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

Lab 3:Task-Priority kinematic control (1A)

Introduction

The aim of this lab is to create visual representations of the movements within the null space of a planar 3-link manipulator equipped with 3 revolute joints. Building on what was learned in Lab 2, control of the robot will be achieved through the resolved-rate motion control algorithm, and exploration of the null space will be conducted using the null space projector. In Exercise 2, The objective of this activity is to apply the Task-Priority control algorithm, focusing on a two-task hierarchy (utilizing the analytical solution), to manage the manipulator that was simulated in Exercise 1. This involves specifying the Jacobians and errors for each task and executing the control loop.

Methodology

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 linear terms of the Jacobian are taken

$$J_p = J_p(\boldsymbol{q}) = J_v(\boldsymbol{q}) \in \mathbb{R}^{3\times n} \tag{1}$$

Then the null space projector is calculated using the formula

$$P = (I - J^{\dagger}(\boldsymbol{q})J(\boldsymbol{q})) \quad \text{Null space projector} \tag{2}$$

Then the control signal with respect to time is calculated using the formula:

$$\boldsymbol{\xi} = J^{\dagger}(\boldsymbol{q})\dot{\boldsymbol{x}}_E + (I - J^{\dagger}(\boldsymbol{q})J(\boldsymbol{q}))\boldsymbol{y} \tag{3}$$

Here I have visualised the null space projector as well as the joints problem with respect to time. 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 $\theta_3 = q_3 = 0.2$.

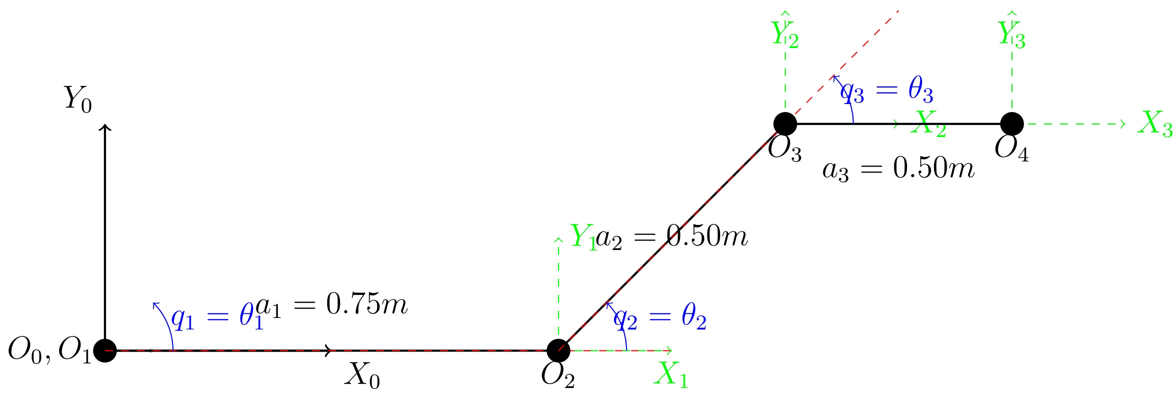


Figure 1: A simplified representation of the robot arm.

In Exercise 2, Initially, the error vectors for both tasks were computed using the following formula:

$$\tilde{\sigma}_{ji} = \sigma_{ji,d} - \sigma_{ji} \text{ Joint Position task error} \tag{4}$$

$$\tilde{\sigma}_p = \eta_{1,d} - \eta_1 \text{ End Effector position task error} \tag{5}$$

The task-priority control algorithm was employed to manage two specific tasks: determining the position of the end-effector and the position of the first joint using the following formula.

$$J_p = J_p(\boldsymbol{q}) = J_v(\boldsymbol{q}) \in \mathbb{R}^{3\times n} \text{ Position of End Effector task Jacobian} \tag{6}$$

$$P = (I - J^\dagger(\boldsymbol{q})J(\boldsymbol{q})) \quad \text{Null space projector} \tag{7}$$

$$J_{ji} = [1, 0, 0] \in \mathbb{R}^{1\times n} \text{ Joint Position task Jacobian} \tag{8}$$

