Fundamentals of Robotics: Mini Project 1

Robot Frame Figure

DH Table

θ	d	a	α
θ_1	L_1	0	-90
$\theta_2 - 90$	0	L_2	180
θ_3	0	L_3	180
$\theta_4 + 90$	0	0	90
θ_5	$L_4 + L_5$	0	0

The $\theta_2 - 90$ and $\theta_4 + 90$ expressions were added to account for initial arm position.

Forward Kinematics Equations

The forward kinematics of the 5-DOF robot arm is computed using the homogeneous transformation matrices derived from the DH parameters. The transformation from one frame to the next is given by:

$$T_i^{i+1} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i \cos\alpha_i & \sin\theta_i \sin\alpha_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\theta_i \cos\alpha_i & -\cos\theta_i \sin\alpha_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The full transformation from the base to the end-effector is obtained by multiplying the individual transformations:

$$T_0^5 = T_0^1 T_1^2 T_2^3 T_3^4 T_4^5$$

where each T_i^{i+1} is derived using the corresponding DH parameters.

Using matrix multiplication, the final position and orientation of the end-effector relative to the base frame can be computed from T_0^5:

$$T_0^5 = \begin{bmatrix} R & d \\ 0 & 1 \end{bmatrix}$$

where R represents the rotation matrix and P represents the position vector of the end-effector.

Jacobian Matrix Derivation

To determine the linear velocity Jacobian J_v for the 5-DOF robot arm, we use the standard Jacobian formulation:

$$J_v = \begin{bmatrix} \frac{\partial x}{\partial \theta_1} & \frac{\partial x}{\partial \theta_2} & \cdots & \frac{\partial x}{\partial \theta_5} \\ \frac{\partial y}{\partial \theta_1} & \frac{\partial y}{\partial \theta_2} & \cdots & \frac{\partial y}{\partial \theta_5} \\ \frac{\partial z}{\partial \theta_1} & \frac{\partial z}{\partial \theta_2} & \cdots & \frac{\partial z}{\partial \theta_5} \end{bmatrix}$$

where each column corresponds to a joint and describes how the end-effector's position (x, y, z) changes with respect to the joint angles.

Derivation

For the joints, the linear velocity contribution is given by:

$$J_{v_i} = Z_0^i \times (P_5 - P_i)$$

where: - Z_0^i is the axis of rotation, obtained from the transformation matrix T_0^i by multiplying with:

$$Z_0^i = T_0^i \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

- P_5 is the end-effector position, extracted from T_0^5.
- P_i is the position of joint i, extracted from T_0^i.
- J_v is built by crossing Z_0^i with $P_5 - P_i$ for each joint.

The final Jacobian matrix for the linear velocity of the end-effector is:

$$J_v = \begin{bmatrix} Z_0^1 \times (P_5 - P_1) \\ Z_0^2 \times (P_5 - P_2) \\ \cdots \\ Z_0^5 \times (P_5 - P_5) \end{bmatrix}$$

where each column represents the contribution of a joint to the linear velocity.

Code

Simulation Functions

Forward Position Kinematics

```
def calc_forward_kinematics(self, theta: list, radians=False):
    """
    Calculate forward kinematics based on the provided joint angles.

    Args:
        theta: List of joint angles (in degrees or radians).
        radians: Boolean flag to indicate if input angles are in radians.
    """

    # Want to use radians to control the simulation since all the functions are written to u
    if not radians:
        theta = np.radians(theta)

    # DH Table, derivation shown above
    self.DH = [
        [theta[0], self.l1, 0, -np.pi/2],
        [theta[1] - np.pi/2, 0, self.l2, np.pi],
        [theta[2], 0, self.l3, np.pi],
        [theta[3] + np.pi/2, 0, 0, np.pi/2],
        [theta[4], self.l4 + self.l5, 0, 0],
    ]

    # This vertically stacks the transformation matricies from the DH table
    # self.T represents the transformation required to go from joint i-1 to joint i
    self.T = np.stack(
        [
            dh_to_matrix(self.DH[0]),
            dh_to_matrix(self.DH[1]),
            dh_to_matrix(self.DH[2]),
            dh_to_matrix(self.DH[3]),
            dh_to_matrix(self.DH[4]),
        ],
        axis=0,
    )


    # Calculate robot points (positions of joints)
    self.calc_robot_points()
```

