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

Lab 5:Task-Priority kinematic control (2A)

Introduction

This lab report extends the implementation of the recursive Task-Priority algorithm to support set-based (inequality) tasks for enhanced robot control. Building upon successful obstacle avoidance task implementation in a simulated 3-link planar manipulator, this work focuses on developing and testing a joint limits task. The objective is to demonstrate the algorithm's ability to handle joint limit constraints, ensuring safe and reliable robot operation within its physical boundaries. This implementation highlights the framework's adaptability and potential for broader applications in robotics.

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 $\theta_3 = q_3 = 0.2$.

From the lab 2, the common files are taken which computes the DH parameters, kinematic, Jacobian and DLS function. In the exercise 1, at first, the obstacle avoidance task is implemented where we calculated the Jacobian using the following formula:

$$[J_r = J_r(\mathfrak{q}) = J_v(\mathfrak{q}) \in \mathbb{R}^{3\times n}] \tag{1}$$

Then the error is calculated from the current position to obstacle position using the following formula:

$$\dot{x}_{li}(q) = \frac{n_{1,ee}(q) - P}{|n_{1,ee}(q) - P|} \tag{2}$$

The safe distance is calculated using the following formula:

$$\sigma_r = \sigma_r(q) = |n_{1,ee}(q) - P| \tag{3}$$

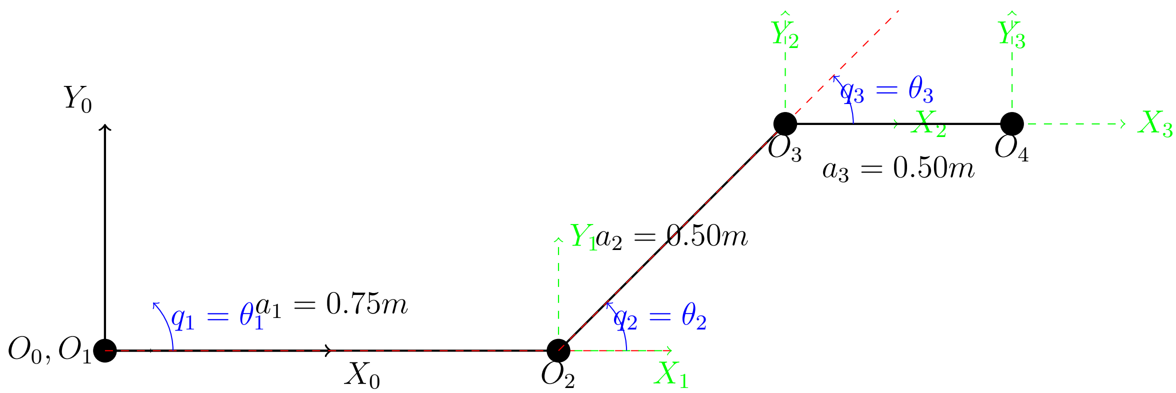


Figure 1: A simplified representation of the robot arm.

The activation function is calculated using the following formula. In the equation, $r_\alpha \in \mathbb{R}$ is the activation threshold, $r_\delta \in \mathbb{R}$ is the deactivation threshold, and the condition $r_\delta > r_\alpha$ must be taken into consideration to avoid chatter.

$$a_r(q) = \begin{cases} 1, & \text{if } a_r = 0 \wedge |n_{1,ee}(q) - P| \leq r_\alpha \\ 0, & \text{if } a_r = 1 \wedge |n_{1,ee}(q) - P| \geq r_\delta \end{cases} \tag{4}$$

Then the task is defined as three obstacle task with three different position as [0.0, 1.0], [0.5, -1.3] and [-1.5, -0.5]. Then three different radius is given as 0.5, 0.6 and 0.4. Then the end effector position task is defined. Then the following task priority algorithm is implemented.

