Hands-on Intervention

Lab #6 – Task Priority Kinematic Control (2B)

Delivered by:
Rihab Laroussi, u1100330

Supervisor:
Patryk Cieślak

Date of Submission:
30/03/2025

Contents

1 Introduction

A mobile manipulator is a system that is composed of a differential drive mobile base and a 3-link planar manipulator mounted on top. In this lab, we will update the Task Priority algorithm to simulate its motion while avoiding joint limits and reaching a desired end-effector position.
The lab is divided into three exercises. In the first, we build the kinematic model and implement the recursive TP control scheme. In the second, we integrate a weighted version of DLS and investigate how different joint weights influence motion. In the third exercise, we assess the effect of different base integration methods (linear vs. arc-based) on task execution and error evolution. Throughout, simulation and visualization are used to evaluate the performances.

2 Methodology and Results

2.1 Exercice1

In the first exercise, the goal was to implement a **MobileManipulator** class that integrates a differential drive mobile base and a 3-link manipulator. The robot operates in a 2D environment where the mobile base can move along the X and Y axes while the manipulator controls its orientation and position.

The main challenge was to integrate the kinematics of both the mobile base and the manipulator. The kinematic chain of the manipulator was modeled using Denavit-Hartenberg parameters, and its end-effector's control was achieved using the task-priority method.

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 :

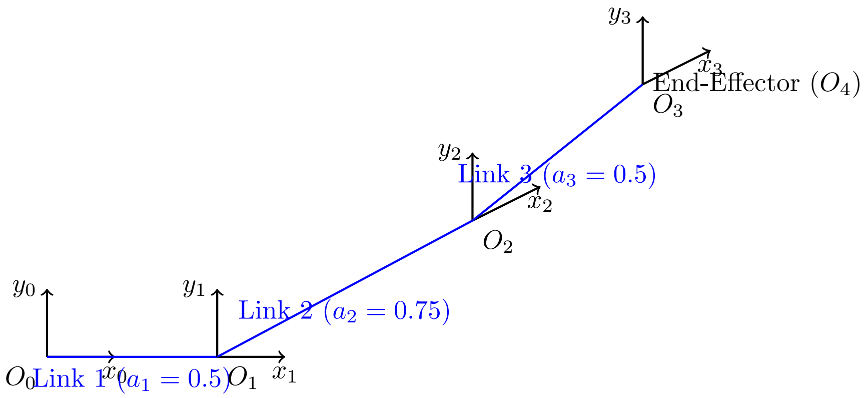


Figure 1: Robot model with DH parameters and coordinate systems.

I first implemented the **MobileManipulator** class. During each time step, the `update()` method was called to simulate the movement of the robot. This method works by first updating the joint angles based on the velocities provided as input. For the mobile base, the method calculates the position and orientation using basic kinematics, updating the `self.eta` vector, which represents the mobile base's pose (X, Y, and θ). The kinematic transformations are computed by concatenating the DH parameters for both the manipulator and the mobile base. These transformations are used to update the manipulator's state. Then I modified the kinematic function, which can now take into consideration the mobile base. The transformation matrix T_b is given as:

$$T_b = \begin{bmatrix} \cos(\eta_{2,0}) & -\sin(\eta_{2,0}) & 0 & \eta_{0,0} \\ \sin(\eta_{2,0}) & \cos(\eta_{2,0}) & 0 & \eta_{1,0} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

The two main tasks used in this exercise are:

1. **JointLimits**: Ensures that the robot's joints stay within the safe range of motion.

2. **Position2D**: Moves the end-effector to a desired position in the 2D plane.

Finally, I implemented **Algorithm 1**, which loops through each task, computes its Jacobian, and determines the desired joint velocities using the Damped Least Squares (DLS) method. The null-space projector P is updated to ensure that lower-priority tasks are executed in the null space of higher-priority tasks.

