Task-Priority kinematic control

Raúl López Musito

07 March 2025

1 Introduction

This report presents the results obtained from simulating a 3-degree-of-freedom (DoF) robotic manipulator in a 2D environment while performing task-priority control. This control strategy utilizes the null space of the Jacobian associated with the first-priority task, taking advantage of the redundancy in the robot's kinematics to achieve secondary objectives without interfering with the primary task.

The Lab is divided into two main parts:

1. In the first part, the primary task is to maintain the end effector at a fixed position while applying a set of predefined joint velocities. This scenario tests the ability of the controller to reject disturbances and maintain the desired position.
2. In the second part, the primary task is to move the end effector to a random target position, while a secondary task aims to keep the first joint at 0°. The secondary task is executed within the null space of the Jacobian and is overridden when necessary to ensure successful completion of the primary task.

2 Exercise 1: Null Space Motions of a Planar 3-Link Manipulator

The first exercise focuses on executing a primary task while simultaneously applying a set of predefined joint velocities to all joints.

To accomplish this, the first step is to define the parameters of the robot. These parameters consist of five vectors, each of size $1 \times n$, where n is the number of joints in the robot (in this case, $n = 3$). The vectors are defined as follows:

- $\mathbf{b}$ – vector of displacements along the z-axis.
- $\mathbf{q}$ – vector of rotations around the z-axis.
- $\boldsymbol{\alpha}$ – vector of rotations around the x-axis.
- $\mathbf{a}$ – vector of displacements along the x-axis.
- **revolut** - vector indicating the type of joint

Once the robot is defined, the Denavit-Hartenberg parameters are arranged to generate the transformation matrices for all joints and stored in a matrix as follows:

$$T_i = \begin{bmatrix} R_i & O_i \\ 0 & 1 \end{bmatrix} \tag{1}$$

$$\mathbf{T}_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

$$T_i = T_{i-1} \cdot DH(b_i, q_i, a_i, \alpha_i) \text{In our case, } i \in \{1, 2, 3\} \tag{3}$$

$$T = \begin{bmatrix} T_1^0 & T_2^0 & T_3^0 \end{bmatrix} \tag{4}$$

With the corresponding transformation for each joint, we can use the velocity propagation method to compute the Jacobian:

$$\mathbf{J} = \begin{bmatrix} \mathbf{z_1} \times (O - O_1) & \mathbf{z_2} \times (O - O_2) & \mathbf{z_3} \times (O - O_3) \\ \mathbf{z_1} & \mathbf{z_2} & \mathbf{z_3} \end{bmatrix} \tag{5}$$

Where:

- O_i is the position vector of joint i, located in the top-right 3×1 part of the transformation matrix T_i, i.e., $T_i[0:3, 3]$.
- $\mathbf{z_i}$ is the unit vector along the axis of joint i, which is extracted from the third column of the rotation matrix $\mathbf{R_i}$, i.e., $\mathbf{z_i} = T_i[0:3, 2]$.
- O is the position of the end-effector, which is $T_n[0:3, 3]$, the last column of the final transformation matrix.

In order to complete the first task, we need the error. Since the controlled variable is the end-effector position, we can easily extract the current one from the last transformation in $\mathbf{T}$, where $\sigma = \mathbf{T}[-1][:2, 3]$ corresponds to the first two elements of the last column of $\mathbf{T}$ (x, y).

Having current sigma is possible to compute the error of the system by:

$$\mathbf{e} = \sigma_{\text{desired}} - \sigma \tag{6}$$

With the error computed, we can now accomplish the first task by using the relationship between the end effector velocity, with the Jacobian and the joint velocity:

$$\dot{\mathbf{e}} = \mathbf{J}\dot{\mathbf{q}} \tag{7}$$

$$\dot{q} = \mathbf{J}^{+}\dot{e} \tag{8}$$

- $\mathbf{e}$ is the error.
- σ_{desired} is the desired end-effector position.
- σ is the current end-effector position.
- $\mathbf{J}$ is the Jacobian matrix.
- $\dot{q}$ is the joint velocity.
- $\mathbf{J}^{+}$ is the pseudo inverse of the jacobian.