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: ζ_0 = 0^n, P_0 = I^{n×n}
for i ∈ 1...k do
    if a_i(q) ≠ 0 then
        J_i(q) = J_i(q) P_{i−1}
        ζ_i = ζ_{i−1} + J_i^†(q)(a_i(q) ẋ_i(q) − J_i(q) ζ_{i−1})
        P_i = P_{i−1} − J_i^†(q) J_i(q)
    else
        ζ_i = ζ_{i−1}, P_i = P_{i−1}
    end if
end for

return ζ_k
```

In the exercise 2, The joint limit task is defined where the error is 1 and the jocobain is defined for joint 1. The activation function is defined as the following equations where $\alpha_{li} \in \mathbb{R}$ is the activation threshold, $\delta_{li} \in \mathbb{R}$ is the deactivation threshold and $\delta_{li} > \alpha_{li}$ is used to avoid chatter.

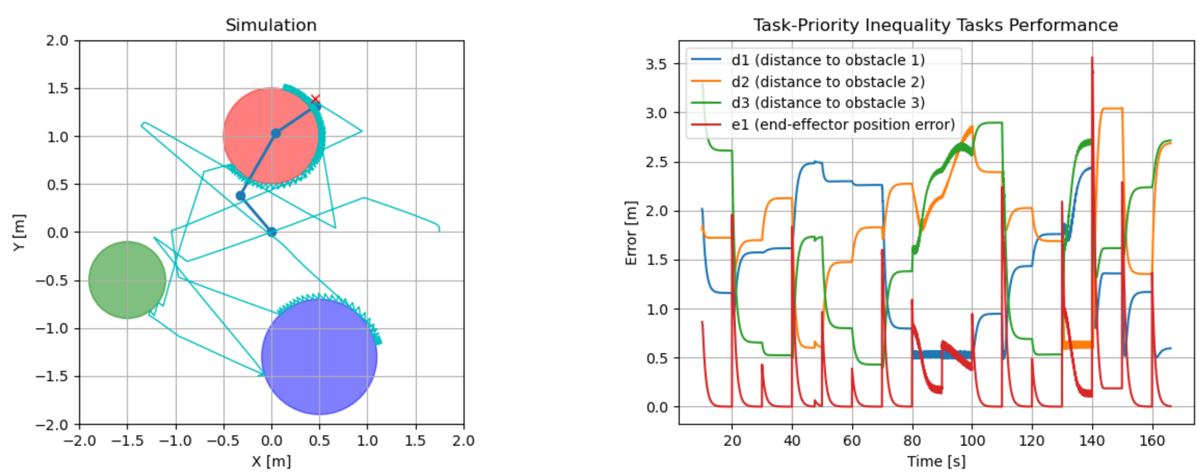
$$a_{li}(q) = \begin{cases} -1, & a_{li} = 0 \land q_i \geq q_{i,\max} - \alpha_{li} \\ 1, & a_{li} = 0 \land q_i \leq q_{i,\min} + \alpha_{li} \\ 0, & a_{li} = -1 \land q_i \leq q_{i,\max} - \delta_{ji} \\ 0, & a_{li} = 1 \land q_i \geq q_{i,\min} + \delta_{ji} \end{cases} \tag{5}$$

The joint limit task is set at the top and end-effector position task is set at the bottom. In the joint limit task, activation and deactivation threshold is set as 0.01 and 0.04 and the safe set is defined as -0.5 (min) and 0.5 (max). Then the algorithm 1 is implemented.

Results

In Exercise 1,In each figure the left plot illustrates motion of robot structure in 2D plane with endeffector goal and the plot on the right shows distance from the obstacles.

The image right image illustrates the results of a simulation of the lab. The y-axis, labeled "Error [m]," represents the end-effector position error, where a lower error indicates proximity to the intended position. The x-axis, labeled "Time[s]," denotes time in seconds. The labels "d1 (distance to obstacle 1)," "d2 (distance to obstacle 2)," and "d3 (distance to obstacle 3)" refer to the distances between the robot's end effector and three obstacles in the environment over time. Higher values on these lines suggest greater distances from the end effector to specific obstacles. The plot suggests a trade-off between proximity to obstacles and end-effector positioning accuracy, where the robot ideally navigates close to obstacles while maintaining a low end-effector position error.



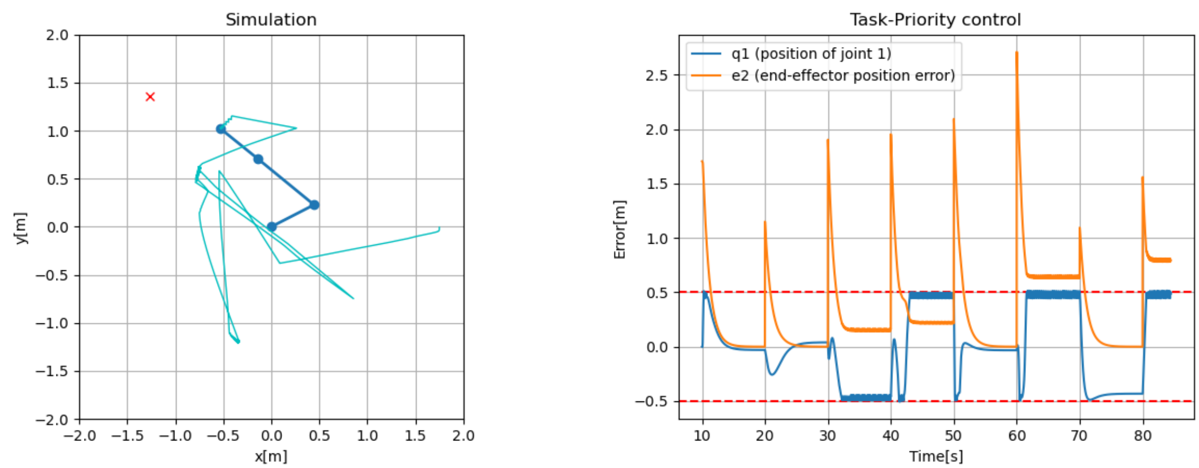
(a) Simulation of the manipulator including end-effector goal and obstacles.

(b) Evolution of the TP control error and distance to obstacles over time.