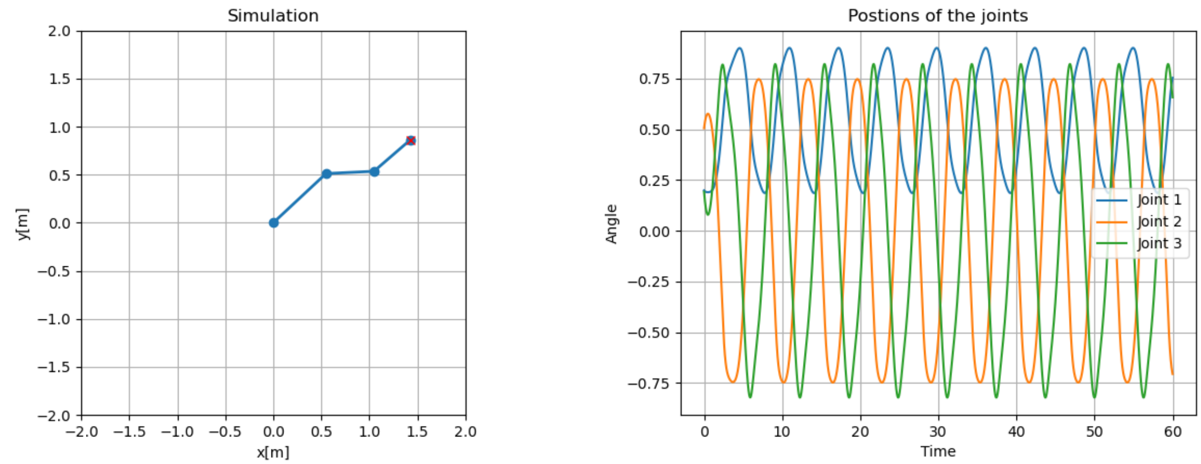
Then the Jacobian of the second task is multiplied with the null space projector. The velocity for the first task is calculated like in lab 2. The Task-Priority algorithm was executed using an analytical solution for the pair of tasks. Moreover, the output velocity derived from the Task-Priority algorithm was modified at every time step to ensure it remained within the boundaries of the maximum joint velocity limit. For combining the task, we used the following formula:

$$\dot{\xi}_i = \dot{\xi}_{i-1} + J_i^\dagger(\boldsymbol{q})(\dot{x}_i(\boldsymbol{q}) - J_i(\boldsymbol{q})\dot{\xi}_{i-1}) \tag{9}$$

Following this, the motion of the robot was simulated in a two-dimensional space, with an observation of the varying joint velocities throughout the duration of the simulation. The same process is followed again, just in reverse priority task order, keeping the first joint velocity task in task 1 and end-effector position task in second priority.

Results

In the first exercise, the computation of the resolved rate motion control and the null space projector is carried out within a control feedback loop. Figure 2 depicts a simulation of null space movements, with the left graph displaying the motion of the robot's structure on a two-dimensional plane, targeting the end-effector's position, and the right graph presenting the variation of the robot's joint positions over time. Figure 3 showcases the simulation iterations prioritizing the end-effector



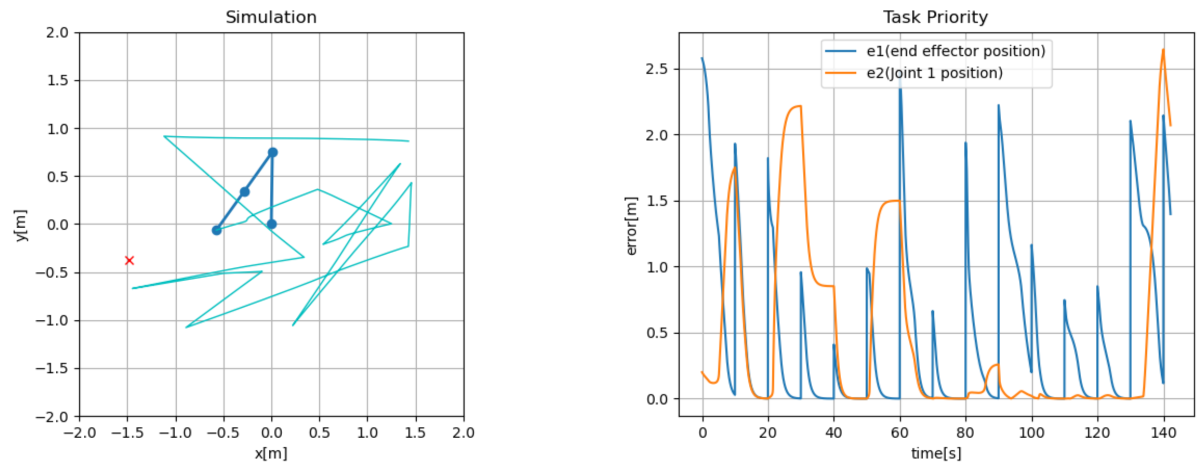
(a) Manipulator with end-effector goal.

