Hands-on Intervention

Lab #5 – Task Priority Kinematic Control (2A)

Delivered by:
Rihab Laroussi, u1100330

Supervisor:
Patryk Cieślak

Date of Submission:
23/03/2025

Contents

1 Introduction

In the previous lab, I extended the Task-Priority (TP) algorithm to its recursive form to control a planar 3-link manipulator, enabling the handling of an arbitrary hierarchy of tasks. In this lab, I will introduce obstacles into the workspace and develop an obstacle avoidance task to enhance the robot's ability to navigate complex environments.
In Exercise 1, I implement and simulate two set-based tasks, focusing on obstacle avoidance and end-effector positioning. In Exercise 2, I test the joint limits task, ensuring that the robot operates within predefined safe joint ranges.

2 Exercice 1

2.1 Methodology

The robot used in this lab is a 3-link planar manipulator with three revolute joints. It has four coordinate systems:

- O_0: Base frame (fixed reference frame).
- O_1, O_2, O_3: Frames attached to Joint 1, Joint 2, and Joint 3, respectively.
- O_4: End-effector frame.

The robot's kinematic structure is defined using the Denavit-Hartenberg (DH) parameters, as shown in Table 1.

Table 1: Denavit-Hartenberg Parameters

Link	θ (rad)	d (m)	a (m)	α (rad)
1	0	0	0.5	0
2	0.6	0	0.75	0
3	0.3	0	0.5	0

The drawing of the robot model with its DH parameters and coordinate systems is shown in :

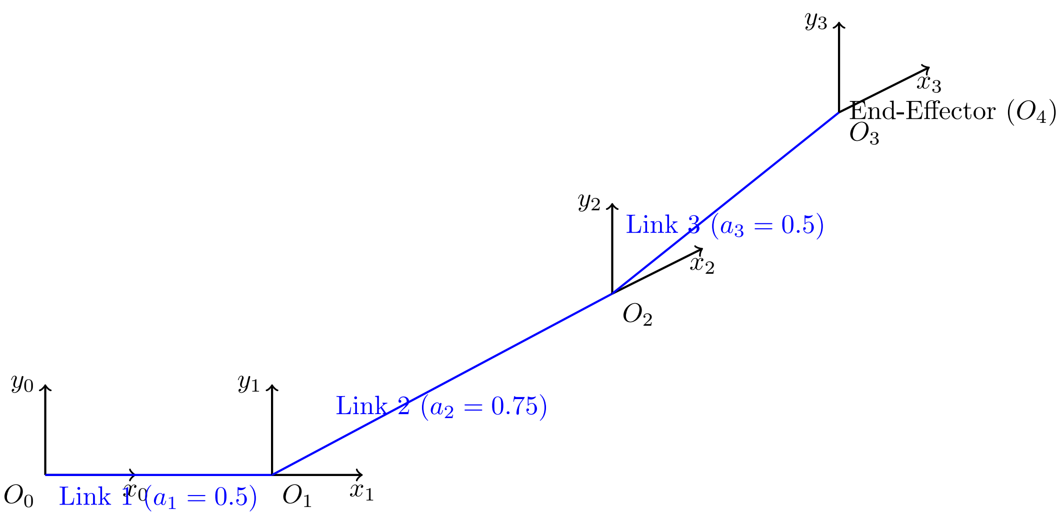


Figure 1: Robot model with DH parameters and coordinate systems.

To implement a set-based obstacle avoidance task for the manipulator, I first defined Obstacle2D task as a subclass of the base Task class. The task ensures that the robot maintains a safe distance from obstacles in the workspace.

First, I computed the Jacobian using using the end-effector Jacobian, as the task depends on the position of the end-effector relative to the obstacle, and it is given by:

$$J_r = J_r(\mathbf{q}) = J_v(\mathbf{q}) \in R^{3\times n}$$

Then the error is defined as the distance between the current position and the obstacle position. The error vector is computed as:

$$\dot{x}_\mu(q) = \frac{\eta_{1,ee}(q) - P}{|\eta_{1,ee}(q) - P|}$$

The task is activated when the end-effector enters a predefined safety zone around the obstacle. The activation and deactivation thresholds are defined as:

- r_α: Activation threshold (distance at which the task becomes active).
- r_δ: Deactivation threshold (distance at which the task becomes inactive).

