rand(1);
66     q3 =  range3(1) + (range3(2)-range3(1)).*rand(1);
67     q4 =  range4(1) + (range4(2)-range4(1)).*rand(1);
68     q5 =  range5(1) + (range5(2)-range5(1)).*rand(1);
69 %     q1 = 0; q2 = 0; q3 = 0; q4 = 0; q5 = 0;
70 %     q1 = -25.27; q2 = 100.5; q3 = -41.19; q4 = -13.62; q5 = 154.6;
71
72     % function calculates compound transformation matrices
73     [link1 link2 link3 link4 link5] = forwardKinematics(q1,q2,q3,q4,q5);
74
75     % print FK input and output
76     fprintf('FK input angles: theta 1 = %.2f, theta 2 = %.2f, theta ...
          3 = %.2f, theta 4 = %.2f, theta 5 = %.2f\n',q1,q2,q3,q4,q5)
77     FK_p1 = getP(link1); FK_p2 = getP(link2); FK_p3 = getP(link3); ...
          FK_p4 = getP(link4); FK_p5 = getP(link5);
78     L_pitch = sqrt( (FK_p5(2) - FK_p4(2))^2 + (FK_p5(1) - ...
          FK_p4(1))^2 ); % calculate length of link 4-5 in XY plane
79     FK_pitch = atan2d( (FK_p5(3) - FK_p4(3)), L_pitch ); % calculate ...
          pitch
80     fprintf('FK end-effector position: x = %.2f, y = %.2f, z = %.2f, ...
          pitch = %.2f\n',FK_p5(1),FK_p5(2),FK_p5(3), FK_pitch)
```

```
81
82      % Plot the robotic arm in 3D space
83      plotFK('Forward Kinematics',link1,link2,link3,link4,link5,1,1)
84
85  end
86
87  %% Workspace
88  if workspace_flag == 1
89      workspace(range1,range2,range3,range4,N_steps);
90  end
91
92  %% Inverse Kinematics
93  if IK_flag == 1
94
95      % set input of inverse kinematics to result of forward kinematcs ...
            otherwise to default
96      if exist('FK_p5','var')
97          x = FK_p5(1); y = FK_p5(2); z = FK_p5(3); pitch = FK_pitch;
98      else
99          x = xyzpIK(1); y = xyzpIK(2); z = xyzpIK(3); pitch = xyzpIK(4);
100     end
101
102     % print target
103     fprintf('\nTarget IK end-effector position: x = %.2f, y = %.2f, ...
            z = %.2f, pitch = %.2f\n\n',x,y,z,pitch)
104
105     % perform inverse kinematics
106     result = inverseKinematics(x,y,z,pitch);
107
108     % plot results
109     for ii = 1:size(result,1)
```

```
110         plotFK('Inverse ...
                Kinematics',result(ii,1),result(ii,2),result(ii,3),result(ii,4),result(ii,
111     end
112
113 end
114
115 %% Task
116 if task_flag == 1
117     % perform task
118     hold off; figure;
119     pos = zeros(4,N*(size(xyzp,1)-1));
120     kk = 1;
121     [pos, kk] = task(xyzp,'Pick and Place Task',pos,kk);
122
123 end
124
125 %% Free Motion Trajectory
126 if free_motion_flag == 1
127     hold off; figure;
128 %     xyzp = ...
      [0,300,0,-90;0,250,300,0;250,0,300,0;0,-250,300,0;0,-300,0,-90]; ...
      % positions/orientations
129     posEEfree = freeMotion(xyzp,N);
130 end
131
132 %% Straight Line Trajectory
133 if straight_line_motion_flag == 1
134     posEElin = straightLine(xyzp,N);
135 end
136
137 %% Cubic Polynomial Trajectory
138 if poly_motion_flag == 1
```

```
139     posEEpoly = polynomialTraj(xyzp,time_steps,N);
140 end
141 
142 %% Quintic Polynomial Trajectory
143 if quintic_poly_motion_flag == 1
144     posEEquint = quinticPolyTraj(xyzp,time_steps,N);
145 end
146 
147 %% Linear with Parabolic Blend Trajectory
148 if lpbb_motion_flag == 1
149 %     time_steps = 10; % set number of time steps
150 %     xyzp = ...
      [0,300,0,-90;80,250,300,0;250,0,300,0;90,-250,300,0;0,-300,0,-90]; ...
      % positions/orientations
151     parBlendTraj(xyzp,N);
152 end
153 
154 %% Obstacle Avoidance
155 if obstacle_avoidance_flag == 1
156     xyzp = ...
          [0,300,0,-90;80,250,300,0;80,0,300,0;250,0,300,0;90,-250,300,0;0,-300,0,-90];
          % positions/orientations
157     posEEobs = obsAvoid(xyzp,N,time_steps);
158 end
159 
160 %% plot trajectories
161 if  ¬isempty(posEElin) && ¬isempty(posEEfree) && ¬isempty(posEEpoly) ...
      && ¬isempty(posEEquint)
162     hold off
163     figure
164     hold on
165     title('Trajectories')
```

```
166     plot3(posEElin(:,1),posEElin(:,2),posEElin(:,3),'*g-')
167     plot3(posEEfree(:,1),posEEfree(:,2),posEEfree(:,3),'sr-')
168     plot3(posEEpoly(:,1),posEEpoly(:,2),posEEpoly(:,3),'<b-')
169     plot3(posEEquint(:,1),posEEquint(:,2),posEEquint(:,3),'dm-')
170     xlabel('x');ylabel('y');zlabel('z');
171     legend('Straight Line','Free Motion','Cubic Polynomial','Quintic ...
            Polynomial')
172 end
```

armLink.m

```
1  %   Arm Link class for lynxmotion robotic arm.
2  %   Robotic Fundamentals UFMF4X-15-M
3  %   armLink.m
4  %
5  %   by Aidan Scannell
6  classdef armLink < handle
7      properties
8          T % transformation matrix wrt previous link
9          To % transformation matrix wrt to origin
10         a_n_1 % link length, the distance from z(i?1) to z(i) ...
               measured along x(i?1)
11         alpha_n_1 % twist angle, the angle between z(i?1) to z(i) ...
               measured about x(i?1)
12         d % offset length, the distance from x(i?1) to x(i) measured ...
               along z(i)
13         theta % Joint angle, the angle between x(i?1) to x(i) ...
               measured about z(i)
14         p % 3D coordinates of the end of the link
15         i % link number
```

```
16      end
17      methods
18          function obj = armLink(a_n_1,alpha_n_1,d,theta,i)
19            if nargin > 0
20                if i == 4
21                    obj.theta = theta + 90;
22                else
23                    obj.theta = theta;
24                end
25                obj.a_n_1 = a_n_1;
26                obj.alpha_n_1 = alpha_n_1;
27                obj.d = d;
28                obj.T = [ cosd(theta)                  -sind(theta) ...
                                    0                 a_n_1;
29                          sind(theta)*cosd(alpha_n_1)  ...
                               cosd(theta)*cosd(alpha_n_1)   ...
                               -sind(alpha_n_1)  -sind(alpha_n_1)*d;
30                          sind(theta)*sind(alpha_n_1)  ...
                               cosd(theta)*sind(alpha_n_1)   ...
                               cosd(alpha_n_1)   cosd(alpha_n_1)*d;
31                          0                            0 ...
                                                           0                 1];
32            end
33          end
34          function calcTo(obj,To_prev)
35              if To_prev == 0
36                  obj.To = obj.T;
37              else
38                  obj.To = To_prev * obj.T;
39              end
40          end
41          function p = getP(obj)
```

```
42              obj.p = obj.To(1:3,4)';
43              p = obj.p;
44          end
45          function theta = getTheta(obj)
46              theta = obj.theta;
47          end
48      end
49  end
```

forwardKinematics.m

