

Student ID:		Lab Section:	
Name:		Lab Group:	

Software Lab 3

Forward Kinematics in ROS

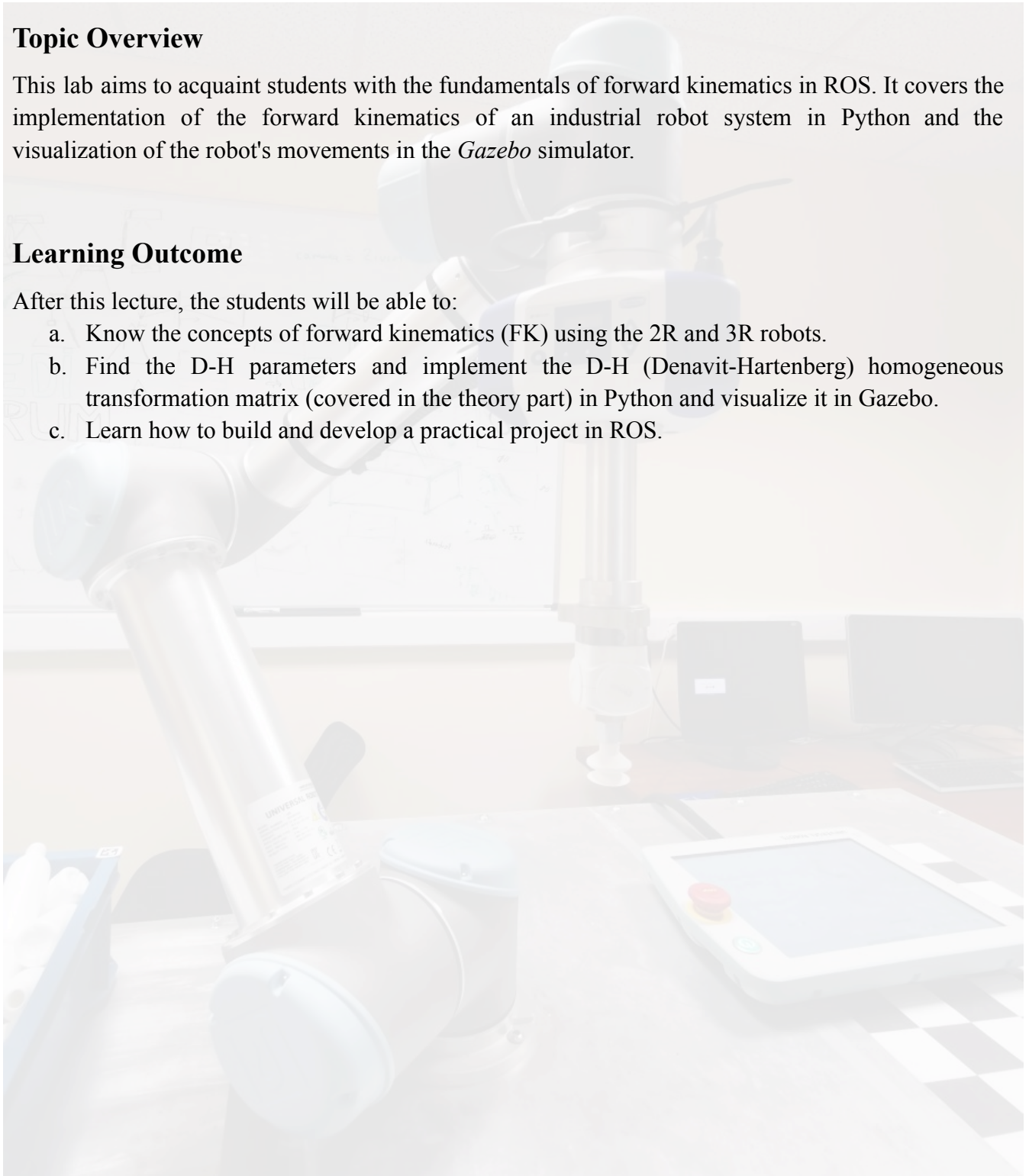
Topic Overview

This lab aims to acquaint students with the fundamentals of forward kinematics in ROS. It covers the implementation of the forward kinematics of an industrial robot system in Python and the visualization of the robot's movements in the *Gazebo* simulator.

Learning Outcome

After this lecture, the students will be able to:

- Know the concepts of forward kinematics (FK) using the 2R and 3R robots.
- Find the D-H parameters and implement the D-H (Denavit-Hartenberg) homogeneous transformation matrix (covered in the theory part) in Python and visualize it in Gazebo.
- Learn how to build and develop a practical project in ROS.



3.1 Forward Kinematics in Robotics

3.1.1 Forward Kinematics

Robot kinematics refers to the study of the motion of robot manipulators. It focuses on understanding and describing the geometric relationships between the different components of a robot, such as the links and joints, and how these relationships determine the robot's overall motion.

Forward kinematics is one of the primary focuses of Robot kinematics. Forward kinematics deals with determining the position (x, y, z) and orientation of the robot's end effector based on the joint angles $(\theta_1, \theta_2, \theta_3, \dots, \theta_n)$ for revolute joints or joint displacements (d) for prismatic joints. This analysis is essential for controlling and planning the robot's movements, as it allows us to know the exact location of the end effector given the joint configurations.

3.1.2 Forward Kinematics of a 2R Robot Using Simple Trigonometric Identity

For ease of understanding, we first look into the kinematics of a 2R robot. A **2R robot** is a two linked robot having two parallel revolute joints and operates in a plane.

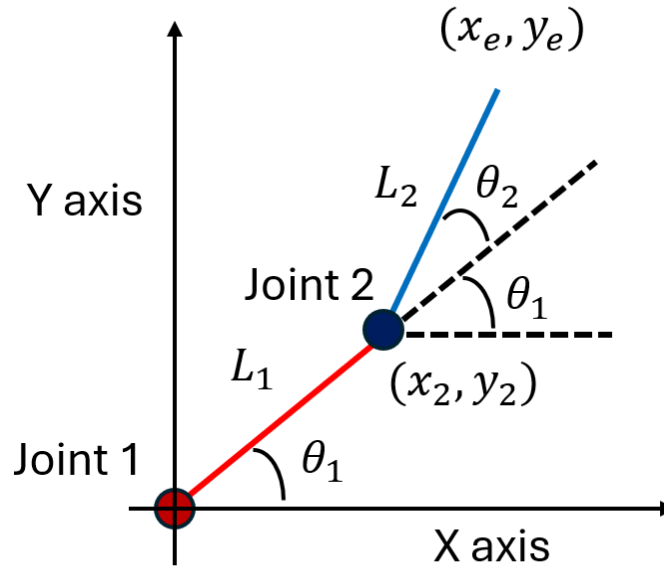


Figure 1: Structure of a 2R robot

The end point of the robot has coordinates (x_e, y_e) . As a roboticist, we are only capable of controlling the joint angles θ_1 and θ_2 . After setting the joints to a certain value, we wish to know where the end effector of the robot is at.

