Tinkerers' Lab
IIT Hyderabad

Mechatronics Summer School 2025

Prepared by:

The Senior Mechatronics Team

Abhijith Raj, *Senior Core*
Satyavada Sri Harini, *Senior Core*
Taha Mohiuddin, *Senior Core*

Tinkerers' Lab, IIT Hyderabad
Between A, B, G, H Hostel Block, IIT Hyderabad
IITH Main Road, Near NH-65, Sangareddy, Telangana 502285

Tinkerers' Lab
IIT Hyderabad

.

Module 4
Robot Kinematics

Contents

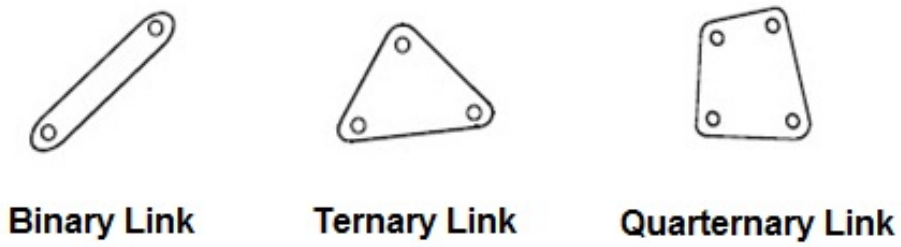
1 The Basics

1.1 Links

A **link** is a solid, inflexible component that connects with other components through joints. It serves as the basic building block of any mechanism. You can think of a link as a bone in the human body—rigid, connected to others via joints, and capable of transmitting forces and motion.

Links can vary based on how many joints they connect to:

- **Binary link**: connects two joints.
- **Ternary link**: connects three joints.
- **Quaternary link**: connects four joints.



In a robotic arm, for example, the straight segments between two rotating joints are links. Each link holds a specific position and orientation, which, when combined with joint movements, determines the overall behavior of the mechanism.

1.2 Joints

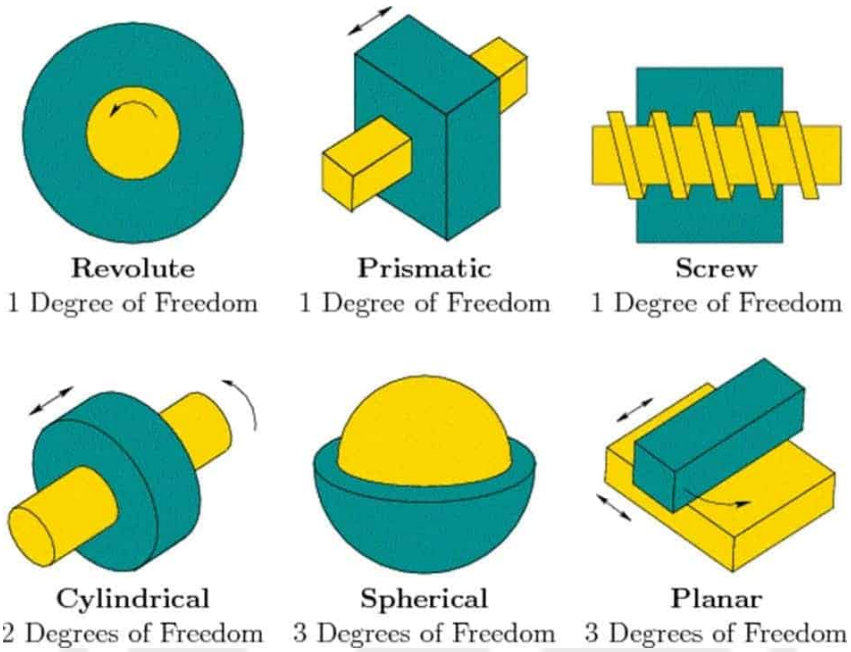
A **joint** (also called a **kinematic pair**) connects two links and allows relative motion between them. Joints determine the type and number of movements possible within a mechanism, which is critical to how a robot or machine functions.