```
1  function [link1 link2 link3 link4 link5] = ...
       forwardKinematics(q1,q2,q3,q4,q5)
2  %    forwardKinematics -> Creates armLink objects for each arm - ...
       perform forward kinematics
3  %
4  %    by Aidan Scannell
5  %
6  %    ===== Inputs =====
7  %    qN - angle of joint N
8
9  %    ===== Outputs ====
10 %    linkN - armLink object for link N
11 global l1 l2 l3 l4
12
13      %% Initialise arm links
14      link1 = armLink(0,0,l1,q1,1);
15      link2 = armLink(0,90,0,q2,2);
16      link3 = armLink(l2,0,0,q3,3);
17      link4 = armLink(l3,0,0,q4-90,4);
```

```
18      link5 = armLink(0,-90,l4,q5,5);
19
20      %% Calculate compound transformation matrices
21      link1.calcTo(0);
22      link2.calcTo(link1.To);
23      link3.calcTo(link2.To);
24      link4.calcTo(link3.To);
25      link5.calcTo(link4.To);
26
27  end
```

workspace.m

```
1   function workspace(range1,range2,range3,range4,N_steps)
2   %   workspace -> Plots workspace for lynxmotion arm
3   %
4   %   by Aidan Scannell
5   %
6   %   ===== Inputs =====
7   %   rangeN - range of joint N
8   %   N_steps - number of steps within each joints range
9
10  %   ===== Outputs ====
11  %   prints number of iterations
12  %   plots 3D workspace
13
14      %% calcualte range step sizes for given N
15      step1 = diff(range1)/(N_steps-1);
16      step2 = diff(range2)/(N_steps-1);
17      step3 = diff(range3)/(N_steps-1);
```

```
18      step4 = diff(range4)/(N_steps-1);
19
20      %% Loop through joint angles and store end-effector coordinates
21
22      ii = 1; % counter
23
24      % initialise x, y, z arrays to speed up computation
25      N = N_steps^4 + 1;
26      x = zeros(N,1); y = zeros(N,1); z = zeros(N,1);
27
28      % matrix to plot different rgb intensities for each point
29      fill = zeros(N,3);
30
31      i = 1;
32      for q1 = range1(1):step1:range1(2)
33          fprintf('Workspace Iteration = %d\n', i)
34          for q2 = range2(1):step2:range2(2)
35              for q3 = range3(1):step3:range3(2)
36                  for q4 = range4(1):step4:range4(2)
37                      [link1 link2 link3 link4 link5] = ...
                            forwardKinematics(q1,q2,q3,q4,0); % ...
                            initialise objects
38
39                      p = [getP(link5)];
40  %                     range1 = [-90,90]; range2 = [0,180]; range3 = ...
      [-160,0]; range4 = [-130,90]; range5 = [0,360]; % set joint ranges
41  % range4 = [-90,90];
42  % range1 = [-90,90]; range2 = [0,180]; range3 = [-165,0]; range4 = ...
      [-180,0]; range5 = [0,360]; % set joint ranges
43                      fill(ii,1) = 1 - (q2 / diff(range2));
44                      fill(ii,2) = ((q3+diff(range3))/diff(range3));
45                      fill(ii,3) = ((q4+range4(2))/diff(range4));
```

```
46                      x(ii) = p(1);
47                      y(ii) = p(2);
48                      z(ii) = p(3);
49
50                      ii = ii + 1; % increment counter
51                  end
52              end
53          end
54          i = i + 1;
55      end
56
57      % print number of iterations
58      fprintf('Number of iterations = %d\n',ii);
59
60      %% Plot workspace
61      hold off; hold on
62      figure
63      title('Workspace')
64      scatter3(x,y,z,10,fill,'filled')
65      xlabel('x'); ylabel('y'); zlabel('z') % axis labels
66  %     hold off; hold on
67  %     figure
68  %     scatter(x,y,10,fill,'filled')
69  %     hold off; hold on
70  %     figure
71  %     scatter(y,z,10,fill,'filled')
72  %     hold off; hold on
73  %     figure
74  %     scatter(x,z,10,fill,'filled')
75
76  %     hold off; hold on;
77  %     iy = y(x ≥ 0);
```

```
78 %     iz = z(x ≥ 0);
79 %     figure
80 %     scatter(iy, iz)
81 %     grid on
82 %     title('Plot Z(Y,X=1)')
83 %     ix = x(y ≥ 0);
84 %     iz = z(y ≥ 0);
85 %     hold off; hold on;
86 %     figure
87 %     scatter(ix, iz)
88 %     grid on
89 %     title('Plot Z(X,Y=3*\pi/4)')
90 %     hold off
91
92 end
```

inverseKinematics.m

```
1  function solutions = inverseKinematics(x,y,z,pitch)
2  %   inverseKinematics -> Calculates arm angles for target end-effector
3  %   position (inverse kinematics)
4  %
5  %   by Aidan Scannell
6  %
7  %   ===== Inputs =====
8  %   x - target x position
9  %   y - target y position
10 %   z - target y position
11 %   pitch - target pitch
12
```

```
%   ===== Outputs ====
%   solutions - array of armLink objects for each valid solution
global l1 l2 l3 l4
global range1 range2 range3 range4 range5

    %% calculate theta 1 (XY plane)
    theta1 = atan2d(y,x);

    %% calculate theta 3 (XZ plane)
    %calculate joint 4 coordinares
    x4 = sqrt(( x^2)+(y^2)) - (l4*cosd(pitch) );

    % x4 = x - l4 * cosd(pitch);
    z4 = z - l4 * sind(pitch) - l1;

    % calculate cos(theta3)
    c3 = (x4^2 + z4^2 - l2^2 - l3^2)/(2 * l2 * l3);

    if c3^2 > 1
        fprintf('Target position not in workspace\n')
        return
    end

    % calculate theta3
    theta3(1) = atan2d(sqrt(1-c3^2),c3);
    theta3(2) = atan2d(-sqrt(1-c3^2),c3);

    %% calculate theta 2 (XZ plane)
    % calculate cos(theta2) and sin(theta2)
    c2(1) = (x4*(l2 + l3*cosd(theta3(1))) + ...
        z4*l3*(sind(theta3(1))))/(x4^2 + z4^2);
```

```
43      c2(2) = (x4*(l2 + l3*cosd(theta3(2))) + ...
            z4*l3*(sind(theta3(2))))/(x4^2 + z4^2);
44
45      % calculate theta2
46      theta2(1) = atan2d(sqrt(1-c2(1)^2),c2(1));
47      theta2(2) = atan2d(sqrt(1-c2(2)^2),c2(2));
48      theta2(3) = atan2d(-sqrt(1-c2(1)^2),c2(1));
49      theta2(4) = atan2d(-sqrt(1-c2(2)^2),c2(2));
50
51      %% calculate theta 4
52  %     theta4(1) = pitch - theta2(1) - theta3(1) - 90; % solution 1
53  %     theta4(2) = pitch - theta2(3) - theta3(1) - 90; % solution 2
54  %     theta4(3) = pitch - theta2(2) - theta3(2) - 90; % solution 3
55  %     theta4(4) = pitch - theta2(4) - theta3(2) - 90; % solution 4
56      theta4(1) = pitch - theta2(1) - theta3(1); % solution 1
57      theta4(2) = pitch - theta2(3) - theta3(1); % solution 2
58      theta4(3) = pitch - theta2(2) - theta3(2); % solution 3
59      theta4(4) = pitch - theta2(4) - theta3(2); % solution 4
60      thetas = {[theta2(1), theta3(1), theta4(1)],[theta2(3), ...
            theta3(1), theta4(2)],...
61           [theta2(2), theta3(2), theta4(3)],[theta2(4), theta3(2), ...
                theta4(4)]};
62
63      %% determine correct solutions
64      % perform FK for each solution and create a list of arm objects ...
            for each solution
65      [link11 link21 link31 link41 link51] = ...
            forwardKinematics(theta1,theta2(1),theta3(1),theta4(1),0);
66      [link12 link22 link32 link42 link52] = ...
            forwardKinematics(theta1,theta2(3),theta3(1),theta4(2),0);
67      [link13 link23 link33 link43 link53] = ...
            forwardKinematics(theta1,theta2(2),theta3(2),theta4(3),0);
```