The required safe distance is calculated as:

$$\sigma_r = \sigma_r(q) = |\, \eta_{1,ee}(q) - P \,|$$

I implemented the task activation as follows:

$$a_r(\mathbf{q}) = \begin{cases} 1, & a_r = 0 \wedge |\eta_{1,ee}(\mathbf{q}) - P| \leq r_\alpha \\ 0, & a_r = 1 \wedge |\eta_{1,ee}(\mathbf{q}) - P| \geq r_\delta \end{cases}$$

Keep in mind that to avoid chattering (rapidly switching between active and inactive states), take $r_\delta > r_\alpha$.

Next, I applied the recursive formulation of the TP algorithm, which computes the joint velocities $\dot{q}$ required to achieve multiple tasks while respecting their priorities. I defined one end-effector position task at [1.0, 0.5], and three obstacle tasks each with a different position:

```
obstacle_pos1 = [0.0, 1.0]
obstacle_pos2 = [1.0, -0.5]
obstacle_pos3 = [-0.5, -1.0]
```

The task-priority algorithm is implemented as follows:

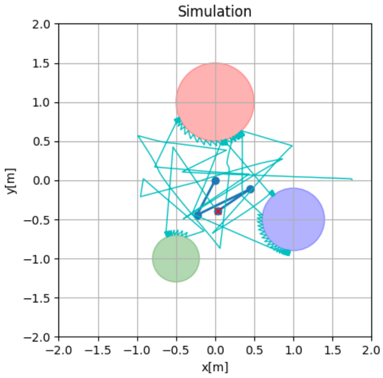
Algorithm 1 Extended Recursive Task-Priority Algorithm

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

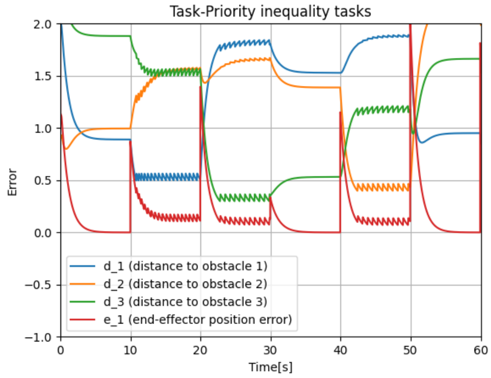
Initialize $\zeta_0 = 0^n$, $P_0 = I^{n\times n}$
for $i \in 1...k$ **do**
  if $a_i(q) \neq 0$ **then**
    $J_i(q) = J_i(q)P_{i-1}$
    $\zeta_i = \zeta_{i-1} + J_i^+(q)(a_i(q)\dot{x}_i(q) - J_i(q)\zeta_{i-1})$
    $P_i = P_{i-1} - J_i^+(q)J_i(q)$
  else
    $\zeta_i = \zeta_{i-1}$, $P_i = P_{i-1}$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

2.2 Results

The results below are visualised using Matplotlib, showing the robot's motion and the evolution of the end-effector position error plus obstacle distances over time, showing effective obstacle avoidance. The labels d1, d2, and d3 are the distances between the robot's end effector and each obstacle in the workspace.



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



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

Figure 2: Task1:Obstacle2D & Task2: Position2D.

As we can see, the simulation results demonstrate the effectiveness of the recursive TP algorithm in handling inequality tasks. The robot successfully navigates around all three obstacles while accurately reaching the desired end-effector position. The second plot highlights the trade-off between the robot's proximity to obstacles and its positioning accuracy. Ideally, the robot maintains a safe distance from obstacles while minimizing the end-effector position error, ensuring both safety and precision in its motion.

3 Exercice 2

3.1 Methodology

In this exercise, I implemented another set-based task: the joints limits task. I first defined JointLimits as a subclass of Task class , where the Jacobian for the joint limits task is defined for joint 1.

Since the task only affects joint 1, the Jacobian is a row vector: J = [1,0,0].

The task activation logic is implemented as follows:

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

Where $q_{i,\max}$ is the maximum safe-set, $q_{i,\min}$ is the minimum safe-set, α is the activation threshold, and δ is the deactivation threshold. Keep in mind that $\delta > \alpha$ is used to avoid chatter.

