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

Lab 4:Task-Priority kinematic control (1B)

Introduction

The objective of the exercise 1 is to apply the Task-Priority algorithm in a recursive manner, enabling a hierarchical organization of tasks. This implementation is divided into two sections. The initial section involves creating various tasks as Python class definitions, while the second focuses on the recursive application of the Task-Priority algorithm, demonstrated through a simulation involving a 3-link planar manipulator. The aim of this exercise 2 is to build upon the code from Exercise 1 by incorporating additional functionalities that enhance the versatility in defining tasks. These enhancements comprise the ability to select links for position and orientation tasks, the introduction of gain matrices coupled with a weighted DLS (Damped Least Squares) implementation, and the inclusion of a feedforward velocity component for tracking purposes.

Methodology

The robot that I have used for the model comprises three revolute joints, with the origins of the coordinate systems denoted by O_0, O_1, O_2, O_3, and O_4. There are five coordinate systems in total: one for the base frame, three for the robot joints, and one for the end-effector. The Denavit-Hartenberg parameter values used in the code are as follows: the link lengths (distance along the x-axis) are $a_1 = 0.75$, $a_2 = 0.50$, and $a_3 = 0.50$. The link offsets (distance along the z-axis) are $d_1 = 0$, $d_2 = 0$, and $d_3 = 0$. The link twist angles (rotation around the X-axis) are $\alpha_1 = 0$, $\alpha_2 = 0$, $\alpha_3 = 0$, and the joint angles (for the revolute joints) are variables that will update with each time step; the initial values are set as $\theta_1 = q_1 = 0.2$, $\theta_2 = q_2 = 0.5$, and $\theta_3 = q_3 = 0.2$.

From the lab 2, the common files are taken which computes the DH parameters, kinematic, Jacobian and DLS function. In the exercise 1, at first all the sub classes for the task are build using the following algorithm 1.

Then according to different task hierarchies mentioned in the instruction manual all the simulation are run to visualize the results. In here, the task priority algorithm(shown in algorithm 2) is used to visualize the results. In the exercise 2, using the algorithm 1 all the subclasses are created. In the exercise 1 there was no values for the gain K and link and now we have taken values for them and no values for FFvelocity. Here, while creating the subclass link jacobian is used instead of Jocobian (shown in algorithm 3). The task priority algorithm remains the same where

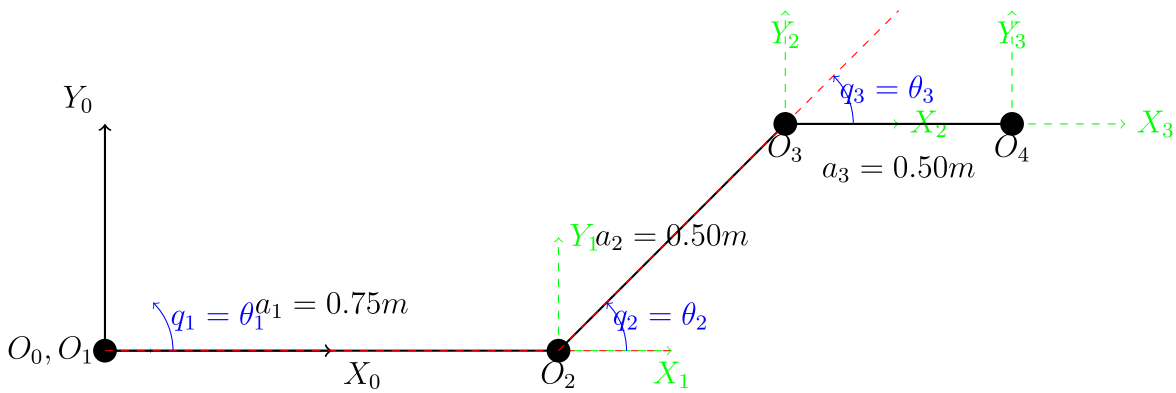
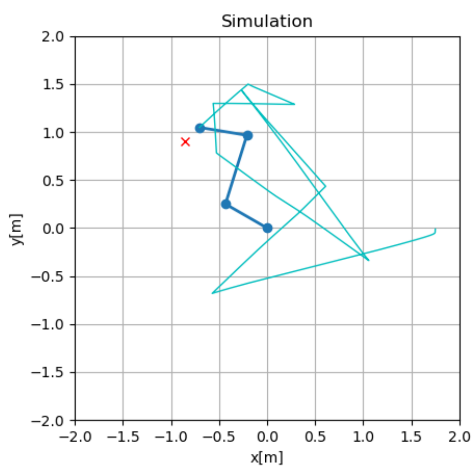


Figure 1: A simplified representation of the robot arm.