```
68      [link14 link24 link34 link44 link54] = ...
            forwardKinematics(theta1,theta2(4),theta3(2),theta4(4),0);
69      links = {[link11 link21 link31 link41 link51],[link12 link22 ...
            link32 link42 link52],...
70           [link13 link23 link33 link43 link53],[link14 link24 link34 ...
                link44 link54]};
71
72      % loop through solutions
73      solutions = [];
74      for i = 1:4
75          pos4 = getP(links{i}(4)); % coordinates of joint 4
76          pos5 = getP(links{i}(5)); % coordinates of joint 5
77
78          % ensure joint angles are within joint ranges and that ...
                end-effector position is correct
79          if round(pos5(1),2) == round(x,2) && round(pos5(2),2) == ...
                round(y,2) && round(pos5(3),2)...
80                   == round(z,2) && theta1 ≥ range1(1) && theta1 ≤ ...
                         range1(2) && thetas{i}(1) ≥ range2(1) ...
81                   && thetas{i}(1)≤ range2(2) && thetas{i}(2) ≥ ...
                         range3(1) && thetas{i}(2) ≤ range3(2) && ...
82                   thetas{i}(3) ≥ range4(1) && thetas{i}(3) ≤ range4(2)
83
84              % calculate pitch
85              L_pitch = sqrt( (pos5(2) - pos4(2))^2 + (pos5(1) - ...
                    pos4(1))^2 ); % calculate length of link 4-5 in XY plane
86              FK_pitch = atan2d( (pos5(3) - pos4(3)), L_pitch ); % ...
                    calculate pitch
87
88              % print joint configuration
89              fprintf('--------- INVERSE KINEMATICS SOLUTION %d ...
                    ---------\n',i)
```

```
90              fprintf('IK solution %d: theta 1 = %.2f, theta 2 = %.2f, ...
                    theta 3 = %.2f, theta 4 = ...
                    %.2f\n',i,theta1,thetas{i}(1),thetas{i}(2),thetas{i}(3))
91
92              % print end-effector position and pitch
93              fprintf('IK end-effector position for solution %d: x = ...
                    %.2f, y = %.2f, z = %.2f, pitch = %.2f\n'...
94                  ,i,pos5(1),pos5(2),pos5(3),FK_pitch)
95
96              solutions = [solutions; links{i}];
97          end
98      end
99
100 end
```

task.m

```
1  function [pos, kk] = task(xyzp,title,pos,kk)
2  %   task -> performs inverse kinemtics for each position/orientation in
3  %   matrix xyzp and prints results
4  %
5  %   by Aidan Scannell
6  %
7  %   ===== Inputs =====
8  %   xyzp - matrix containing array of target positions/orientations ...
       [x; y; z; pitch]
9  %   title - string containing title of figure
10 %
11 %   ===== Outputs ====
12 %   prints target position and joint coordinates and angles
```

```
13      for i = 1:length(xyzp)
14          fprintf('\n------------------------------------ Position %d ...
                ------------------------------------\n',i)
15
16          % perform inverse kinematics
17          result = ...
                inverseKinematics(xyzp(i,1),xyzp(i,2),xyzp(i,3),xyzp(i,4));
18
19          % plot arm position on the same figure
20          p(:,:,i) = ...
                [[0,0,0,0];getP(result(1,1)),getTheta(result(1,1));getP(result(1,2)),...
21               getTheta(result(1,2));getP(result(1,3)),getTheta(result(1,3));getP(result
22               getTheta(result(1,4));getP(result(1,5)),getTheta(result(1,5))];
23          plotFK(title,result(1,1),result(1,2),result(1,3),result(1,4),result(1,5),0,0)
24
25          % print results
26          fprintf('TARGET: \nx = %d, y = %d, z = %d, pitch = ...
                %d\n\n\n',xyzp(i,1),xyzp(i,2),xyzp(i,3),xyzp(i,4))
27          fprintf('JOINT COORDINATES/ANGLES:\n')
28          for j = 2:6
29              fprintf('x%d = %.2f, y%d = %.2f, z%d = %.2f, theta%d = ...
                    %.2f\n',j-1,p(j,1,i),j-1,p(j,2,i),j-1,p(j,3,i),j-1,p(j,4,i))
30          end
31          fprintf(' \n\n')
32  %         pause(0.2)
33
34          % store joint angles for every time step
35          pos(1,kk) = getTheta(result(1,1));
36          pos(2,kk) = getTheta(result(1,2));
37          pos(3,kk) = getTheta(result(1,3));
38          pos(4,kk) = getTheta(result(1,4));
39          pos(5,kk) = getTheta(result(1,5));
```

```
40          kk = kk + 1;
41      end
42  end
```

freeMotion.m

```
1  function posEEfree = freeMotion(xyzp,N)
2  fprintf('\n\n------------------------------------ FREE MOTION ...
       TRAJECTORY ------------------------------------\n\n')
3      pos = zeros(4,N*(size(xyzp,1)-1));
4      kk = 1;
5      posEEfree = [];
6      for j = 1 : size(xyzp,1)-1
7          % inverse kinematics to get joint configurations for task ...
               position i and i+1
8          result1 = ...
               inverseKinematics(xyzp(j,1),xyzp(j,2),xyzp(j,3),xyzp(j,4));
9          result2 = ...
               inverseKinematics(xyzp(j+1,1),xyzp(j+1,2),xyzp(j+1,3),xyzp(j+1,4));
10
11         % create theta variables
12         theta11 = getTheta(result1(1,1)); theta21 = ...
               getTheta(result1(1,2)); theta31 = getTheta(result1(1,3));
13         theta41 = getTheta(result1(1,4)); theta51 = ...
               getTheta(result1(1,5)); theta12 = getTheta(result2(1,1));
14         theta22 = getTheta(result2(1,2)); theta32 = ...
               getTheta(result2(1,3)); theta42 = getTheta(result2(1,4));
15         theta52 = getTheta(result2(1,5));
16
17         % calculate angle step size for each joint and pitch
```

```
18 %         pitch = linspace(xyzp(j,4),xyzp(j+1,4),N); % vector of ...
       pitch angles
19         pitch = xyzp(j,4);
20         pitch_d = (xyzp(j+1,4) - xyzp(j,4))/N; % vector of pitch angles
21         theta1_d = (theta12 - theta11)/N;
22         theta2_d = (theta22 - theta21)/N;
23         theta3_d = (theta32 - theta31)/N;
24         theta4_d = (theta42 - theta41)/N;
25         theta5_d = (theta52 - theta51)/N;
26
27         for i = 1 : N
28             % function calculates compound transformation matrices
29             [link1 link2 link3 link4 link5] = ...
                   forwardKinematics(theta11,theta21,theta31,theta41,theta51);
30
31             % plot arm position on the same figure
32             clear p;
33             p(:,:,i) = ...
                   [[0,0,0,0];getP(link1),theta11;getP(link2),theta21;getP(link3),theta31
34             plotFK('Free Motion',link1,link2,link3,link4,link5,0,0)
35
36             posEEfree = [posEEfree; getP(link5)];
37
38             % print results
39             fprintf('------------------------------------ Position ...
                   %d ------------------------------------\n',i)
40             fprintf('JOINT COORDINATES/ANGLES:\n')
41             for j = 2:6
42                 fprintf('x%d = %.2f, y%d = %.2f, z%d = %.2f, theta%d ...
                       = ...
                       %.2f\n',j-1,p(j,1,i),j-1,p(j,2,i),j-1,p(j,3,i),j-1,p(j,4,i))
43             end
```

