Hands-on Intervention

Lab #4 - Task Priority Kinematic Control (1B)

Delivered by:

Rihab Laroussi, u1100330

Supervisor:

Patryk Cieślak

Date of Submission:

16/03/2025

Table of Contents

Introduction

In the previous lab, we discovered how to control a planar 3-link manipulator using the Task-Priority control algorithm. In this lab, we extend the TP algorithm to its recursive form, enabling the handling of an arbitrary hierarchy of tasks. In Exercise 1, we implement and simulate four distinct task hierarchies to demonstrate the flexibility and effectiveness of the recursive TP algorithm. While, Exercise 2 enhances the algorithm by introducing feedforward velocity, gain matrices, and link selection, making the control system more robust and adaptable to complex robotic tasks.

Exercice 1

1. Methodology

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

- O0: Base frame (fixed reference frame).
- O1, O2, O3: Frames attached to Joint 1, Joint 2, and Joint 3, respectively.
- O4: 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

Joint i	θ_i (rad)	d_i (m)	a_i (m)	α_i (rad)
1	$q_1 = 0$	$d_1 = 0$	$a_1 = 0.75$	$\alpha_1 = 0$
2	$q_2 = 0.6$	$d_2 = 0$	$a_2 = 0.5$	$\alpha_2 = 0$
3	$q_3 = 0.3$	$d_3 = 0$	$a_3 = 0.5$	$\alpha_3 = 0$

The drawing of the robot model with its DH parameters and coordinates systems are in Fig1.

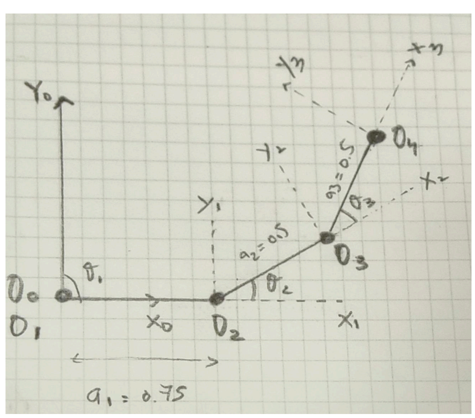


Fig.1: A simple representation of the robot configuration

To implement the recursive Task-Priority algorithm, I first defined each task as a subclass of the base Task class. Each task is responsible for computing its error and Jacobian, which are used by the TP algorithm to compute the joint velocities required to achieve the task. Below, I describe the four tasks implemented in this exercise:

End-Effector Position

The goal is to move the end-effector to a desired 2D position. The task is implemented under class **Position2D()**, where the error is calculated as the difference between the desired and current end-effector position, and the Jacobian is the related joint velocities to the end-effector linear velocities.

End-Effector Orientation

The goal is to achieve the desired end-effector orientation. The task is implemented under class **Orientation2D()**, where the error is calculated as the difference between the desired and current orientation, and the Jacobian is the related joint velocities to the end-effector angular velocity.

End-Effector Configuration

The goal is to achieve both position and orientation simultaneously. The task is implemented under class **Configuration2D()**, where the error is calculated as the combined position and orientation errors, and the Jacobian is the related joint velocities to the end-effector's linear and angular velocities.

Joint 1 Position

The goal is to move a specific joint to a desired angle. The task is implemented under class **Joint_Position()**, where the error is calculated as the difference between the desired and current joint angle, and the Jacobian is the related joint velocities to joint angle changes.

Next, I applied the recursive formulation of the TP algorithm, which computes the joint velocities q dot required to achieve multiple tasks while respecting their priorities. The algorithm proceeds as follows:

For each task i ∈1…k:

1. Initialize the null space projector $P_0 = I_{n \times n}$ and the joint velocities $\zeta_0 = 0_n$.
2. For each task $i \in 1 \ldots k$:
   - Compute the task error e_i and Jacobian J_i.
   - Compute the modified Jacobian $\bar{J}_i = J_i P_{i-1}$.
   - Compute the joint velocity contribution for task i:
   $$\zeta_i = \zeta_{i-1} + \bar{J}_i^{\dagger}(e_i - J_i \zeta_{i-1})$$
   - Update the null space projector:
   $$P_i = P_{i-1} - \bar{J}_i^{\dagger} \bar{J}_i$$
3. The final joint velocities ζ_k are used to update the robot's state.