Types of joints include:

- **Revolute Joint (R Joint)**: Allows rotational motion around a fixed axis, similar to a door hinge.
- **Prismatic Joint (P Joint)**: Allows linear or sliding motion (e.g., piston in a cylinder).
- **Helical Joint**: Produces screw-like motion, combining rotation and translation.
- **Cylindrical Joint**: Allows both rotation and linear sliding along the same axis.

- **Spherical Joint**: Allows rotation in all three directions (like a human shoulder joint).

Each joint imposes specific constraints on the motion of links and thus affects the movement of the overall system.



1.3 Degree of Freedom (DOF) Calculation

The **Degree of Freedom (DOF)** of a mechanical system indicates how many independent parameters are needed to fully define its configuration. In simpler terms, it tells us how many ways the system can move.

For **planar mechanisms**, Kutzbach's Equation is used:

$$\text{DOF} = 3(n-1) - 2j - h$$

Where:

- n = total number of links (including the ground),
- j = number of lower pair joints connecting two links (e.g., revolute or prismatic, each with 1 DOF),
- h = number of higher pair joints (e.g., cams, gears, each with 2 DOF).

For **spatial mechanisms** (3D):

$$\text{DOF} = 6(n-1) - \sum f_i$$

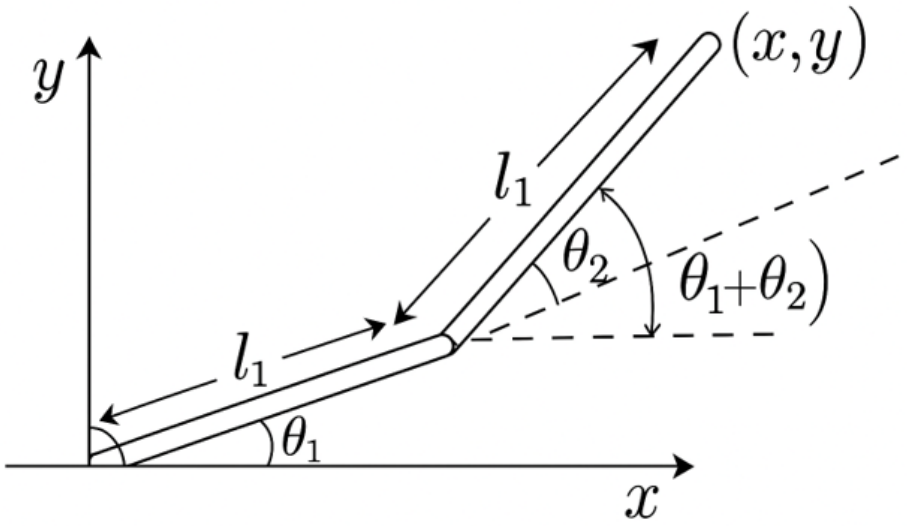
Where f_i is the number of constraints imposed by the i^{th} joint (e.g., a revolute joint imposes 5 constraints, allowing 1 DOF).

2 Forward Kinematics

Forward Kinematics (FK) is the method used to determine the position and orientation of the end-effector (e.g., the hand of a robotic arm) given the values of its joint parameters.

FK takes joint angles (for revolute joints) or displacements (for prismatic joints) as inputs and outputs the pose (position and orientation) of the end-effector.

FK is fundamental in robot control, motion planning, and simulation tasks.



Let:

- l_1: Length of the first link
- l_2: Length of the second link
- θ_1: Joint angle at the base
- θ_2: Joint angle between the first and second links
- (x, y): Cartesian position of the end-effector

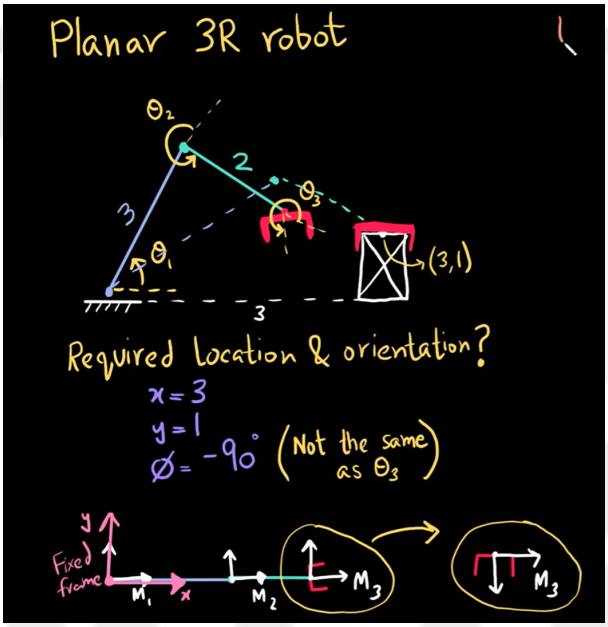
From the diagram:

$$x = l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2)$$

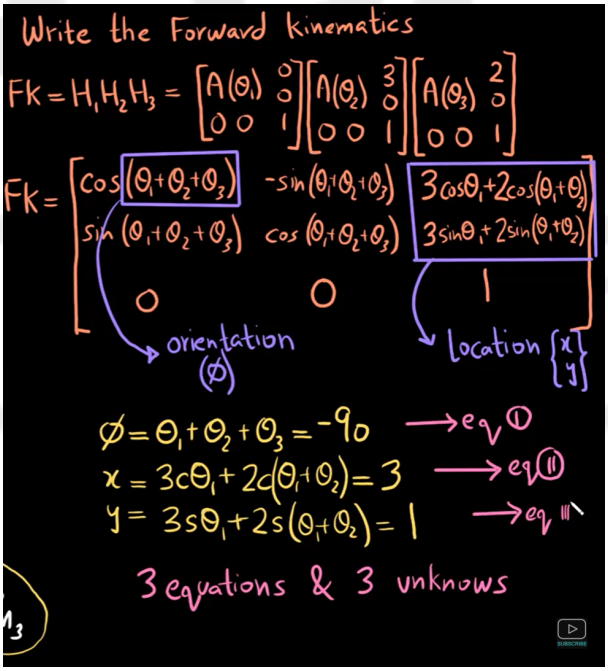
$$y = l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2)$$

3 Inverse Kinematics

Inverse Kinematics as the name suggests is the opposite of forward i.e here we know the coordinates of the end effector and are supposed to find the respective joint angles and their combinations to get the required end effector location. So for example lets take a planar 3 DOF (3 revolute joint) robotic arm.

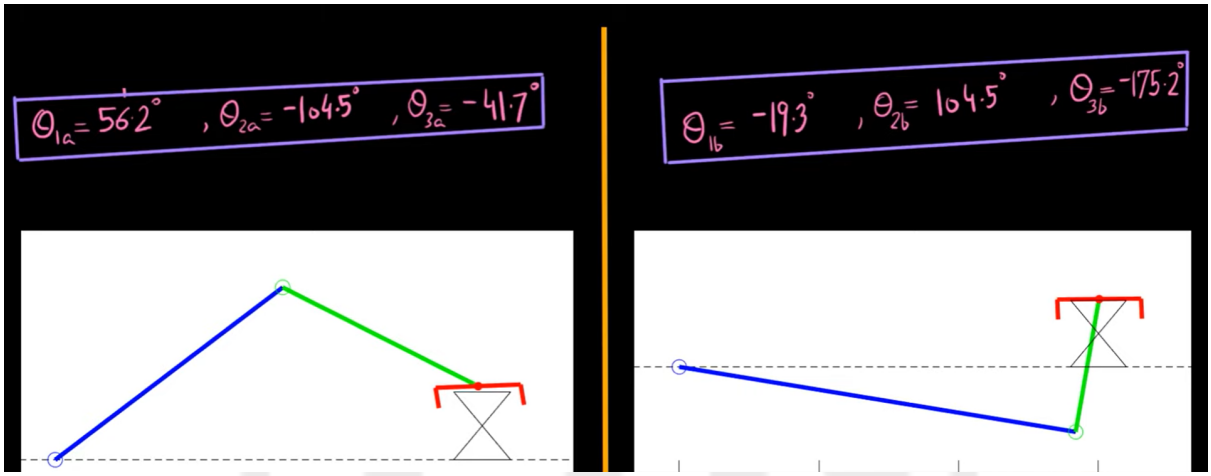


Each the joint has its own frame of reference and angle with respect to the previous link in ACW direction. Therefore the angle/orientation of end effector w.r.t the horizontal is not θ_3 in this notation.