(b) Positions of robot's joints

Figure 2: Null space motions simulation

position task above the first joint position task. For each iteration, the left image illustrate the movement of the robotic structure, while the right image track the progression of the norm of control errors for the tasks over time. In this sets of figures, it's observed that the end-effector consistently attains its intended target position (when possible) regardless of whether the position of joint 1 is maintained at a specific desired location, owing to the fact that the end-effector position now has the highest priority. This is evident in figures, which indicate that at every 10-second interval, the end effector is assigned a new random target position and strives to reduce the error between its current and target positions (e1), eventually reaching zero to signify the goal has been met. However, the error in joint 1's position (e2) does not always diminish, as reaching the end-effector's target position requires deviations from joint 1's desired position.

Figure 4 displays the simulation, with priority given to maintaining the joint position. In the figure, the robot's movement is depicted in the left image, while the right image shows how the control error norms for the defined tasks evolve over time. It's noticeable that in these simulations, the robot consistently prioritises reaching the preset position for joint 1. After securing joint 1's position, it attempts to move the end-effector to its designated location. The simulations reveal that the robot often does not succeed in positioning the end-effector as intended because it is focusing on keeping joint 1 in place. This is demonstrated in the left figure, where the error for joint 1 (e1) is reduced to zero as it hits the target position and then remains at zero, signifying that joint 1's position is maintained consistently, irrespective of whether the end-effector achieves its intended position or not.



(a) Motion of the Manipulator.

(b) Control errors of the tasks over time

Figure 3: End-effector position task at the top of the hierarchy

Question and Answer

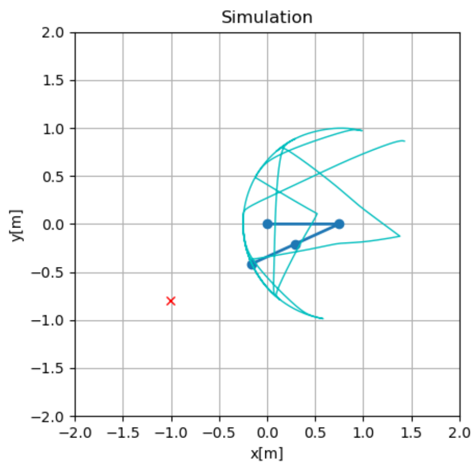
Q1: What are the advantages and disadvantages of redundant robotic systems?

Redundant robotic systems are those with more degrees of freedom (DoF) than are necessary for a given task. This redundancy can be advantageous or disadvantageous depending on the context and application. Here are some advantages and disadvantages: Advantages

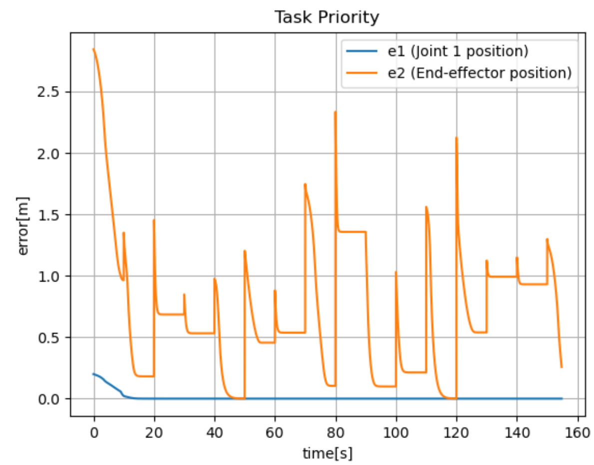
- Increased Flexibility: Redundancy allows for more paths and configurations to reach a given position.
- Greater Dexterity: More degrees of freedom can lead to improved dexterity in complex tasks.
- Fault Tolerance: The system can still function even if a part fails, using the remaining degrees of freedom.
- Improved Manipulability: Redundant robots can optimize manipulability for specific tasks.
- Obstacle Avoidance: These systems can maneuver to avoid obstacles while maintaining end-effector positioning.
- Workspace Enlargement: Additional degrees of freedom can expand the robot's operational workspace.

Disadvantages

- Complex Control Schemes: Additional degrees of freedom require more sophisticated control algorithms.