```
44              fprintf(' \n\n')
45  %             pause(0.2)
46
47              % store joint angles for every time step
48              pos(1,kk) = theta11;
49              pos(2,kk) = theta21;
50              pos(3,kk) = theta31;
51              pos(4,kk) = theta41;
52              pos(5,kk) = theta51;
53              kk = kk + 1;
54
55              % update angles for next iteration
56              theta11 = theta11 + theta1_d;
57              theta21 = theta21 + theta2_d;
58              theta31 = theta31 + theta3_d;
59              theta41 = theta41 + theta4_d;
60              theta51 = theta51 + theta5_d;
61              pitch = pitch + pitch_d;
62
63          end
64      end
65
66      % store end-effector positions and plot
67      posEEfree = [posEEfree;xyzp(end,1),xyzp(end,2),xyzp(end,3)];
68      plot3(posEEfree(:,1),posEEfree(:,2),posEEfree(:,3),'*-')
69
70      % Plot joint position
71      plotPos(pos,kk)
72
73      % Plot joint speed
74      plotSpeed(pos,kk)
75
```

```
76      % Plot joint acceleration
77      plotAcc(pos,kk)
78  end
```

straightLine.m

```
1   function posEElin = straightLine(xyzp,N)
2       fprintf('\n\n---------------------------------- STRAIGHT LINE ...
            TRAJECTORY ----------------------------------\n\n')
3       hold off; figure;
4       pos = zeros(4,N*(size(xyzp,1)-1));
5       kk = 1;
6       posEElin = [];
7
8       for j = 1 : size(xyzp,1)-1
9
10          % create vectors of x,y,z,p positions for N steps
11          x = linspace(xyzp(j,1),xyzp(j+1,1),N); % vector of x positions
12          y = linspace(xyzp(j,2),xyzp(j+1,2),N); % vector of y positions
13          z = linspace(xyzp(j,3),xyzp(j+1,3),N); % vector of z positions
14          p = linspace(xyzp(j,4),xyzp(j+1,4),N); % vector of pitch angles
15
16          position = [x;y;z;p]';
17          clear p;
18          for i = 1:length(position)
19              fprintf('\n------------------------------------ ...
                    Position %d ------------------------------------\n',i)
20
21              % perform inverse kinematics
```

```
22              result = ...
                    inverseKinematics(position(i,1),position(i,2),position(i,3),position(
23
24              % plot arm position on the same figure
25              p(:,:,i) = ...
                    [[0,0,0,0];getP(result(1,1)),getTheta(result(1,1));getP(result(1,2)),
26                   getTheta(result(1,2));getP(result(1,3)),getTheta(result(1,3));getP(re
27                   getTheta(result(1,4));getP(result(1,5)),getTheta(result(1,5))];
28              plotFK('Straight Line ...
                    Trajectory',result(1,1),result(1,2),result(1,3),result(1,4),result(1,5
29
30              % print results
31              fprintf('TARGET: \nx = %d, y = %d, z = %d, pitch = ...
                    %d\n\n',position(i,1),position(i,2),...
32                   position(i,3),position(i,4))
33              fprintf('JOINT COORDINATES/ANGLES:\n')
34              for j = 2:6
35                  fprintf('x%d = %.2f, y%d = %.2f, z%d = %.2f, theta%d ...
                        = %.2f\n',j-1,p(j,1,i),j-1,...
36                       p(j,2,i),j-1,p(j,3,i),j-1,p(j,4,i))
37              end
38              fprintf(' \n\n')
39      %         pause(0.2)
40
41              % store joint angles for every time step
42              pos(1,kk) = getTheta(result(1,1));
43              pos(2,kk) = getTheta(result(1,2));
44              pos(3,kk) = getTheta(result(1,3));
45              pos(4,kk) = getTheta(result(1,4));
46              pos(5,kk) = getTheta(result(1,5));
47              kk = kk + 1;
48
```

```
49              % store positions of end effector
50              [link1 link2 link3 link4 link5] = ...
                    forwardKinematics(getTheta(result(1,1)),getTheta(result(1,2)),...
51                   getTheta(result(1,3)),getTheta(result(1,4)),getTheta(result(1,5)));
52              posEElin = [posEElin; getP(link5)];
53          end
54
55
56
57      end
58
59      % plot straight line between points
60      hold on
61      plot3(posEElin(:,1),posEElin(:,2),posEElin(:,3),'*-')
62
63      % Plot joint position
64      plotPos(pos,kk)
65
66      % Plot joint speed
67      plotSpeed(pos,kk)
68
69      % Plot joint acceleration
70      plotAcc(pos,kk)
71  end
```

polynomialTraj.m

```
1  function posEEpoly = polynomialTraj(xyzp,time_steps,N)
2      fprintf('\n\n------------------------------------ POLYNOMIAL ...
           TRAJECTORY ------------------------------------\n\n')
```

```
3       hold off; figure;
4       pos = zeros(4,N*(size(xyzp,1)-1));
5       kk = 1;
6       posEEpoly = [];
7       for j = 1 : size(xyzp,1)
8           % inverse kinematics to get joint configurations for task ...
                position i and i+1
9           result = ...
                inverseKinematics(xyzp(j,1),xyzp(j,2),xyzp(j,3),xyzp(j,4));
10
11          % create theta variables
12          for i = 1:5
13              theta(i,j) = getTheta(result(1,i));
14          end
15
16      end
17
18      for j = 1 : size(xyzp,1)-1
19          for i = 1:5
20              q_0 = theta(i,j);
21              q_f = theta(i,j+1);
22              t_0 = 0;
23              t_f = 1;
24              v_0 = 0;
25              v_f = 0;
26
27              A = [1,  t_0,  t_0^2,  t_0^3;
28                   0,  1,    2*t_0,  3*t_0^3;
29                   1,  t_f,  t_f^2,  t_f^3;
30                   0,  1,    2*t_f,  3*t_f^2;];
31
32              B = [q_0; v_0; q_f; v_f];
```

```
33
34              X{j}(i,:) = linsolve(A,B);
35          end
36      end
37
38      % iterate through time steps
39      x = linspace(0,t_f,time_steps);
40      for j = 1 : size(xyzp,1)-1
41          for ii = 1 : time_steps
42
43              for i = 1:5
44                  theta_(i) = X{j}(i,4)*x(ii)^3 + X{j}(i,3)*x(ii)^2 + ...
                        X{j}(i,2)*x(ii)^1 + X{j}(i,1);
45              end
46
47              % function calculates compound transformation matrices
48              [link1 link2 link3 link4 link5] = ...
                    forwardKinematics(theta_(1),theta_(2),theta_(3),theta_(4),theta_(5));
49
50              % plot arm position on the same figure
51              plotFK('Cubic Polynomial ...
                    Motion',link1,link2,link3,link4,link5,0,0)
52
53              % store joint angles for every time step
54              pos(1,kk) = theta_(1);
55              pos(2,kk) = theta_(2);
56              pos(3,kk) = theta_(3);
57              pos(4,kk) = theta_(4);
58              pos(5,kk) = theta_(5);
59              kk = kk + 1;
60
61              % store end-effector position
```

```
62              posEEpoly = [posEEpoly; getP(link5)];
63          end
64      end
65
66      % plot straight line between points
67      hold on
68      plot3(posEEpoly(:,1),posEEpoly(:,2),posEEpoly(:,3),'*-')
69
70      % Plot joint position
71      plotPos(pos,kk)
72
73      % Plot joint speed
74      plotSpeed(pos,kk)
75
76      % Plot joint acc
77      plotAcc(pos,kk)
78  end
```

quinticPolyTraj.m