Once you have the frames labelled you can go ahead and do the homogeneous trans-

forms like in forward kinematics. From the resulting matrix you get equations for both the orientation and location of the end effector solving which (it can also be done manually) gives you solutions for the required state.



As it can be seen from the solution for the above case all the solutions need not be possible physically and hence must be taken care of by keeping in mind the allowable Range of Motion (ROM) of each joint.

4 Simulation of a Mechanism:-

4.1 The Loop Closure Method

Now for a 4-bar mechanism define 4 position vectors along each link to get a loop. (These vectors can be written for both cartesian and complex coordinate system). The result of this will be an equation as follows:

Vector Loop for A Fourbar

Define four position vectors to obtain a loop (closed chain). ▶

For notational simplification, rename the vectors. ▶

The four vectors form a vector loop equation:

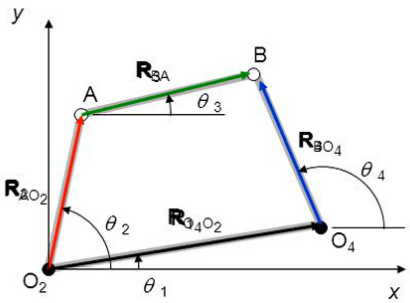
$$\mathbf{R}_2 + \mathbf{R}_3 = \mathbf{R}_1 + \mathbf{R}_4 \quad (1)$$

Rearranged the equation as

$$\mathbf{R}_2 + \mathbf{R}_3 - \mathbf{R}_4 - \mathbf{R}_1 = \mathbf{0} \quad (2)$$

Define a Cartesian *x*-*y* reference frame at a convenient position and orientation. ▶

Define angles for the vectors according to "our convention". ▶



Now the vector loop equation can be transformed from vector form to algebraic form.

The equation can then be seperated into 2 different ones each for the respective directional components (X & Y) as they are independent.

Algebraic Form of Vector Loop Equation

The vector loop equation for the fourbar in its vector form is expressed as:

$$\mathbf{R}_2 + \mathbf{R}_3 - \mathbf{R}_4 - \mathbf{R}_1 = \mathbf{0} \qquad (2)$$

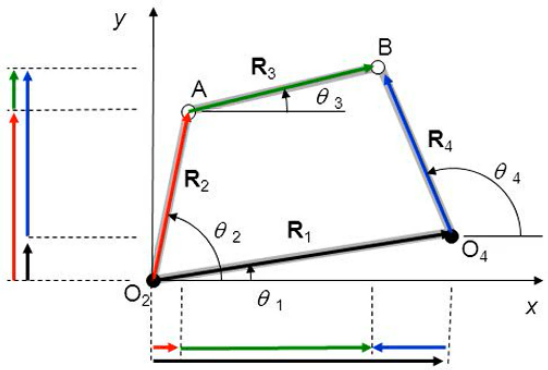
This equation is projected onto the *x*-axis to obtain one algebraic equation as:

$$R_2 \cos\theta_2 + R_3 \cos\theta_3 - R_4 \cos\theta_4 - R_1 \cos\theta_1 = 0 \qquad \text{(3-x)}$$

The projection of Eq. (2) onto the *y*-axis yields:

$$R_2 \sin\theta_2 + R_3 \sin\theta_3 - R_4 \sin\theta_4 - R_1 \sin\theta_1 = 0 \qquad \text{(3-y)}$$

Note that the *plus* and *minus* signs in the algebraic equations (3-x) and (3-y) follow the signs in the vector loop equation (2).



Then assuming the link 1 is fixed and link 2 is your input, $\theta_1 and \theta_2$ are known. Your unknowns would be the rest of the link orientation angles (if the lengths are known). Solving these equations gives you the different states of the mechanism for different crank angles.