(a) Motion of the Manipulator.



(b) Control errors of the tasks over time

Figure 4: Joint position task at the top of the hierarchy

- Increased Costs: More actuators and sensors lead to higher hardware and maintenance costs.
- Higher Probability of Mechanical Failure: More components can lead to a greater chance of mechanical failure.
- Complex Kinematic Solutions: Inverse kinematics for redundant systems can yield multiple solutions.
- Increased Weight and Energy Consumption: Additional components can result in a heavier, more energy-consuming system.
- Complex Programming and Integration: Advanced programming expertise is required, complicating system integration.

Q2: What is the meaning and practical use of a weighting matrix W, that can be introduced in the pseudo inverse/DLS implementation?

A weighting matrix W is utilized within the pseudo-inverse or Damped Least Squares (DLS) method for multiple purposes. It serves as a tool to scale and prioritize different tasks or directions of motion.

Meaning:

The weighting matrix W is typically a diagonal or full-rank matrix that influences the robot's behavior by weighting certain tasks or motion directions more heavily in the optimization process. It modifies the calculation of the pseudo-inverse of the Jacobian, which is pivotal in determining the joint velocities required to achieve a specified end-effector velocity. This allows for:

- Differentiation between the importance of various motions.

- Compensation for scale differences in joint or task space coordinates.
- Management of sensor feedback influence.
- Consideration for the robot's physical constraints, such as limited joint ranges.

Practical Use:

In practical scenarios, the weighting matrix is applied in several ways:

1. **Task Prioritization:** Assigning higher weights to critical tasks ensures their precision and preference in execution.
2. **Regularization:** It plays a role in the DLS method to avoid numerical instabilities, ensuring smooth and stable robot motion.
3. **Joint Limit Avoidance:** Dynamically adjusting weights can help prevent joints from reaching their limits.
4. **Redundancy Resolution:** In redundant robotic systems, W can guide the selection of joint movements.
5. **Compliance and Stiffness Control:** Adjusting W can control the robot's compliance, useful in tasks involving contact or human interaction.
6. **Energy Efficiency:** Weights may be chosen to favor energy-efficient joint movements.

Overall, the weighting matrix W in pseudo-inverse or DLS algorithms allows for enhanced control strategies in robotic manipulation, fine-tuning task execution to meet operational goals and constraints.

Conclusion

Robotic systems with redundancy are engineered to perform various tasks at the same time. These tasks, which are dependent on the configuration of the system, can be categorized based on their level of importance. The system's redundant capabilities can be utilized by integrating optimization tasks of lesser importance into the control scheme. It is essential, however, to ensure that these additional lower-priority tasks do not interfere with the completion of tasks that are more critical, particularly those related to safety and operations. Analytical approaches have been traditionally favored over iterative techniques due to their ability to provide precise solutions and their typically quicker computation times. In this laboratory session, we have focused on learning the implementation of task-priority control using an analytical method.

Appendix

Common python code

```python
import numpy as np

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
    '''
    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]])

    Tz = np.array([[1, 0, 0, 0],
                   [0, 1, 0, 0],
                   [0, 0, 1, d],
                   [0, 0, 0, 1]])

    Tx = np.array([[1, 0, 0, a],
                   [0, 1, 0, 0],
                   [0, 0, 1, 0],
                   [0, 0, 0, 1]])

    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]])

    T = Rz @ Tz @ Tx @ Rx

    return T

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

```python

        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 i in range(len(d)):
        T_current = DH(d[i], theta[i], a[i], alpha[i])
        T_accumulated = T[-1] @ T_current
        T.append(T_accumulated)

    return T


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

        Returns:
        (Numpy array): end-effector Jacobian
    '''
    n = len(T)-1
    J = np.zeros((6, n))

    O = np.array([T[-1][:3, 3]]).T
    Z = np.array([[0, 0, 1]]).T

    for i in range(n):
        R_i = T[i][:3, :3]
        O_i = np.array([T[i][:3, 3]]).T
        Z_i = R_i @ Z

        if revolute[i]:
            J[:3, i] = np.cross(Z_i.T, (O - O_i).T).T[:, 0]
            J[3:, i] = Z_i[:, 0]
        else:
```

```python
            J[:3, i] = Z_i[:, 0]

    return J



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



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
```

Exercise 1

```python
# Import necessary libraries
from lab2_robotics import * # Includes numpy import
import matplotlib.pyplot as plt
import matplotlib.animation as anim