```
1   function posEEquint = quinticPolyTraj(xyzp,time_steps,N)
2       fprintf('\n\n------------------------------------ POLYNOMIAL ...
            TRAJECTORY ------------------------------------\n\n')
3       hold off; figure;
4       pos = zeros(4,N*(size(xyzp,1)-1));
5       kk = 1;
6       posEEquint = [];
7       for j = 1 : size(xyzp,1)
8           % inverse kinematics to get joint configurations for task ...
                position i and i+1
```

```
9           result = ...
                inverseKinematics(xyzp(j,1),xyzp(j,2),xyzp(j,3),xyzp(j,4));
10
11          % create theta variables
12          for i = 1:5
13              theta(i,j) = getTheta(result(1,i));
14          end
15
16      end
17
18      for j = 1 : size(xyzp,1)-1
19          for i = 1:5
20              q_0 = theta(i,j);
21              q_f = theta(i,j+1);
22              t_0 = 0;
23              t_f = 1;
24              v_0 = 0;
25              v_f = 0;
26              a_0 = 0;
27              a_f = 0;
28
29              A = [1,  t_0,  t_0^2,  t_0^3,   t_0^4,    t_0^5;
30                   0,  1,    2*t_0,  3*t_0^3, 4*t_0^3,  5*t_0^4;
31                   0,  0,    2,      6*t_0,   12*t_0^2, 20*t_0^3;
32                   1,  t_f,  t_f^2,  t_f^3,   t_f^4,    t_f^5;
33                   0,  1,    2*t_f,  3*t_f^2, 4*t_f^3,  5*t_f^4;
34                   0,  0,    0,      6*t_f,   12*t_f^2, 20*t_f^3];
35
36              B = [q_0; v_0; a_0; q_f; v_f; a_f];
37
38              X{j}(i,:) = linsolve(A,B);
39          end
```

```
40      end
41
42      % iterate through time steps
43      x = linspace(0,t_f,time_steps);
44      for j = 1 : size(xyzp,1)-1
45          for ii = 1 : time_steps
46
47              for i = 1:5
48                  theta_(i) = X{j}(i,6)*x(ii)^5 + X{j}(i,5)*x(ii)^4 + ...
                        X{j}(i,4)*x(ii)^3 + ...
49                      X{j}(i,3)*x(ii)^2 + X{j}(i,2)*x(ii)^1 + X{j}(i,1);
50              end
51
52              % function calculates compound transformation matrices
53              [link1 link2 link3 link4 link5] = ...
                    forwardKinematics(theta_(1),theta_(2),theta_(3),theta_(4),theta_(5));
54
55              % plot arm position on the same figure
56              plotFK('Quintic Motion',link1,link2,link3,link4,link5,0,0)
57
58              % store joint angles for every time step
59              pos(1,kk) = theta_(1);
60              pos(2,kk) = theta_(2);
61              pos(3,kk) = theta_(3);
62              pos(4,kk) = theta_(4);
63              pos(5,kk) = theta_(5);
64              kk = kk + 1;
65
66              % store end-effector position
67              posEEquint = [posEEquint; getP(link5)];
68          end
69      end
```

```
70
71      % plot straight line between points
72      hold on
73      plot3(posEEquint(:,1),posEEquint(:,2),posEEquint(:,3),'*-')
74
75      % Plot joint position
76      plotPos(pos,kk)
77
78      % Plot joint speed
79      plotSpeed(pos,kk)
80
81      % Plot joint acc
82      plotAcc(pos,kk)
83  end
```

obsAvoid.m

```
1   function posEEobs = obsAvoid(xyzp,N,time_steps)
2       fprintf('\n\n------------------------------------ OBSTACLE ...
            AVOIDANCE TRAJECTORY ------------------------------------\n\n')
3       hold off; figure;
4       pos = zeros(4,N*(size(xyzp,1)-1));
5       kk = 1;
6       posEEobs = [];
7   %     N = 10;
8   %     time_steps = 20;
9   %     xyzp = [0;0;0;0];
10  %     for j = 1 : size(position,1)-1
11  %
12  %         % create vectors of x,y,z,p positions for N steps
```

```
13 %         x = linspace(position(j,1),position(j+1,1),N); % vector of ...
       x positions
14 %         y = linspace(position(j,2),position(j+1,2),N); % vector of ...
       y positions
15 %         z = linspace(position(j,3),position(j+1,3),N); % vector of ...
       z positions
16 %         pitch = linspace(position(j,4),position(j+1,4),N); % ...
       vector of pitch angles
17 %
18 %         xyzp = [xyzp(1,:), x(2:end); xyzp(2,:), y(2:end); ...
       xyzp(3,:), z(2:end); xyzp(4,:), pitch(2:end)];
19 %     end
20 %     xyzp = xyzp(:,2:end);
21 %     for j = 1 : size(xyzp,2)
22 %         % inverse kinematics to get joint configurations for task ...
       position i and i+1
23 %         result = ...
       inverseKinematics(xyzp(1,j),xyzp(2,j),xyzp(3,j),xyzp(4,j));
24 %
25 %         % create theta variables
26 %         for i = 1:5
27 %             theta(i,j) = getTheta(result(1,i));
28 %         end
29 %     end
30 %     for j = 1 : size(xyzp,2)-1
31 %         for i = 1:5
32 %             q_0 = theta(i,j);
33 %             q_f = theta(i,j+1);
34 %             t_0 = 0;
35 %             t_f = 1;
36 %             v_0 = 0;
37 %             v_f = 0;
```

```
38 %             a_0 = 0;
39 %             a_f = 0;
40 %
41 %             A = [1,  t_0,  t_0^2,  t_0^3,   t_0^4,    t_0^5;
42 %                  0,  1,    2*t_0,  3*t_0^3, 4*t_0^3,  5*t_0^4;
43 %                  0,  0,    2,      6*t_0,   12*t_0^2, 20*t_0^3;
44 %                  1,  t_f,  t_f^2,  t_f^3,   t_f^4,    t_f^5;
45 %                  0,  1,    2*t_f,  3*t_f^2, 4*t_f^3,  5*t_f^4;
46 %                  0,  0,    0,      6*t_f,   12*t_f^2, 20*t_f^3];
47 %
48 %             B = [q_0; v_0; a_0; q_f; v_f; a_f];
49 %
50 %             X{j}(i,:) = linsolve(A,B);
51 %         end
52 %     end
53 %
54 %     % iterate through time steps
55 %     x = linspace(0,t_f,time_steps);
56 %     for j = 1 : size(xyzp,1)-1
57 %         for ii = 1 : time_steps
58 %
59 %             for i = 1:5
60 %                 theta_(i) = X{j}(i,6)*x(ii)^5 + X{j}(i,5)*x(ii)^4 ...
    + X{j}(i,4)*x(ii)^3 + ...
61 %                     X{j}(i,3)*x(ii)^2 + X{j}(i,2)*x(ii)^1 + X{j}(i,1);
62 %             end
63 %
64 %             % function calculates compound transformation matrices
65 %             [link1 link2 link3 link4 link5] = ...
    forwardKinematics(theta_(1),theta_(2),theta_(3),theta_(4),theta_(5));
66 %
67 %             % plot arm position on the same figure
```

