Student Name:
Mir Mohibullah Sazid [sazidarnob@gmail.com]
Group Member [Syma Afsha]

Lab 6: Task-Priority kinematic control (2B)

Introduction

This lab focuses on the development and analysis of a simulated mobile manipulator robot. The robot model combines a differential drive base with a 3-link planar manipulator, utilizing a Task-Priority control algorithm. There are three primary objectives:

Objective 1: Implement a new class representing the mobile manipulator's integrated kinematics.

Objective 2: Implement and analyze the weighted Damped Least Squares (DLS) algorithm, demonstrating the impact of weight adjustments on the robot's motion.

Objective 3: Evaluate the simulation error introduced by simplified linear updates of the mobile base kinematics. This will be compared against a more accurate update accounting for the robot's movement along an arc.

This lab contributes to the understanding of mobile manipulator systems, the impact of weighting within motion control algorithms, and the development of more accurate simulation models for robotics research.

Methodology

The robot that I have used for the model comprises three revolute joints, with the origins of the coordinate systems denoted by O_0, O_1, O_2, O_3, and O_4. There are five coordinate systems in total: one for the base frame, three for the robot joints, and one for the end-effector. The Denavit-Hartenberg parameter values used in the code are as follows: the link lengths (distance along the x-axis) are $a_1 = 0.75$, $a_2 = 0.50$, and $a_3 = 0.50$. The link offsets (distance along the z-axis) are $d_1 = 0$, $d_2 = 0$, and $d_3 = 0$. The link twist angles (rotation around the X-axis) are $\alpha_1 = 0$, $\alpha_2 = 0$, $\alpha_3 = 0$, and the joint angles (for the revolute joints) are variables that will update with each time step; the initial values are set as $\theta_1 = q_1 = 0.2$, $\theta_2 = q_2 = 0.5$, and $\theta_3 = q_3 = 0.2$.

In the first exercise, a mobile manipulator class is implemented. It includes fields to store critical parameters like the mobile base pose and the distance between the robot's center and the manipulator's base. The code's core functions are:

Constructor: Initializes kinematic parameters, configurations, and sets initial joint and base positions.

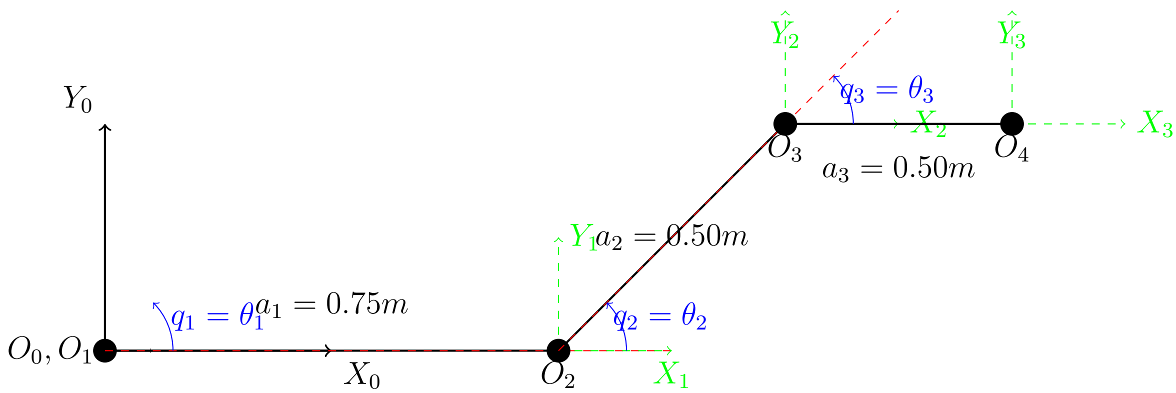


Figure 1: A simplified representation of the robot arm.

Update: Updates joint positions, base pose, and recalculates kinematic matrices based on input velocities and time.

GetEEJacobian: Computes and returns the end-effector Jacobian.

GetEETransform: Returns the end-effector transformation matrix.

GetJointPos, GetBasePose, GetDOF: Provide information on joint position, base pose, and degrees of freedom, respectively.

Drawing: Returns data for robot visualization.

The kinematic function is modified which can now take consideration of the mobile base.The base is given as follows:

$$T_b = \begin{bmatrix} \cos(\eta_{2,0}) & -\sin(\eta_{2,0}) & 0 & \eta_{0,0} \\ \sin(\eta_{2,0}) & \cos(\eta_{2,0}) & 0 & \eta_{1,0} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

Then using the algorithm 1, mobile manipulator task is calculated which initializes a projector matrix and a vector for differential joint velocities. For each active task, it updates the task based on the robot's state, calculates the task's Jacobian (adjusted by the projector matrix), and uses the Damped Least Squares (DLS) method to compute corrective joint velocities. The projector matrix is continuously updated to prioritize tasks and prevent conflicts between them. Here joint limit task and end effector position task is observed where the safe is [-0.5, 0.5] and the threshold is [0.03, 0.05]. This task defination and formulation is taken from the previous lab

In the exercise 2, at first weighted DLS is calculated which is same as DLS with added weight matrix. then using the algorithm 1, only using end effector configuration is calculated to observe the velocity, position and orientation of the end effector. For the weighted DLS different weight [2, 4, 6, 8, 9] is given.

The exercise 3 evaluates the simulation error introduced by using a simplified linear base update instead of a correct arc-based update.In the exercise 3, three implementations of the mobile base kinematics integration is done. Here instead of random desired end effector configuration some

Algorithm 1 Extension of the recursive TP

Require: List of tasks $J_i(q), \dot{x}_i(q), a_i(q)$, for $i \in 1 \dots k$
Ensure: Quasi-velocities $\zeta_k \in \mathbb{R}^n$

Initialise: $\zeta_0 = 0^n$, $P_0 = I^{n \times n}$
for $i \in 1 \dots k$ **do**
  if $a_i(q) \neq 0$ **then**
    $J_i(q) = J_i(q) P_{i-1}$
    $\zeta_i = \zeta_{i-1} + J_i^{\dagger}(q)(a_i(q)\dot{x}_i(q) - J_i(q)\zeta_{i-1})$
    $P_i = P_{i-1} - J_i^{\dagger}(q) J_i(q)$
  else
    $\zeta_i = \zeta_{i-1}$, $P_i = P_{i-1}$
  end if
end for

return ζ_k

selected vectors are used [[-1.5, -1.5], [1.5, 1.5], [0, 0], [0.1, 1.5], [0.3, -0.1], [-0.4, 1.2]] and in the weighted DLS weights is changed to [0.1, 0.1, 0.1, 0.1, 0.1] The following equations are used to implement each of the three integration ideas.