First, let us define the position of joint 2 as (x_2, y_2) in terms of the origin. Using simple trigonometric identity, we can find the position of joint 2 as:

$$x_2 = L_1 \cos \theta_1$$

$$y_2 = L_1 \sin \theta_1$$

Now we calculate the displacement of the end effector from the position of joint-2 as:

$$\Delta x = L_2 \cos(\theta_1 + \theta_2)$$

$$\Delta y = L_2 \sin(\theta_1 + \theta_2)$$

Finally, we can calculate the end effector position in terms of the base frame i.e. the origin. This is found as:

$$x_e = x_2 + \Delta x = L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2)$$

$$y_e = y_2 + \Delta y = L_1 \sin\theta_1 + L_2 \sin(\theta_1 + \theta_2)$$

These equations directly calculate the coordinates of the end effector w.r.t the base frame.

Simply finding the end effector position (x, y, z) in the 3D space is not sufficient most of the time. In case of the 2R robot, we can also find the end effector orientation θ_e (only one angle is sufficient since the robot cannot rotate/twist along the x or y axes)

$$\theta_e = \theta_1 + \theta_2$$

Forward kinematics of a planar 2R robot

$$x_2 = L_1 \cos\theta_1$$

$$y_2 = L_1 \sin\theta_1$$

$$x_e = L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2)$$

$$y_e = L_1 \sin\theta_1 + L_2 \sin(\theta_1 + \theta_2)$$

$$\theta_e = \theta_1 + \theta_2$$

3.1.3 Simulation of a 2R robot

We will now simulate the motion of a planar 2R robot using python and the forward kinematics model. Open a terminal window and run the python file: “kinematics_2R.py” (see Appendix) using the following command:

```
ubuntu@ubuntu2004:~$ python kinematics/kinematics_2R.py
```

We assume that the python file is kept under the folder named “kinematics”.

For the current simulation, the revolute joints θ_1 and θ_2 are defined as functions of time such that

$$\theta_1(t) = \sin(2\pi t/10)$$

$$\theta_2(t) = t$$

After running the simulation, can you explain the motion of the robot in terms of the joint angles?

Task 3.1.3.a: Modify two lines of the code to set $\theta_1(t) = t$ and $\theta_2(t) = 0$. Can you explain the motion of the robot?

Task 3.1.3.b: Now we wish to determine the end effector position for specific joint angles.

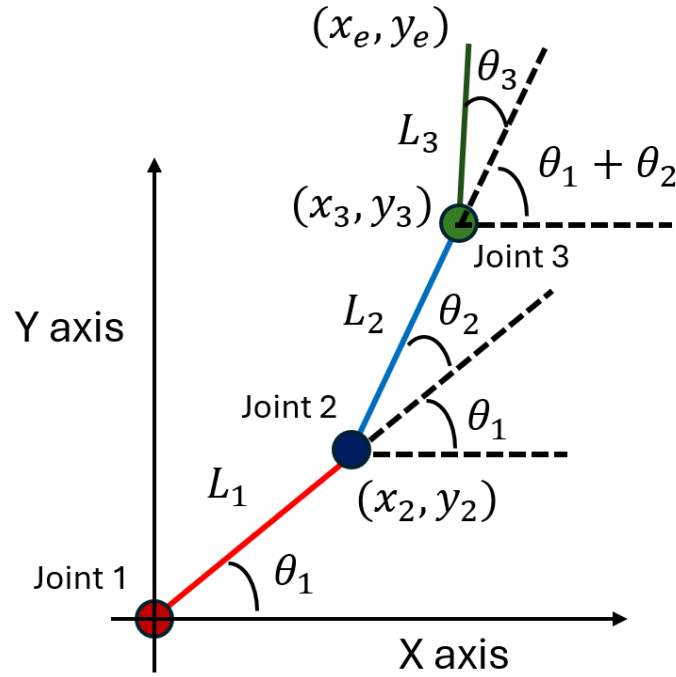
Uncomment the print statement inside the *update()* function to see the end effector position at each time step. Now using the following values for θ_1 and θ_2 , fill up the table containing the end effector positions.

Table 1: End effector positions of a 2R robot

θ_1	θ_2	x_e	y_e
$\pi/4$	$-\pi/4$		
$3\pi/4$	$\pi/4$		
$-3\pi/4$	$\pi/4$		
$\pi/2$	$-\pi/2$		
$-\pi/2$	$\pi/6$		

3.1.4 Forward Kinematics of a 3R Robot Using Simple Trigonometric Identity

The 3R robot is a simple extension of the 2R robot with one additional joint and link.

**Figure 2:** Structure of a planar 3R robot

The end point of the robot has coordinates (x_e, y_e) in the joint-1 frame. We define the position of joint 2 w.r.t the origin as (x_2, y_2) , the displacement of joint 3 from the position of joint 2 as $(\Delta x_2, \Delta y_2)$ and the displacement of the end effector from the position of joint 3 as $(\Delta x_3, \Delta y_3)$.

Using the same calculations as before, we can find each of these terms:

$$\begin{aligned}
 x_2 &= L_1 \cos \theta_1 \\
 y_2 &= L_1 \sin \theta_1 \\
 \Delta x_2 &= L_2 \cos(\theta_1 + \theta_2) \\
 \Delta y_2 &= L_2 \sin(\theta_1 + \theta_2) \\
 \Delta x_3 &= L_3 \cos(\theta_1 + \theta_2 + \theta_3)
 \end{aligned}$$

$$\Delta y_3 = L_3 \sin(\theta_1 + \theta_2 + \theta_3)$$

Finally, we can calculate the pose of the end effector w.r.t the origin as:

Forward kinematics of a planar 3R robot

$$\begin{aligned}x_2 &= L_1 \cos\theta_1 \\y_2 &= L_1 \sin\theta_1 \\x_3 &= L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2) \\y_3 &= L_1 \sin\theta_1 + L_2 \sin(\theta_1 + \theta_2) \\x_e &= L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2) + L_3 \cos(\theta_1 + \theta_2 + \theta_3) \\y_e &= L_1 \sin\theta_1 + L_2 \sin(\theta_1 + \theta_2) + L_3 \sin(\theta_1 + \theta_2 + \theta_3) \\\theta_e &= \theta_1 + \theta_2 + \theta_3\end{aligned}$$

3.1.5 Simulation of a 3R robot

Task 3.1.5.a: We will now simulate the motion of a planar 3R robot using python and the forward kinematics model. But first, we need to implement the kinematics of the different joints and end effectors of the robot. Namely, we need to complete the following part of the code “kinematics_3R.py”:

```
##### TODO #####
# Compute the position of joint 2
x2 = 0
y2 = 0

# Compute the position of joint 3
x3 = 0
y3 = 0

# Compute the position of end effector
xe = 0
ye = 0

#####
```

Upon completion, open a terminal window and run the python file: “kinematics_3R.py” using the following command:

```
ubuntu@ubuntu2004:~$ python kinematics/kinematics_3R.py
```

We assume that the python file is kept under the folder named “kinematics”.