Velocity Kinematics

```
def calc_velocity_kinematics(self, vel: list):
    """
    Calculate the joint velocities required to achieve the given end-effector velocity.

    Args:
```

```
        vel: Desired end-effector velocity (3x1 vector).
    """


    # Computes the total transformation matricies to get from frame 0 to frame i
    T_cumulative = [np.eye(4)]
    for i in range(self.num_dof):
        T_cumulative.append(T_cumulative[-1] @ self.T[i])

    # Extracts the translational component to get from frame 0 to the end-effector frame
    d = T_cumulative[-1] @ np.vstack([0, 0, 0, 1])

    # Calculates the jacobian by crossing the distance to the end-effector and the z compone
    for i in range(0, 5):
        T_i = T_cumulative[i]
        z = T_i @ np.vstack([0, 0, 1, 0])
        d1 = T_i @ np.vstack([0, 0, 0, 1])
        r = np.array([d[0] - d1[0], d[1] - d1[1], d[2] - d1[2]]).flatten()
        self.J[i] = np.cross(z[:3].flatten(), r.flatten())

    # Uses psuedoinverse to calculate inverse of jacobian
    # This is done since the jacobian is not square
    J_inv = np.linalg.pinv(self.J)
    # Multiplies the velocity vector by the inverse jacobian to get angular velocities of e
    theta_dot = np.dot(np.array(vel), J_inv)

    # Control cycle time step
    dt = 0.01
    # Calculates next theta values by multiplying angular velocities by time step
    self.theta = self.theta + (theta_dot * dt)
    # Calls forward kinematics with new theta values
    self.calc_forward_kinematics(self.theta, radians=True)
```

Robot Functions

Velocity Kinematics

```
def set_arm_velocity(self, cmd: ut.GamepadCmds):
    """Calculates and sets new joint angles from linear velocities.

    Args:
        cmd (GamepadCmds): Contains linear velocities for the arm.
    """
    J = np.zeros((5, 3))

    vel = [cmd.arm_vx, cmd.arm_vy, cmd.arm_vz]
```