# Robot definition (3 revolute joint planar manipulator)
d = np.zeros(3)                          # displacement along Z-axis
q = np.array([0.2, 0.5, 0.2]).reshape(3,1)# rotation around Z-axis (theta)
a = np.array([0.75, 0.5, 0.5])           # displacement along X-axis
alpha = np.zeros(3)                      # rotation around X-axis
revolute = [True, True, True]    # flags specifying the type of joints
k=np.eye(2)                      # gain

# Setting desired position of end-effector to the current one
T = kinematics(d, q.flatten(), a, alpha) # flatten() needed if q defined as
    column vector !
sigma_d = T[-1][0:2,3].reshape(2,1)

# Simulation params
dt = 1.0/60.0
Tt = 10 # Total simulation time
tt = np.arange(0, Tt, dt) # Simulation time vector

# 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.set_xlabel('x[m]')
ax.set_ylabel('y[m]')
ax.grid()
line, = ax.plot([], [], 'o-', lw=2) # Robot structure
path, = ax.plot([], [], 'c-', lw=1) # End-effector path
point, = ax.plot([], [], 'rx') # Target

# storing data
PPx = []
PPy = []
q1_position = []
q2_position = []
q3_position = []
time = []
# Simulation initialization
def init():
    line.set_data([], [])
    path.set_data([], [])
    point.set_data([], [])
    return line, path, point

```

```python
# Simulation loop
def simulate(t):
    global q, a, d, alpha, revolute, sigma_d,k
    global PPx, PPy, q1_position, q2_position, q3_position, time

    # Update robot
    T = kinematics(d, q.flatten(), a, alpha)
    J = jacobian(T, revolute)

    # Update control
    sigma = T[-1][:2, 3].reshape(2,1) # Current position of the end-effector
    err = sigma_d-sigma               # Error in position
    Jbar = J[:2, :3]                  # Task Jacobian
    P = np.eye(3)-np.linalg.pinv(Jbar)@Jbar      # Null space projector
    y =  np.array([[np.sin(t), np.cos(t), np.sin(t)]]).T  # Arbitrary joint velocity
    dq = np.linalg.pinv(Jbar)@k@err+P@y         # Control signal
    q = q + dt * dq # Simulation update

    q1_position.append(q[0])
    q2_position.append(q[1])
    q3_position.append(q[2])
    time.append(t)

    # Update drawing
    PP = robotPoints2D(T)
    line.set_data(PP[0,:], PP[1,:])
    PPx.append(PP[0,-1])
    PPy.append(PP[1,-1])
    path.set_data(PPx, PPy)
    point.set_data(sigma_d[0], sigma_d[1])

    return line, path, point



# Run simulation
animation = anim.FuncAnimation(fig, simulate, np.arange(0, 60, dt),
                                interval=10, blit=True, init_func=init, repeat=False)
plt.show()

fig = plt.figure()
plt.plot(time, q1_position, label='Joint 1')
plt.plot(time, q2_position, label='Joint 2')
plt.plot(time, q3_position, label='Joint 3')
plt.ylabel('Angle')
plt.xlabel('Time')
plt.title('Positions of the joints')
plt.grid(True)
plt.legend()
```

```
plt.show()
```

Exercise 2

```
# Import necessary libraries
from lab2_robotics import * # Includes numpy import
import matplotlib.pyplot as plt
import matplotlib.animation as anim

# Robot definition (3 revolute joint planar manipulator)
d = np.zeros(3)         # displacement along Z-axis
q = np.array([0.2, 0.5, 0.2]).reshape(3,1)# rotation around Z-axis (theta)
a = np.array([0.75, 0.5, 0.5]) # displacement along X-axis
alpha = np.zeros(3) # rotation around X-axis
revolute = [True, True, True]     # flags specifying the type of joints
k=np.eye(2)                       # gain
max_velocity = 0.5

# Desired values of task variables
# sigma1_d = np.array([0.0, 1.0]).reshape(2,1) # Position of the end-effector
sigma2_d = np.array([[0.0]]) # Position of joint 1

# Simulation params
dt = 1.0/60.0
Tt = 10 # Total simulation time
tt = np.arange(0, Tt, dt) # Simulation time vector
count = -1 # for the loop