Figure 2: Results of Exercise 1

Exercise 2 involved extending our task hierarchy to prioritize both end-effector positioning and joint limits. We implemented the joint limits task (specifically for joint 1) as a set-based constraint using the 3-link planar manipulator model from exercise 1.

The figure illustrates the performance of a robot arm manipulator aiming to reach a specific position while adhering to joint limits. On the right side, two key metrics are plotted against time. The y-axis, labeled "Error [m]," indicates the end-effector position error , with lower values suggesting proximity to the desired position. The x-axis, labeled "Time[s]," represents time in seconds. A red dotted line delineates the safe operational zone for the joint limits task. Oscillations in the end-effector position error are observed, particularly when approaching maximum and minimum joint limits. These fluctuations result from control system adjustments to maintain safety while achieving the desired end-effector position. The plot suggests a potential trade-off between precision and adhering to joint limits, as the controller balances these objectives, occasionally causing slight deviations from the ideal trajectory.



(a) Simulation of the manipulator including end-effector goal.

(b) Evolution of the end-effector position error and the first joint position (including limits).

Figure 3: Exercise 2 results

Conclusion

This exercise demonstrated the successful integration of set-based (inequality) tasks into a general robotic system's task hierarchy. Our method enables the handling of multiple scalar set-based tasks with designated priorities alongside equality tasks. The primary strength lies in the system's ability to always satisfy the constraints imposed by set-based tasks while simultaneously working to fulfill equality-based goals. This approach significantly enhances the flexibility and adaptability of robotic control systems, opening up potential applications where both hard constraints and more flexible objectives must be managed.

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):
    '''
        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 = [np.eye(4)] # Base transformation
    # 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_accumulated = T[-1] @ T_current

        # Append the accumulated transformation to the list
        T.append(T_accumulated)

    return T

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