Once we have fully defined the first task, we must apply the predefined set of joint velocities. These velocities are added to the computed velocities of the first task. However, this addition should be done in such a way that it does not interfere with the motion required by the first task. To achieve this, we "filter" the velocities through the projection of the first task's null space. This filtering ensures that the additional velocities, which belong to the null space, do not affect the first task's motion, allowing for task priority control.

The projection matrix $\mathbf{P_1}$ is computed using the Jacobian of the first task, and is given by:

$$\mathbf{P_1} = \mathbf{I} - \mathbf{J_1}^{+}\mathbf{J_1} \tag{9}$$

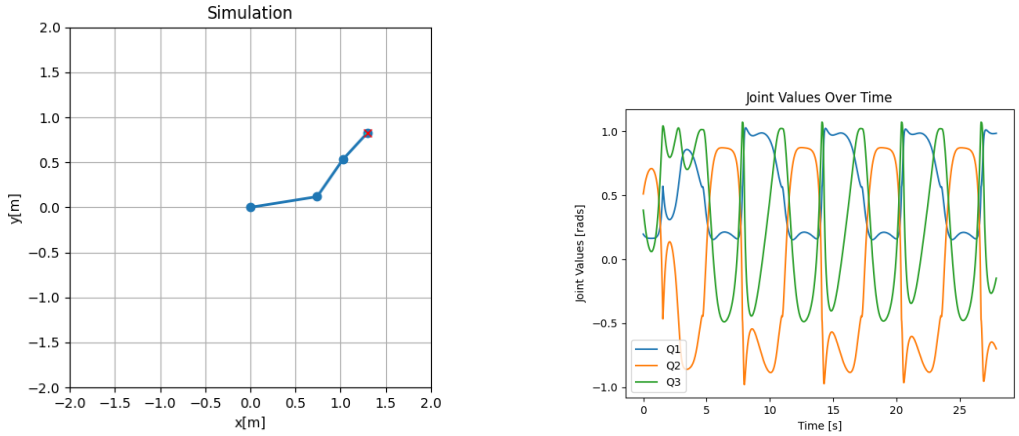

Figure 1: Robot simulation and evolution of robot's joints positions over time

The null space projector is used to ensure that the additional joint velocities only affect the joints in directions that do not influence the motion of the end-effector in the first task. The final joint velocity for both tasks can be computed by adding the joint velocities from the first task and the second task as:

$$\dot{q} = \mathbf{J_1}^{+}\dot{e_1} + \mathbf{P_1}\dot{q_2} \tag{10}$$

Where:

- $\dot{e_1}$ is the error velocity of the first task.
- $\dot{q_2}$ is the predefined joint velocities.

By using this approach, we ensure that the first task's motion is prioritized while still allowing the second task to proceed in the null space of the first task.

This can be done because the robot has more degrees of freedom (DOF) than the task space requires. As a result, the system is kinematically redundant. Kinematic redundancy allows for the robot to move in directions that do not affect the end-effector's position or orientation, which is crucial when performing multiple tasks with different priorities. The additional DOF enable the robot to execute secondary motions in the null space of the primary task's Jacobian, allowing for the simultaneous accomplishment of both tasks without interference.

In Figure 1, we can observe how the robot continuously modifies the positions of all its joints over time, rather than remaining stationary at a single configuration. The velocities applied to the joints follow periodic functions, specifically sine, cosine, and tangent, with respect to the simulation time. This periodic pattern is clearly visible in the evolution of the joint positions, as each joint moves in a repetitive manner. Despite the movements of the joints, the end-effector always remains in the desired position, demonstrating that the primary task is successfully accomplished, even while the robot performs secondary motions in the null space.

```
# 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([0.2,0.5,0.4]).reshape(3,1)         # rotation around Z-axis (theta)
alpha = np.zeros(3)                              # displacement along X-axis
```