The algebraic position equations for a fourbar are:

$$R_2 \cos\theta_2 + R_3 \cos\theta_3 - R_4 \cos\theta_4 - R_1 \cos\theta_1 = 0 \qquad \text{(3-x)}$$

$$R_2 \sin\theta_2 + R_3 \sin\theta_3 - R_4 \sin\theta_4 - R_1 \sin\theta_1 = 0 \qquad \text{(3-y)}$$

These equations are called "position equations" or "position constraints". The equations contain the following constants:

$$R_1 , R_2 , R_3 , R_4 , \theta_1$$

The equations contain the following variables:

$$\theta_2 , \theta_3 , \theta_4$$

For a specific fourbar, the constants have known (given) values.

If a value is assigned to one of the variables (for example, the angle of the input link), the two equations can be solved for the other two variables.

The position equations are *nonlinear* in the coordinates (angles).

Differentiating the Positional Analysis equations gives you the velocity and acceleration equations.

$$- R_2 \sin\theta_2\omega_2 - R_3 \sin\theta_3\omega_3 + R_4 \sin\theta_4\omega_4 = 0$$

$$R_2 \cos\theta_2\omega_2 + R_3 \cos\theta_3\omega_3 - R_4 \cos\theta_4\omega_4 = 0$$

$$- R_2 \sin\theta_2\alpha_2 - R_3 \sin\theta_3\alpha_3 + R_4 \sin\theta_4\alpha_4 = R_2 \cos\theta_2\omega_2^2 + R_3 \cos\theta_3\omega_3^2 - R_4 \cos\theta_4\omega_4^2$$

$$R_2 \cos\theta_2\alpha_2 + R_3 \cos\theta_3\alpha_3 - R_4 \cos\theta_4\alpha_4 = R_2 \sin\theta_2\omega_2^2 + R_3 \sin\theta_3\omega_3^2 - R_4 \sin\theta_4\omega_4^2$$

4.2 Python Simulation

An example code for simulation of a 4bar mechanism by solving the loop closure equations using f-solve has been attached. Use that as a reference for the task and it also contains velocity analysis

```
    import numpy as np
import matplotlib.pyplot as plt

# Defaults
N = 100

# These link lengths ensure we have a crank-rocker
l1, l2, l3, l4 = 7, 3, 6, 8
alpha = np.pi / 4
a = 1.2

# Initializations
theta_3_and_4 = np.array([np.pi, np.pi / 2])
theta2 = np.linspace(0, 4 * np.pi, N)
theta3 = np.zeros(N)
theta4 = np.zeros(N)

# Function defining loop-closure equations
def loop_closure_4bar(x, theta2):
    F1 = l1 + l4 * np.cos(x[1]) + l3 * np.cos(x[0]) - l2 * np.cos(theta2)
    F2 = l4 * np.sin(x[1]) + l3 * np.sin(x[0]) - l2 * np.sin(theta2)
    return np.array([F1, F2])

# Compute Jacobian matrix
def jacobian(x):
    J = np.zeros((2, 2))
    J[0, 0] = -l3 * np.sin(x[0])  # dF1/d(theta3)
    J[0, 1] = -l4 * np.sin(x[1])  # dF1/d(theta4)
    J[1, 0] = l3 * np.cos(x[0])   # dF2/d(theta3)
    J[1, 1] = l4 * np.cos(x[1])   # dF2/d(theta4)
    return J

# Newton-Raphson iteration
def newton_raphson(theta2, initial_guess, tol=1e-9, max_iter=50):
    x = initial_guess
    for _ in range(max_iter):
        F = loop_closure_4bar(x, theta2)
```

```
        J = jacobian(x)
        dx = np.linalg.solve(J, -F)  # Solve J * dx = -F
        x += dx  # Update solution
        if np.linalg.norm(dx) < tol:
            break
    return x

# Solve for each theta2 using Newton-Raphson
for i in range(N):
    theta_3_and_4 = newton_raphson(theta2[i], theta_3_and_4)
    theta3[i], theta4[i] = theta_3_and_4

# Coupler curve coordinates
coupler_curve_x_coord = l1 + l4 * np.cos(theta4) + a * l3 * np.cos(theta3 - alpha)
coupler_curve_y_coord = l4 * np.sin(theta4) + a * l3 * np.sin(theta3 - alpha)

# Plotting the motion of the mechanism and coupler curve
plt.figure(figsize=(10, 6))

for i in range(N):
    x_coord = [0, l1, l1 + l4 * np.cos(theta4[i]), l2 * np.cos(theta2[i]), 0]
    y_coord = [0, 0, l4 * np.sin(theta4[i]), l2 * np.sin(theta2[i]), 0]
    P_x_coord = [
        l1 + l4 * np.cos(theta4[i]),
        l1 + l4 * np.cos(theta4[i]) + a * l3 * np.cos(theta3[i] - alpha),
        l2 * np.cos(theta2[i]),
    ]
    P_y_coord = [
        l4 * np.sin(theta4[i]),
        l4 * np.sin(theta4[i]) + a * l3 * np.sin(theta3[i] - alpha),
        l2 * np.sin(theta2[i]),
    ]

    plt.plot(x_coord, y_coord, 'b--')
    plt.plot(P_x_coord, P_y_coord, 'b-')
    plt.plot(coupler_curve_x_coord[:i], coupler_curve_y_coord[:i], 'r-')
    plt.plot(x_coord, y_coord, 'ro')
    plt.axis([-l1 - l4, 2 * l1 + l4, -l1 - l4, l1 + l4])
    plt.grid(True)
    plt.title("Crank-rocker", fontsize=18, fontname="Palatino")
    plt.xlabel(r'$x$', fontsize=14, fontname="Palatino")
    plt.ylabel(r'$y$', fontsize=14, fontname="Palatino")
    plt.pause(0.05)
    plt.cla()  # Clear the current axis for the next frame

plt.show()
```