```
68 %             plotFK('Quintic Motion',link1,link2,link3,link4,link5,0,0)
69 %
70 %             % store joint angles for every time step
71 %             pos(1,kk) = theta_(1);
72 %             pos(2,kk) = theta_(2);
73 %             pos(3,kk) = theta_(3);
74 %             pos(4,kk) = theta_(4);
75 %             pos(5,kk) = theta_(5);
76 %             kk = kk + 1;
77 %
78 %             % store end-effector position
79 %             posEEobs = [posEEobs; getP(link5)];
80 %         end
81 %     end
82 %
83 %     % plot straight line between points
84 %     hold on
85 %     plot3(posEEobs(:,1),posEEobs(:,2),posEEobs(:,3),'*-')
86 %
87 %     % plot cylinder
88 %     [X,Y,Z] = cylinder(40,100);
89 %     plot3(X+150,Y+150,Z*400,'s-')
90 %
91 %     % Plot joint position
92 %     plotPos(pos,kk)
93 %
94 %     % Plot joint speed
95 %     plotSpeed(pos,kk)
96 %
97 %     % Plot joint acc
98 %     plotAcc(pos,kk)
99
```

```
100     for j = 1 : size(xyzp,1)-1
101
102         % create vectors of x,y,z,p positions for N steps
103         x = linspace(xyzp(j,1),xyzp(j+1,1),N); % vector of x positions
104         y = linspace(xyzp(j,2),xyzp(j+1,2),N); % vector of y positions
105         z = linspace(xyzp(j,3),xyzp(j+1,3),N); % vector of z positions
106         pitch = linspace(xyzp(j,4),xyzp(j+1,4),N); % vector of pitch ...
                angles
107
108         position = [x;y;z;pitch]';
109         clear p;
110         for i = 1:length(position)-1
111             fprintf('\n------------------------------------ ...
                    Position %d ------------------------------------\n',i)
112
113             % perform inverse kinematics
114             result = ...
                    inverseKinematics(position(i,1),position(i,2),position(i,3),position(
115
116             % plot arm position on the same figure
117             p(:,:,i) = ...
                    [[0,0,0,0];getP(result(1,1)),getTheta(result(1,1));getP(result(1,2)),
118                  getTheta(result(1,2));getP(result(1,3)),getTheta(result(1,3));getP(re
119                  getTheta(result(1,4));getP(result(1,5)),getTheta(result(1,5))];
120             plotFK('Obstacle Avoidance ...
                    Trajectory',result(1,1),result(1,2),result(1,3),result(1,4),result(1,5
121
122             % print results
123             fprintf('TARGET: \nx = %d, y = %d, z = %d, pitch = ...
                    %d\n\n',position(i,1),position(i,2),...
124                  position(i,3),position(i,4))
125             fprintf('JOINT COORDINATES/ANGLES:\n')
```

```
126             for j = 2:6
127                 fprintf('x%d = %.2f, y%d = %.2f, z%d = %.2f, theta%d ...
                        = %.2f\n',j-1,p(j,1,i),j-1,...
128                     p(j,2,i),j-1,p(j,3,i),j-1,p(j,4,i))
129             end
130             fprintf(' \n\n')
131     %         pause(0.2)
132
133             % store joint angles for every time step
134             pos(1,kk) = getTheta(result(1,1));
135             pos(2,kk) = getTheta(result(1,2));
136             pos(3,kk) = getTheta(result(1,3));
137             pos(4,kk) = getTheta(result(1,4));
138             pos(5,kk) = getTheta(result(1,5));
139             kk = kk + 1;
140
141             % store positions of end effector
142             [link1 link2 link3 link4 link5] = ...
                    forwardKinematics(getTheta(result(1,1)),getTheta(result(1,2)),...
143                 getTheta(result(1,3)),getTheta(result(1,4)),getTheta(result(1,5)));
144             posEEobs = [posEEobs; getP(link5)];
145         end
146
147
148
149     end
150
151     % plot straight line between points
152     hold on
153     plot3(posEEobs(:,1),posEEobs(:,2),posEEobs(:,3),'*-')
154
155     [X,Y,Z] = cylinder(40,100);
```

```
156     plot3(X+150,Y+150,Z*400,'s-')
157
158     % Plot joint position
159     plotPos(pos,kk)
160
161     % Plot joint speed
162     plotSpeed(pos,kk)
163
164     % Plot joint acceleration
165     plotAcc(pos,kk)
166 end
```

plotPos.m

```
1  function plotPos(pos,kk)
2      % Plot joint position
3      hold off
4      figure
5      hold on
6      plot([0:size(pos(1,:),2)-1],pos(1,:),'->')
7      plot([0:size(pos(1,:),2)-1],pos(2,:),'-*')
8      plot([0:size(pos(1,:),2)-1],pos(3,:),'-s')
9      plot([0:size(pos(1,:),2)-1],pos(4,:),'-d')
10 %      plot([1:kk-1],pos(5,:),'-^')
11     xlabel('Time Step, i')
12     ylabel('Joint Position, (degrees)')
13     legend('Joint 1','Joint 2','Joint 3','Joint 4')
14 %      legend('Joint 1','Joint 2','Joint 3','Joint 4','Joint 5')
15 end
```

plotSpeed.m

```
function plotSpeed(pos,kk)
    % Plot joint position
    dy1 = diff(pos(1,:));% ./ diff([0:kk-2]);
    dy2 = diff(pos(2,:));% ./ diff([0:kk-2]);
    dy3 = diff(pos(3,:));% ./ diff([0:kk-2]);
    dy4 = diff(pos(4,:));% ./ diff([0:kk-2]);
    dy5 = diff(pos(5,:));% ./ diff([0:kk-2]);

    hold off
    figure
    hold on
    plot([0:size(pos(1,:),2)-2],dy1,'->')
    plot([0:size(pos(1,:),2)-2],dy2,'-*')
    plot([0:size(pos(1,:),2)-2],dy3,'-s')
    plot([0:size(pos(1,:),2)-2],dy4,'-d')
%     plot([2:kk-1],dy5,'-^')
    xlabel('Time Step, i')
    ylabel('Joint Speed, (degrees/s)')
    legend('Joint 1','Joint 2','Joint 3','Joint 4')
%     legend('Joint 1','Joint 2','Joint 3','Joint 4','Joint 5')
end
```

plotAcc.m

```
function plotAcc(pos,kk)
    % Plot joint position
    dy1 = diff(pos(1,:));% ./ diff([1:kk-1]);
```

```
4       dy2 = diff(pos(2,:));% ./ diff([1:kk-1]);
5       dy3 = diff(pos(3,:));% ./ diff([1:kk-1]);
6       dy4 = diff(pos(4,:));% ./ diff([1:kk-1]);
7       dy5 = diff(pos(5,:));% ./ diff([1:kk-1]);
8
9   %     dy12 = diff(dy1)
10  %     dy22 = diff(dy2)
11  %     dy32 = diff(dy3)
12  %     dy42 = diff(dy4)
13  %     dy52 = diff(dy5)
14
15      dy12 = diff(dy1);% ./ diff([1:kk-2]);
16      dy22 = diff(dy2);% ./ diff([1:kk-2]);
17      dy32 = diff(dy3);% ./ diff([1:kk-2]);
18      dy42 = diff(dy4);% ./ diff([1:kk-2]);
19      dy52 = diff(dy5);% ./ diff([1:kk-2]);
20
21      hold off
22      figure
23      hold on
24      plot([0:size(pos(1,:),2)-3],dy12,'->')
25      plot([0:size(pos(1,:),2)-3],dy22,'-*')
26      plot([0:size(pos(1,:),2)-3],dy32,'-s')
27      plot([0:size(pos(1,:),2)-3],dy42,'-d')
28  %     plot([2:kk-1],dy5,'-^')
29      xlabel('Time Step, i')
30      ylabel('Joint Acceleration, (degrees/s^2)')
31      legend('Joint 1','Joint 2','Joint 3','Joint 4')
32  %     legend('Joint 1','Joint 2','Joint 3','Joint 4','Joint 5')
33  end
```

plotFK.m

```
function ...
    plotFK(t,link1,link2,link3,link4,link5,new_figure_flag,joint_angles_flag)
%   plotFK -> Plot the robotic arm in 3D space for given foward ...
    kinematics
%
%   by Aidan Scannell
%
%   ===== Inputs =====
%   linkN - armLink object for link N