1. Move Forward and then Rotate:

$$
\begin{aligned}
x(t) &= x(t-1) + V\cos(\theta) \cdot dt \\
y(t) &= x(y-1) + V\sin(\theta) \cdot dt \\
\theta(t) &= \theta(t-1) + \omega \cdot dt
\end{aligned}
$$

2. Rotate and then Move Forward:

$$
\begin{aligned}
\theta(t) &= \theta(t-1) + \omega \cdot dt \\
x(t) &= x(t-1) + V\cos(\theta) \cdot dt \\
y(t) &= x(y-1) + V\sin(\theta) \cdot dt
\end{aligned}
$$

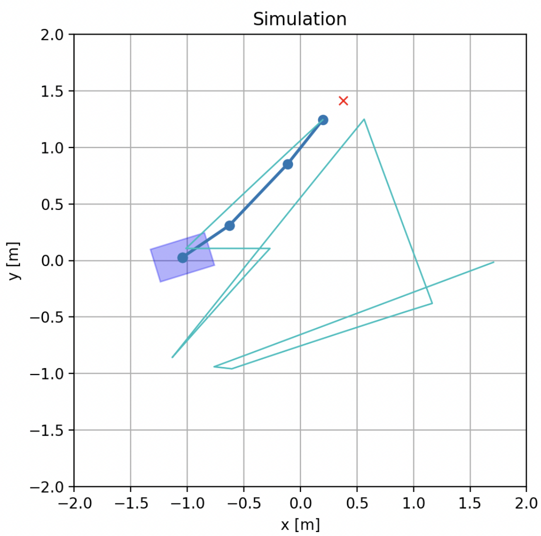
3. Rotate and then Move Together:

$$
\begin{aligned}
x(t) &= x(t-1) - R\sin(\theta) + R\sin(\omega \cdot dt + \theta) \\
y(t) &= y(t-1) - R\cos(\theta) + R\cos(\omega \cdot dt + \theta) \\
\theta(t) &= \theta(t-1) + \omega \cdot dt
\end{aligned}
$$

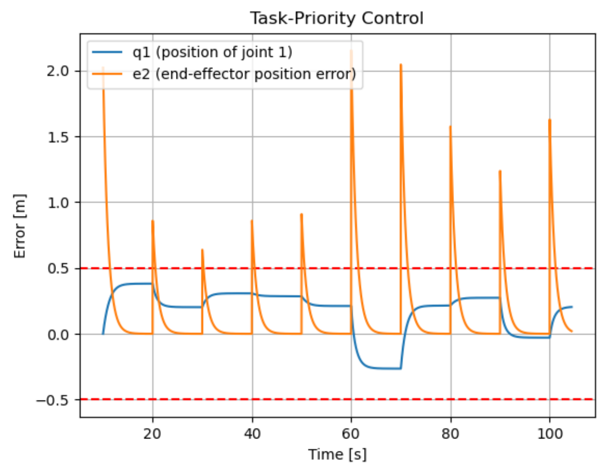
Here, $R = \frac{V}{\omega}$

Results

In the exercise 1, The figure displays a graph that tracks the progression of two different types of errors over time during a robot control simulation: the position error of joint 1 (q1), and the position error of the robot's end-effector (e2). The errors fluctuate over time, indicating the dynamic response of the robot to the control algorithm. The blue line represents the error in the position of joint 1, while the orange line represents the end-effector position error. The dashed red lines represent threshold limits that the system is trying to maintain for joint 1. This suggests a repeated adjustment process, as the control system continually corrects the robot's movement to follow the desired task, due to encountering limits or obstacles.



(a) Simulation of the moobile manipulator

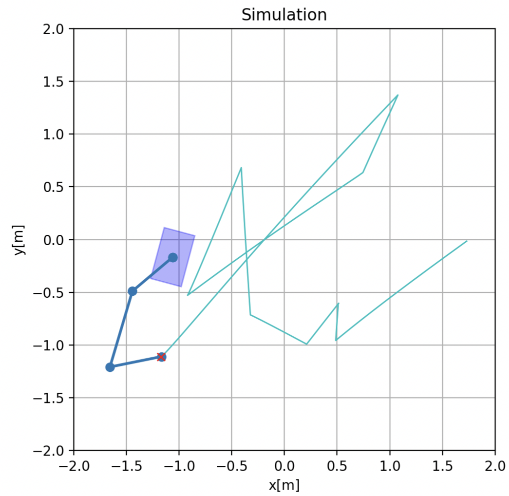


(b) Evolution of the TP control error

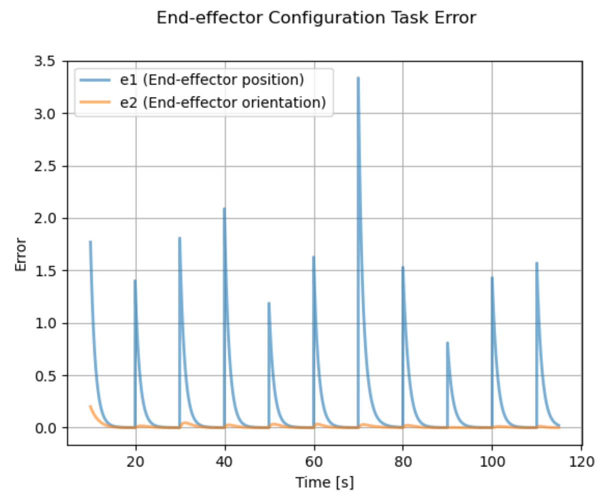
Figure 2: Results of Exercise 1

For the exercise 2, image plots the errors in end-effector configuration over time. The blue line (e1) shows the error in the end-effector's position, while the orange line (e2) displays the error in the end-effector's orientation.

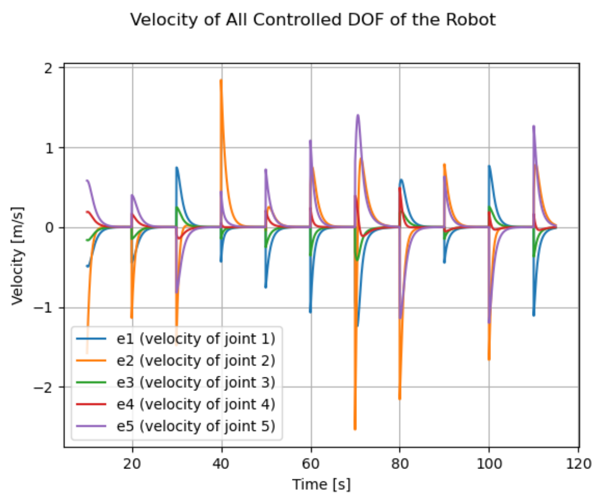
The second image illustrates the velocities of all controlled degrees of freedom (DOF) of the robot over time. Each line represents the velocity of a different joint, labeled e1 through e5. The graph shows that the velocities of the joints vary considerably over time, indicating dynamic adjustments by the control system to achieve the desired movement as part of a task-priority control.