2. Code

---Exercice1.py---

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

# Robot model - 3-link manipulator
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([0.75, 0.5, 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
max_velocity = 1     # Maximum joint velocity for normalization

# Task hierarchy definition
tasks = [
            Position2D("End-effector position", np.array([1.0, 0.5]).reshape(2,1)),
            #Orientation2D("End-effector orientation", np.array([[np.pi]])),
            #Configuration2D("End-effector configuration", np.array([1.0, 0.5, np.pi/2]).reshape(3,
            #Joint_Position("Joint 1 position", np.array([0]),0)
        ]

# 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
PPx = []
PPy = []
time = [] # List to store simulation time for error plotting

# Simulation initialization
def init():
    global tasks , i
    line.set_data([], [])
    path.set_data([], [])
    point.set_data([], [])
    if tasks[0].name == "End-effector configuration":
        tasks[0].setDesired(np.random.uniform(-1,1,size = (3,1)))   # Random configuration
    else:
        tasks[0].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 tasks
    global robot
    global PPx, PPy, i

    ### Recursive Task-Priority algorithm
    # 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)
        # 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
        # Normalize joint velocities to respect maximum velocity limits
        s = np.max(dq/max_velocity)
        if s>1 :
            dq = dq/s

        # 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[0].getDesired()[0], tasks[0].getDesired()[1])

    # Append current time for error plotting
    time.append(t + i )

    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 (two tasks)")
ax.set_xlabel("Time[s]")
ax.set_ylabel("Error")
ax.grid()
# Plot task errors over time
plt.plot(time, tasks[0].error, label="e1 ({})".format(tasks[0].name))
plt.plot(time, tasks[1].error, label="e2 ({})".format(tasks[1].name))

ax.legend()

plt.show()
```

Lab #4 - Task-Priority kinematic control (1B) - Rihab Laroussi

---lab4_robotics.py---

```
'''
    Subclass of Task, representing the 2D position task.
'''
class Position2D(Task):                              # represent a 2D position task, where the
goal is to control the position of a point (the EE or a specific link) in the 2D plane
    def __init__(self, name, desired):
        super().__init__(name, desired)
        self.J =  np.zeros((len(desired),3))     # Initialize Jacobian with proper dimensions (2 x
        3 for 2D position)
        self.err = np.zeros(desired.shape)   # Initialize with proper dimensions

    def update(self, robot):
        self.J = robot.getEEJacobian()[: len(self.sigma_d), :]  # Update task Jacobian , keep
        only the rows ccorresponding to the linear velocities (x_d, y_d)
        sigma = robot.getEETransform()[: len(self.sigma_d), 3].reshape(self.sigma_d.
        shape)          # Get the current position of the end-effector (x, y)
        self.err =  self.getDesired()- sigma            # 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 orientation task.
'''
class Orientation2D(Task):
    def __init__(self, name, desired):
        super().__init__(name, desired)
        self.J = np.zeros ((len(desired),3)) # Initialize with proper dimensions (1 x 3 for
        orientation)
        self.err =  np.zeros(desired.shape) # Initialize with proper dimensions

    def update(self, robot):
        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):
        super().__init__(name, desired)
        self.J = np.zeros((3,3))     # Initialize with proper dimensions (3 x 3 for configuration)
        self.err = np.zeros(desired.shape) # Initialize with proper dimensions

    def update(self, robot):
        self.J = robot.getEEJacobian()[[0, 1, -1], :]      # Update task Jacobian
        sigma_p = robot.getEETransform()[:2, -1]       # Get the current position of the
        end-effector (x, y)
        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):
        super().__init__(name, desired)
        self.J = np.zeros((1,3))     # Initialize with proper dimensions (1 x 3 for joint position)
        self.err = np.zeros(desired.shape) # Initialize with proper dimensions
        self.Joint = joint