VELOCITY ANALYSIS:-

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve

# Defaults
N = 100

# These link lengths ensure we have a crank-rocker
l1, l2, l3, l4 = 7, 3, 6, 8
alpha = np.pi / 4
a = 1.2

# Initializations
theta_3_and_4 = [np.pi, np.pi / 2]
theta2 = np.linspace(0, 4 * np.pi, N)
```

```
theta3 = np.zeros_like(theta2)
theta4 = np.zeros_like(theta2)

# Function defining loop-closure equations
def loop_closure_4bar(x, l1, l2, l3, l4, theta2):
    F1 = l1 + l4 * np.cos(x[1]) + l3 * np.cos(x[0]) - l2 * np.cos(theta2)
    F2 = l4 * np.sin(x[1]) + l3 * np.sin(x[0]) - l2 * np.sin(theta2)
    return [F1, F2]

# Solving nonlinear equations for position
for i in range(N):
    xsol = fsolve(loop_closure_4bar, theta_3_and_4, args=(l1, l2, l3, l4, theta2[i]))
    theta3[i] = xsol[0]
    theta4[i] = xsol[1]
    theta_3_and_4 = [theta3[i], theta4[i]]

# Velocity Analysis
omega_3_and_4 = [0, 0]
omega2 = np.full(N, 0.2)
omega3 = np.zeros_like(omega2)
omega4 = np.zeros_like(omega2)

def loop_closure_4bar_velocity(x, l2, l3, l4, theta2, theta3, theta4, omega2):
    F1 = l4 * x[1] * np.sin(theta4) + l3 * x[0] * np.sin(theta3) - l2 * omega2 * np.sin(theta2
    )
    F2 = l4 * x[1] * np.cos(theta4) + l3 * x[0] * np.cos(theta3) - l2 * omega2 * np.cos(theta2
    )
    return [F1, F2]

# Solving nonlinear equations for velocity
for i in range(N):
    xsol1 = fsolve(loop_closure_4bar_velocity, omega_3_and_4, args=(l2, l3, l4, theta2[i],
    theta3[i], theta4[i], omega2[i]))
    omega3[i] = xsol1[0]
    omega4[i] = xsol1[1]
    omega_3_and_4 = [omega3[i], omega4[i]]

# Coupler curve coordinates
coupler_curve_x_coord = l1 + l4 * np.cos(theta4) + a * l3 * np.cos(theta3 - alpha)
coupler_curve_y_coord = l4 * np.sin(theta4) + a * l3 * np.sin(theta3 - alpha)