(a) Simulation of the moobile manipulator



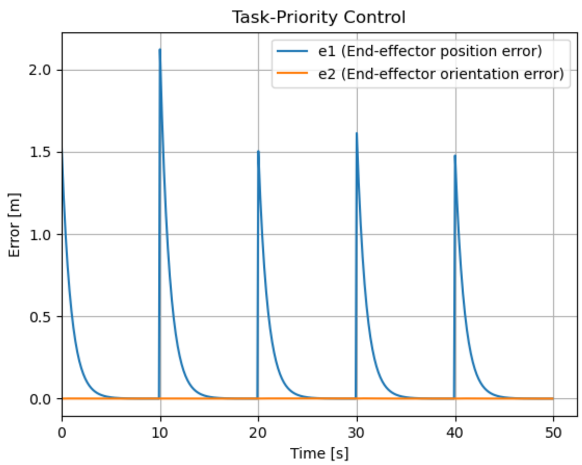
(b) Evolution of the TP control error



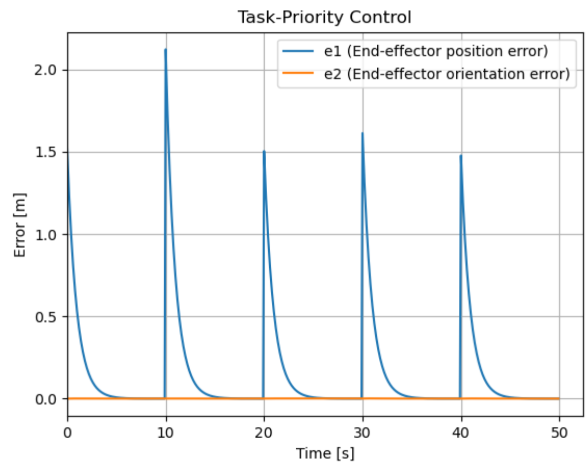
(c) Evolution of velocity

Figure 3: Results of Exercise 2

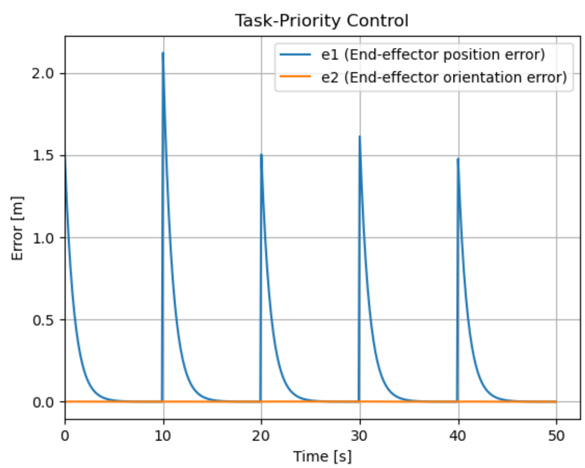
In the exercise 3, the error evaluation is introduced in the simulation, by the simplified, linear update of the mobile base kinematics, compared to the correct update, taking into account that the robot is moving along an arc. The following shows different error overtimes.



(a) Move and then Rotate



(b) Rotate and then Move



(c) Move and Rotate together

Figure 4: Results of Exercise 3

The figure shows the position of a mobile manipulator on the x-y plane. The text labels for the axis are “y[m]” and “x[m]”. The figure shows four different traces, each representing the position of the end effector and the base of the mobile manipulator according to two different scenarios. The scenarios are labeled “F>R”, “R>F”, "F and R" which are forward then rotate, rotate then forward and forward and rotate.

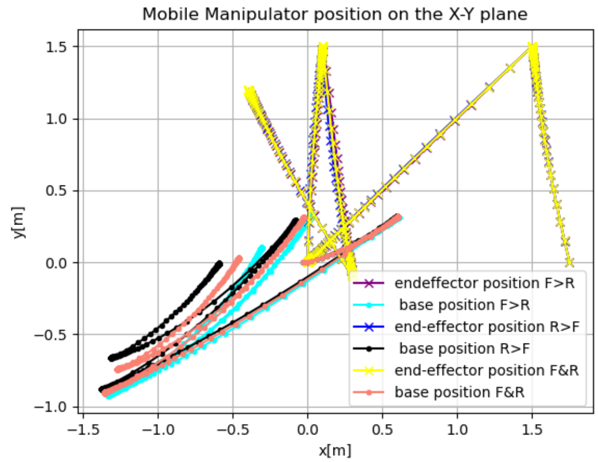


Figure 5: Mobile base position and the end-effector position

Conclusion

This lab successfully implemented a mobile manipulator class, integrated the weighted DLS algorithm, and analyzed simulation errors due to base update simplification. The results demonstrate the impact of weights on motion control and highlight the importance of accurate kinematic modeling for mobile manipulators.

Appendix

Common python code

```python
import numpy as np # Import Numpy

def DH(d, theta, a, alpha):
    '''
        Function builds elementary Denavit-Hartenberg transformation matrices
        and returns the transformation matrix resulting from their
            multiplication.

        Arguments:
        d (double): displacement along Z-axis
        theta (double): rotation around Z-axis
        a (double): displacement along X-axis
        alpha (double): rotation around X-axis

        Returns:
        (Numpy array): composition of elementary DH transformations
    '''
    # 1. Build matrices representing elementary transformations (based on input
        parameters).
    # 2. Multiply matrices in the correct order (result in T).
    # Convert angles from degrees to radians for consistency


    # Rotation about Z-axis by theta
    Rz = np.array([[np.cos(theta), -np.sin(theta), 0, 0],
                   [np.sin(theta), np.cos(theta), 0, 0],
                   [0, 0, 1, 0],
                   [0, 0, 0, 1]])

    # Translation along Z-axis by d
    Tz = np.array([[1, 0, 0, 0],
                   [0, 1, 0, 0],
                   [0, 0, 1, d],
                   [0, 0, 0, 1]])

    # Translation along X-axis by a
    Tx = np.array([[1, 0, 0, a],
                   [0, 1, 0, 0],
                   [0, 0, 1, 0],
                   [0, 0, 0, 1]])

    # Rotation about X-axis by alpha
    Rx = np.array([[1, 0, 0, 0],
                   [0, np.cos(alpha), -np.sin(alpha), 0],
                   [0, np.sin(alpha), np.cos(alpha), 0],
                   [0, 0, 0, 1]])