    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
```

3. Results

The results below are visualised using Matplotlib, showing the robot's motion and the evolution of task errors over time for each task.

Result 1:

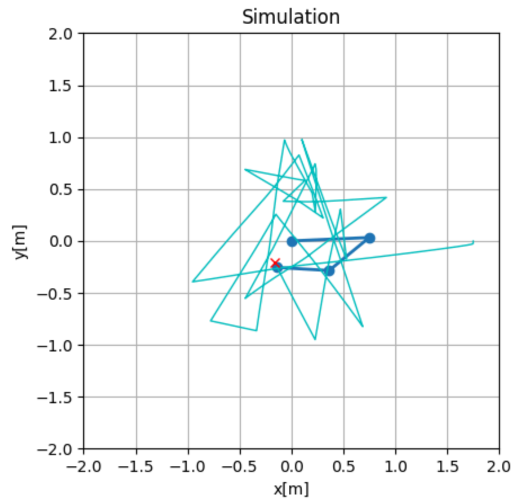


Fig 2.a. Simulation of the manipulator with end-effector goal

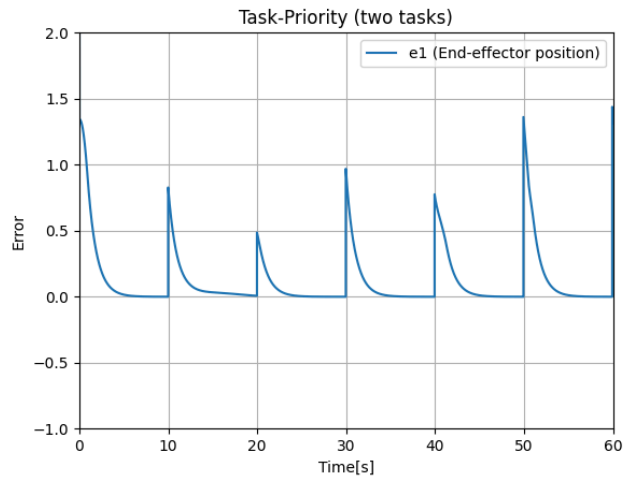


Fig 2.b. Evolution of the TP control errors

Fig 2. One task -> 1: end-effector position

Result 2:

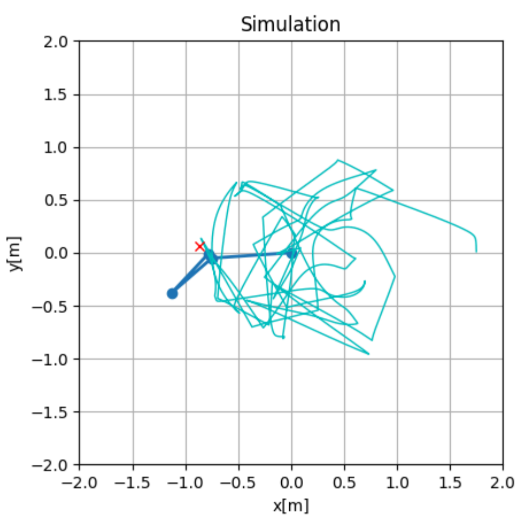


Fig 3.a. Simulation of the manipulator with end-effector goal

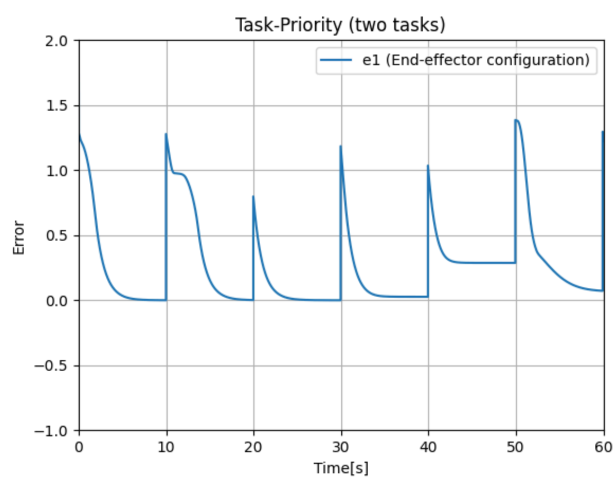
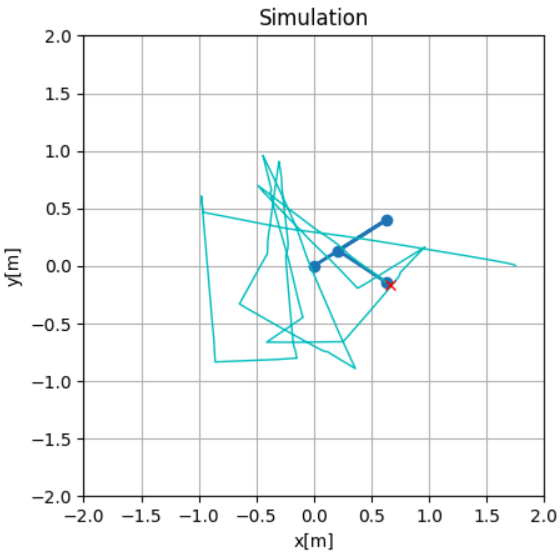


Fig 3.b. Evolution of the TP control errors

Fig 3. One task -> 1: end-effector configuration

Result 3:



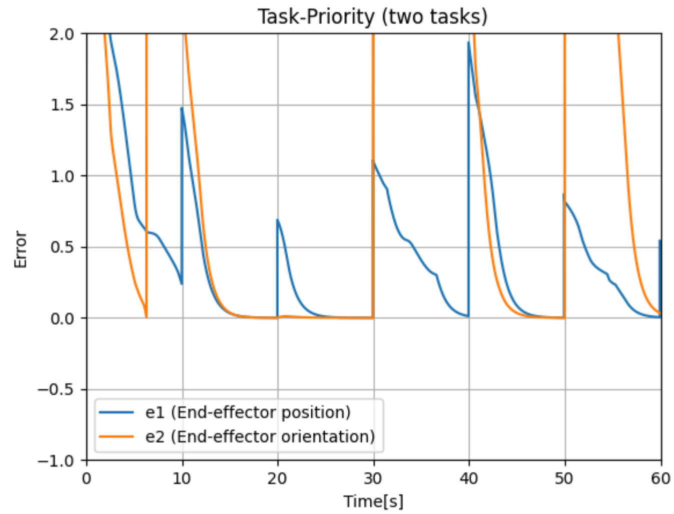
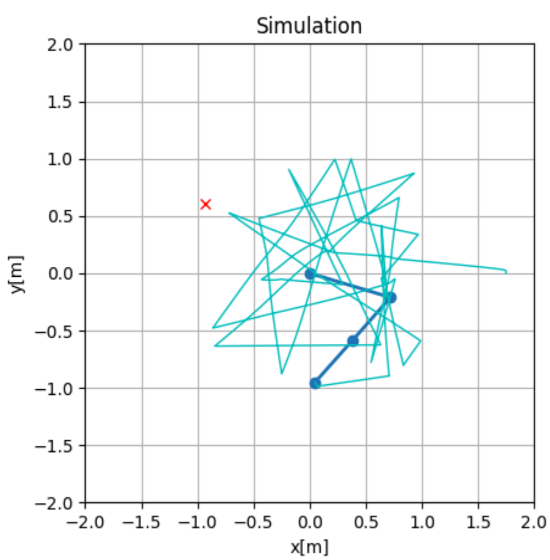


Fig 4.a. Simulation of the manipulator with end-effector goal

Fig 4.b. Evolution of the TP control errors

Fig 4. Two tasks -> 1: end-effector position, 2: end-effector orientation

Result 4:



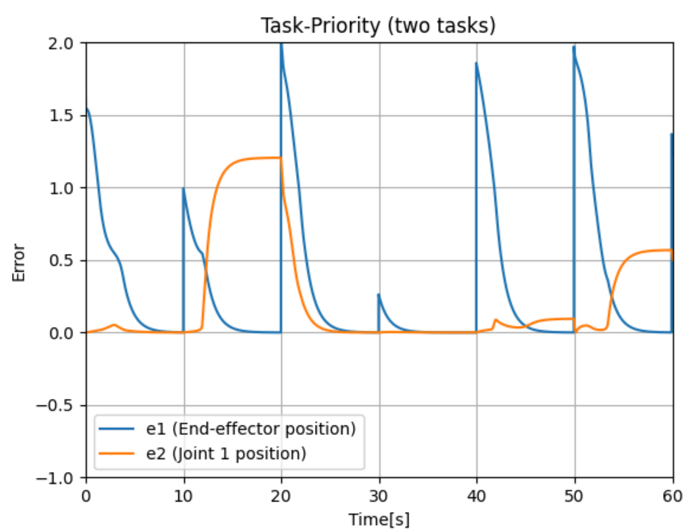


Fig 5.a. Simulation of the manipulator with end-effector goal

Fig 5.b. Evolution of the TP control errors

Fig 5. Two tasks -> 1: end-effector position, 2: joint 1 position

As we can see, the simulation results demonstrate the effectiveness of the recursive TP algorithm in handling tasks. For **Task 1 (End-Effector Position),** the end-effector moves smoothly to the desired position, with the position error converging to zero. **In Task 2 (End-Effector Configuration),** the end-effector moves toward the desired position while simultaneously adjusting its orientation, their errors are minimized simultaneously. **For Task 3 (End-Effector Position + Orientation)**, the robot first reaches the desired position, and then adjusts its orientation without disturbing the position, the robot's position error decreases then converges to zero, and the orientation error does the same when the position is achieved. **In Task 4 (End-Effector Position + Joint 1 Position)**, the robot first achieves the desired position, then adjusts Joint 1 to the target angle q1,d=0 without affecting the end-effector's position, with errors converging in sequence.