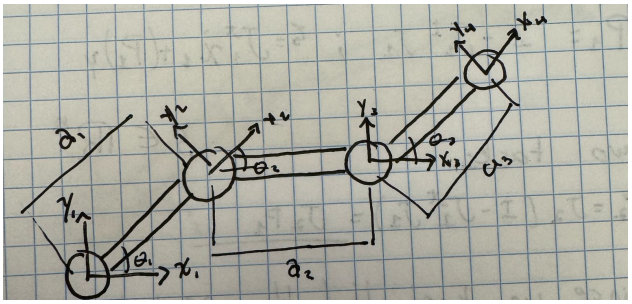

Figure 2: Robot parameters

```
a = np.array([0.75, 0.5, 0.4])                    # rotation around X-axis
revolute = [True,True,True]                       # flags specifying the type of joints

# 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 = 60 # 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
PPx = []
PPy = []
qvec1 = []
qvec2 = []
qvec3 = []
# 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, sigma_d
    global PPx, PPy

    # Update robot
```

```
    T = kinematics(d, q.flatten(), a, alpha)
    J = jacobian(T, revolute)[:2,:]


    # Update control
    sigma = T[-1][:2, 3].reshape(2,1)                    # Current position of the end-e

    err =  sigma_d - sigma                    # Error in position
    Jbar = J                                  # Task Jacobian
    pin_J = np.linalg.pinv(J)                 # Pseudo-inverse of the Jacobian
    P =  np.eye(3) - pin_J@ J          # Null space projector
    y = np.array([0*np.sin(t), 1*np.cos(t), 0*np.tan(t)]).reshape(3,1)       # Arbitrary j
    dq = np.linalg.pinv(Jbar)@ err + P @ y # Control signal
    q = q + dt * dq # Simulation update
    qvec1.append(q[0][0]) #append(q) # Save joint values for plotting
    qvec2.append(q[1][0]) #append(q) # Save joint values for plotting
    qvec3.append(q[2][0]) #append(q) # Save joint values for plotting


    # 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()

# Plot error distances
plt.figure()
plt.plot(tt[:len(qvec1)], qvec1, label="Q1")
plt.plot(tt[:len(qvec2)], qvec2, label="Q2")
plt.plot(tt[:len(qvec3)], qvec3, label="Q3")
plt.title('Joint Values Over Time')
plt.xlabel('Time [s]')
plt.ylabel('Joint Values [rads]')

plt.legend()
plt.show()
```

3 Exercise 2

For the second part, instead of applying a predefined vector of velocities, a second task is applied. This second task is considered the low-priority task. Therefore, the second task, as previously mentioned, will only take place in the null space of the Jacobian of the first task. To achieve this, we multiply the Jacobian of the second task by the projection of the null space of the Jacobian from the first task. This allows us to ensure that the second task only affects the system in the null space of the first task, leaving the first task