After running the simulation, can you explain the motion of the robot in terms of the joint angles?

Task 3.1.5.b: Now we wish to determine the end effector position for specific joint angles.

Uncomment the **print** statement inside the **update()** function to see the end effector position at each time step. Now using the following values for $\theta_1, \theta_2, \theta_3$ - fill up the table containing the end effector positions.

Table 2: End effector positions of a 3R robot

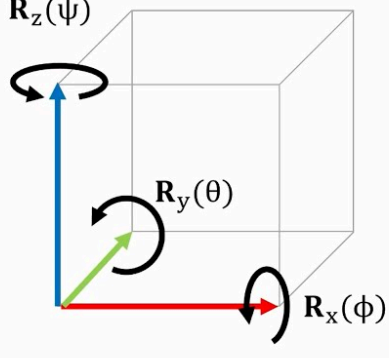
θ_1	θ_2	θ_3	x_e	y_e
$\pi/4$	$-\pi/4$	$\pi/3$		
$3\pi/4$	$\pi/4$	$-\pi/6$		
$-3\pi/4$	$\pi/4$	$\pi/2$		
$\pi/2$	$-\pi/2$	$-\pi/2$		
$-\pi/2$	$\pi/6$	$\pi/12$		

3.2 Homogeneous Transformations

Homogeneous transformations are the easiest linear transformation to switch between different coordinate frames in a kinematic system. Homogeneous transformation matrices have two key components: **rotational** and **translational** parts. As we develop robots with higher degrees of freedoms and complex structures, forward kinematics calculation using plain trigonometry becomes cumbersome, and this is where homogeneous transformations shine.

3.2.1 Rotation Matrix

In the 3D euclidean coordinate system, rotation matrices can transform one coordinate system to the other following a rotation operation along any axis. Rotation along x, y and z axis of own frame are denoted by the matrices R_x , R_y , R_z respectively.

$$\begin{aligned} R_x(\phi) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix} \\ R_y(\theta) &= \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \\ R_z(\psi) &= \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$


3.2.2 Translation Matrix

When a coordinate system is translated along x, y and/or z axis without changing orientation, the transformation to the new coordinate system is found as:

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} d_x + p_x \\ d_y + p_y \\ d_z + p_z \\ 1 \end{bmatrix}$$

Assuming the point ${}^B P = [p_x, p_y, p_z]^T$ is in frame B, and the displacement of frame B w.r.t frame A is given by $[d_x, d_y, d_z]^T$, then we can find the position of point V w.r.t frame A as:

$${}^A P = {}^A T_B * {}^B P$$

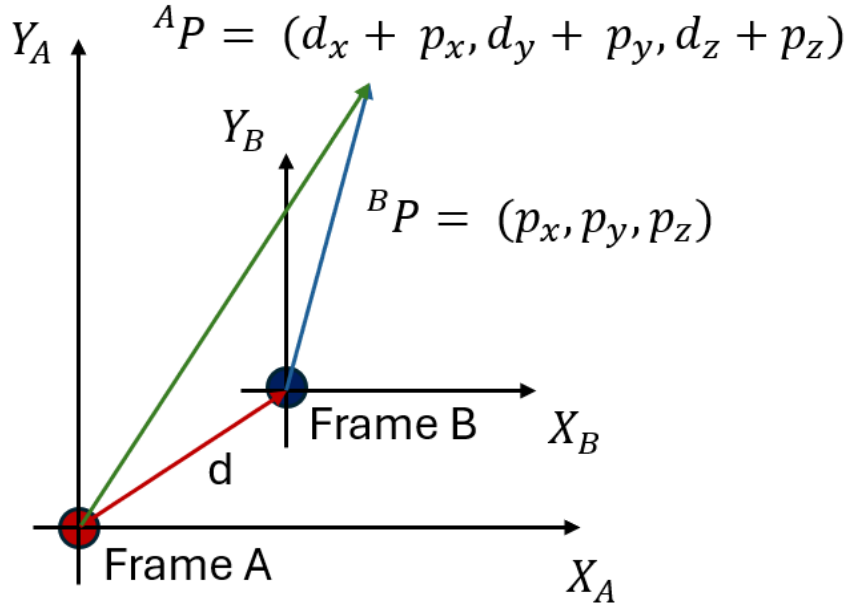


Figure: Coordinate transform for translation operation

Where ${}^A T_B$ is the transform from frame B to frame A. Note that the translation matrix requires appending a '1' to the bottom of the position vector. This is only required for the linear algebraic calculations and is removed afterwards.

3.2.3 Homogeneous Transformation Matrix

Suppose we have two different coordinate frames A and B. The coordinate transform from frame A to frame B is given by the homogeneous transformation matrix ${}^A T_B$. Suppose we can find the coordinate frame B by doing a translation d along the axes of coordinate frame A, getting the new coordinate frame A', then doing a rotation R along the z-axis of temporary frame A' to get the frame B. Suppose we have a point P defined in coordinate frame B as ${}^B P$

Now, the translation transformation from A to A' frame is given as:

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The rotation from A' frame to B is given as:

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{21} & R_{22} & R_{23} & 0 \\ R_{31} & R_{32} & R_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Thus, the homogeneous transformation from frame A to frame B will be found as:

$${}^A T_B = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{21} & R_{22} & R_{23} & 0 \\ R_{31} & R_{32} & R_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & d_x \\ R_{21} & R_{22} & R_{23} & d_y \\ R_{31} & R_{32} & R_{33} & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Finally, for any point ${}^B P$ given in frame B, we can find the coordinate of point P in frame A:

$${}^A P = {}^A T_B * {}^B P$$

Example:

Suppose we have a point P located in (2, 2, 0) position in frame B. The transformation of frame B from frame A is given by a translation of (3, 2, 0) along (x, y, z) axes of frame A, followed by a rotation of 90 degrees along the z-axis. We wish to find the position of point P in terms of frame A.