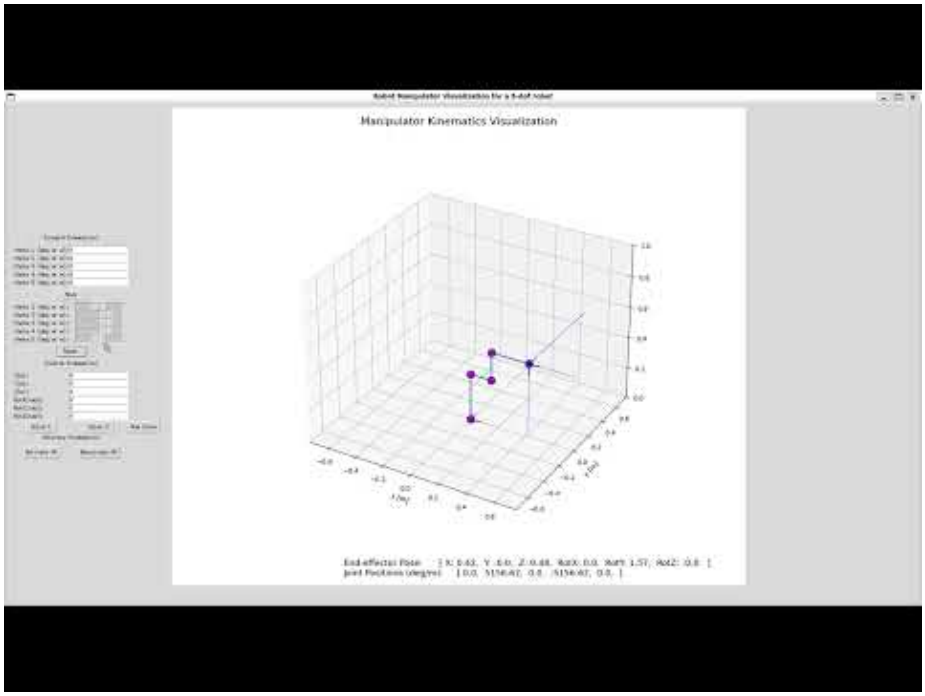
```
# Sets initial theta values to be the current joint angles
# This is needed to accurately calculate next theta values
theta = self.joint_values[:5]

# DH Table derived above
DH = [
    [theta[0], L1, 0, -90],
    [theta[1] - 90, 0, L2, 180],
    [theta[2], 0, L3, 180],
    [theta[3] + 90, 0, 0, 90],
    [theta[4], L4 + L5, 0, 0],
]

# This vertically stacks the transformation matricies from the DH table
# T represents the transformation required to go from joint i-1 to joint i
T = np.stack(
    [
        ut.dh_to_matrix(DH[0]),
        ut.dh_to_matrix(DH[1]),
        ut.dh_to_matrix(DH[2]),
        ut.dh_to_matrix(DH[3]),
        ut.dh_to_matrix(DH[4]),
    ],
    axis=0,
)

# Computes the total transformation matricies to get from frame 0 to frame i
T_cumulative = [np.eye(4)]
for i in range(5):
    T_cumulative.append(T_cumulative[-1] @ T[i])

# Extracts the translational component to get from frame 0 to the end-effector frame
d = T_cumulative[-1] @ np.vstack([0, 0, 0, 1])

# Calculates the jacobian by crossing the distance to the end-effector and the z compone
for i in range(0, 5):
    T_i = T_cumulative[i]
    z = T_i @ np.vstack([0, 0, 1, 0])
    d1 = T_i @ np.vstack([0, 0, 0, 1])
    r = np.array([d[0] - d1[0], d[1] - d1[1], d[2] - d1[2]]).flatten()
    J[i] = np.cross(z[:3].flatten(), r.flatten())

# Uses psuedoinverse to calculate inverse of jacobian
# This is done since the jacobian is not square
J_inv = np.linalg.pinv(J)
```

```
    # Multiplies the velocity vector by the inverse jacobian to get angular velocities of ea
    thetalist_dot = np.dot(np.array(vel), J_inv)
    # Add an extra entry at the end of the array to account for the end-effector opening and
    thetalist_dot = np.append(thetalist_dot, 0.0)

    print(f'[DEBUG] Current thetalist (deg) = {self.joint_values}')
    print(f'[DEBUG] linear vel: {[round(vel[0], 3), round(vel[1], 3), round(vel[2], 3)]}')
    print(f'[DEBUG] thetadot (deg/s) = {[round(td,2) for td in thetalist_dot]}')

    # Update joint angles
    dt = 0.5 # Fixed time step
    K_vel = 1 # mapping gain for velocity control
    K = 10 # mapping gain for individual joint control
    new_thetalist = [0.0]*6

    # linear velocity control
    for i in range(5):
        new_thetalist[i] = self.joint_values[i] + dt * K_vel * thetalist_dot[i]
    # individual joint control
    new_thetalist[0] += dt * K * cmd.arm_j1
    new_thetalist[1] += dt * K * cmd.arm_j2
    new_thetalist[2] += dt * K * cmd.arm_j3
    new_thetalist[3] += dt * K * cmd.arm_j4
    new_thetalist[4] += dt * K * cmd.arm_j5
    new_thetalist[5] = self.joint_values[5] + dt * K * cmd.arm_ee

    new_thetalist = [round(theta,2) for theta in new_thetalist]
    print(f'[DEBUG] Commanded thetalist (deg) = {new_thetalist}')

    # set new joint angles
    self.set_joint_values(new_thetalist, radians=False)
```