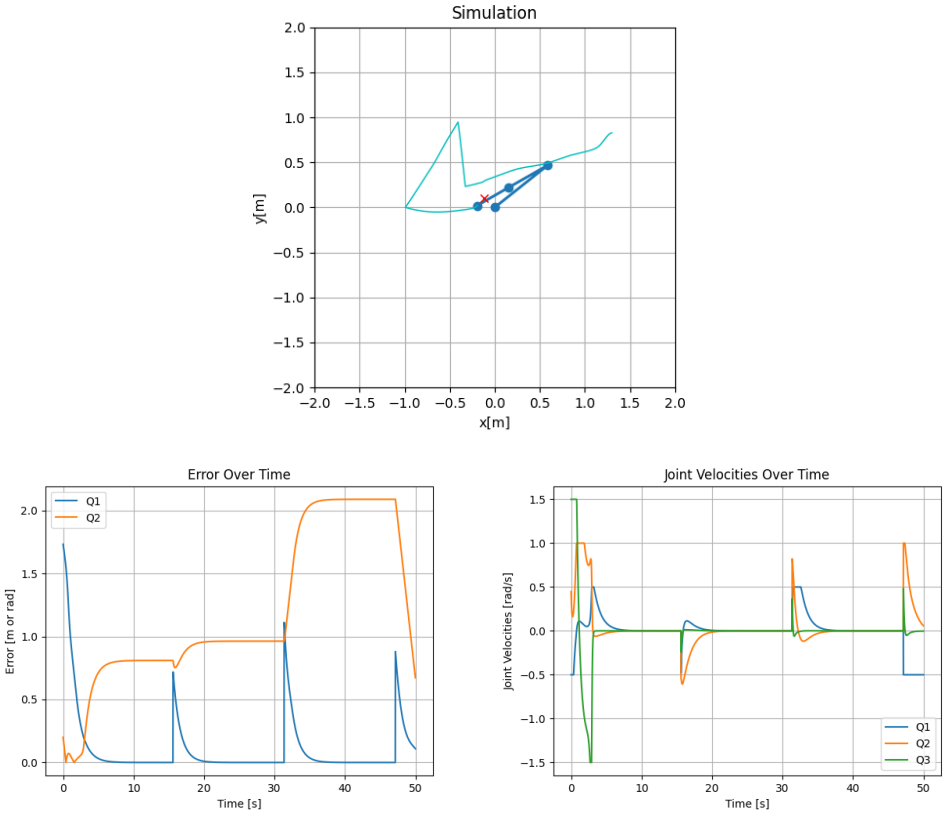

Figure 3: End effector position as High priority task, clipped velocities.

unchanged. This second task requires the first joint to remain = 0

Null Space Projection Matrix P_1:

$$P_1 = I - J_1^{\dagger} J_1$$

Where $J_1^{\dagger}$ is the pseudoinverse of J_1 and I is the identity matrix of appropriate size.

Augmented Jacobian for Combined Tasks:

$$J_2^{\text{bar}} = J_2 P_1$$

$$\dot{q}_{12} = \dot{q}_1 + \text{DLS}(J_2^{\text{bar}}, 0.2)\,((\mathbf{e}_2 - J_2 \dot{q}_1))$$

Where: $\dot{q}_1$ is the joint velocity for the first task, $\dot{q}_{12}$ is the combined joint velocity for both tasks, DLS$(\cdot)$ is the Damped Least Squares (DLS) method, J_2^{bar} is the augmented Jacobian for the second task, $\mathbf{e}_2$ is the error of the second task.

Additionally, a velocity clipping was applied for each joint velocity to ensure that the robot does not reach unsuitable or excessive velocities. As seen in Figure 3, the trajectories of the simulation are not as smooth or straight as those shown in Figure 4. This difference is due to the velocity limitations imposed by the clipping, which restricts the maximum velocity for each joint, resulting in a less idealized but safer motion profile.

Lastly, the priorities of the two tasks were swapped. This means that the task of keeping the first joint at 0 degrees became the highest priority, while the goal position task was relegated to a lower priority. As a result, the robot was unable to reach the goal in several instances, as the first joint was no longer utilized for positioning. This behavior can be observed in the simulation shown in Figure 5. Furthermore, in Figure 5,