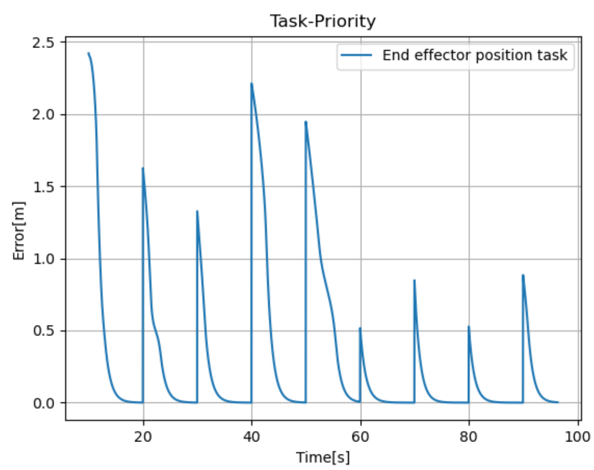
quasi-velocity formula is updated which now use the feedforward velocity and the gain matrix.

Results

In Exercise 1, a recursive task-priority framework is applied and visualized for four distinct task hierarchies, with each task being executed through a Python class. For each task hierarchy, there are two accompanying diagrams: the one on the left depicts the movement of the robot's structure within a 2D space, aiming for a target with its end-effector, while the one on the right tracks the progression of task errors over time. The following figure 2-5 shows different task and their error overtime.



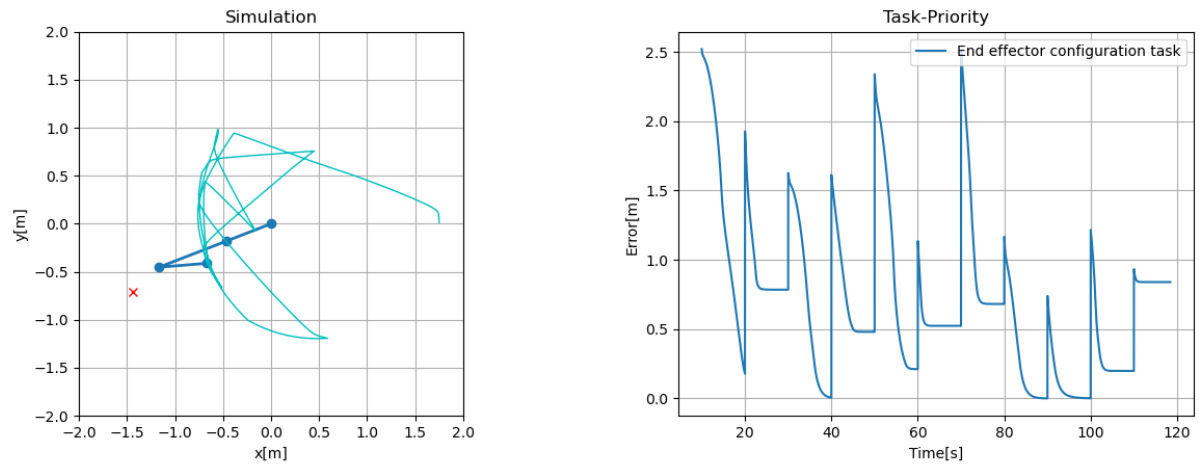
(a) Manipulator with end-effector goal.



(b) Task error overtime

Figure 2: One task -> 1: end-effector position

In Exercise 2, we built upon the foundational work of Exercise 1 by enhancing the code to



(a) Manipulator with end-effector goal.