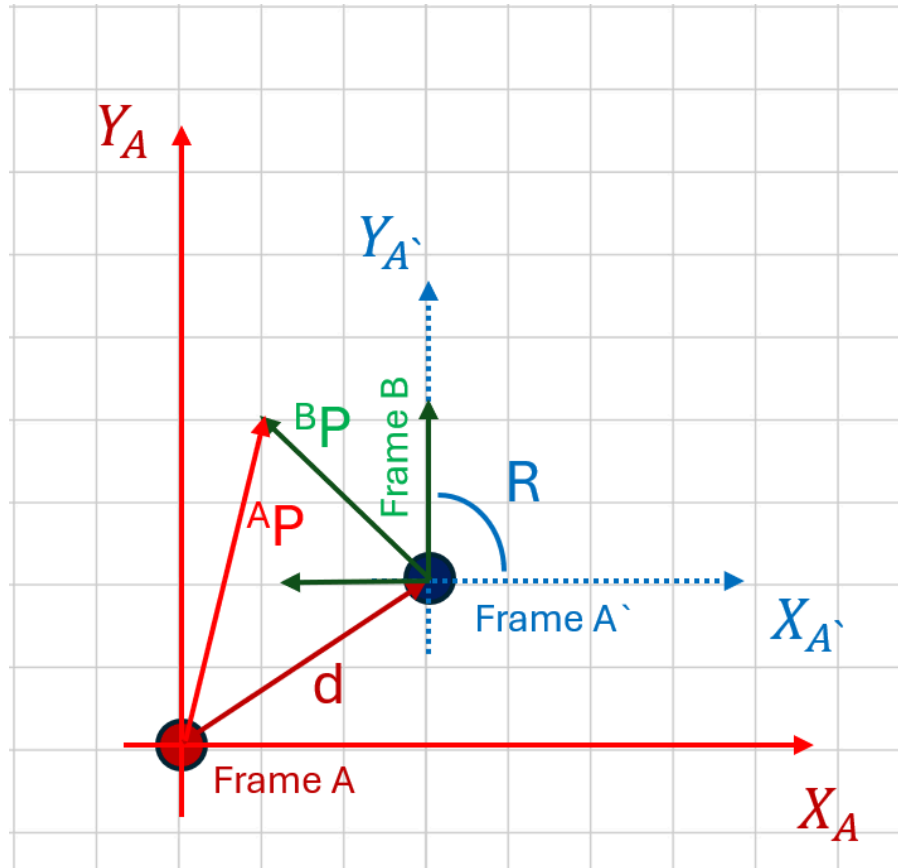


Fig: Translation of (3, 2, 0) followed by rotation of 90 degrees along z axis

We construct the homogeneous transformation matrix step by step:

First, the translation transformation:

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Followed by a rotation of 90 degrees along z axis:

$$\begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Thus, the transform ${}^A T_B$ from frame A to frame B:

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & d_x \\ \sin \theta & \cos \theta & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 & 3 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Finally, we can find the location of point P = [2, 2, 0]^T in coordinate frame A:

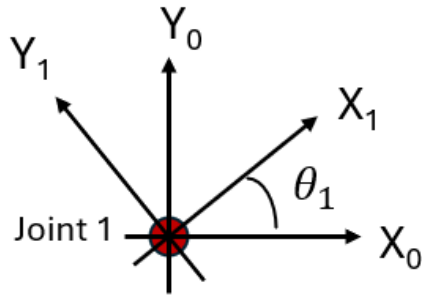
$${}^A P = {}^A T_B X {}^B P = \begin{bmatrix} 0 & -1 & 0 & 3 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ 0 \\ 1 \end{bmatrix}$$

Note: The (x, y, z) coordinates should have 3 values, thus the column vector of a coordinate should be of dimension 3x1. But we are multiplying a vector of size 4x1. This is done by augmenting the 3x1 position vector with an extra 1 in the last position which helps add the translation parameters following the rotation operation. After the resulting vector of size 4x1 is found, we simply discard the 1 in the last position and convert the position vector back to 3x1. Meaning the final result will be [1, 4, 0]^T

3.2.4 Homogeneous Transformation of a Planar 3R robot

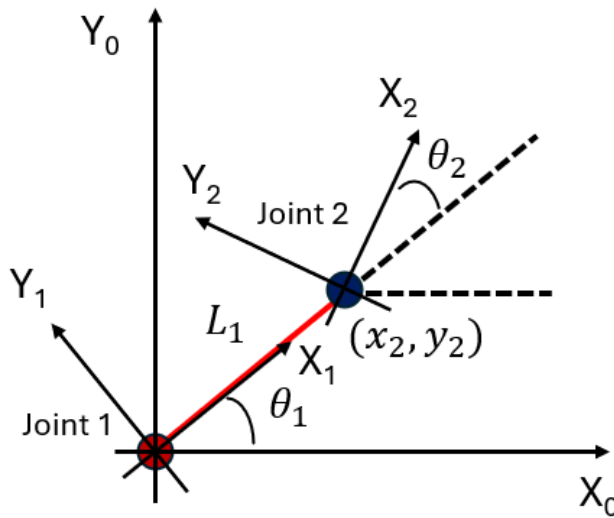
Recall the 3R robot introduced in section 3.1.4 - we wish to derive the forward kinematics of the end point using homogeneous transformations.

Joint-1 adds a rotation of θ_1 along z axis, without any translation



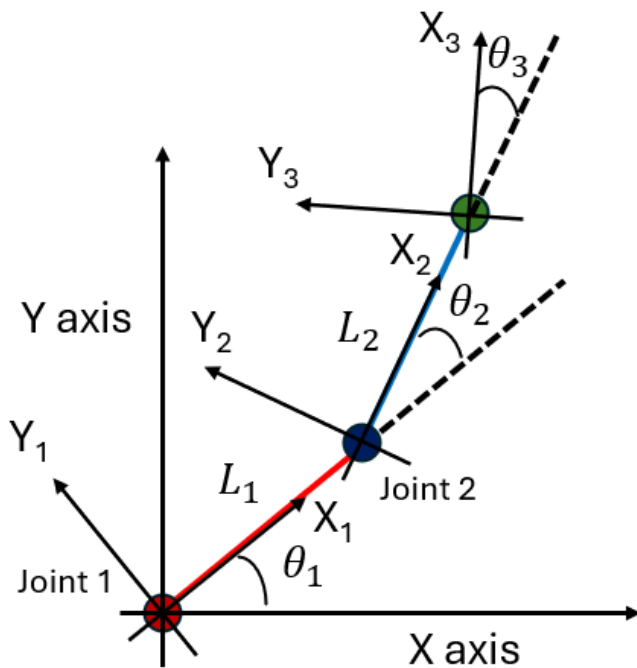
$${}^0T_1 = \begin{bmatrix} \cos \theta_1 & -\sin \theta_1 & 0 & 0 \\ \sin \theta_1 & \cos \theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Joint-2 is placed at a link length L_1 along x axis, followed by a rotation of θ_2 along the z axis.



$${}^1T_2 = \begin{bmatrix} \cos \theta_2 & -\sin \theta_2 & 0 & L_1 \\ \sin \theta_2 & \cos \theta_2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Joint-3 is placed at a link length L_2 along x axis, followed by a rotation of θ_3 along the z axis.



$${}^2T_3 = \begin{bmatrix} \cos \theta_3 & -\sin \theta_3 & 0 & L_2 \\ \sin \theta_3 & \cos \theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Finally, end effector is placed at a distance L_3 along the x axis of coordinate frame-3

$${}^3P = \begin{bmatrix} L_3 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

When the transformation matrices are applied consecutively, we can find the forward kinematics of the end effector:

$${}^0P = {}^0T_1 \times {}^1T_2 \times {}^2T_3 \times {}^3P$$

$$= \begin{bmatrix} \cos \theta_1 & -\sin \theta_1 & 0 & 0 \\ \sin \theta_1 & \cos \theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta_2 & -\sin \theta_2 & 0 & L_1 \\ \sin \theta_2 & \cos \theta_2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta_3 & -\sin \theta_3 & 0 & L_2 \\ \sin \theta_3 & \cos \theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} L_3 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} L_1 \cos(\theta_1) + L_2 \cos(\theta_1 + \theta_2) + L_3 \cos(\theta_1 + \theta_2 + \theta_3) \\ L_1 \sin(\theta_1) + L_2 \sin(\theta_1 + \theta_2) + L_3 \sin(\theta_1 + \theta_2 + \theta_3) \\ 0 \\ 1 \end{bmatrix}$$

The advantage of using homogeneous transforms is that it allows us to find the coordinates of links and joints using efficient matrix operations.