```

```python
    # DH Transformation Matrix
    T = Rz @ Tz @ Tx @ Rx

    return T

def kinematics(d, theta, a, alpha, tb):
    '''
        Functions builds a list of transformation matrices, for a kinematic chain,
        descried by a given set of Denavit-Hartenberg parameters.
        All transformations are computed from the base frame.

        Arguments:
        d (list of double): list of displacements along Z-axis
        theta (list of double): list of rotations around Z-axis
        a (list of double): list of displacements along X-axis
        alpha (list of double): list of rotations around X-axis

        Returns:
        (list of Numpy array): list of transformations along the kinematic chain (from the base frame)
    '''
    T = [tb] # Base transformation
    Transformation = T
    # For each set of DH parameters:
    # 1. Compute the DH transformation matrix.
    # 2. Compute the resulting accumulated transformation from the base frame.
    # 3. Append the computed transformation to T.
    for i in range(len(d)):
        # Compute the DH transformation matrix for the current joint
        T_current = DH(d[i], theta[i], a[i], alpha[i])

        # Compute the accumulated transformation from the base
        T = T @ T_current

        # Append the accumulated transformation to the list
        Transformation = np.vstack((Transformation, np.array(T)))

    return Transformation

# Inverse kinematics
def jacobian(T, revolute):
    '''
        Function builds a Jacobian for the end-effector of a robot,
        described by a list of kinematic transformations and a list of joint types.

        Arguments:
        T (list of Numpy array): list of transformations along the kinematic chain of the robot (from the base frame)
```

```python
        revolute (list of Bool): list of flags specifying if the corresponding
            joint is a revolute joint

        Returns:
        (Numpy array): end-effector Jacobian
    '''
    # 1. Initialize J and O.
    # 2. For each joint of the robot
    #   a. Extract z and o.
    #   b. Check joint type.
    #   c. Modify corresponding column of J.

    n = len(T)-1  # Number of joints/frames
    J = np.zeros((6, n))  # Initialize the Jacobian matrix with zeros

    O = np.array([T[-1][:3, 3]]).T  # End-effector's origin (position)
    Z = np.array([[0, 0, 1]]).T  # Z-axis of the base frame

    for i in range(n):
        # Extract the rotation matrix and origin from the transformation matrix
        R_i = T[i][:3, :3]
        O_i = np.array([T[i][:3, 3]]).T

        # Extract the z-axis from the rotation matrix
        Z_i = R_i @ Z

        if revolute[i]:
            # For revolute joints, use the cross product of z and (O - O_i)
            J[:3, i] = np.cross(Z_i.T, (O - O_i).T).T[:, 0]
            J[3:, i] = Z_i[:, 0]
        else:
            # For prismatic joints, the linear velocity is along the z-axis,
                and angular velocity is zero
            J[:3, i] = Z_i[:, 0]
            # The angular part is zero for prismatic joints, which is already
                set by the initialization with zeros

    return J

# Damped Least-Squares
def DLS(J, damping):
    '''
        Function computes the damped least-squares (DLS) solution to the matrix
            inverse problem.

        Arguments:
        A (Numpy array): matrix to be inverted
        damping (double): damping factor

        Returns:
        (Numpy array): inversion of the input matrix
```

```
    '''
    I = len(J)  # Identity matrix for a two-jointed robot

    damped_J = np.transpose(J) @ np.linalg.inv(J @ np.transpose(J) + ((damping ** 2) * np.identity(I)))


    return damped_J# Implement the formula to compute the DLS of matrix A.

def Weighted_DLS(A, damping, weights):
    '''
    Function computes the damped least-squares (DLS) solution to the
    matrix inverse problem.
    Arguments:
    A (Numpy array): matrix to be inverted
    damping (double): damping factor
    weights (list): weight of each DOF of the robot
    Returns:
    (Numpy array): inversion of the input matrix
    '''
    l=len(A)
    W = np.diag(weights)
    return np.linalg.pinv(W) @ np.transpose(A) @ np.linalg.pinv(A @ np.linalg.pinv(W) @ np.transpose(A)+((damping ** 2)* np.identity(l)))

# Extract characteristic points of a robot projected on X-Y plane
def robotPoints2D(T):
    '''
        Function extracts the characteristic points of a kinematic chain on a 2D plane,
        based on the list of transformations that describe it.

        Arguments:
        T (list of Numpy array): list of transformations along the kinematic chain of the robot (from the base frame)

        Returns:
        (Numpy array): an array of 2D points
    '''
    P = np.zeros((2,len(T)))
    for i in range(len(T)):
        P[:,i] = T[i][0:2,3]
    return P
```

Lab-5 Common file

```
from lab2_robotics import * # Includes numpy import

def jacobianLink(T, revolute, link): # Needed in Exercise 2
    '''
```

```
        Function builds a Jacobian for the end-effector of a robot,
        described by a list of kinematic transformations and a list of joint
            types.

        Arguments:
        T (list of Numpy array): list of transformations along the kinematic
            chain of the robot (from the base frame)
        revolute (list of Bool): list of flags specifying if the corresponding
            joint is a revolute joint
        link(integer): index of the link for which the Jacobian is computed

        Returns:
        (Numpy array): end-effector Jacobian
    '''
    # Code almost identical to the one from lab2_robotics...
    # Number of joints up to the specified link
    n = len (T)-1

    # Initialize the Jacobian matrix
    J = np.zeros((6, n))

    # Position of the end-effector
    p_n = T[link][:3, 3]

    for i in range(link):
        # Extract the rotation matrix and position vector for the current joint
        R_i = T[i][:3, :3]
        p_i = T[i][:3, 3]

        # Compute the z-axis (rotation/translation axis) for the current joint
        z_i = R_i[:, 2]

        # Compute the vector from the current joint to the end-effector
        r = p_n - p_i

        if revolute[i]:
            # For revolute joints, compute the linear velocity component
            J[:3, i] = np.cross(z_i, r)
            # And the angular velocity component
            J[3:, i] = z_i
        else:
            # For prismatic joints, the linear velocity component is the z-axis
            J[:3, i] = z_i
            # And the angular velocity component is zero
            J[3:, i] = 0

    return J


'''
    Class representing a robotic manipulator.
```

```python
'''
class Manipulator:
    '''
        Constructor.

        Arguments:
        d (Numpy array): list of displacements along Z-axis
        theta (Numpy array): list of rotations around Z-axis
        a (Numpy array): list of displacements along X-axis
        alpha (Numpy array): list of rotations around X-axis
        revolute (list of Bool): list of flags specifying if the corresponding
            joint is a revolute joint
    '''
    def __init__(self, d, theta, a, alpha, revolute):
        self.d = d
        self.theta = theta
        self.a = a
        self.alpha = alpha
        self.revolute = revolute
        self.dof = len(self.revolute)
        self.q = np.zeros(self.dof).reshape(-1, 1)
        self.update(0.0, 0.0)

    '''
        Method that updates the state of the robot.

        Arguments:
        dq (Numpy array): a column vector of joint velocities
        dt (double): sampling time
    '''
    def update(self, dq, dt):
        self.q += dq * dt
        for i in range(len(self.revolute)):
            if self.revolute[i]:
                self.theta[i] = self.q[i]
            else:
                self.d[i] = self.q[i]
        self.T = kinematics(self.d, self.theta, self.a, self.alpha)

    '''
        Method that returns the characteristic points of the robot.
    '''
    def drawing(self):
        return robotPoints2D(self.T)

    '''
        Method that returns the end-effector Jacobian.
    '''
    def getEEJacobian(self):
        return jacobian(self.T, self.revolute)