%   ===== Outputs ====
%   plots arm in 3D space
    if new_figure_flag == 1
        figure
    end
    title(t)
    hold on; grid on
    xlabel('x'); ylabel('y'); zlabel('z') % z-axis label
    theta1 = getTheta(link1); theta2 = getTheta(link2); theta3 = ...
        getTheta(link3); theta4 = getTheta(link4);
    theta5 = getTheta(link5);
    p1 = getP(link1); p2 = getP(link2); p3 = getP(link3); p4 = ...
        getP(link4); p5 = getP(link5);
    p = ...
        [[0,0,0];getP(link1);getP(link2);getP(link3);getP(link4);getP(link5)];
    if joint_angles_flag == 1
        text(0,0,0,num2str(theta1,4));
        text(p2(1),p2(2),p2(3),num2str(theta2,4));
        text(p3(1),p3(2),p3(3),num2str(theta3,4));
```

```
25          text(p4(1),p4(2),p4(3),num2str(theta4,4));
26          text(p5(1),p5(2),p5(3),num2str(theta5,4));
27      end
28
29  %       text(p5(1),p5(2),p5(3),['pos ',num2str(iii)]);
30  %       grid on
31  %       daspect([1 1 1])
32  %       minx = min([0, p2(1), p3(1), p4(1), p5(1)]);
33  %       maxx = max([0, p2(1), p3(1), p4(1), p5(1)]);
34  %       miny = min([0, p2(2), p3(2), p4(2), p5(2)]);
35  %       maxy = max([0, p2(2), p3(2), p4(2), p5(2)]);
36  %       minz = min([0, p2(3), p3(3), p4(3), p5(3)]);
37  %       maxz = max([0, p2(3), p3(3), p4(3), p5(3)]);
38  %       minA = min([minx, miny, minz]);
39  %       maxA = max([maxx, maxy, maxz]);
40  %       axis([minA maxA minA maxA minA maxA])
41      plot3(p(:,1),p(:,2),p(:,3), 'b')
42  end
```

PARALLEL KINEMATICS

parallelKinematics.m

```
1  %  Planar parallel Kinematics.
2  %  Robotic Fundamentals UFMF4X-15-M
3  %  paralellKinamtics.m
4  %
5  %  Created by AidanScannell.
6  %  Copyright l' 2017 AidanScannell. All rights reserved.
```

```
7
8  %% Setup
9  clear all
10 clc
11 close all
12
13 arm; % import arm class
14
15 % set flags
16 IK_flag = 1;
17 workspace_flag = 1;
18
19 % Specify design parameters
20 s_a = 170; % lower section
21 L = 130; % upper section
22 r_p = 130; % platform joint circle radius
23 r_b = 290; % base joint circle radius
24
25 %% Inverse Kinematics
26 if IK_flag == 1
27
28     % Input parameters
29     X_c = 200;
30     Y_c = 150;
31     a = -30;
32
33     % Calculate global variables
34     BC = [X_c-r_b*cosd(30);Y_c-r_b*sind(30);0]; % Define vector BC
35     R_BC = [cosd(a), -sind(a), 0; sind(a), cosd(a), 0; 0, 0, 1]; % ...
           Define 2D rotation matrix for platform with rotation a
36
37     % Create arm objects
```

```
38      arm1 = arm(1,R_BC,BC,s_a,r_p,L,r_b,a,X_c,Y_c);
39      arm2 = arm(2,R_BC,BC,s_a,r_p,L,r_b,a,X_c,Y_c);
40      arm3 = arm(3,R_BC,BC,s_a,r_p,L,r_b,a,X_c,Y_c);
41
42      % determine if target position is within workspace
43      [x,y,ii,singularity_flag] = ...
            singularity(arm1,arm2,arm3,X_c,Y_c,1,[],[]);
44
45      if singularity_flag == 0
46          % Get joint coordinates
47          posArm1 = getJointCo(arm1,s_a,L);
48          posArm2 = getJointCo(arm2,s_a,L);
49          posArm3 = getJointCo(arm3,s_a,L);
50
51          % Create lists of arm parameters for looping
52          arms = ...
                {getJointCo(arm1,s_a,L),getJointCo(arm2,s_a,L),getJointCo(arm3,s_a,L)};
53          theta = {arm1.getTheta, arm2.getTheta, arm3.getTheta};
54          psi = {arm1.getPsi, arm2.getPsi, arm3.getPsi};
55
56          % Plot
57          figure
58          hold on; grid on
59  %         title('Inverse Kinematics')
60          xlabel('x'); ylabel('y'); % axis labels
61          axis equal;
62          plot(X_c,Y_c,'X') % target position (for center of platform)
63          tri_b = [0 2*r_b*cosd(30) r_b*cosd(30) 0; 0 0 (r_b*sind(30) ...
                + r_b) 0]; % base triangle
64          tri_p = [posArm1(1,3) posArm2(1,3) posArm3(1,3) ...
                posArm1(1,3); posArm1(2,3) posArm2(2,3) posArm3(2,3) ...
                posArm1(2,3)]; % platform triangle
```

```
65          plot(tri_p(1,:), tri_p(2,:)) % plot base
66          plot(tri_b(1,:), tri_b(2,:)) % plot platform
67
68          for i = 1:3
69              text(arms{i}(1,3),arms{i}(2,3),num2str(i)); % add PP label
70              text(arms{i}(1,1),arms{i}(2,1),num2str(theta{i}(1))); % ...
                    add joint theta values
71              text(arms{i}(1,2),arms{i}(2,2),num2str(psi{i}(1))); % ...
                    add joint psi values
72              plot(arms{i}(1,:),arms{i}(2,:)); % plot arm
73          end
74      else
75          fprintf('Target position outside of workspace (singularity)\n')
76      end
77  end
78
79  %% Workspace
80  if workspace_flag == 1
81
82      % Input parameters
83      x = zeros(1,120);
84      y = zeros(1,120);
85      a = 0;
86
87      % Calculate workspace by neglecting singularities
88      ii = 1;
89      for X_c = 5:5:600
90          for Y_c = 5:5:600
91              % Calculate global variables
92              BC = [X_c-r_b*cosd(30);Y_c-r_b*sind(30);0]; % Define ...
                    vector BC
```

```
93              R_BC = [cosd(a), -sind(a), 0; sind(a), cosd(a), 0; 0, 0, ...
                    1]; % Define 2D rotation matrix for platform with ...
                    rotation a
94
95              % Create arm objects
96              arm1 = arm(1,R_BC,BC,s_a,r_p,L,r_b,a,X_c,Y_c);
97              arm2 = arm(2,R_BC,BC,s_a,r_p,L,r_b,a,X_c,Y_c);
98              arm3 = arm(3,R_BC,BC,s_a,r_p,L,r_b,a,X_c,Y_c);
99
100             % Determine which points lie within the workspace
101             [x,y,ii,singularity_flag] = ...
                    singularity(arm1,arm2,arm3,X_c,Y_c,ii,x,y);
102
103         end
104     end
105
106     % Determine workspace by plotting range of each joint
107     jj = 1;
108     for theta = 1:5:360
109         for psi = 1:5:360
110
111             % x,y positions for each joint
112             x1(jj) = s_a*cosd(theta) + L*cosd(psi) + r_p*cosd(30-a);
113             x2(jj) = s_a*cosd(theta+120) + L*cosd(psi+120) + ...
                    r_p*cosd(30+120-a) + 2*r_b*cosd(30);
114             x3(jj) = s_a*cosd(theta+240) + L*cosd(psi+240) + ...
                    r_p*cosd(30+240-a) + r_b*cosd(30);
115             y1(jj) = s_a*sind(theta) + L*sind(psi) + r_p*sind(30-a);
116             y2(jj) = s_a*sind(theta+120) + L*sind(psi+120) + ...
                    r_p*sind(30+120-a);
117             y3(jj) = s_a*sind(theta+240) + L*sind(psi+240) + ...
                    r_p*sind(30+240-a) + (r_b*sind(30) + r_b);
```