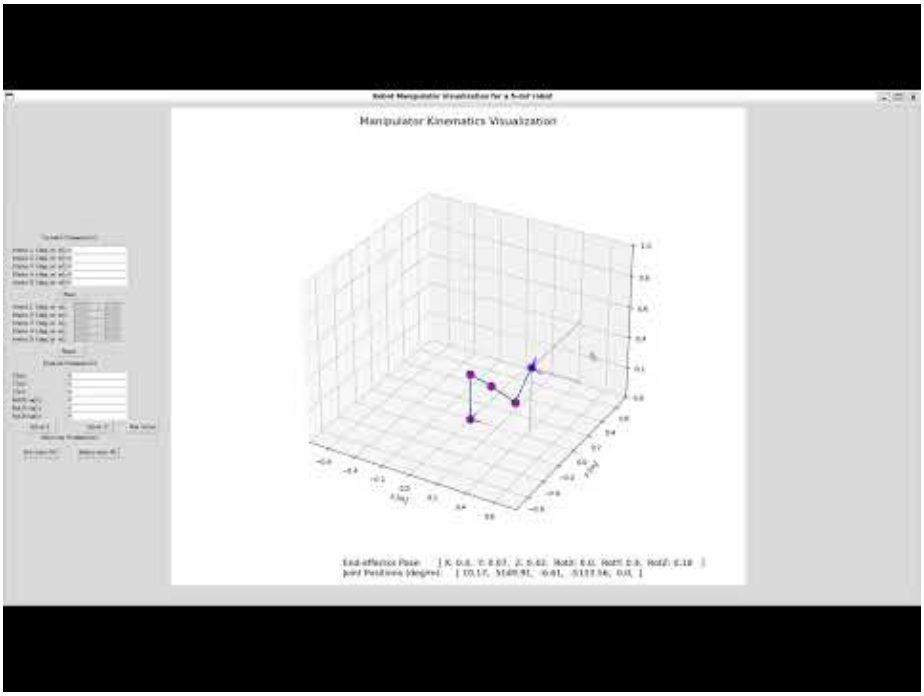
Simulation Verification

Click the images to view the videos

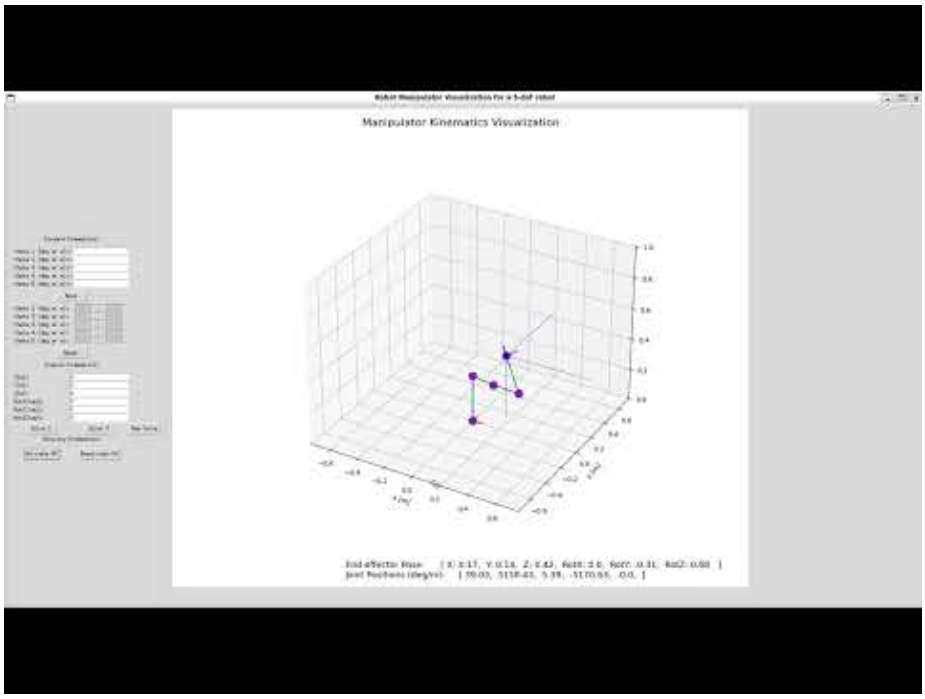
Forward Position Kinematics



Velocity Kinematics



Seperate Motion



Combined Motion

Gamepad Control

Velocity Kinematics