```

```
    '''
        Method that returns the end-effector transformation.
    '''
    def getEETransform(self):
        return self.T[-1]

    '''
        Method that returns the position of a selected joint.

        Argument:
        joint (integer): index of the joint

        Returns:
        (double): position of the joint
    '''
    def getJointPos(self, joint):
        return self.q[joint,0]

    '''
        Method that returns number of DOF of the manipulator.
    '''
    def getDOF(self):
        return self.dof

    def getLinkTransform(self, link):
        return self.T[link]

    '''
        Method that returns the link Jacobian.
    '''
    def getLinkJacobian(self, link):
        return jacobianLink(self.T, self.revolute, link)


'''
    Base class representing an abstract Task.
'''
class Task:
    '''
        Constructor.

        Arguments:
        name (string): title of the task
        desired (Numpy array): desired sigma (goal)
    '''
    def __init__(self, name, desired, FFVelocity= None, K=None):
        self.name = name # task title
        self.sigma_d = desired # desired sigma
        self.FFVelocity = FFVelocity #feedforward velocity
        self.K = K #gain matrix

```

```
    '''
        Method updating the task variables (abstract).

        Arguments:
        robot (object of class Manipulator): reference to the manipulator
    '''
    def update(self, robot):
        pass

    '''
        Method setting the desired sigma.

        Arguments:
        value(Numpy array): value of the desired sigma (goal)
    '''
    def setDesired(self, value):
        self.sigma_d = value

    '''
        Method returning the desired sigma.
    '''
    def getDesired(self):
        return self.sigma_d

    '''
        Method returning the task Jacobian.
    '''
    def getJacobian(self):
        return self.J

    '''
        Method returning the task error (tilde sigma).
    '''
    def getError(self):
        return self.err

    def setFFVelocity(self, value):
        self.FFVelocity = value

    '''
        Method returning the feedforward velocity vector.
    '''
    def getFFVelocity(self):
        return self.FFVelocity

    '''
        Method setting the gain matrix K.

        Arguments:
        value(Numpy array): value of the gain matrix K.
    '''
```

```
    def setK(self, value):
        self.K = value

    '''
        Method returning the gain matrix K.
    '''
    def getK(self):
        return self.K




'''
    Subclass of Task, representing the 2D position task.
'''
class Position2D(Task):
    def __init__(self, name, desired, FFVelocity=None, K= None, link= None):
        super().__init__(name, desired, FFVelocity, K)
        self.J = np.zeros((2,3)) # Initializing with proper dimensions
        self.err = np.zeros((2,1)) # Initializing with proper dimensions
        self.FFVelocity = np.zeros((2,1)) # Initializing with proper dimensions
        self.K = np.eye((2)) # Initializing with proper dimensions
        self.link = link

    def update(self, robot):
        self.J=robot.getEEJacobian()[:2,:]
        sigma = robot.getEETransform()[:2,3].reshape(2,1)  # Current position
            of the task
        self.err =  self.getDesired() - sigma #task error

    def isActive(self):
        return 1
'''
    Subclass of Task, representing the 2D orientation task.
'''
class Orientation2D(Task):
    def __init__(self, name, desired, FFVelocity, K, link):
        super().__init__(name, desired, FFVelocity, K)
        self.J = np.zeros((1,3)) # Initialize with proper dimensions
        self.err = np.zeros((1,1)) # Initialize with proper dimensions
        self.FFVelocity = np.zeros((1,1)) # Initialize with proper dimensions
        self.K = np.eye((1)) # Initialize with proper dimensions
        self.link = link

    def update(self, robot):
        #<<<<<<<<Exercise-1>>>>>>>>>>>
        self.J = robot.getEEJacobian()[5,:].reshape(1,3)
        angle = np.arctan2(robot.getEETransform()[1,0], robot.getEETransform()
            [0,0])
        self.err = (self.getDesired() - angle)

```

```python

'''
    Subclass of Task, representing the 2D configuration task.
'''
class Configuration2D(Task):
    def __init__(self, name, desired, FFVelocity = None, K = None, link = None)
        :
        super().__init__(name, desired, FFVelocity, K)
        self.J = np.zeros((3,5)) # Initializing with proper dimensions
        self.err = np.zeros((3,1)) # Initializing with proper dimensions
        self.FFVelocity = np.zeros((3,1)) # Initializing with proper dimensions
        self.K = np.eye((3)) # Initializing with proper dimensions
        self.link = link

    def update(self, robot):
        #<<<<<<<Exercise-1>>>>>>>>
        self.J[:2,:] = robot.getEEJacobian()[:2,:]
        self.J[2,:] = robot.getEEJacobian()[5,:]
        R = robot.getEETransform() [:3,:3]
        angle = angle=np.arctan2(R[1,0],R[0,0])
        self.err[:2]= self.getDesired()[:2] - robot.getEETransform()[:2,3].
            reshape(2,1)
        self.err[2] = self.getDesired()[2] - angle

    def isActive(self):
        return 1

'''
    Subclass of Task, representing the joint position task.
'''
class JointPosition(Task):
    def __init__(self, name, desired, FFVelocity = None, K = None):
        super().__init__(name, desired, FFVelocity, K)
        self.J = np.zeros((1,3)) # Initializing with proper dimensions
        self.err = np.zeros((1,1)) # Initializing with proper dimensions
        self.FFVelocity = np.zeros((1,1)) # Initializing with proper dimensions
        self.K = np.eye((1)) # Initializing with proper dimensions

    def update(self, robot):
        self.J[0,0] = 1 #for joint 1
        self.err =  self.getDesired() - robot.getJointPos(0)


'''
    Subclass of Task, representing 2D circular obostracle.
'''
class Obstacle2D(Task):
    def __init__(self, name, obstacle_pos, radius):
        super().__init__(name, 0)
        self.obstacle_pos = obstacle_pos
        self.radius = radius
```

```python
        self.J = np.zeros((2,3)) # Initializing with proper dimensions
        self.err = np.zeros((2,1)) # Initializing with proper dimensions
        self.active = 0
        self.distance = 0

    def isActive(self):
        return self.active

    def update(self, robot):
        self.J = robot.getEEJacobian()[:2, :]
        current_pos = robot.getEETransform()[:2,3].reshape(2,1)
        self.err =  (current_pos - self.obstacle_pos)/np.linalg.norm(current_pos - self.obstacle_pos)
        self.distance = current_pos - self.obstacle_pos
        if self.active==0 and np.linalg.norm(current_pos - self.obstacle_pos) <= self.radius[0]:
            self.active=1
        elif self.active==1 and np.linalg.norm(current_pos - self.obstacle_pos) >= self.radius[1]:
            self.active=0

'''
    Subclass of Task, representing the joint limit task.
'''
class JointLimit(Task):
    def __init__(self, name, joint, safe_set, thresholds):
        super().__init__(name, 0)
        self.J = np.zeros((1,3)) # Initializing with proper dimensions
        self.err = np.zeros((1,1)) # Initializing with proper dimensions
        self.safe_set = safe_set
        self.thresholds = thresholds
        self.joint = joint
        self.active = 0

    def update(self, robot):

        self.J[0,self.joint] = 1
        self.err =  1 * self.active

        #Activation update
        if self.active==0 and robot.getJointPos(self.joint) >= self.safe_set[1] - self.thresholds[0]:
            self.active = -1
        elif self.active== 0 and robot.getJointPos(self.joint) <= self.safe_set[0] + self.thresholds[0]:
            self.active = 1
        elif self.active== -1 and robot.getJointPos(self.joint) <= self.safe_set[1] - self.thresholds[1]:
            self.active = 0
        elif self.active==1 and robot.getJointPos(self.joint) >= self.safe_set[0] + self.thresholds[1]:
```

```
            self.active = 0


    def isActive(self):
        return self.active
```