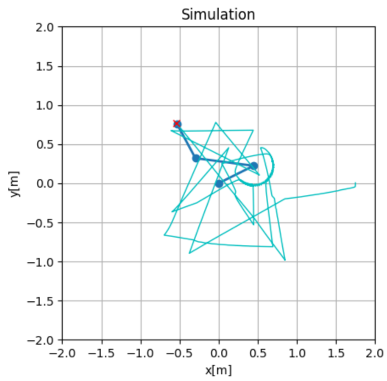
I defined a task hierarchy with two tasks:

Joint Limits Task: Ensures that joint 1 stays within the safe set [0.5,0.5].
End-Effector Position Task: Moves the end-effector to the desired position [1.0,0.5].

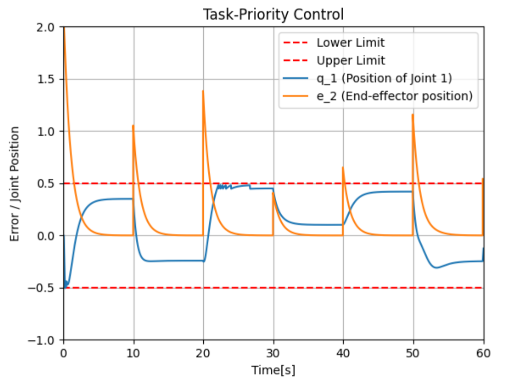
The recursive TP algorithm is applied to compute the joint velocities $\dot{q}$ while respecting the priorities of the tasks.

3.2 Results

The results below are visualised using Matplotlib, showing the robot's motion and the evolution of the end-effector position error plus joint 1 position over time, showing effective joint limit enforcement.



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



(b) Evolution of the end-effector position error and the 1st joint position (including limits).

Figure 3: Task1:JointLimits & Task2: Position2D.

We observe the robot maintains joint 1 within the safe set [0.5,0.5] while reaching the desired end-effector position. Two red dotted lines describe the safe operational zone for the joint limits tasks. As the joint near its maximum or minimum boundaries, oscillations in the end-effector (EE) position error become noticeable, and it is because the control system continuously adjusts to maintain joint safety while also striving to reach the target position. The controller balances these objectives, and occasionally results in minor deviations from the ideal trajectory, emphasizing the complexity of managing multiple constraints simultaneously.