# 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.set_xlabel('x[m]')
ax.set_ylabel('y[m]')
ax.grid()
line, = ax.plot([], [], 'o-', lw=2) # Robot structure
path, = ax.plot([], [], 'c-', lw=1) # End-effector path
point, = ax.plot([], [], 'rx') # Target

# storing data
PPx = []
PPy = []
time = []
error1 = []
error2 = []

# Simulation initialization
def init():
    line.set_data([], [])
```

```
    path.set_data([], [])
    point.set_data([], [])
    return line, path, point

# Simulation loop
def simulate(t):
    global q, a, d, alpha, revolute, sigma1_d, sigma2_d, max_velocity
    global PPx, PPy, count

    # random position of the end-effector
    if t == 0:
        sigma1_d = np.array([np.random.uniform(-1.5,1.5),np.random.uniform
            (-1.5,1.5)]).reshape(2,1)
        count = count + 1

    # Update robot
    T = kinematics(d, q.flatten(), a, alpha)
    J = jacobian(T, revolute)
    #<<<<<<<<<<<<<<<<<<End-effector position task at the top of the hierarchy
        .>>>>>>>>>>>>>>>>>>>>>>>
    # # Update control
    # # TASK 1
    # sigma1 = T[-1][:2,3].reshape(2,1) # Current position of the end-effector
    # err1 =  sigma1_d - sigma1          # Error in Cartesian position
    # J1 =  J[:2,:3]                     # Jacobian of the first task
    # P1 =  np.eye(3) - np.linalg.pinv(J1) @ J1 # Null space projector

    # # TASK 2
    # sigma2 = q[0]                  # Current position of joint 1
    # err2 = sigma2_d - sigma2       # Error in joint position
    # J2 =  np.array([[1, 0, 0]])    # Jacobian of the second task
    # J2bar = J2 @ P1                # Augmented Jacobian
    # # Combining tasks
    # dq1 =  DLS(J1,0.1) @ err1       # Velocity for the first task
    # dq12 = dq1 + DLS(J2bar, 0.1) @ (err2 - J2 @ dq1)    # Velocity for both
        tasks

    # #velocity scalling
    # s = np.max(dq12/max_velocity)
    # if s > 1:
    #     dq12 = dq12/s

    # q = q + dq12 * dt # Simulation update
    #<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<Joint position task at the top of the
        hierarchy>>>>>>>>>>>>>>>>>>>
    # Update control
    # TASK 1
    sigma1 = q[0]                  # Current position of joint 1
    err1 = sigma2_d - sigma1       # Error in joint position
    J1 =  np.array([[1, 0, 0]])    # Jacobian of the second task
    P1 =  np.eye(3) - np.linalg.pinv(J1) @ J1 # Null space projector
```

```python


    # TASK 2
    sigma2 = T[-1][0:2,3].reshape(2,1) # Current position of the end-effector
    err2 =  sigma1_d - sigma2          # Error in Cartesian position
    J2 =  J[:2,:3]                     # Jacobian of the first task
    J2bar = J2 @ P1                    # Augmented Jacobian

    # Combining tasks
    dq1 =  DLS(J1,0.1) @ err1       # Velocity for the first task
    dq12 = dq1 + DLS(J2bar, 0.1) @ (err2 - J2 @ dq1) # Velocity for both tasks

    #velocity scalling
    s = np.max(dq12/max_velocity)
    if s > 1:
        dq12 = dq12/s

    q = q + dq12 * dt # Simulation update
    # Update drawing
    PP = robotPoints2D(T)
    line.set_data(PP[0,:], PP[1,:])
    PPx.append(PP[0,-1])
    PPy.append(PP[1,-1])
    path.set_data(PPx, PPy)
    point.set_data(sigma1_d[0], sigma1_d[1])
    time.append(t + 10 * count)
    error1.append(np.linalg.norm(err1))
    error2.append(np.linalg.norm(err2))

    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()

# Plotting simulation
fig = plt.figure()
# for end effector in the task 1
# plt.plot(time, error1, label = 'e1(end effector position)')
# plt.plot(time, error2, label = 'e2(Joint 1 position)' )
# for end effector in the task 2 and joint-1 in task 1
plt.plot(time, error1, label='e1 (Joint 1 position)')
plt.plot(time, error2, label='e2 (End-effector position)')
plt.ylabel('Error[m]') #Title of the Y axis
plt.ylabel('error[m]')
plt.xlabel('time[s]')
plt.title('Task Priority')
plt.grid(True)
plt.legend()
```

```
plt.show()
```