Lab-6 Common file

```
import math
from lab5_robotics import *

class MobileManipulator:
    '''
        Constructor.

        Arguments:
        d (Numpy array): list of displacements along Z-axis
        theta (Numpy array): list of rotations around Z-axis
        a (Numpy array): list of displacements along X-axis
        alpha (Numpy array): list of rotations around X-axis
        revolute (list of Bool): list of flags specifying if the corresponding
           joint is a revolute joint
    '''
    def __init__(self, d, theta, a, alpha, revolute):
        self.d = d
        self.theta = theta
        self.a = a
        self.alpha = alpha
        self.radius = 0
        self.revolute = revolute
        self.revoluteExt = [True, False] # List of joint types extended with
           base joints
        self.revoluteExt.extend(self.revolute)
        self.r = 0            # Distance from robot centre to manipulator base
        self.dof = len(self.revoluteExt)  # Number of DOF of the system
        self.q = np.zeros((len(self.revolute),1)) # Vector of joint positions (
           manipulator)
        self.eta = np.zeros((3,1)) # Vector of base pose (position &
           orientation)
        self.update(np.zeros((self.dof,1)), 0.0) # Initialise robot state

    '''
        Method that updates the state of the robot.

        Arguments:
        dQ (Numpy array): a column vector of quasi velocities
        dt (double): sampling time
    '''
    def update(self, dQ, dt):
        self.q += dQ[2:, 0].reshape(-1, 1) * dt
```

```
        for i in range(len(self.revolute)):
            if self.revolute[i]:
                self.theta[i] = self.q[i]
            else:
                self.d[i] = self.q[i]

        # Update mobile base pose
        # Update mobile base pose (move forward then rotate)
        # self.eta[0,0] += dQ[1, 0] *np.cos(self.eta[2,0])* dt
        # self.eta[1,0] += dQ[1, 0] *np.sin(self.eta[2,0])* dt
        # self.eta[2,0]+=dQ[0, 0]* dt

        # Update mobile base pose ( rotate then move forward)
        self.eta[2,0]+=dQ[0, 0]* dt
        self.eta[0,0] += dQ[1, 0] *np.cos(self.eta[2,0])* dt
        self.eta[1,0] += dQ[1, 0] *np.sin(self.eta[2,0])* dt

# Update mobile base pose ( rotate and move forward)
        # if math.isclose(dQ[0,0], 0.0):
        #     pass
        # else:
        #     #calculate radius
        #     self.radius = dQ[1,0]/dQ[0,0]
        #     self.eta[0,0] += - self.radius*np.sin(self.eta[2,0])+ self.radius
            *np.sin(dQ[0,0]*dt+(self.eta[2,0]))
        #     self.eta[1,0] += self.radius*np.cos(self.eta[2,0])- self.radius*
            np.cos(dQ[0,0]*dt+(self.eta[2,0]))
        #     self.eta[2,0] += dQ[0,0]*dt

        # Base kinematics
        Tb = np.array([[np.cos(self.eta[2,0]),-np.sin(self.eta[2,0]),0,self.eta
            [0,0]],
             [np.sin(self.eta[2,0]),np.cos(self.eta[2,0]),0,self.eta[1,0]],
             [0,0,1,0],
             [0,0,0,1]])

        # Combined system kinematics (DH parameters extended with base DOF)
        self.theta[0]+=-np.pi/2
        dExt = np.concatenate([np.array([ 0 , self.r ]), self.d])
        thetaExt = np.concatenate([np.array([np.pi/2 , 0 ]), self.theta])
        aExt = np.concatenate([np.array([ 0 , 0 ]), self.a])
        alphaExt = np.concatenate([np.array([ np.pi/2 , -np.pi/2 ]), self.alpha
            ])
        self.T = kinematics(dExt, thetaExt, aExt, alphaExt, Tb)
    '''
        Method that returns the characteristic points of the robot.
    '''
    def drawing(self):
        return robotPoints2D(self.T)

    '''
```

```python
        Method that returns the end-effector Jacobian.
    '''
    def getEEJacobian(self):
        return jacobian(self.T, self.revoluteExt)

    '''
        Method that returns the end-effector transformation.
    '''
    def getEETransform(self):
        return self.T[-1]

    '''
        Method that returns the position of a selected joint.

        Argument:
        joint (integer): index of the joint

        Returns:
        (double): position of the joint
    '''
    def getJointPos(self, joint):
        return self.q[joint-2,0]


    def getBasePose(self):
        return self.eta

    '''
        Method that returns number of DOF of the manipulator.
    '''
    def getDOF(self):
        return self.dof

    ###
    def getLinkJacobian(self, link):
        return jacobianLink(self.T, self.revoluteExt, link)

    def getLinkTransform(self, link):
        return self.T[link]
```

Exercise 1

```python
from lab6_robotics import *  # This imports MobileManipulator, JointLimit,
    Position2D classes
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patch
import matplotlib.animation as anim
import matplotlib.transforms as trans