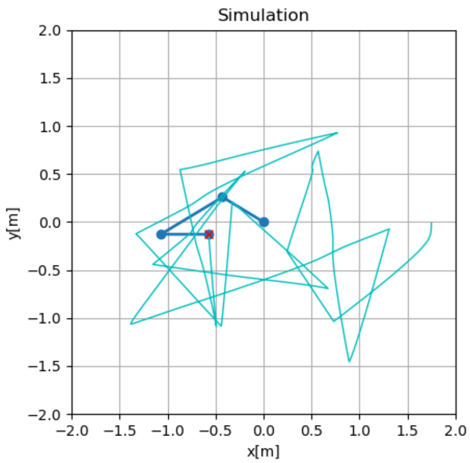
(b) Task error overtime

Figure 3: One task -> 1: end-effector configuration

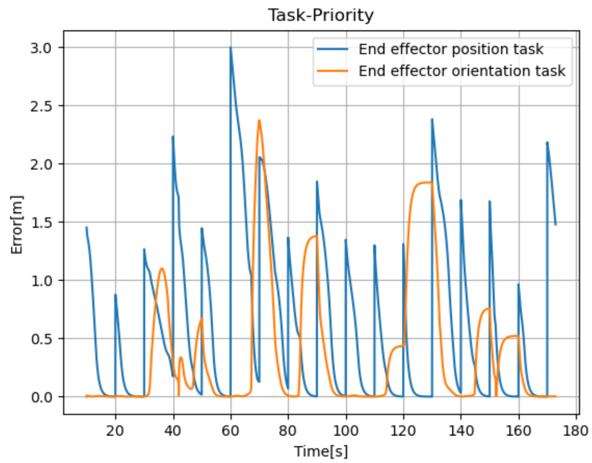
allow more adaptable task definitions. New capabilities include the ability to choose specific links for position and orientation tasks, the integration of gain matrices, and the implementation of a feedforward velocity component. Figure 6 displays a visual snapshot of a simulation involving a dual-task hierarchy, with the primary task being the end-effector's position and the secondary task being the maintenance of a zero orientation for the second link. Subsequently, Figure 7 showcases a trio of graphs that plot the trajectory of the norm of control errors for the two specified tasks over time, juxtaposed against three distinct K matrix values (as denoted in the figure) associated with the first task, which is the end-effector's position. We observe that by amplifying the gain, K, for the first task, the task swiftly achieves the target position. This is evidenced in Figure 7, where the error for the first task, Error 1, precipitously declines to zero with the increment of the gain.

Conclusion

Redundant robotic systems are engineered to handle several tasks concurrently. These tasks vary based on the system setup and can be categorized according to their importance. Leveraging system redundancy involves integrating lower-priority optimization tasks into the control framework. Nevertheless, it's crucial to ensure that these lower-priority tasks don't compromise the performance of higher-priority safety and operational tasks.



(a) Manipulator with end-effector goal.



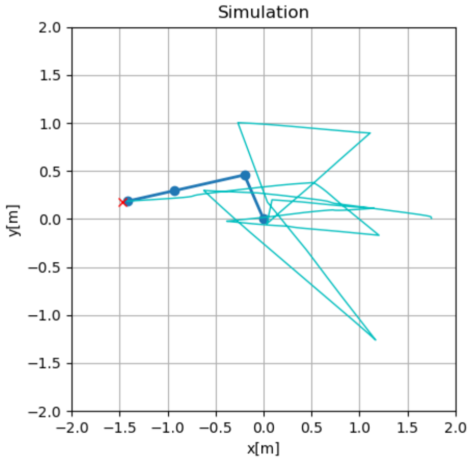
(b) Task error overtime

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

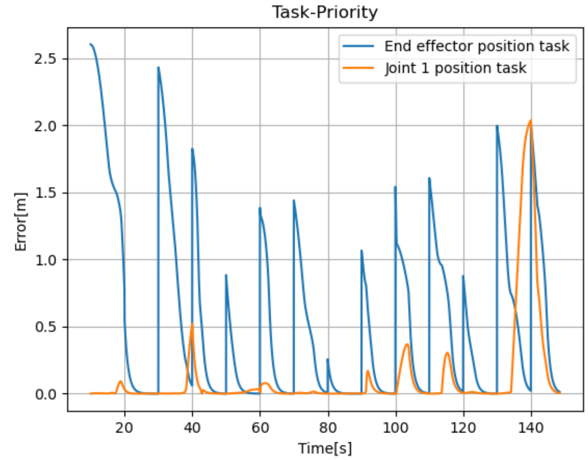
Appendix

Common python code

```
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]])
```



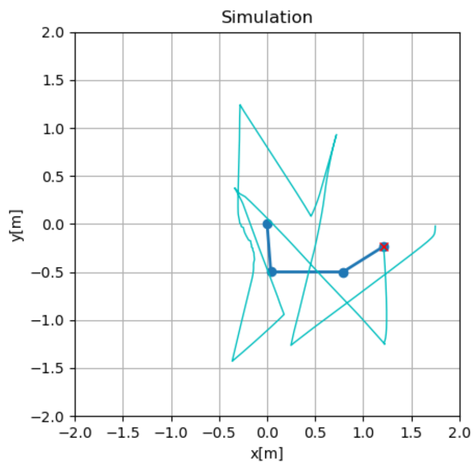
(a) Manipulator with end-effector goal.