Exercice 2

1. Methodology

In this exercise, I extended the code of Exercice1 by adding new features such as link selection for position and orientation tasks, gain matrices (with associated weighted DLS implementation) and the feedforward velocity component (tracking); these features allow for flexible task definition.

Link selection for position and orientation tasks: Tasks can now be defined for any link in the manipulator, not just the end-effector. This was done by updating the function **jacobianLink()** under the class Manipulator, the tasks -Position2D, Orientation2D, Configuration2D- now accept a link parameter to specify the target link.

Gain matrices with associated weighted DLS implementation: Each task can have an associated gain matrix K, which weights the task error. This allows for weighted control, where some tasks are prioritized more than others. This was done by adding **setGainMatrix()** and **getGainMatrix()** functions under the class Task.

Feedforward velocity component: A feedforward velocity component can be added to the control law to improve tracking performance. This was done by adding **setFeedforward()** and **getFeedforward()** under the class Task.

The recursive TP algorithm was updated to include the gain matrix and the feedforward velocity in the control law, where q dot is now computed as the following:

$$\dot{\mathbf{q}} = \mathrm{DLS}(\mathbf{J}_{\mathrm{aug}}, \lambda) \cdot ((\mathbf{K} \cdot \mathbf{e} + \mathbf{v}_{\mathrm{ff}}) - (\mathbf{J} \cdot \dot{\mathbf{q}}))$$

2. Code

--Exercice2.py---

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

# Robot model - 3-link manipulator
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([0.75, 0.5, 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
max_velocity = 1  # Maximum joint velocity limit

# Task hierarchy definition
tasks = [
    Position2D("End-effector position", np.array([1.0, 0.5]).reshape(2,1),3),
    Orientation2D("2nd-link orientation", np.array([[0]]),2),