```

```python
# Robot model initialization
d = np.zeros(3)  # Displacement along Z-axis
theta = np.array([0.2, 0.5, 0.2])  # Rotation around Z-axis
alpha = np.zeros(3)  # Rotation around X-axis
a = [0.5, 0.75, 0.5]  # Displacement along X-axis
revolute = [True, True, True]  # All joints are revolute
robot = MobileManipulator(d, theta, a, alpha, revolute)

# Define tasks for the robot
tasks = [
    JointLimit("Joint limits", 2, np.array([-0.5, 0.5]), np.array([0.03, 0.05])
        ),
    Position2D("End-effector position", np.array([1.0, 0.5]).reshape(2, 1))
]

# Simulation parameters
dt = 1.0 / 60.0  # Time step
count = -1  # Counter for the simulation steps

# Set up the drawing for the simulation
fig = plt.figure(1)
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-2, 2), ylim=(-2, 2))
ax.set_title('Simulation')
ax.set_aspect('equal')
ax.grid(True)
ax.set_xlabel('x [m]')
ax.set_ylabel('y [m]')
rectangle = patch.Rectangle((-0.25, -0.15), 0.5, 0.3, color='blue', alpha=0.3)
veh = ax.add_patch(rectangle)
line, = ax.plot([], [], 'o-', lw=2)  # Robot structure
path, = ax.plot([], [], 'c-', lw=1)  # Path of the end-effector
point, = ax.plot([], [], 'rx')  # Target point
pp_x = []  # X-coordinates of the path
pp_y = []  # Y-coordinates of the path
time = []  # Time points for plotting
errors = [[], []]  # Errors for joints and end-effector

# Initialize simulation
def init():
    global tasks, count
    count += 1
    # Set a new desired position for the end-effector
    tasks[-1].setDesired(np.array([np.random.uniform(-1.5, 1.5), np.random.
        uniform(-1.5, 1.5)]).reshape(2, 1))
    line.set_data([], [])
    path.set_data([], [])
    point.set_data([], [])
    return line, path, point

# Simulation step function
def simulate(t):
```

```python
    global robot, pp_x, pp_y
    p_matrix = np.eye(robot.getDOF())  # Projector matrix
    dq = np.zeros(robot.getDOF()).reshape(robot.getDOF(), 1)  # Differential
        joint velocities
    for index, task in enumerate(tasks):
        task.update(robot)
        if task.isActive():
            j_bar = task.getJacobian() @ p_matrix
            dq += DLS(j_bar, 0.1) @ (task.getError() - task.getJacobian() @ dq)
            p_matrix -= np.linalg.pinv(j_bar) @ j_bar
        errors[index].append(np.linalg.norm(task.getError()) if index else
            robot.getJointPos(task.joint))

    robot.update(dq, dt)
    pp = robot.drawing()
    line.set_data(pp[0, :], pp[1, :])
    pp_x.append(pp[0, -1])
    pp_y.append(pp[1, -1])
    time.append(t + 10 * count)
    path.set_data(pp_x, pp_y)
    point.set_data(tasks[-1].getDesired()[0], tasks[-1].getDesired()[1])
    eta = robot.getBasePose()
    veh.set_transform(trans.Affine2D().rotate(eta[2]) + trans.Affine2D().
        translate(eta[0], eta[1]) + ax.transData)
    return line, veh, path, point

# Run simulation
ani = anim.FuncAnimation(fig, simulate, frames=np.arange(0, 10, dt), interval
    =10, blit=True, init_func=init, repeat=True)
plt.show()

# Error plotting after simulation
fig2 = plt.figure(2)
plt.axhline(y=-0.5, color='r', linestyle='--')
plt.axhline(y=0.5, color='r', linestyle='--')
plt.plot(time, errors[0], label='q1 (position of joint 1)')
plt.plot(time, errors[1], label='e2 (end-effector position error)')
plt.ylabel('Error [m]')
plt.xlabel('Time [s]')
plt.title('Task-Priority Control')
plt.grid(True)
plt.legend()
plt.show()
```

Exercise 2

```python
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patch
```

```python
import matplotlib.animation as anim
import matplotlib.transforms as trans
from lab6_robotics import *

# Define constants and parameters
dt = 1.0 / 60.0
counter = -1
weights = [2, 4, 6, 8, 9]

# Initialize lists for storing data
PPx = []
PPy = []
time = []
error = [[], []]
velocities = []

# Initialize simulation figure
fig = plt.figure(1)
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-2, 2), ylim=(-2, 2))
ax.set_title('Simulation')
ax.set_aspect('equal')
ax.grid()
ax.set_xlabel('x[m]')
ax.set_ylabel('y[m]')

# Initialize robot model
d = np.zeros(3)
theta = np.array([0.2, 0.5, 0.2])
alpha = np.zeros(3)
a = [0.5, 0.75, 0.5]
revolute = [True, True, True]
robot = MobileManipulator(d, theta, a, alpha, revolute)

# Define tasks
tasks = [
    Configuration2D("end-effector configuration", np.array([1.0, 0.5, 0]).
        reshape(3, 1))
]

# Define drawing elements
rectangle = patch.Rectangle((-0.25, -0.15), 0.5, 0.3, color='blue', alpha=0.3)
veh = ax.add_patch(rectangle)
line, = ax.plot([], [], 'o-', lw=2)  # Robot structure
path, = ax.plot([], [], 'c-', lw=1)  # End-effector path
point, = ax.plot([], [], 'rx')  # Target

# Initialize animation
def init():
    global tasks, counter
    counter += 1
```

```python
    tasks[-1].setDesired(np.array([np.random.uniform(-1.5, 1.5), np.random.
        uniform(-1.5, 1.5), 0.2]).reshape(3, 1))
    line.set_data([], [])
    path.set_data([], [])
    point.set_data([], [])
    return line, path, point

# Run simulation loop
def simulate(t):
    global tasks, robot, PPx, PPy, time, error, velocities
    P = np.eye(robot.getDOF())
    dq = np.zeros(robot.getDOF()).reshape(robot.getDOF(), 1)
    for task in tasks:
        task.update(robot)
        if task.isActive():
            J_bar = task.getJacobian() @ P
            dq += Weighted_DLS(J_bar, 0.1, weights) @ (task.getError() - task.
                getJacobian() @ dq)
            P -= np.linalg.pinv(J_bar) @ J_bar
            error[0].append(np.linalg.norm(task.getError()[0:2]))
            error[1].append(np.linalg.norm(task.getError()[2]))
            velocities.append(dq)
    robot.update(dq, dt)
    PP = robot.drawing()
    line.set_data(PP[0, :], PP[1, :])
    PPx.append(PP[0, -1])
    PPy.append(PP[1, -1])
    time.append(t + 10 * counter)
    path.set_data(PPx, PPy)
    point.set_data(tasks[-1].getDesired()[0], tasks[-1].getDesired()[1])
    eta = robot.getBasePose()
    veh.set_transform(trans.Affine2D().rotate(eta[2, 0]) + trans.Affine2D().
        translate(eta[0, 0], eta[1, 0]) + ax.transData)
    return line, veh, path, point

# Initialize and run animation
animation = anim.FuncAnimation(fig, simulate, np.arange(0, 10, dt), interval
    =10, blit=True, init_func=init, repeat=True)
plt.show()

# Plot for end-effector configuration task error
fig, ax1 = plt.subplots()
fig.suptitle('End-effector Configuration Task Error')
ax1.set_xlabel('Time [s]')
ax1.set_ylabel('Error')
ax1.plot(time, error[0], alpha=0.6, linewidth=2, label='e1 (End-effector
    position)')
ax1.plot(time, error[1], alpha=0.6, linewidth=2, label='e2 (End-effector
    orientation)')
ax1.legend()
ax1.grid()
```

```
plt.show()

# Plot for velocities of all controlled DOF of the robot
fig, ax2 = plt.subplots()
fig.suptitle('Velocity of All Controlled DOF of the Robot')
ax2.set_xlabel('Time [s]')
ax2.set_ylabel('Velocity [m/s]')
ax2.plot(time, [v[0, 0] for v in velocities], label='e1 (velocity of joint 1)')
ax2.plot(time, [v[1, 0] for v in velocities], label='e2 (velocity of joint 2)')
ax2.plot(time, [v[2, 0] for v in velocities], label='e3 (velocity of joint 3)')
ax2.plot(time, [v[3, 0] for v in velocities], label='e4 (velocity of joint 4)')
ax2.plot(time, [v[4, 0] for v in velocities], label='e5 (velocity of joint 5)')
ax2.legend()
ax2.grid()
plt.show()
```