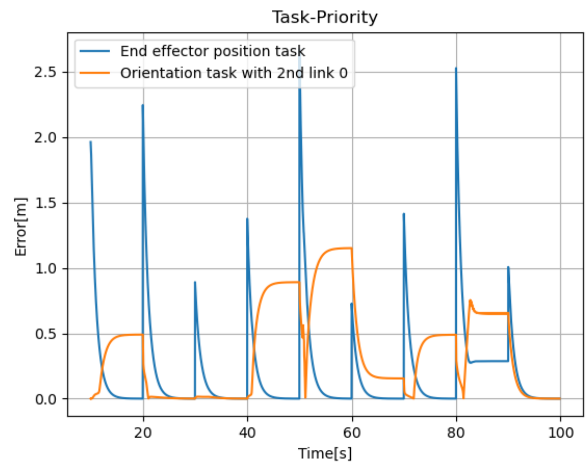
(b) Task error overtime

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

```

    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.

        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:
```



(a) Manipulator with end-effector goal.



(b) Task error overtime

Figure 6: Simulation of the robot with a hierarchy composed of two tasks: 1) position of the end-effector, 2) orientation of the second link equal 0.

```
        (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
```

```python
    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:
            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
```

Lab-4 Common file

```
from lab2_robotics import * # Includes numpy import

def jacobianLink(T, revolute, link): # Needed in Exercise 2
    '''
        Function builds a Jacobian for the end-effector of a robot,
        described by a list of kinematic transformations and a list of joint
            types.

        Arguments:
        T (list of Numpy array): list of transformations along the kinematic
            chain of the robot (from the base frame)
        revolute (list of Bool): list of flags specifying if the corresponding
            joint is a revolute joint
        link(integer): index of the link for which the Jacobian is computed

        Returns:
        (Numpy array): end-effector Jacobian
    '''
    # Code almost identical to the one from lab2_robotics...
    # Number of joints up to the specified link
    n = len (T)-1

    # Initialize the Jacobian matrix
    J = np.zeros((6, n))

    # Position of the end-effector
    p_n = T[link][:3, 3]

    for i in range(link):
        # Extract the rotation matrix and position vector for the current joint
        R_i = T[i][:3, :3]
        p_i = T[i][:3, 3]

        # Compute the z-axis (rotation/translation axis) for the current joint
        z_i = R_i[:, 2]

        # Compute the vector from the current joint to the end-effector
        r = p_n - p_i
```

```python

        if revolute[i]:
            # For revolute joints, compute the linear velocity component
            J[:3, i] = np.cross(z_i, r)
            # And the angular velocity component
            J[3:, i] = z_i
        else:
            # For prismatic joints, the linear velocity component is the z-axis
            J[:3, i] = z_i
            # And the angular velocity component is zero
            J[3:, i] = 0

    return J


'''
    Class representing a robotic manipulator.
'''
class Manipulator:
    '''
        Constructor.

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

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

        Arguments:
        dq (Numpy array): a column vector of joint velocities
        dt (double): sampling time
    '''
    def update(self, dq, dt):
        self.q += dq * dt
        for i in range(len(self.revolute)):
            if self.revolute[i]:
```

```
                self.theta[i] = self.q[i]
            else:
                self.d[i] = self.q[i]
        self.T = kinematics(self.d, self.theta, self.a, self.alpha)

    '''
        Method that returns the characteristic points of the robot.
    '''
    def drawing(self):
        return robotPoints2D(self.T)

    '''
        Method that returns the end-effector Jacobian.
    '''
    def getEEJacobian(self):
        return jacobian(self.T, self.revolute)

    '''
        Method that returns the end-effector transformation.
    '''
    def getEETransform(self):
        return self.T[-1]

    '''
        Method that returns the position of a selected joint.

        Argument:
        joint (integer): index of the joint

        Returns:
        (double): position of the joint
    '''
    def getJointPos(self, joint):
        return self.q[joint]

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

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

    '''
        Method that returns the link Jacobian.
    '''
    def getLinkJacobian(self, link):
        return jacobianLink(self.T, self.revolute, link)