4 Code

4.1 Lab4robotics.py code

```python
def jacobianLink(T, revolute, link):
    '''
    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
    '''
    J = np.zeros((6, len(T)-1 ))  # Initialize the Jacobian matrix with zeros
    O_link = np.array([T[link][:3, 3]]).T  # Position of the link's frame in the base frame
    Z = np.array([[0, 0, 1]]).T  # Z-axis of the base frame
    for i in range(link):
        Ri = T[i][:3, :3]
        Oi = np.array([T[i][:3, 3]]).T
        Zi = Ri @ Z
        if revolute[i]:
            J[:3, i] = np.cross(Zi.T, (O_link - Oi).T).T[:, 0]
            J[3:, i] = Zi[:, 0]
        else:
            J[:3, i] = Zi[:, 0]
    return J
```

```python

class Manipulator:
    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)

    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)

    def drawing(self):
        return robotPoints2D(self.T)

    def getEEJacobian(self):
        return jacobian(self.T, self.revolute)

    def getEETransform(self):
        return self.T[-1]

    def getJointPos(self, joint):
        return self.q[joint, 0]

    def getDOF(self):
        return self.dof

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

    def getJacobianLink(self, link):
        return jacobianLink(self.T, self.revolute, link)

class Task:
    def __init__(self, name, desired):
        self.name = name
        self.sigma_d = desired
        self.error = []
        self.K = None
        self.feedforward = None

    def update(self, robot):
        pass

    def setDesired(self, value):
        self.sigma_d = value

    def getDesired(self):
        return self.sigma_d

    def getJacobian(self):
        return self.J

    def getError(self):
        return self.err

    def setFeedforward(self, value):
        self.feedforward = np.ones(self.sigma_d.shape) * value

    def getFeedforward(self):
        return self.feedforward

    def setGainMatrix(self, value):
        self.K = self.K * value
```

```python

    def getGainMatrix(self):
        return self.K

    def isActive(self):
        return 1

class Position2D(Task):
    def __init__(self, name, desired):
        super().__init__(name, desired)
        self.J = np.zeros((len(desired), 3))
        self.err = np.zeros(desired.shape)

    def update(self, robot):
        self.J = robot.getEEJacobian()[: len(self.sigma_d), :]
        sigma = robot.getEETransform()[: len(self.sigma_d), 3].reshape(self.sigma_d.shape)
        self.err = self.getDesired() - sigma
        self.error.append(np.linalg.norm(self.err))

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

    def update(self, robot):
    #-----------------------------Exercice 1-----------------------------------
        self.J = robot.getEEJacobian()[-1, :].reshape(1,3)          # Update task Jacobian
        sigma = np.arctan2(robot.getEETransform()[1,0],robot.getEETransform()[0,0])  # Get the current orientation of the end-effector (theta)
        self.err = self.getDesired() - sigma.reshape(self.sigma_d.shape)              # Update task error
        self.error.append(np.linalg.norm(self.err))   # Append the norm of the error for tracking

'''
    Subclass of Task, representing the 2D configuration task.
'''
class Configuration2D(Task):
    def __init__(self, name, desired, link = 2):
        super().__init__(name, desired)
        self.J = np.zeros((3,3))    # Initialize with proper dimensions
        self.err = np.zeros(desired.shape) # Initialize with proper dimensions
        self.link = link
        self.feedforward = np.zeros(desired.shape)  #3,1
        self.K = np.zeros(len(desired)) # 3

    def update(self, robot):
        #-----------------------------Exercice 1-----------------------------------
        self.J = robot.getEEJacobian()[[0, 1, -1], :]     # Update task Jacobian
        sigma_p = robot.getEETransform()[:2, -1]
        sigma_o = np.arctan2(robot.getEETransform()[1,0],robot.getEETransform()[0,0]) # Get the current orientation of the end-effector (theta)
        sigma = np.block([sigma_p, sigma_o])        # Combine position and orientation into a single task variable
        self.err = self.getDesired() - sigma.reshape(3,1)
        self.error.append(np.linalg.norm(self.err)) # Append the norm of the error for tracking'''
'''
    Subclass of Task, representing the joint position task.
'''
# class JointPosition(Task):

class Joint_Position(Task):
    def __init__(self, name, desired, joint):
```

```python
        super().__init__(name, desired)
        self.J = np.zeros((1,3))    # Initialize with proper dimensions
        self.err = np.zeros(desired.shape) # Initialize with proper dimensions
        self.Joint = joint
        self.feedforward = np.zeros(desired.shape)  # 2,1
        self.K = np.zeros(len(desired)) # 2

    def update(self, robot):
        self.J[0,self.Joint] = 1    # Update task Jacobian
        sigma = robot.getJointPos(self.Joint)          # Get the current position of the joint
        self.err = self.getDesired() - sigma
        self.error.append(np.linalg.norm(self.err))     # Append the norm of the error for tracking

    """
    Subclass of Task,  a class representing a 2D obstacle avoidance task.
    This task ensures that the robot's end-effector maintains a safe distance from a circular obstacle.
    """
class Obstacle2D(Task):

    def __init__(self, name, obstacle_pos,obstacle_r):
        super().__init__(name,None)
        self.J = np.zeros((2,3))
        self.err = np.zeros((2,1))
        self.obstacle_pos = obstacle_pos
        self.r = obstacle_r
        self.active = 0
        self.distance = 0


    def isActive(self):
        """
        Override the default isActive method to return the task's activation status.
        """
        return self.active

    def update(self, robot):
        # Update task Jacobian (2x3)
        self.J = robot.getEEJacobian()[:2, :]

        # Get the current position of the end-effector (2x1)
        sigma = robot.getEETransform()[:2, 3].reshape(2, 1)

        # Update task error (2x1),
        # Error is the normalized vector pointing from the obstacle to the end-effector
        self.err = (sigma - self.obstacle_pos) / np.linalg.norm(sigma - self.obstacle_pos)

        # Update task activation status
        self.distance = sigma - self.obstacle_pos
        if self.active == 0 and np.linalg.norm(self.distance) < self.r[0]:
            self.active = 1  # Activate the task
        elif self.active == 1 and np.linalg.norm(self.distance) > self.r[1]:
            self.active = 0  # Deactivate the task

        self.error.append(np.linalg.norm(self.distance)) # Append the norm of the error for tracking'''

    """
    Subclass of Task, a class representing a joint limits task.
    This task ensures that a joint stays within predefined safe limits.
    """
class JointLimits(Task):

    def __init__(self, name, safe_set, limit):
        super().__init__(name,0)
        self.J = np.zeros((1,3))   # Task Jacobian (1x3 matrix)
        self.err = np.zeros((1,1)) # Task error (1x1 vector)
        self.safe_set = safe_set  # Safe operational range [q_min, q_max]
        self.limit = limit   # Activation and deactivation thresholds [alpha, sigma]
        self.active = 0
```

```


    def update(self, robot):
        self.J[0,0] = 1    # Update task Jacobian (only affects joint 1)
        sigma = robot.getJointPos(0)   # Get the current position of joint 1
        self.err = 1 * self.active      # Error is 1 if active, 0 otherwise

        # Task activation logic
        if self.active == 0 and sigma >= self.safe_set[1] - self.limit[0] :
            self.active = -1     # Activate task (joint approaching upper limit)
        elif self.active == 0 and sigma <= self.safe_set[0] + self.limit[0]:
            self.active = 1      # Activate task (joint approaching lower limit)
        elif self.active == -1 and sigma <= self.safe_set[1] - self.limit[1]:
            self.active = 0       # Deactivate task (joint moving away from upper limit)
        elif self.active == 1 and sigma >= self.safe_set[0] + self.limit[1]:
            self.active = 0       # Deactivate task (joint moving away from lower limit)

        self.error.append(robot.getJointPos(0))   # Append the joint position for
    tracking

    def isActive(self):
        """
       Override the default isActive method to return the task's activation status.
         """
        return self.active
```