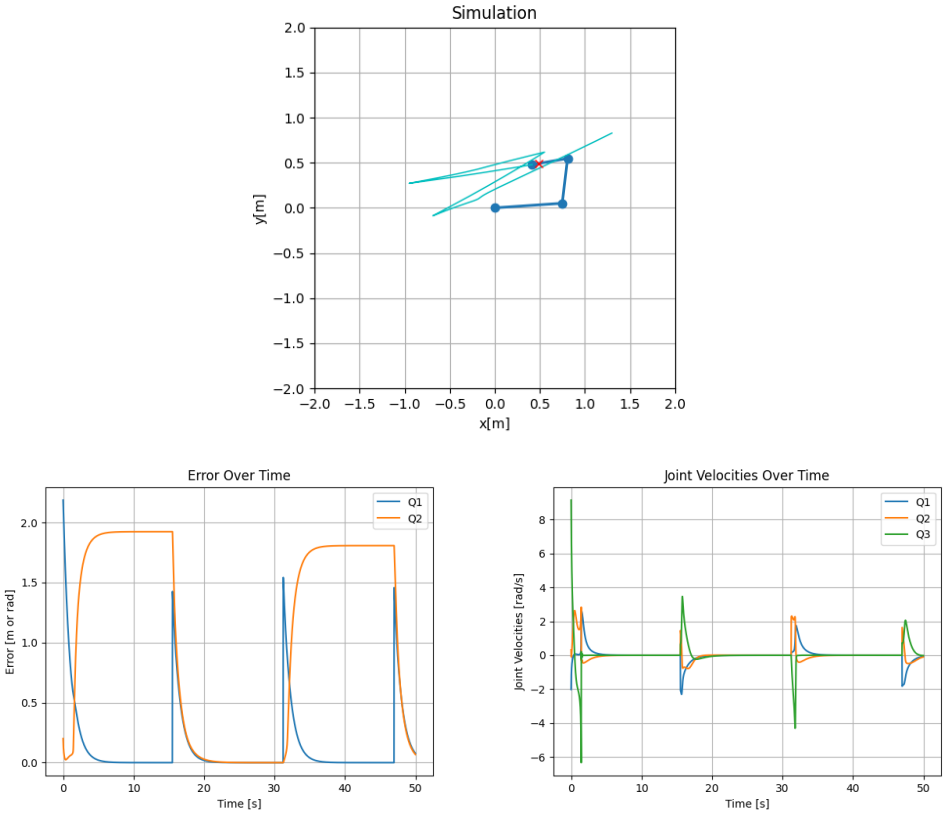

Figure 4: End effector position as High priority task, Not clipped velocities.

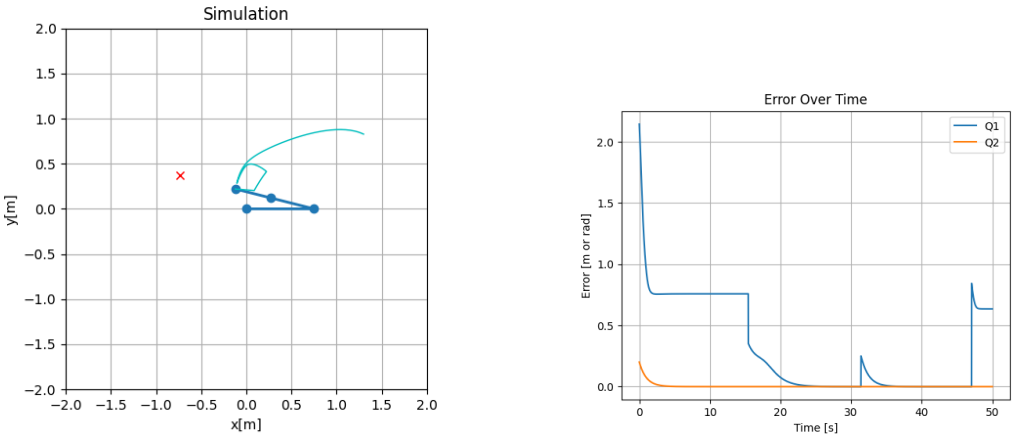

Figure 5: End effector position as Low priority task.

we can see the error over time for both tasks. The error in the joint angle approaches 0 and remains there, while the error in position stabilizes at a constant value when the goal is not achievable. In contrast, when the goal was achievable (as in the first priority task scenario), the position error would always decrease to 0, and the joint angle error would increase only when necessary to maintain the system's constraints.

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

# Robot definition (3 revolute joint planar manipulator)
d = np.zeros(3)                                  # displacement along Z-axis
q = np.array([0.2,0.5,0.4]).reshape(3,1)         # rotation around Z-axis (theta)
alpha = np.zeros(3)                              # rotation around X-axis
a = np.array([0.75, 0.5, 0.4])                   # displacement along X-axis
revolute = [True,True,True]                      # flags specifying the type of joints

# Desired values of task variables

def set_random_goal():
    sigma1_d = np.random.rand(2,1)*2-1
    return sigma1_d

start_time = time.time() # Start time
sigma1_d = set_random_goal() # Position of the end-effector
sigma2_d = np.array([[0.0]]) # Position of joint 1