3.2.5 Simulation of a 3R robot with homogeneous transforms

Task 3.2.5.a: We will now simulate the motion of a planar 3R robot using python and the forward kinematics model. But first, we need to implement the kinematics of the different joints and end effectors of the robot by finding the homogeneous transforms. Namely, we need to complete the following part of the code “kinematics_3R_homogeneous.py”:

```
##### TODO #####

# Joint 1 transform
T01 = np.array([[1, 0, 0, 0],
                [0, 1, 0, 0],
                [0, 0, 1, 0],
                [0, 0, 0, 1]])

# Joint 2 transform
T12 = np.array([[1, 0, 0, 0],
                [0, 1, 0, 0],
                [0, 0, 1, 0],
                [0, 0, 0, 1]])
```

```

# Joint 3 transform
T23 = np.array([[1, 0, 0, 0],
                [0, 1, 0, 0],
                [0, 0, 1, 0],
                [0, 0, 0, 1]])

# End effector position in joint-3 frame
P3 = np.array([0, 0, 0, 1]).T

#####

```

Upon completion, open a terminal window and run the python file: “kinematics_3R.py” using the following command:

```
ubuntu@ubuntu2004:~$ python kinematics/kinematics_3R_homogeneous.py
```

We assume that the python file is kept under the folder named “kinematics”.

After running the simulation, can you explain the motion of the robot in terms of the joint angles?

Task 3.2.5.b: Now we wish to observe the transformation matrices for the different joints from the origin, namely we wish to observe 0T_1 , 0T_2 , 0T_3 . Edit the **print** statement inside the **update()** function to observe the matrices T01, T02, T03. Now using the following values for $\theta_1, \theta_2, \theta_3$ - fill up the table containing the end effector positions. **From the last columns of the three matrices, can you determine the joint positions for joint-1, joint-2 and joint-3?**

Table 3: Joint transforms of a 3R robot

θ_1	θ_2	θ_3	0T_1	0T_2	0T_3
$\pi/4$	$-\pi/4$	$\pi/3$			
$3\pi/4$	$\pi/4$	$-\pi/6$			

3.3 Denavit-Hartenberg (D-H) Convention

A commonly used convention for selecting frames of reference in robotic applications is the **Denavit-Hartenberg** or **D-H convention**.

The D-H parameters are a widely used convention in robotics to describe the kinematic properties and geometry of robot manipulators. They provide a systematic way to represent the relationships between consecutive links and joints in a robot manipulator. The D-H parameters consist of four parameters associated with each link-joint connection.

The **link offset** (d) represents the distance between the z-axes of consecutive coordinate frames along the common normal. The **joint angle** (θ) denotes the angle between the x-axes of consecutive coordinate frames about the common normal. It represents the rotation about the z-axis to align the x-axis of the current frame with the x-axis of the next frame. The **link length** (a) represents the distance between the origins of the two consecutive frames along the common x-axis. Finally, the **link twist** (α) denotes the angle between the z-axes of consecutive coordinate frames about the common x-axis.

Using the D-H (Denavit-Hartenberg) homogeneous transformation matrix, we can solve the forward kinematics problem of a robot with its D-H parameters.