```python

        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
```

```python

    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.

# 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 2
           D 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-4 Common file

```python
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.
'''
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]

```

```python
    '''
        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):
        #<<<<<<<<Exercise-1>>>>>>>>>>
        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)


'''
    Subclass of Task, representing the 2D configuration task.
'''
class Configuration2D(Task):
    def __init__(self, name, desired, FFVelocity, K, link):
        super().__init__(name, desired, FFVelocity, K)
        self.J = np.zeros((3,3)) # 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,:]
```

```
        angle = np.arctan2(robot.getEETransform()[1,0],robot.getEETransform()[0,0])
        self.err[:2]= self.getDesired()[:2] - robot.getEETransform()[:2,3].reshape(2,1)
        self.err[2] = self.getDesired()[2] - angle

'''
    Subclass of Task, representing the joint position task.
'''
class JointPosition(Task):
    def __init__(self, name, desired, FFVelocity, K):
        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
        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, 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.active = 0

    def update(self, robot):

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

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


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

Exercise 1

```
from lab4_robotics import *
import matplotlib.pyplot as plt
import matplotlib.animation as anim
import matplotlib.patches as patch

# Robot model parameters
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 = Manipulator(d, theta, a, alpha, revolute)  # Instantiate the
    manipulator

# Set up tasks for obstacle avoidance and end-effector targeting
obstacle_pos = np.array([0.0, 1.0]).reshape(2, 1)  # Position of the first
    obstacle
obstacle_r = 0.5  # Radius of the first obstacle
obstacle_pos2 = np.array([0.5, -1.3]).reshape(2, 1)  # Position of the second
    obstacle
obstacle_r2 = 0.6  # Radius of the second obstacle
obstacle_pos3 = np.array([-1.5, -0.5]).reshape(2, 1)  # Position of the third
    obstacle
obstacle_r3 = 0.4  # Radius of the third obstacle

tasks = [
    Obstacle2D("Obstacle avoidance", obstacle_pos, np.array([obstacle_r,
        obstacle_r + 0.05])),
    Obstacle2D("Obstacle avoidance", obstacle_pos2, np.array([obstacle_r2,
        obstacle_r2 + 0.05])),
    Obstacle2D("Obstacle avoidance", obstacle_pos3, np.array([obstacle_r3,
        obstacle_r3 + 0.05])),
    Position2D("End-effector position", np.array([1.0, 0.5]).reshape(2, 1))
]

# Define simulation parameters
dt = 1.0 / 60.0  # Time step for simulation
Storage = -1  # Index used for storing simulation data across runs

# Set up the visualization environment
fig = plt.figure()
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]')
# Visualize obstacles as patches
ax.add_patch(patch.Circle(obstacle_pos.flatten(), obstacle_r, color='red',
    alpha=0.5))
ax.add_patch(patch.Circle(obstacle_pos2.flatten(), obstacle_r2, color='blue',
    alpha=0.5))
ax.add_patch(patch.Circle(obstacle_pos3.flatten(), obstacle_r3, color='green',
    alpha=0.5))

# Elements for animation
line, = ax.plot([], [], 'o-', lw=2)  # Visual representation of the robot
path, = ax.plot([], [], 'c-', lw=1)  # Path of the end-effector
point, = ax.plot([], [], 'rx')       # Desired target position for the end-
    effector

# Data storage for plotting
```

```python
PPx, PPy = [], []  # Path position storage
time = []  # Time storage for x-axis in plots
error = [[], [], [], []]  # Error storage for the different tasks

# Initialize the simulation
def init():
    global tasks, Storage
    Storage += 1

    # Set a random desired position for the end-effector task
    tasks[-1].setDesired(np.array([np.random.uniform(-1.5, 1.5),
                                   np.random.uniform(-1.5, 1.5)]).reshape(2, 1))

    # Clear data from previous frames
    line.set_data([], [])
    path.set_data([], [])
    point.set_data([], [])
    return line, path, point

# Simulation loop called by the animation
# Simulation loop called by the animation
def simulate(t):
    global tasks, robot, PPx, PPy

    # Use a recursive task-priority framework for control
    P = np.eye(robot.getDOF())  # Null-space projector
    dq = np.zeros(robot.getDOF()).reshape(robot.getDOF(), 1)  # Initialize joint velocities

    for index, task in enumerate(tasks):  # Iterate over each task
        task.update(robot)  # Update task information based on current robot state

        if task.isActive():  # Check if the task is active
            J_bar = task.getJacobian() @ P  # Compute the dynamically consistent inverse
            dq += DLS(J_bar, 0.1) @ (task.getError() - task.getJacobian() @ dq)  # Compute the joint velocity increment
            P -= np.linalg.pinv(J_bar) @ J_bar  # Update the projector

        # Record errors for plotting. Use getError() for end-effector task and distance for obstacle tasks

        if task.name == "End-effector position":
            error[index].append(np.linalg.norm(task.getError()))
        else:
            #distance form obstacles
            error[index].append(np.linalg.norm(task.distance))
    # Update robot state with computed joint velocities
    robot.update(dq, dt)
```