]

# Gain Matrix
K = 1
tasks[0].setGainMatrix(K)   # Set the gain matrix for the first task
# FeedForward Velocity
ff = 0.0
tasks[0].setFeedforward(ff)  # Set the feedforward velocity for the first task

# 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
PPx = []
PPy = []
time = []        # List to store simulation time

# Simulation initialization
def init():
    global tasks , i
    line.set_data([], [])
    path.set_data([], [])
    point.set_data([], [])
    if tasks[0].name == "End-effector configuration":
        tasks[0].setDesired(np.random.uniform(-1,1,size = (3,1)))   # Random configuration
    else:
        tasks[0].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 tasks
    global robot
    global PPx, PPy, i

    ### Recursive Task-Priority algorithm
    # 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)
        # Compute augmented Jacobian
        J = task.getJacobian()
        J_bar = J @ P
        # Compute task velocity
        e = task.getError()                # get the task error
        K = task.getGainMatrix()           # get the task gain matrix
        ff = task.getFeedforward()         # get the feedforward velocity
        dq_acc = DLS(J_bar, 0.1) @ ( (K @ e + ff ) - (J @ dq))
        # Accumulate velocity
        dq += dq_acc
        # Normalize joint velocities to respect maximum velocity limits
        s = np.max(dq/max_velocity)         # Check if velocities exceed the limit
        if s>1 :
            dq = dq/s
        # 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[0].getDesired()[0], tasks[0].getDesired()[1])

    time.append(t + i )   # Update the simulation time
```

```python
# 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 the norm control errors
fig_joint = plt.figure()
ax = fig_joint.add_subplot(111, autoscale_on=False, xlim=(0, 60), ylim=(-1, 2))
ax.set_title("Task-Priority (two tasks)")
ax.set_xlabel("Time[s]")
ax.set_ylabel("Error")
ax.grid()
plt.plot(time, tasks[0].error, label="e1 ({})".format(tasks[0].name))
plt.plot(time, tasks[1].error, label="e2 ({})".format(tasks[1].name))
ax.legend()
plt.show()
```