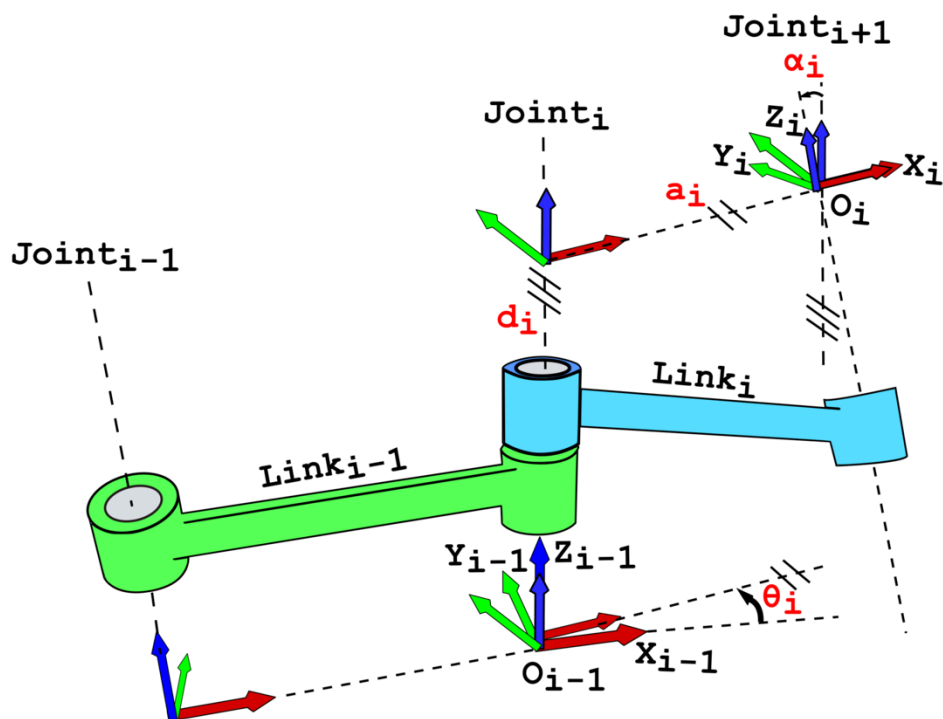
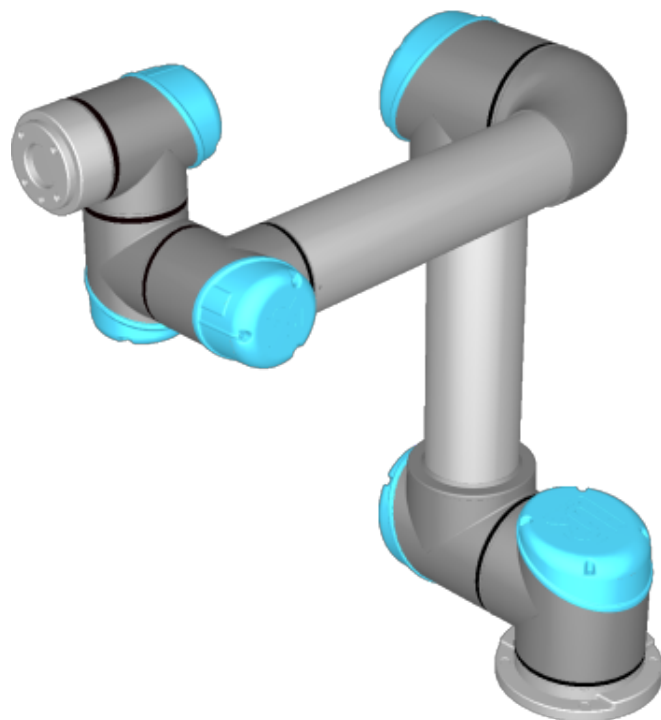
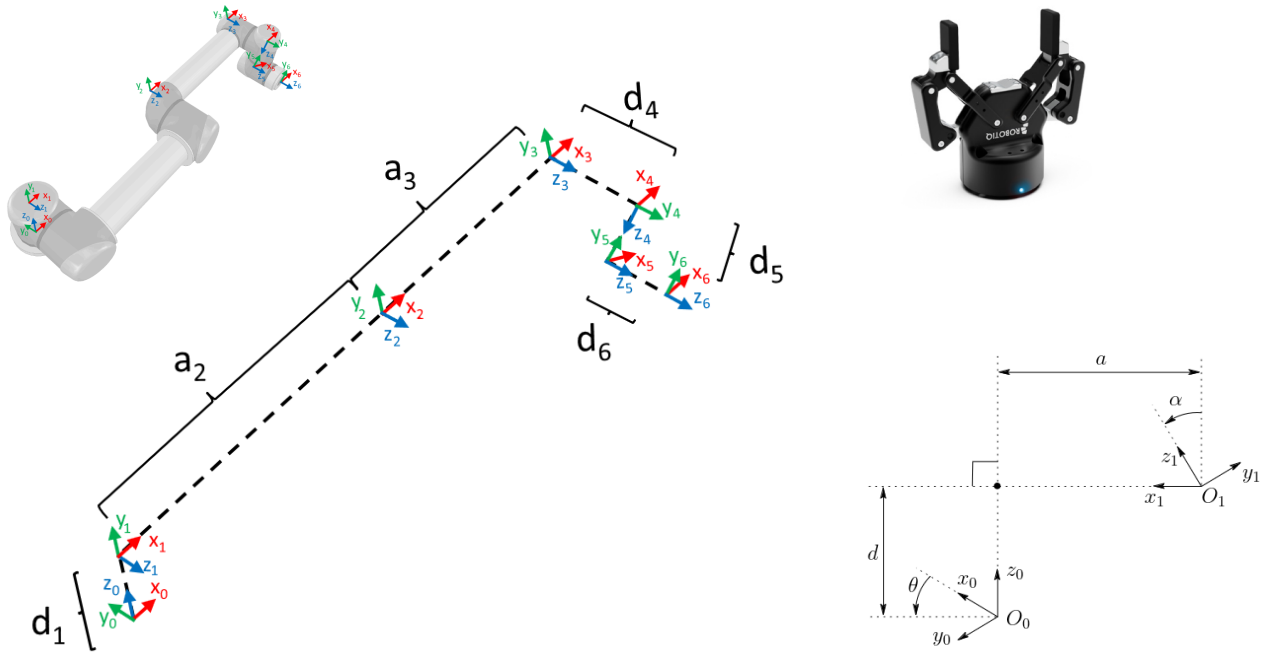


Fig: Different components of the DH-convention (source: [Wikipedia](https://en.wikipedia.org/wiki/Denavit-Hartenberg_parameters))

3.4 Forward Kinematics of the Industrial Robot



3.4.1 D-H Parameters and D-H Homogeneous Transformation Matrix for the robot

We implement the forward kinematics of an industrial robot system in this lab. The D-H parameters for this robot arm with a gripper are given in the following. Find out the α (**radians**) parameters from the figure above ($a = r$). The gripper attached to the arm adds **16.28 cm** to the end of the arm.

Table 4. D-H Parameters for the Arm-Gripper System

Joint	α (radians)	a (meters)	d (meters)	θ (radians)
Joint 1	?	0	0.089159	θ_1
Joint 2	?	-0.425	0	θ_2
Joint 3	?	-0.39225	0	θ_3
Joint 4	?	0	0.10915	θ_4
Joint 5	?	0	0.09465	θ_5
Joint 6	?	0	0.0903 + ?	θ_6

The D-H homogeneous matrix for the joint i is given in the following ($c = \cos$, $s = \sin$).

$${}^{i-1}T_i = \begin{bmatrix} c\theta_i & -c\alpha_i s\theta_i & s\alpha_i s\theta_i & a_i c\theta_i \\ s\theta_i & c\alpha_i c\theta_i & -s\alpha_i c\theta_i & a_i s\theta_i \\ 0 & s\alpha_i & c\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Multiplying the transformation matrix from the $(i - 1)^{th}$ to the i^{th} frame $[{}^{i-1}T_i]$ with the pose of the $(i + 1)^{th}$ joint in the i^{th} coordinate frame gives us the pose of the $(i + 1)^{th}$ joint in the $(i - 1)^{th}$ coordinate frame. Using linear algebra, we can consecutively multiply the transformation matrix for each corresponding joint to find the pose of the end effector in the base frame. This allows us to calculate the end effector position from the revolute joint angles.

Finally, ${}^0T_6 = {}^0T_1 {}^1T_2 {}^2T_3 {}^3T_4 {}^4T_5 {}^5T_6$

3.4.2 Python Implementation of FK

The workspace of this lab is located in the folder [~/lab3_catkin_ws/](#)

The forward kinematics of the robot is implemented in the file:

[~/lab3_catkin_ws/src/motion_planning/scripts/kinematics.py](#)

The D-H parameters are to be specified in the following code block:

```
# TODO: 1. Define robot arm parameters
d    = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
a    = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
alph = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

Then, the D-H matrix has to be implemented here. The variable `th` contains all the θ_i 's in radians.

```
def DH(i, th):
    # TODO: 2. Return the DH matrix for joint i
    return None
```


Finally, return the end effector position in terms of the six joint angles by multiplying the transformation matrices. The matrix multiplication operator is @ in numpy.

```
def forward(th):
    A_1 = DH(1, th)
    A_2 = DH(2, th)
    A_3 = DH(3, th)
    A_4 = DH(4, th)
    A_5 = DH(5, th)
    A_6 = DH(6, th)

    T_06 = None # TODO: 3. Return the final FK matrix

    return T_06
```

3.4.3 Building the Project

We move to the Lab 3 workspace and initialize ROS. Then, we build the workspace using catkin.