```

```
'''
    Base class representing an abstract Task.
'''
class Task:
    '''
        Constructor.

        Arguments:
        name (string): title of the task
        desired (Numpy array): desired sigma (goal)
    '''
    def __init__(self, name, desired, FFVelocity, K):
        self.name = name # task title
        self.sigma_d = desired # desired sigma
        self.FFVelocity = FFVelocity #feedforward velocity
        self.K = K #gain matrix

    '''
        Method updating the task variables (abstract).

        Arguments:
        robot (object of class Manipulator): reference to the manipulator
    '''
    def update(self, robot):
        pass

    '''
        Method setting the desired sigma.

        Arguments:
        value(Numpy array): value of the desired sigma (goal)
    '''
    def setDesired(self, value):
        self.sigma_d = value

    '''
        Method returning the desired sigma.
    '''
    def getDesired(self):
        return self.sigma_d

    '''
        Method returning the task Jacobian.
    '''
    def getJacobian(self):
        return self.J

    '''
        Method returning the task error (tilde sigma).
    '''
    def getError(self):
```

```
            return self.err

    def setFFVelocity(self, value):
        self.FFVelocity = value

    '''
        Method returning the feedforward velocity vector.
    '''
    def getFFVelocity(self):
        return self.FFVelocity

    '''
        Method setting the gain matrix K.

        Arguments:
        value(Numpy array): value of the gain matrix K.
    '''
    def setK(self, value):
        self.K = value

    '''
        Method returning the gain matrix K.
    '''
    def getK(self):
        return self.K




'''
    Subclass of Task, representing the 2D position task.
'''
class Position2D(Task):
    def __init__(self, name, desired, FFVelocity, K, link):
        super().__init__(name, desired, FFVelocity, K)
        self.J = np.zeros((2,3)) # Initializing with proper dimensions
        self.err = np.zeros((2,1)) # Initializing with proper dimensions
        self.FFVelocity = np.zeros((2,1)) # Initializing with proper dimensions
        self.K = np.eye((2)) # Initializing with proper dimensions
        self.link = link

    def update(self, robot):
        #<<<<<<<<Exercise-1>>>>>>>>>>
        # self.J=robot.getEEJacobian()[:2,:]
        # sigma = robot.getEETransform()[:2,3].reshape(2,1)  # Current position of the task
        # self.err =  self.getDesired() - sigma #task error

        #<<<<<<<<Exercise-2>>>>>>>>>>>
        self.J = robot.getLinkJacobian(self.link)[:2,:]
```

```
        sigma = robot.getLinkTransform(self.link)[:2,3].reshape(2,1)  # Current
            position of the link
        self.err =  self.getDesired() - sigma #task error
'''
    Subclass of Task, representing the 2D orientation task.
'''
class Orientation2D(Task):
    def __init__(self, name, desired, FFVelocity, K, link):
        super().__init__(name, desired, FFVelocity, K)
        self.J = np.zeros((1,3)) # Initialize with proper dimensions
        self.err = np.zeros((1,1)) # Initialize with proper dimensions
        self.FFVelocity = np.zeros((1,1)) # Initialize with proper dimensions
        self.K = np.eye((1)) # Initialize with proper dimensions
        self.link = link

    def update(self, robot):
        #<<<<<<<<Exercise-1>>>>>>>>>>>
        # self.J = robot.getEEJacobian()[5,:].reshape(1,3)
        # angle = np.arctan2(robot.getEETransform()[1,0], robot.getEETransform
            ()[0,0])
        # self.err = (self.getDesired() - angle)

        #<<<<<<<<Exercise-2>>>>>>>>>>>
        self.J = robot.getLinkJacobian(self.link)[5,:].reshape(1,3)
        angle = np.arctan2(robot.getLinkTransform(self.link)[1,0], robot.
            getLinkTransform(self.link)[0,0])
        self.err = (self.getDesired() - angle)

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

    def update(self, robot):
        #<<<<<<<Exercise-1>>>>>>>>
        # self.J[:2,:] = robot.getEEJacobian()[:2,:]
        # self.J[2,:] = robot.getEEJacobian()[5,:]
        # angle = np.arctan2(robot.getEETransform()[1,0],robot.getEETransform()
            [0,0])
        # self.err[:2]= self.getDesired()[:2] - robot.getEETransform()[:2,3].
            reshape(2,1)
        # self.err[2] = self.getDesired()[2] - angle

        #<<<<<<Exercise-2>>>>>>>>>
```

```
        self.J[:2,:] = robot.getLinkJacobian(self.link)[:2,:]
        self.J[2,:] = robot.getLinkJacobian(self.link)[5,:]
        angle = np.arctan2(robot.getLinkTransform(self.link)[1,0],robot.
            getLinkTransform(self.link)[0,0])
        self.err[:2]= self.getDesired()[:2] - robot.getLinkTransform(self.link)
            [:2,3].reshape(2,1)
        self.err[2] = self.getDesired()[2] - angle
'''
    Subclass of Task, representing the joint position task.
'''
class JointPosition(Task):
    def __init__(self, name, desired, FFVelocity, K):
        super().__init__(name, desired, FFVelocity, K)
        self.J = np.zeros((1,3)) # Initializing with proper dimensions
        self.err = np.zeros((1,1)) # Initializing with proper dimensions
        self.FFVelocity = np.zeros((1,1)) # Initializing with proper dimensions
        self.K = np.eye((1)) # Initializing with proper dimensions

    def update(self, robot):
        self.J[0,0] = 1 #for joint 1
        self.err =  self.getDesired() - robot.getJointPos(0)
```