# Simulation params
dt = 1.0/60.0
Tt = 50 # 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
PPx = []
PPy = []

errorvec1 = [] # vector containing the norm error of the end effector position of each ite
errorvec2 = []  # vector containing the error of the joint position of each iteration

velocity_q1 = [] # vector containing the velocity of joint 1 of each iteration
velocity_q2 = [] # vector containing the velocity of joint 2 of each iteration
velocity_q3 = [] # vector containing the velocity of joint 3 of each iteration
```

```
# Limit the maximum joint velocity
#max_joint_velocity = 1 # rad/s
dq_max = np.array([.5  ,1  ,1.5]).reshape(3,1)  # Suponiendo 3 DOF

# 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, start_time
    global PPx, PPy

    # Update robot
    T = kinematics(d, q.flatten(), a, alpha)
    J = jacobian(T, revolute)[:2,:]

    # Update control
    # TASK 1
    sigma1 = T[-1][:2, 3].reshape(2,1)                    # Current position of the end-ef
    err1 = sigma1_d - sigma1                              # Error in Cartesian position
    J1 = J                                                # Jacobian of the first task
    P1 = (np.eye(3)-np.linalg.pinv(J1)@J1)                # Null space projector

    # TASK 2
    coord = T[1][:2, 3].reshape(2,1)                      # Current position of joint 1
    sigma2 = m.atan2(coord[1], coord[0])
    err2 = sigma2_d - sigma2                              # Error in joint position
    J2 =  np.array([[1,0,0]])                             # Jacobian of the second task
    P2 = (np.eye(3)-np.linalg.pinv(J2)@J2)                # Null space projector

    # Save error values
    errorvec1.append(m.sqrt(err1[0]**2+err1[1]**2))
    errorvec2.append(abs(err2[0]))

    """                          Combining tasks
"""
    """         Using the speudo inverse didnt work, so we used the DLS method
"""
    """                 Uncomment one of the two options below
"""


    """ First task = end effector position | Second task = joint position """
#-----------|
    #dq1 = DLS(J1, 0.1)@ err1                                          # Velocities fo
|
    #J2bar = (J2@P1)                                                   # Augmented Jac
|
    #dq12 = dq1 + DLS(J2bar, 0.2)@ ((err2-J2@dq1).reshape(1,1))        # Velocity for
|
```

```
                                                                    #
|
    """ First task = joint position | Second task = end effector position """#
|
    dq2 = DLS(J2, 0.1)@ err2                                        # Velocities fo
|
    J1bar = (J1@P2)                                                 # Augmented Ja
|
    dq12 = dq2 + DLS(J1bar, 0.2)@ ((err1-J1@dq2).reshape(2,1))      # Velocity for
|



    # Clip the joint velocities
    dq12 = np.clip(dq12, -dq_max, dq_max)

    # Save joint velocities
    velocity_q1.append(dq12[0][0])
    velocity_q2.append(dq12[1][0])
    velocity_q3.append(dq12[2][0])


    q = q + dq12 * dt # Simulation update

    current_time = time.time()

    # Verify 10 sec
    if (current_time - start_time) >= 10 or errorvec1[-1] < 0.1:
        sigma1_d = set_random_goal()
        start_time = current_time  # Reiniciar el tiempo


    # 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])

    return line, path, point

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

# Plot error distances
plt.figure()
plt.plot(tt[:len(errorvec1)], errorvec1, label="Q1")
plt.plot(tt[:len(errorvec2)], errorvec2, label="Q2")
plt.title('Error Over Time')
plt.xlabel('Time [s]')
plt.ylabel('Error [m or rad]')
```

```
plt.grid()
plt.legend()
plt.show()

# Plot joint velocities
plt.figure()
plt.plot(tt[:len(velocity_q1)], velocity_q1, label="Q1")
plt.plot(tt[:len(velocity_q2)], velocity_q2, label="Q2")
plt.plot(tt[:len(velocity_q3)], velocity_q3, label="Q3")
plt.title('Joint Velocities Over Time')
plt.xlabel('Time [s]')
plt.ylabel('Joint Velocities [rad/s]')
plt.grid()
plt.legend()
plt.show()
```