```
ubuntu@ubuntu2004:~$ cd ~/lab3_catkin_ws/
ubuntu@ubuntu2004:~/lab3_catkin_ws$ source /opt/ros/noetic/setup.bash
ubuntu@ubuntu2004:~/lab3_catkin_ws$ catkin build

-----
Profile:                                default
Extending:                             [env] /opt/ros/noetic
Workspace:                             /home/ubuntu/lab3_catkin_ws
-----
Build Space:                           [exists] /home/ubuntu/lab3_catkin_ws/build
Devel Space:                           [exists] /home/ubuntu/lab3_catkin_ws/devel
Install Space:                         [unused] /home/ubuntu/lab3_catkin_ws/install
Log Space:                             [missing] /home/ubuntu/lab3_catkin_ws/logs
Source Space:                          [exists] /home/ubuntu/lab3_catkin_ws/src
DESTDIR:                               [unused] None
-----
...

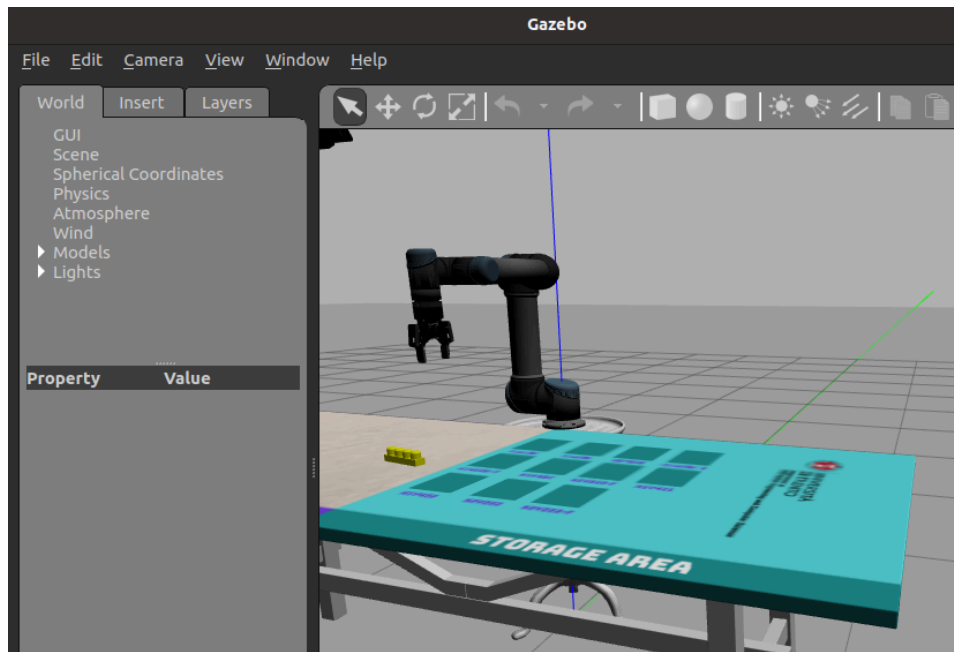
[build] Summary: All 21 packages succeeded!
[build] Ignored: None.
[build] Warnings: 4 packages succeeded with warnings.
[build] Abandoned: None.
[build] Failed: None.
[build] Runtime: 56.3 seconds total.
[build] Note: Workspace packages have changed, please re-source setup files to use them.
```

Then, we register our workspace which makes sure that we can conveniently use all the packages.

```
ubuntu@ubuntu2004:~/lab3_catkin_ws$ source $PWD/devel/setup.bash
ubuntu@ubuntu2004:~/lab3_catkin_ws$ echo "source $PWD/devel/setup.bash" >> $HOME/.bashrc
```

Next, the lego world environment is launched in Gazebo. This will take some time.

```
ubuntu@ubuntu2004:~/lab3_catkin_ws$ roslaunch bricks lego_world.launch
```



To add a lego brick run:

```
ubuntu@ubuntu2004:~/lab3_catkin_ws$ rosrun bricks add_bricks.py
Added 1 bricks
All done. Ready to start.
```

To initialize the kinematics node, we first install `pyquaternion` and then run the node.

```
ubuntu@ubuntu2004:~/lab3_catkin_ws$ pip3 install pyquaternion
ubuntu@ubuntu2004:~/lab3_catkin_ws$ rosrun motion_planning motion_planning.py
-----
Initializing node of kinematics
```

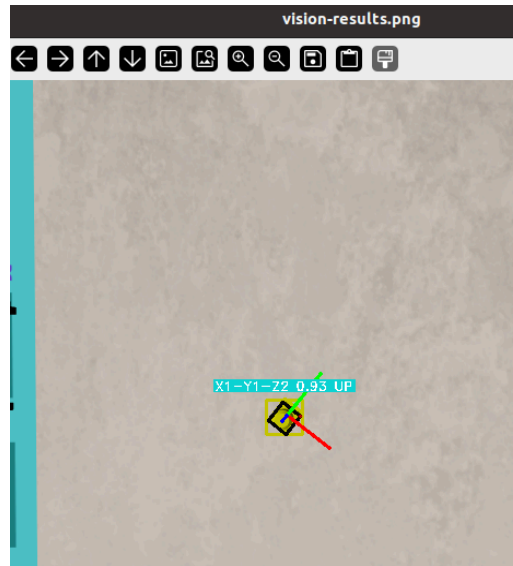
Now, we run the vision node to find the brick and communicate with the kinematics node for the motion planning. Vision node uses the YOLOv5 model for processing the camera images.

```
ubuntu@ubuntu2004:~/lab3_catkin_ws$ rosrun vision vision.py -show
Loading model best.pt
YOLOv5 🚀 v7.0-295-gac6c4383 Python-3.8.10 torch-2.2.2+cu121 CPU
...
```

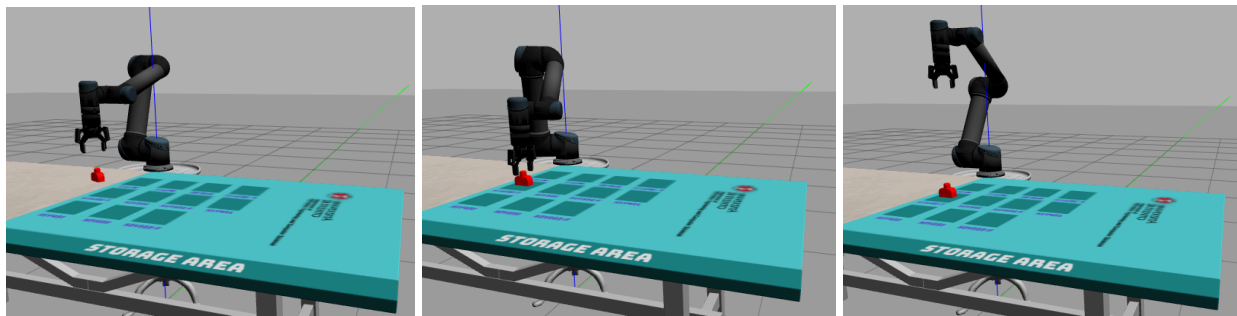
We have to unpause physics to start the full system using a Gazebo service.