Exercise 3

```
from lab6_robotics import * # Includes numpy import
import matplotlib.pyplot as plt
import matplotlib.patches as patch
import matplotlib.animation as anim
import matplotlib.transforms as trans

# Robot model
d = np.zeros(3)                             # displacement along Z-axis
theta = np.array([0.2, 0.5, 0.2])           # rotation around Z-axis
alpha = np.zeros(3)                         # rotation around X-axis
a = [0.5, 0.75, 0.5]                        # displacement along X-axis
revolute = [True, True, True]               # flags specifying the type of
    joints
robot = MobileManipulator(d, theta, a, alpha, revolute)
weights = [0.1, 0.1, 0.1, 0.1, 0.1] #weight of each DOF
tasks = [
        Configuration2D("end-effector configuration", np.array([1.0, 0.5,0]).
            reshape(3,1))
        ]
# Simulation params
dt = 1.0/10.0
counter = -2
# Drawing preparation
fig = plt.figure(1)
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-2, 2), ylim=(-2,2))
ax.set_title('Simulation')
ax.set_aspect('equal')
ax.grid()
ax.set_xlabel('x[m]')
ax.set_ylabel('y[m]')
rectangle = patch.Rectangle((-0.25, -0.15), 0.5, 0.3, color='blue', alpha=0.3)
veh = ax.add_patch(rectangle)
```

```python
line, = ax.plot([], [], 'o-', lw=2) # Robot structure
path, = ax.plot([], [], 'c-', lw=1) # End-effector path
point, = ax.plot([], [], 'rx') # Target
PPx = []
PPy = []
time= []
error = [[],[]]
velocities =[]
EE_pos=[[],[]] # to store the ee position of each joint to be used in the second plot.
base_pos=[[],[]]
i=0


# Simulation initialization
def init():
    global tasks, i, counter
    Desired = tasks[-1].getDesired()
    desired = np.array([[-1.5, -1.5], [1.5, 1.5], [0, 0], [0.1, 1.5], [0.3, -0.1], [-0.4, 1.2]])
    Desired[0:2] = desired[i].reshape(2, 1)
    line.set_data([], [])
    path.set_data([], [])
    point.set_data([], [])
    i += 1
    counter = counter + 1
    return line, path, point

# Simulation loop
def simulate(t):
    global tasks
    global robot
    global PPx, PPy
    ### Recursive Task-Priority algorithm
    P = np.eye(robot.getDOF())
    # Initialize output vector (joint velocity)
    dq = np.zeros(robot.getDOF()).reshape(robot.getDOF(),1)

    for task in tasks:
        task.update(robot)
        if task.isActive():
            J_bar = task.getJacobian() @ P
            dq += Weighted_DLS(J_bar, 0.1, weights) @ (task.getError() - task.getJacobian() @ dq)
            P -= np.linalg.pinv(J_bar) @ J_bar
            error[0].append(np.linalg.norm(task.getError()[0:2]))
            error[1].append(np.linalg.norm(task.getError()[2]))
            velocities.append(dq)

            EE_pos[0].append(robot.getEETransform()[0, 3])
            EE_pos[1].append(robot.getEETransform()[1, 3])
```

```
                base_pos[0].append(robot.getBasePose()[0, 0])
                base_pos[1].append(robot.getBasePose()[1, 0])
                # q_base.append(robot.getBasePose()[:2,0].reshape(2,1))
        # Update robot
        robot.update(dq, dt)
        # Update drawing
        # -- Manipulator links
        PP = robot.drawing()
        line.set_data(PP[0,:], PP[1,:])
        PPx.append(PP[0,-1])
        PPy.append(PP[1,-1])
        time.append(t + 10 * counter)
        path.set_data(PPx, PPy)
        point.set_data(tasks[-1].getDesired()[0], tasks[-1].getDesired()[1])
        # -- Mobile base
        eta = robot.getBasePose()
        veh.set_transform(trans.Affine2D().rotate(eta[2,0]) + trans.Affine2D().
            translate(eta[0,0], eta[1,0]) + ax.transData)
        return line, veh, path, point

# Run simulation
animation = anim.FuncAnimation(fig, simulate, np.arange(0, 10, dt),interval=10,
     blit=True, init_func=init, repeat=True)
plt.show()
# Plot errors over time
plt.plot(time, error[0], label='e1 (End-effector position error)')
plt.plot(time, error[1], label='e2 (End-effector orientation error)')
plt.ylabel('Error [m]')
plt.xlabel('Time [s]')
plt.title('Task-Priority Control')
plt.xlim(left=0)
plt.grid(True)
plt.legend()
plt.show()
# Save EE_pos and base_pos to file
# np.save('move forward then rotate.npy',[EE_pos,base_pos])
np.save(' rotate then move forward.npy',[EE_pos,base_pos])
# np.save(' rotate and move forward.npy',[EE_pos,base_pos])
```

Plotting

```
import matplotlib.pyplot as plt #Plotting library
import numpy as np # linear algebra library


# Define three arrays loaded from disk files
F_R=np.load('move forward then rotate.npy',allow_pickle=True)
R_F=np.load(' rotate then move forward.npy',allow_pickle=True)
F_and_R=np.load(' rotate and move forward.npy',allow_pickle=True)

```

```python
# Create a new figure object
fig = plt.figure()
# Add subplot to the current figure on a "1x1 grid, with ID: first subplot",
    with autoscale.
ax = fig.add_subplot(111, autoscale_on=True)
# Title of the plot
ax.set_title('Mobile Manipulator position on the X-Y plane')
# Label of x axis
ax.set_xlabel('x [m]')
# Label of y axis
ax.set_ylabel('y [m]')
# Aspect of axes (the ratio of y-unit to x)
ax.set_aspect('auto')
# Grid of the subplot
ax.grid()
# Plot presenting the evolution of the mobile base position and the end-
    effector position on the X-Y plane
ax.plot(F_R[0][0], F_R[0][1], marker="x", color='purple', label='end-effector
    position F>R')
ax.plot(F_R[1][0], F_R[1][1], marker=".", color='cyan', label='base position F>
    R')
ax.plot(R_F[0][0], R_F[0][1], marker="x", color='blue', label='end-effector
    position R>F')
ax.plot(R_F[1][0], R_F[1][1], marker=".", color='black', label='base position R
    >F')
ax.plot(F_and_R[0][0], F_and_R[0][1], marker="x", color='yellow', label='end-
    effector position F&R')
ax.plot(F_and_R[1][0], F_and_R[1][1], marker=".", color='salmon', label='base
    position F&R')
# Legend
ax.legend()
# Display the plot
plt.show()
```