--lab4_robotics.py--

```python
from Common 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...

    # 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.

    # 1. Initialize J and O.
    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
    # 2. For each joint of the robot up to the specified link
    for i in range(link):
        #   a. Extract z and o.
        # Extract the rotation matrix and origin from the transformation matrix
        Ri = T[i][:3, :3]
        Oi = np.array([T[i][:3, 3]]).T

        # Extract the z-axis from the rotation matrix
        Zi = Ri @ Z
        #   b. Check joint type.
        #   c. Modify corresponding column of J.
        if revolute[i]:
            # For revolute joints, use the cross product of z and (O - O_i)
            J[:3, i] = np.cross(Zi.T, (O_link - Oi).T).T[:, 0]
            J[3:, i] = Zi[:, 0]
        else:
            # For prismatic joints, the linear velocity is along the z-axis, and angular velocity is zero
            J[:3, i] = Zi[:, 0]

    return J
```

Lab #4 - Task-Priority kinematic control (1B) - Rihab Laroussi

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

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

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

        Arguments:
        dq (Numpy array): a column vector of joint velocities
        dt (double): sampling time
    '''
    def update(self, dq, dt):       # given the joint velocities dq and a time step dt,
                                    # update the joint positions and recompute the forward kinematics
        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)          #for velocity control
    '''
        Method that returns the end-effector transformation.
    '''
    def getEETransform(self):                           # for psoition and orientation
        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):          # get the position of the specific joint
        return self.q[joint]