# Plot angular velocities
plt.figure(figsize=(10, 5))
plt.plot(theta2, omega2, label=r'$\omega_2$', linewidth=1)
plt.plot(theta2, omega3, label=r'$\omega_3$', linewidth=1)
plt.plot(theta2, omega4, label=r'$\omega_4$', linewidth=1)
plt.grid(True)
plt.legend()
plt.title("Angular Velocities")
plt.xlabel(r'$\theta_2$')
plt.ylabel(r'$\omega$')
plt.show()
```

5 DIY Tasks

The system consists of a robot with four legs, where each leg is made up of two connected segments joined by two revolute joints, allowing for two degrees of freedom. Each joint is driven by a motor that enables precise movement of the segments. The layout of the legs, including the segment lengths and initial joint angles, is illustrated in the diagrams below.

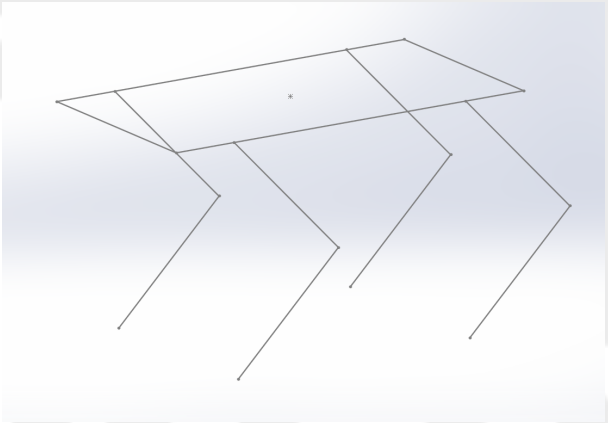


Figure 1: View 1

5.1 Task 1

Implement a function for inverse kinematics on Python for this mechanism. Make sure to define the coordinate system of your choice appropriately.

5.2 Task 2

Using the function you developed for the previous task, make the end effector traverse a circle with the initial position of the end effector as the center, and a radius and time

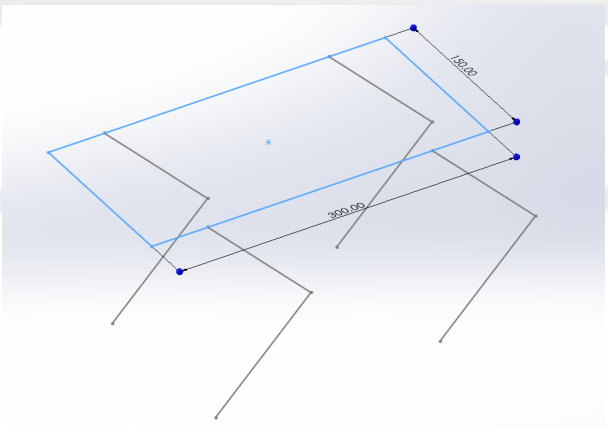


Figure 2:

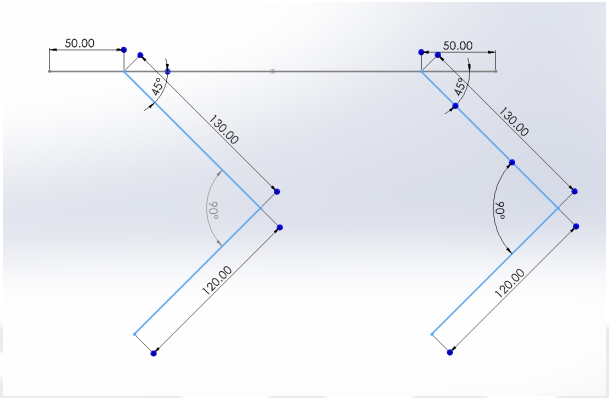


Figure 3:

period of your choice. Then, plot the angular position, velocity and acceleration of the motors with time.

5.3 Task 3

Simulate the trot gait, and plot the angular position, velocity and acceleration of the motors against time.

You may use any library in Python to display the motion.