```
118
119             jj = jj + 1;
120         end
121     end
122
123     % Plot
124     figure
125     hold off; hold on; grid on
126 %     title(['Workspace (a = ' num2str(a) ')'])
127     xlabel('x'); ylabel('y'); % axis labels
128     axis([-10 550 -10 450])
129     tri_b = [0 2*r_b*cosd(30) r_b*cosd(30) 0; 0 0 (r_b*sind(30) + ...
            r_b) 0]; % base triangle
130     plot(tri_b(1,:), tri_b(2,:)) % plot base
131     plot(x,y,'o') % target position (for center of platform)
132
133     % Plot
134     figure
135     hold off; hold on; grid on
136 %     title(['Workspace (a = ' num2str(a) ')'])
137     xlabel('x'); ylabel('y'); % axis labels
138     axis equal
139     tri_b = [0 2*r_b*cosd(30) r_b*cosd(30) 0; 0 0 (r_b*sind(30) + ...
            r_b) 0]; % base triangle
140     plot(tri_b(1,:), tri_b(2,:)) % plot base
141     plot(x1,y1,'.r','MarkerSize',1) % target position (for center of ...
            platform)
142     plot(x2,y2,'.g','MarkerSize',1) % target position (for center of ...
            platform)
143     plot(x3,y3,'.b','MarkerSize',1) % target position (for center of ...
            platform)
144
```

```
145 end
146
147 %% Singularity function
148 function [x,y,ii,singularity_flag] = ...
        singularity(arm1,arm2,arm3,X_c,Y_c,ii,x,y)
149 %   singularity -> determines if the target position is within the
150 %   workspace for the given arm links
151 %
152 %   by Aidan Scannell
153 %
154 %   ===== Inputs =====
155 %   armN - armLink object N
156 %   X_c - target x position
157 %   Y_c - target y position
158 %   ii - iteration number for workspace plotting
159 %   x - vector containing x positions in workspace
160 %   y - vector containing y positions in workspace
161 %
162 %   ===== Outputs ====
163 %   ii - iteration number for workspace plotting
164 %   x - vector containing x positions in workspace
165 %   y - vector containing y positions in workspace
166 %   singularity_flag - 0 if not singularity, 1 if singularity
167     singularity_flag = 1;
168     theta1 = getTheta(arm1);
169     theta2 = getTheta(arm2);
170     theta3 = getTheta(arm3);
171
172     psi1 = getPsi(arm1);
173     psi2 = getPsi(arm2);
174     psi3 = getPsi(arm3);
```

```
175     if (isnan(theta1(1)) == 0) && (isnan(theta2(1)) == 0) && ...
            (isnan(theta3(1)) == 0) && ...
176             (isnan(psi1) == 0) && (isnan(psi2) == 0) && (isnan(psi3) ...
                    == 0)
177         x(ii) = X_c;
178         y(ii) = Y_c;
179         ii = ii + 1;
180         singularity_flag = 0;
181     end
182     if (isnan(theta1(2)) == 0) && (isnan(theta2(2)) == 0) && ...
            (isnan(theta3(2)) == 0) && ...
183             (isnan(psi1) == 0) && (isnan(psi2) == 0) && (isnan(psi3) ...
                    == 0)
184         x(ii) = X_c;
185         y(ii) = Y_c;
186         ii = ii + 1;
187         singularity_flag = 0;
188     end
189 end
```

arm.m

```
1  %   Arm class for planar parallell robot.
2  %   Robotic Fundamentals UFMF4X-15-M
3  %   arm.m
4  %
5  %   by Aidan Scannell
6  classdef arm < handle
7      properties
8          theta % link 1 angle
```

```
9           psi % link 2 angle
10          PBPP % vector from PB to PP
11          CPP % distance from C to PP
12          BPB % vector between centre of base platform and base joint ...
                for arm i
13          PB % x,y position of base
14          points % matrix of [x,y] positions for each joint of arm
15          T % transformation matrix
16      end
17      methods
18          function obj = arm(i,R_BC,BC,s_a,r_p,L,r_b,a,X_c,Y_c)
19              if nargin > 0
20                  % set a and PB for joint i
21                  if i == 1
22                      a = a + 30;
23                      obj.PB(1) = 0;
24                      obj.PB(2) = 0;
25  %                     BC = [r_b*cosd(30)-X_c;r_b*sind(30)-Y_c;0]; % ...
        Define vector BC
26                  elseif i == 2
27                      a = a + 120 + 30;
28                      obj.PB(1) = 2*r_b*cosd(30);
29                      obj.PB(2) = 0;
30  %                     BC = [X_c-r_b*cosd(30);Y_c-r_b*sind(30);0]; % ...
        Define vector BC
31                  elseif i == 3
32                      a = a + 120*2 + 30;
33                      obj.PB(1) = r_b*cosd(30);
34                      obj.PB(2) = r_b*sind(30) + r_b;
35  %                     BC = [X_c-r_b*cosd(30);Y_c-r_b*sind(30);0]; % ...
        Define vector BC
36                  end
```

```
37                  obj.BPB = [-r_b*cosd(a); -r_b*sind(a); 0];
38                  obj.CPP = [-r_p*cosd(a); -r_p*sind(a); 0];
39                  obj.PBPP = R_BC * obj.CPP + BC - obj.BPB; % [xpp, ...
                        ypp, 0]
40
41                  e_1 = -2 * obj.PBPP(2) * s_a;
42                  e_2 = -2 * obj.PBPP(1) * s_a;
43                  e_3 = (obj.PBPP(1))^2 + (obj.PBPP(2))^2 + s_a^2 - L^2;
44
45                  t_1 = ( -e_1 + sqrt(e_1^2 + e_2^2 - e_3^2) ) / (e_3 ...
                        - e_2);
46                  t_2 = ( -e_1 - sqrt(e_1^2 + e_2^2 - e_3^2) ) / (e_3 ...
                        - e_2);
47
48                  obj.T = [R_BC(1,1), R_BC(1,2), R_BC(1,3), X_c;
49                           R_BC(2,1), R_BC(2,2), R_BC(2,3), Y_c;
50                           R_BC(3,1), R_BC(3,2), R_BC(3,3), 0;
51                           0,         0,         0,         1];
52
53                  if t_1 == real(t_1)
54                      obj.theta(1) = 2*atand(t_1);
55                      obj.theta(2) = 2*atand(t_2);
56
57                      c_psi = ( obj.PBPP(1) - s_a*cosd(obj.theta(1)) ) ...
                            / L;
58                      s_psi = ( obj.PBPP(2) - s_a*sind(obj.theta(1)) ) ...
                            / L;
59
60                      obj.psi = atan2d(s_psi,c_psi);
61                  else
62                      obj.theta(1) = NaN;
63                      obj.theta(2) = NaN;
```

```
64                      obj.psi = NaN;
65                  end
66              end
67          end
68          function theta = getTheta(obj)
69              theta = obj.theta;
70          end
71          function psi = getPsi(obj)
72              psi = obj.psi;
73          end
74          function points = getJointCo(obj,s_a,L)
75              points = [[obj.PB(1), obj.PB(1) + ...
                    s_a*cosd(obj.theta(1)), obj.PB(1) + ...
                    s_a*cosd(obj.theta(1)) + L*cosd(obj.psi)]; ...
                    [obj.PB(2), obj.PB(2) + s_a*sind(obj.theta(1)), ...
                    obj.PB(2) + s_a*sind(obj.theta(1)) + L*sind(obj.psi)]];
76              obj.points = points;
77          end
78      end
79  end
```