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

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

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

Lab #4 - Task-Priority kinematic control (1B) - Rihab Laroussi

```python
class Task:
    def __init__(self, name, desired):
        self.name = name # task title
        self.sigma_d = desired # desired sigma , desired goal
        self.error = []
        self.K = None
        self.feedforward = None

    '''
        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 setFeedforward(self, value):
        self.feedforward = np.ones(self.sigma_d.shape)*value # Set the feedforward velocity vector for the task

    def getFeedforward(self):
        return self.feedforward   # Return the stored feedforward velocity vector

    def setGainMatrix(self, value):
        self.K = self.K * value    # Scale the gain matrix K by the specified value

    def getGainMatrix(self):
        return self.K              # Return the current gain matrix K for the task
```

```python
'''
    Subclass of Task, representing the 2D position task.
'''
class Position2D(Task):        # represent a 2D position task, where the goal is to control the position of a point (the EE or a specific link) in the 2D plane
    def __init__(self, name, desired, link = 2):
        super().__init__(name, desired)
        self.J =  np.zeros((len(desired),3))      # Initialize Jacobian with proper dimensions (2 x 3 for 2D position)
        self.err = np.zeros(desired.shape)   # Initialize with proper dimensions
        self.link = link
        self.feedforward = np.zeros(desired.shape)  # 2,1
        self.K = np.eye(len(desired)) # 2

    def update(self, robot):
        #-----------------------------Exercice 1-----------------------------------
        '''self.J = robot.getEEJacobian()[: len(self.sigma_d), :]   # Update task Jacobian , keep only the rows ccorresponding to the linear velocities (x_d, y_d)
        sigma = robot.getEETransform()[: len(self.sigma_d), 3].reshape(self.sigma_d.shape)  # Get the current position of the end-effector (x, y)
        self.err =  self.getDesired()- sigma            # Update task error
        self.error.append(np.linalg.norm(self.err))'''

        #-----------------------------Exercice 2-----------------------------------
        self.J = robot.getJacobianLink(self.link)[: len(self.sigma_d), :]  # Update task Jacobian , keep only the rows ccorresponding to the linear velocities (x_d, y_d)
        sigma = robot.getTransform(self.link)[: len(self.sigma_d), -1].reshape(self.sigma_d.shape)  # Get the current position of the end-effector (x, y)
        self.err =  self.getDesired()- sigma            # Update task error
        self.error.append(np.linalg.norm(self.err))
```

Lab #4 - Task-Priority kinematic control (1B) - Rihab Laroussi

```
'''
    Subclass of Task, representing the 2D orientation task.
'''
class Orientation2D(Task):
    def __init__(self, name, desired, link = 2):
        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.zeros(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])
        self.err = self.getDesired() - sigma.reshape(self.sigma_d.shape)                           # Update task
        self.error.append(np.linalg.norm(self.err))'''
    #----------------------------Exercice 2----------------------------------
        self.J = robot.getJacobianLink(self.link)[-1, :].reshape(1,3)          # Update task Jacobian
        sigma = np.arctan2(robot.getTransform(self.link)[1,0],robot.getTransform(self.link)[0,0])
        self.err = self.getDesired() - sigma.reshape(self.sigma_d.shape)                           # Update task
        self.error.append(np.linalg.norm(self.err))
'''
    Subclass of Task, representing the 2D configuration task.
'''
class Configuration2D(Task):
    def __init__(self, name, desired, link =3):
        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])
        sigma = np.block([sigma_p, sigma_o])
        self.err = self.getDesired() - sigma.reshape(3,1)
        self.error.append(np.linalg.norm(self.err))'''
        #----------------------------Exercice ----------------------------------
        self.J = robot.getJacobianLink(self.link)[[0, 1, -1], :]     # Update task Jacobian
        sigma_p = robot.getTransform(self.link)[:2, -1]
        sigma_o = np.arctan2(robot.getTransform(self.link)[1,0],robot.getTransform(self.link)[0,0])
        sigma = np.block([sigma_p, sigma_o])
        self.err = self.getDesired() - sigma.reshape(3,1)
        self.error.append(np.linalg.norm(self.err))

'''
    Subclass of Task, representing the joint position task.
'''
# class JointPosition(Task):

class Joint_Position(Task):
    def __init__(self, name, desired, joint):
        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)
        self.err = self.getDesired() - sigma
        self.error.append(np.linalg.norm(self.err))
```

3. Results

The following figures illustrate the evolution of the norm of control errors for the two tasks over time, corresponding to three distinct values of the gain matrix K.