4.2 Exercice1.py code

```

from lab4_robotics import * # Includes numpy import
import matplotlib.pyplot as plt
import matplotlib.animation as anim
import matplotlib.patches as patch

# Robot model
d = np.zeros(3)                       # displacement along Z-axis
theta = np.array([0,0.6,0.3])         # rotation around Z-axis
alpha = np.zeros(3)                   # rotation around X-axis
a = np.array([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

# Task hierarchy definition
# Define obstacle positions and radii
obstacle_pos1 = np.array([0.0, 1.0]).reshape(2,1)
obstacle_pos2 = np.array([1.0, -0.5]).reshape(2,1)
obstacle_pos3 = np.array([-0.5, -1.0]).reshape(2,1)
obstacle_r1 = 0.5
obstacle_r2 = 0.4
obstacle_r3 = 0.3

# Define tasks: obstacle avoidance and end-effector position control
tasks = [
          Obstacle2D("Obstacle avoidance", obstacle_pos1, np.array([obstacle_r1,
    obstacle_r1+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))
        ]

# Simulation params
dt = 1.0/60.0

# 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]')
ax.add_patch(patch.Circle(obstacle_pos1.flatten(), obstacle_r1, color='red', alpha=0.3))
ax.add_patch(patch.Circle(obstacle_pos2.flatten(), obstacle_r2, color='blue', alpha=0.3)
    )
ax.add_patch(patch.Circle(obstacle_pos3.flatten(), obstacle_r3, color='green', alpha
    =0.3))
line, = ax.plot([], [], 'o-', lw=2) # Robot structure
path, = ax.plot([], [], 'c-', lw=1) # End-effector path
point, = ax.plot([], [], 'rx') # Target

# Global variables for storing end-effector path and simulation time
PPx = []
PPy = []
time = []

# Simulation initialization
def init():
    global tasks, i
    line.set_data([], [])
    path.set_data([], [])
    point.set_data([], [])
    tasks[-1].setDesired(np.random.uniform(-1,1,size = (2,1)))   # Random position
    if time:
       i = time[-1]  # Continue time from the last simulation
    else: i = 0
    return line, path, point

# Simulation loop
def simulate(t):
    global tasks
    global robot
    global PPx, PPy

    ### Recursive Task-Priority algorithm (w/set-based tasks)
    # The algorithm works in the same way as in Lab4.
    # The only difference is that it checks if a task is active.
    ###
    # Initialize null-space projector
    P = np.eye(robot.getDOF())
    # Initialize output vector (joint velocity)
    dq = np.zeros((robot.getDOF(),1))
    # Loop over tasks

    for task in tasks:
        # Update task state
        task.update(robot)
        if task.isActive():
            # Compute augmented Jacobian
            J = task.getJacobian()
            J_bar = J @ P
            # Compute task velocity
            dq_acc = DLS(J_bar, 0.1) @ ((task.getError()) - (J @ dq))
            # Accumulate velocity
            dq += dq_acc
            # Update null-space projector
            P = P -np.linalg.pinv(J_bar) @ J_bar

    # Update robot
    robot.update(dq, dt)

    # Update drawing
    PP = robot.drawing()
    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(tasks[-1].getDesired()[0], tasks[-1].getDesired()[1])

    time.append(t+i)    # Store simulation time

    return line, path, point

# Run simulation
```

```python
animation = anim.FuncAnimation(fig, simulate, np.arange(0, 10, dt),
                                interval=10, blit=True, init_func=init, repeat=True)
plt.show()
# Evolution of task errors over time
fig_joint = plt.figure()
ax = fig_joint.add_subplot(111, autoscale_on=False, xlim=(0, 60), ylim=(-1, 2))
ax.set_title("Task-Priority inequality tasks")
ax.set_xlabel("Time[s]")
ax.set_ylabel("Error")
ax.grid()
# Plot task errors over time
plt.plot(time, tasks[0].error, label="d_1 (distance to obstacle 1)")
plt.plot(time, tasks[1].error, label="d_2 (distance to obstacle 2)")
plt.plot(time, tasks[2].error, label="d_3 (distance to obstacle 3)")
plt.plot(time, tasks[-1].error, label="e_1 (end-effector position error)")
ax.legend()
plt.show()
```