```
ubuntu@ubuntu2004:~/lab3_catkin_ws$ rosservice call gazebo/unpause_physics
```

At this point, the vision node will receive images and show the output of the object detection.



To start the kinematics, we need to close this window above. After closing the window, the vision node sends necessary information for motion planning. If the forward kinematic is implemented correctly we will observe that the robot is picking up the lego brick and putting it in the appropriate storage box.



Submission

1. kinematics_3R.py, kinematics_3R_homogeneous.py Python files; Table 2 and Table 3 in the report.
2. The DH parameter full table (Table 4) for the robot arm and gripper system in the report.
3. ~/lab3_catkin_ws/src/motion_planning/scripts/kinematics.py Python file.
4. Build commands and screenshots of working arm-gripper system in the report.
5. Answer the following questions:
 - a. What kinds of joints are joints 1, 2, 3, 4, 5, 6 of the arm-gripper system?
 - b. Draw the tentative node-topic communication graph for the system.

Resources

1. <https://app.theconstruct.ai/Desktop> Free virtual machine online hosted by The Construct. They also have an extensive list of tutorials and courses on ROS. <https://app.theconstruct.ai/courses/>
2. ROS official tutorials: <https://wiki.ros.org/ROS/Tutorials>
3. Clearpath Robotics: <http://www.clearpathrobotics.com/assets/guides/noetic/ros/index.html>
4. Homogeneous Transformation: <https://aleksandarhaber.com/homogeneous-transformation-rotation-and-translation/>
5. DH-convention: https://en.wikipedia.org/wiki/Denavit%E2%80%93Hartenberg_parameters

Appendix

kinematics_2R.py

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# Define the link lengths
L1, L2 = 1, 1

# Define the simulation time
t_final = 10          # Total simulation time
t_step = 0.01         # Simulation time step

time_series = np.arange(0, t_final, t_step)

# Create a figure and axes
fig, ax = plt.subplots()

# Create a line object for the first link
link1, = ax.plot([0, L1], [0, 0], 'r-')

# Create a line object for the second link
link2, = ax.plot([L1, L1+L2], [0, 0], 'b-')

# Simulating the motion of the 2R robot
def update(i):

    t = time_series[i]

    # Expression for theta1 and theta2 as functions of time
    theta1 = np.sin(2*np.pi*t/10)
    theta2 = t

    # Compute the position of joint 2
    x2 = L1 * np.cos(theta1)
    y2 = L1 * np.sin(theta1)

    # Compute the position of end effector
    xe = L1 * np.cos(theta1) + L2 * np.cos(theta1 + theta2)
    ye = L1 * np.sin(theta1) + L2 * np.sin(theta1 + theta2)

    # print(xe, ye)

    # update the plot
```

```

    link1.set_data([0, x2], [0, y2])
    link2.set_data([x2, xe], [y2, ye])
    return link1, link2

# Create an animation
ani = FuncAnimation(fig, update, frames=range(len(time_series)),
                    interval=t_step*1000, blit=True)

# Set the axes limits
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)

# Make sure the axes are of equal size
ax.set_aspect('equal')

# Show the plot
plt.show()

```

kinematics_3R.py

```

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# Define the link lengths
L1, L2, L3 = 1, 0.8, 0.6

# Define the simulation time
t_final = 10          # total simulation time
t_step = 0.01         # simulation time step

time_series = np.arange(0, t_final, t_step)

# Create a figure and axes
fig, ax = plt.subplots()

link1, = ax.plot([0, L1], [0, 0], 'r-')
link2, = ax.plot([L1, L1+L2], [0, 0], 'b-')
link3, = ax.plot([L1+L2, L1+L2+L3], [0, 0], 'g-')

# Simulating the motion of the 2R robot
def update(i):
    t = time_series[i]

```

```

# Expression for theta1 and theta2 as functions of time
theta1 = np.sin(2*np.pi*t/10)
theta2 = np.cos(2*np.pi*t/5)
theta3 = 2*np.sin(2*np.pi*t/2)

##### TODO #####
# Compute the position of joint 2
x2 = 0
y2 = 0

# Compute the position of joint 3
x3 = 0
y3 = 0

# Compute the position of end effector
xe = 0
ye = 0

#####
# print(xe, ye)

# update the plot
link1.set_data([0, x2], [0, y2])
link2.set_data([x2, x3], [y2, y3])
link3.set_data([x3, xe], [y3, ye])
return link1, link2, link3

# Create an animation
ani = FuncAnimation(fig, update, frames=range(len(time_series)),
                    interval=t_step*1000, blit=True)

# Set the axes limits
ax.set_xlim(-3, 3)
ax.set_ylim(-3, 3)

# Make sure the axes are of equal size
ax.set_aspect('equal')

# Show the plot
plt.show()

```

kinematics_3R_homogeneous.py

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# Define the link lengths
L1, L2, L3 = 1, 0.8, 0.6

# Define the simulation time

t_final = 10          # total simulation time
t_step = 0.01         # simulation time step

time_series = np.arange(0, t_final, t_step)

# Create a figure and axes
fig, ax = plt.subplots()

link1, = ax.plot([0, L1], [0, 0], 'r-')
link2, = ax.plot([L1, L1+L2], [0, 0], 'b-')
link3, = ax.plot([L1+L2, L1+L2+L3], [0, 0], 'g-')

# Simulating the motion of the 2R robot
def update(i):

    t = time_series[i]

    # Expression for theta1 and theta2 as functions of time
    theta1 = np.sin(2*np.pi*t/10)
    theta2 = np.cos(2*np.pi*t/5)
    theta3 = 2*np.sin(2*np.pi*t/2)

    ##### TODO #####

    # Joint 1 transform
    T01 = np.array([[1, 0, 0, 0],
                    [0, 1, 0, 0],
                    [0, 0, 1, 0],
                    [0, 0, 0, 1]])

    # Joint 2 transform
    T12 = np.array([[1, 0, 0, 0],
                    [0, 1, 0, 0],
                    [0, 0, 1, 0],
                    [0, 0, 0, 1]])
```

```

# Joint 3 transform
T23 = np.array([[1, 0, 0, 0],
                [0, 1, 0, 0],
                [0, 0, 1, 0],
                [0, 0, 0, 1]])

# End effector position in joint-3 frame
P3 = np.array([0, 0, 0, 1]).T

#####

T02 = np.matmul(T01, T12)
T03 = np.matmul(T02, T23)
P0 = np.matmul(T03, P3)

x2, y2 = T02[0, 3], T02[1, 3] # Joint 2 position
x3, y3 = T03[0, 3], T03[1, 3] # Joint 3 position
xe, ye = P0[0], P0[1] # end effector position

print(T02)

# update the plot
link1.set_data([0, x2], [0, y2])
link2.set_data([x2, x3], [y2, y3])
link3.set_data([x3, xe], [y3, ye])
return link1, link2, link3

# Create an animation
ani = FuncAnimation(fig, update, frames=range(len(time_series)),
interval=t_step*1000, blit=True)

# Set the axes limits
ax.set_xlim(-3, 3)
ax.set_ylim(-3, 3)

# Make sure the axes are of equal size
ax.set_aspect('equal')

# Show the plot
plt.show()

```