0.1 Exercise 1 and 2

```
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([0.2, 0.5, 0.2])                        # rotation around Z-
    axis
alpha = np.zeros(3)                          # rotation around X-axis
a = [0.5, 0.75, 0.5]                              # displacement along X-axis
revolute = [True, True, True]                       # flags specifying the type
    of joints
robot = Manipulator(d, theta, a, alpha, revolute) # Manipulator object
max_velocity = 0.5
# Task hierarchy definition
tasks = [   #<<<<<<<<<<<<<<<Exercise-1>>>>>>>>>>>>
            # Position2D("End-effector position", np.array([1.0,0.5]).reshape
                (2,1),0,0,0),
            # Orientation2D("End-effector orientation", np.array([0]).reshape
                (1,1),0,0,0),
            # Configuration2D("End-effector configuration", np.array
                ([1.0,0.5,0.5]).reshape(3,1),0,0,0)
            # JointPosition("Joint 1 position", np.array([0]).reshape(1,1),0,0)


            #<<<<<<<<<<<<<Exercise-2>>>>>>>>>>>>>>>
```

```
            Position2D("End-effector position", np.array([1.0,0.5]).reshape(2,1), 0, np.array([1,1]),3),
            Orientation2D("End-effector orientation", np.array([0]).reshape(1,1), 0, np.array([1,1]),2)
        ]


# Simulation params
dt = 1.0/60.0
count1 = -1

# Drawing preparation
fig = plt.figure()
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-2, 2), ylim=(-2,2))
ax.set_title('Simulation')
ax.set_aspect('equal')
ax.grid()
ax.set_xlabel('x[m]')
ax.set_ylabel('y[m]')
line, = ax.plot([], [], 'o-', lw=2) # Robot structure
path, = ax.plot([], [], 'c-', lw=1) # End-effector path
point, = ax.plot([], [], 'rx') # Target

# stroing data
PPx = []
PPy = []
Time = []
err = [[] for _ in tasks]

# Simulation initialization
def init():
    global tasks, count1
    if tasks[0].name == "End-effector configuration":
        tasks[0].setDesired(np.array([np.random.uniform(-1.5,1.5),
                                      np.random.uniform(-1.5,1.5), 0.2]).reshape(3,1))
    else:
        tasks[0].setDesired(np.array([np.random.uniform(-1.5,1.5),
                            np.random.uniform(-1.5,1.5)]).reshape(2,1))
        # tasks[0].setFFVelocity(np.ones((2,1)))
        tasks[0].setK(np.diag([1,1]))

    count1 = count1 + 1
    line.set_data([], [])
    path.set_data([], [])
    point.set_data([], [])
    return line, path, point

# Simulation loop
def simulate(t):
    global tasks
```

```python
    global robot, count, max_velocity
    global PPx, PPy, Time

    ### Recursive Task-Priority algorithm
    # Initialize null-space projector
    P = np.eye(robot.getDOF())
    # Initialize output vector (joint velocity)
    dq = np.zeros(robot.getDOF()).reshape(robot.getDOF(),1)
    count = 0
    # Loop over tasks
    for task in tasks:
        # Update task state
        task.update(robot)
        # Compute augmented Jacobian
        J_bar= task.getJacobian()@ P

        # <<<<<<<<<<<<<<Exercise 1>>>>>>>>>>>>>>>>>
        # Compute task velocity and Accumulate velocity
        # dq =dq + DLS(J_bar,0.1) @ (task.getError() - task.getJacobian() @ dq)

        # <<<<<<<<<<<<<<Exercise 2>>>>>>>>>>>>>>>>>
        dq = dq + DLS(J_bar,0.1) @ (task.getFFVelocity() + task.getK() @ task.
            getError() - task.getJacobian() @ dq)

        # Update null-space projector
        P = P - np.linalg.pinv(J_bar) @ J_bar

        # err[count].append(np.linalg.norm(task.getError))
        err[count].append(np.linalg.norm(task.getError()))
        count = count + 1

    ###

    # 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])
    Time.append(t + 10 * count1)
    path.set_data(PPx, PPy)
    point.set_data(tasks[0].getDesired()[0], tasks[0].getDesired()[1])

    return line, path, point

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