4.3 Exercice2.py code

```python

from lab4_robotics import *  # Includes numpy import
import matplotlib.pyplot as plt
import matplotlib.animation as anim
import matplotlib.patches as patch

# Robot model
d = np.zeros(3)  # Displacement along Z-axis
theta = np.array([0, 0.6, 0.3])  # Rotation around Z-axis
alpha = np.zeros(3)  # Rotation around X-axis
a = np.array([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

# Task hierarchy definition
tasks = [
    JointLimits("Position of Joint 1", np.array([-0.5, 0.5]), np.array([0.02, 0.05])),
    Position2D("End-effector position", np.array([1.0, 0.5]).reshape(2, 1))
]

# Simulation params
dt = 1.0 / 60.0

# 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
# Global variables for storing end-effector path and simulation time
PPx = []
PPy = []
time = []


# Simulation initialization
def init():
    global tasks, i
    line.set_data([], [])
    path.set_data([], [])
    point.set_data([], [])
    tasks[-1].setDesired(np.random.uniform(-1,1,size = (2,1)))   # Random position
    if time:
        i = time[-1]  # Continue time from the last simulation
    else:
        i = 0
    return line, path, point
```

```python
# Simulation loop
def simulate(t):
    global robot, tasks, PPx, PPy, i
    # Run the Recursive Task-Priority algorithm
    P = np.eye(robot.getDOF())  # Initialize the projector matrix
    dq = np.zeros((robot.getDOF(), 1))  # Initialize joint velocities

    # Loop over tasks, updating each and applying the control law
    for task in tasks:
        task.update(robot)  # Update the task's internal state
        if task.isActive() != 0:
            J = task.getJacobian()
            J_bar = J @ P
            dq_acc = DLS(J_bar, 0.1) @ ((task.getError()) - (J @ dq))
            dq += dq_acc
            P = P - np.linalg.pinv(J_bar) @ J_bar

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

    # Update drawing
    PP = robot.drawing()
    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(tasks[-1].getDesired()[0], tasks[-1].getDesired()[1])

    time.append(t+i) # Store simulation time

    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()

# Evolution of task errors over time
fig_joint = plt.figure()
ax = fig_joint.add_subplot(111, autoscale_on=False, xlim=(0, 60), ylim=(-1, 2))
ax.set_title("Task-Priority Control")
ax.set_xlabel("Time[s]")
ax.set_ylabel("Error / Joint Position")
ax.grid()

# Add horizontal lines for joint limits
ax.axhline(y=tasks[0].safe_set[0], color='r', linestyle='--', label="Lower Limit")
ax.axhline(y=tasks[0].safe_set[1], color='r', linestyle='--', label="Upper Limit")

# Plot task errors over time
plt.plot(time, tasks[0].error, label="q_1 ({})".format(tasks[0].name))  # Joint position
plt.plot(time, tasks[1].error, label="e_2 ({})".format(tasks[-1].name))  # End-effector error
ax.legend()
plt.show()
```

5 Conclusion

In conclusion, this lab focused on implementing and evaluating the Task-Priority algorithm for kinematic control of a 3-link planar manipulator, focusing on set-based tasks: obstacle avoidance and joint limits. Through this lab, by defining and simulating multiple task hierarchies, I demonstrated the algorithm's ability to consistently satisfy the constraints imposed by the set-based tasks while simultaneously implementing equality-based objectives; an approach that is necessary for robotic systems to enhance their flexibility and adaptability.