```python

    # Drawing updates
    PP = robot.drawing()
    line.set_data(PP[0, :], PP[1, :])
    PPx.append(PP[0, -1])
    PPy.append(PP[1, -1])
    time.append(t + 10 * Storage)
    path.set_data(PPx, PPy)
    point.set_data(tasks[-1].getDesired()[0], tasks[-1].getDesired()[1])

    return line, path, point


# Create the animation with the simulate function as the animation driver
animation = anim.FuncAnimation(fig, simulate, np.arange(0, 10, dt),
                               interval=10, blit=True, init_func=init, repeat=
                                   True)
plt.show()

# Plot the errors of each task over time after simulation
fig = plt.figure()
plt.plot(time, error[0], label='d1 (distance to obstacle 1)')
plt.plot(time, error[1], label='d2 (distance to obstacle 2)')
plt.plot(time, error[2], label='d3 (distance to obstacle 3)')
plt.plot(time, error[3], label='e1 (end-effector position error)')
plt.ylabel('Error [m]')
plt.xlabel('Time [s]')
plt.title('Task-Priority Inequality Tasks Performance')
plt.grid(True)
plt.legend()
plt.show()
```

Exercise 2

```python
# Importing necessary libraries
from lab4_robotics import *
import matplotlib.pyplot as plt
import matplotlib.animation as anim
import matplotlib.patches as patch

# Robot model parameters
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 = Manipulator(d, theta, a, alpha, revolute) # Manipulator object

# List of tasks for the robot
```

```
tasks = [
            JointLimit("Joint limits", np.array([-0.5, 0.5]), np.array([0.01, 0.04])),  # Joint limits task with activation and deactivation thresholds 0.01 and 0.04, and safe set of q_min and q_max (-0.5 and 0.5)
            Position2D("End-effector position", np.array([0.25, -0.75]).reshape(2,1))   # End-effector position task
        ]

# Simulation parameters
dt = 1.0/60.0
Storage = -1

# Drawing preparation
fig = plt.figure()
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]')
line, = ax.plot([], [], 'o-', lw=2)  # Robot structure
path, = ax.plot([], [], 'c-', lw=1)  # End-effector path
point, = ax.plot([], [], 'rx')       # Target
# Memory
PPx = []
PPy = []
time= []
error = [[],[]]

# Simulation initialization
def init():
    global tasks, Storage

    Storage += 1
    # Setting end-effector position as final task
    tasks[len(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


# Main simulation loop
def simulate(t):
    global robot, PPx, PPy

    # Run the Recursive Task-Priority algorithm
    P = np.eye(robot.getDOF())  # Initialize the projector matrix
```

```
    dq = np.zeros(robot.getDOF()).reshape(robot.getDOF(),1)  # Initialize joint velocities

    # Loop over tasks, updating each and applying the control law
    for index, task in enumerate(tasks):
        task.update(robot)  # Update the task's internal state
        if task.isActive() != 0:
            J_bar = task.getJacobian() @ P  # Calculate the augmented Jacobian
            dq += DLS(J_bar, 0.1) @ (task.getError() - task.getJacobian() @ dq)  # Calculate the joint velocity
            P = P - np.linalg.pinv(J_bar) @ J_bar  # Update the null-space projector

        # Record the joint position or end-effector position error for plotting
        error[index].append(np.linalg.norm(task.getError()) if index else robot.getJointPos(0))

    # Update the manipulator's state
    robot.update(dq, dt)

    # Update the drawing for animation
    PP = robot.drawing()
    line.set_data(PP[0,:], PP[1,:])
    PPx.append(PP[0,-1])
    PPy.append(PP[1,-1])
    time.append(t + 10 * Storage)
    path.set_data(PPx, PPy)
    point.set_data(tasks[-1].getDesired()[0], tasks[-1].getDesired()[1])

    return line, 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-2
fig2 = plt.figure(2)
# Specifying horizontal line for safe sets
plt.axhline(y = -0.5, color = 'r', linestyle = '--')
plt.axhline(y = 0.5, color = 'r', linestyle = '--')
plt.plot(time, error[0], label='q1 (position of joint 1)')  # Plotting position of joint 1 against time
plt.plot(time, error[1], label='e2 (end-effector position error)')  # Plotting position error of end-effector
plt.ylabel('Error[m]')  # Title of the Y axis
plt.xlabel('Time[s]')  # Title of the X axis
plt.title('Task-Priority control')  # Title of plot-1
plt.grid(True)  # Grid
plt.legend()  # Placing legend
```

```
plt.show()
```