```python
plt.show()

figure = plt.figure()


#<<<<<<<<<<<Exercise 1>>>>>>>>>>>>>>>>
# plt.plot(Time, err[0], label = 'End effector position task') #end-effector position task
# plt.plot(Time, err[1], label = 'End effector orientation task') #end-effector orientation task
# plt.plot(Time, err[0], label = 'End effector configuration task') #end-effector configuration task
# plt.plot(Time, err[1], label = 'Joint 1 position task') #end-joint 1 position task

#<<<<<<<<<<Exercise-2>>>>>>>>>>>>>>>>>>
plt.plot(Time, err[0], label = 'End effector position task') #end-effector position task
plt.plot(Time, err[1], label = 'Orientation task with 2nd link 0') #end-effector orientation task

plt.ylabel('Error[m]') #Title of the Y axis
plt.xlabel('Time[s]') #Title of the X axis
plt.title('Task-Priority')#Title of plot-1
plt.grid(True) #grid
plt.legend() #placing legend
plt.show()
```

Algorithm 1 Task Subclasses Implementation

```
Class Position2D inherits Task
Method __init__(name, desired, FFVelocity, K, link)
  Initialize J, err, FFVelocity, K with appropriate dimensions
  self.link ← link
End Method
Method update(robot)
  Update J and err based on current task and robot state
End Method
End Class

Class Orientation2D inherits Task
Method __init__(name, desired, FFVelocity, K, link)
  Initialize J, err, FFVelocity, K with appropriate dimensions
  self.link ← link
End Method
Method update(robot)
  Update J and err based on current task and robot state
End Method
End Class

Class Configuration2D inherits Task
Method __init__(name, desired, FFVelocity, K, link)
  Initialize J, err, FFVelocity, K with appropriate dimensions
  self.link ← link
End Method
Method update(robot)
  Update J and err based on current task and robot state
End Method
End Class