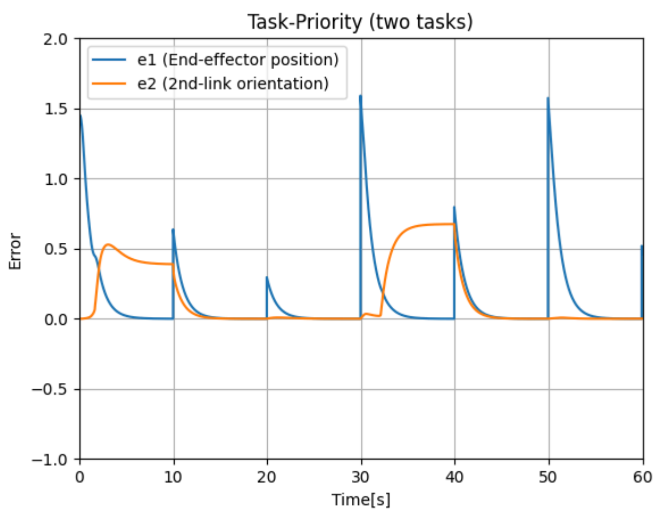


Fig6. The evolution of the errors with K=1

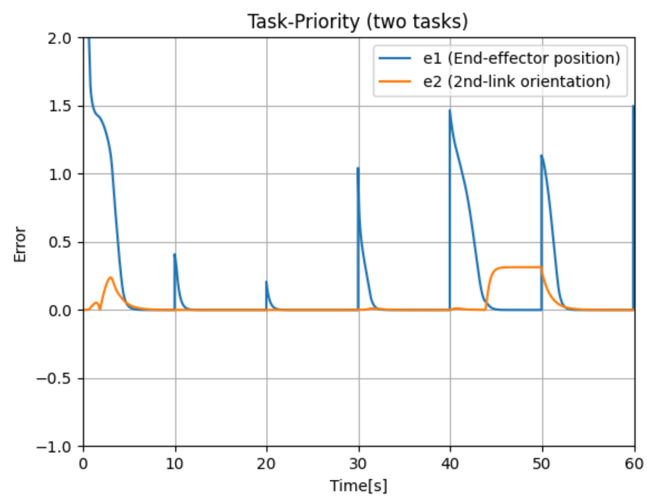


Fig7. The evolution of the errors with K=3

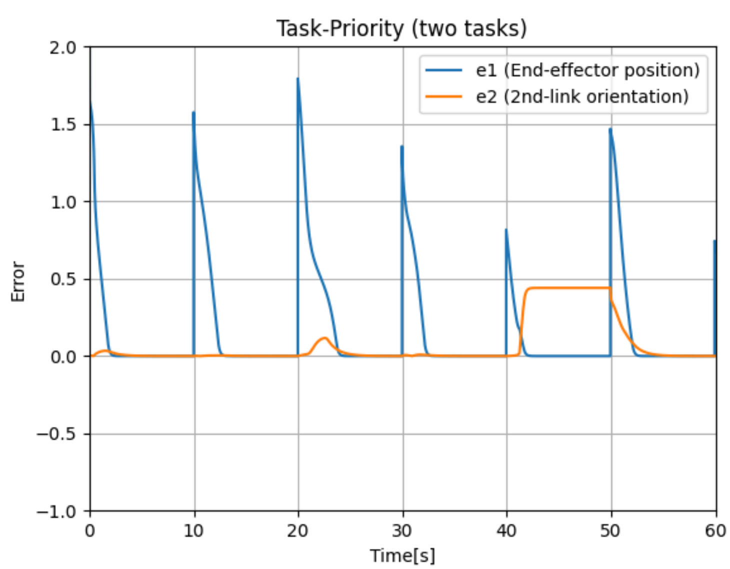


Fig8. The evolution of the errors with K=6

We observe that with lower gain K, the position error converges steadily to zero, and with higher K, it converges faster with higher oscillation; While the manipulator quickly reaches the target position. These results align with control theory principles, where K directly influences the system's responsiveness to errors.

Conclusion

In conclusion, this lab demonstrated the effectiveness of the recursive Task-Priority algorithm in controlling a 3-link planar manipulator. Redundant systems, characterized by their ability to perform tasks with multiple degrees of freedom, require sophisticated control strategies to manage task priorities and exploit their flexibility effectively. Through this lab, by defining and simulating multiple task hierarchies, I showed the algorithm's ability to handle complex tasks with varying priorities, while ensuring that lower-priority tasks don't interfere with the performance of higher-priority tasks.