Class JointPosition inherits Task
Method __init__(name, desired, FFVelocity, K)
  Initialize J, err, FFVelocity, K with appropriate dimensions
End Method
Method update(robot)
  Update J and err based on current task and robot state
End Method
End Class
```

Algorithm 2 Recursive Task-Priority Algorithm

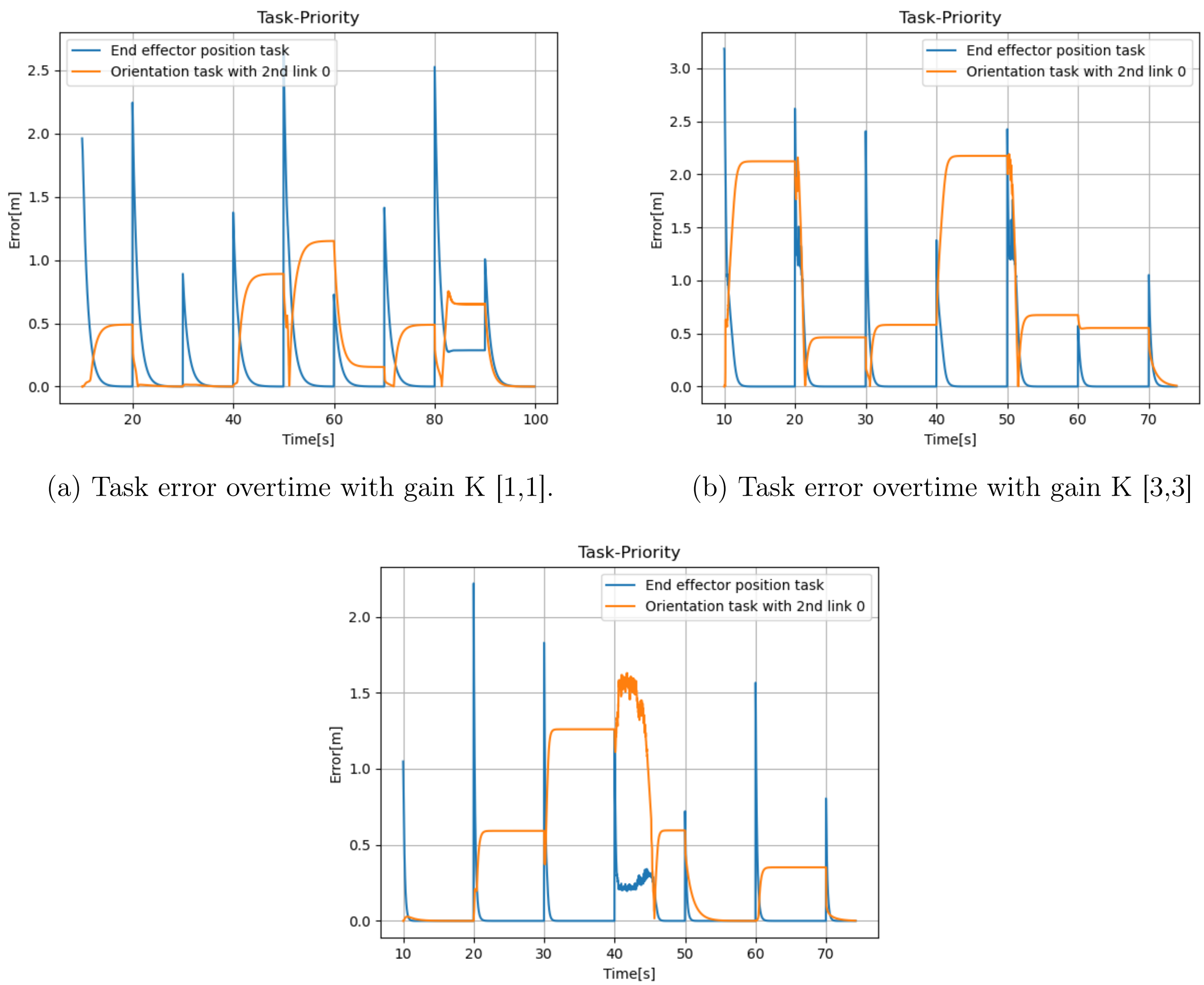
Require: List of tasks $\{J_i(q), \dot{x}_i(q)\}$, $i \in 1 \ldots k$
Ensure: Quasi-velocities $\dot{\xi}_k \in \mathbb{R}^n$

Initialise: $\dot{\xi}_0 = 0^n$, $P_0 = I^{n \times n}$
for $i = 1$ to k **do**
  $J_i(q) = J_i(q) P_{i-1}$
  $\dot{\xi}_i = \dot{\xi}_{i-1} + J_i^{\dagger}(q)(\dot{x}_i(q) - J_i(q)\dot{\xi}_{i-1})$
  $P_i = P_{i-1} - J_i^{\dagger}(q) J_i(q)$
end for
return $\dot{\xi}_k$

Algorithm 3 Calculate Jacobian Link for Robot End-Effector

Require: T: list of transformations, *revolute*: list of joint types, *link*: index of the link
Ensure: J: end-effector Jacobian

$n \leftarrow$ length of T minus 1
Initialize J as a matrix of zeros with shape $(6, n)$
$p_n \leftarrow T[link][:3, 3]$
for i **in** $[0, link - 1]$ **do**
  $R_i \leftarrow T[i][:3, :3]$
  $p_i \leftarrow T[i][:3, 3]$
  $z_i \leftarrow R_i[:, 2]$
  $r \leftarrow p_n - p_i$
  if $revolute[i]$ **then**
    $J[:3, i] \leftarrow \text{cross}(z_i, r)$
    $J[3:, i] \leftarrow z_i$
  else
    $J[:3, i] \leftarrow z_i$
    $J[3:, i] \leftarrow 0$
  end if
end for
return J



(a) Task error overtime with gain K [1,1].

(b) Task error overtime with gain K [3,3]

(c) Task error overtime with gain K [5,5]

Figure 7: Few simulation runs with different values of the K matrix set for the first task.