The task-priority algorithm is implemented as follows:

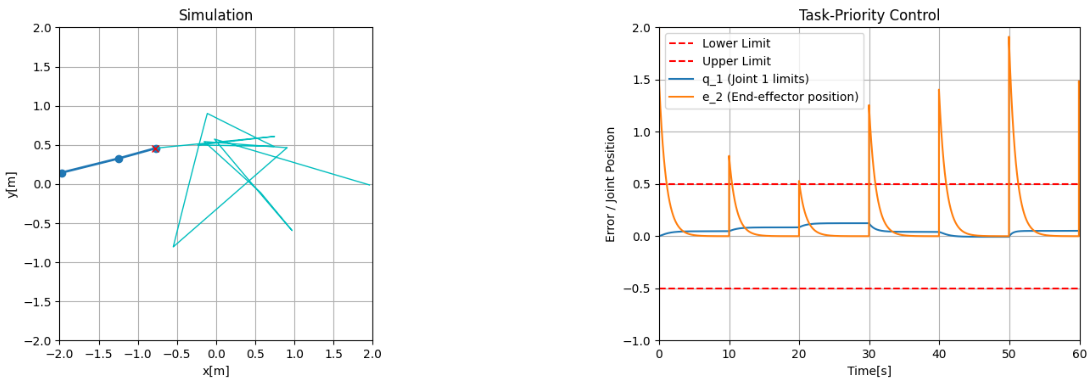
Algorithm 1 Extended Recursive Task-Priority Algorithm

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

Results

I generated the plots to visualize the simulation of the mobile manipulator and the evolution of the TP algorithm control error.



(a) Simulation of the mobile manipulator

(b) Evolution of the TP control error

Figure 2: Results of Exercise 1

As the results show, the robot was able to follow the desired trajectory for the end-effector while respecting the joint limits.

The end-effector error graph shows the norm of the error between the desired and actual end-effector positions. Whereas, the graph of joint position over time shows how the robot's joint position stays within its safe limits, indicated by the horizontal dashed lines.

2.2 Exercice2

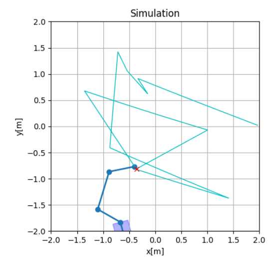
In this exercise, I extended the Task-Priority Control framework to include end-effector configuration control. I introduced the **Weighted Damped Least Squares (WDLS)** method and implemented a new function, named `Weighted_DLS`, within the `Common.py` file. The weighting matrix in the Weighted_DLS method scales the contribution of each joint to the task, enabling control over how much influence each joint has in achieving the desired end-effector position and orientation. Different weights, specifically $\{2, 2, 3, 1, 4\}$, were assigned for this purpose.

Each simulation iteration begins with a randomly generated desired end-effector configuration (target position), ensuring varied results across multiple runs. During the simulation, the robot updates its joint velocities and overall state based on the WDLS algorithm. Once the simulation is complete, the end-effector configuration task error and the velocity outputs from the task-priority control algorithm are plotted for analysis.

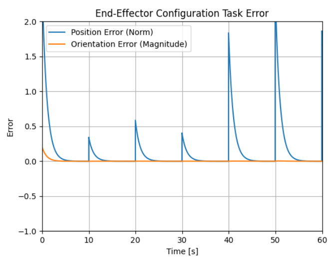
Results

I simulated three different values of the weighting matrix W to observe the effect of the weights on the task performance.
The weight matrices used are defined as follows:

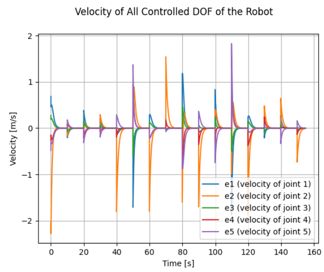
$$W_1 = 0.2 \cdot \text{np.diag}([2, 2, 3, 1, 4])$$
$$W_2 = 0.2 \cdot \text{np.diag}([5, 5, 1, 1, 1])$$
$$W_3 = 0.2 \cdot \text{np.diag}([1, 1, 5, 5, 5])$$



(a) Simulation of the mobile manipulator

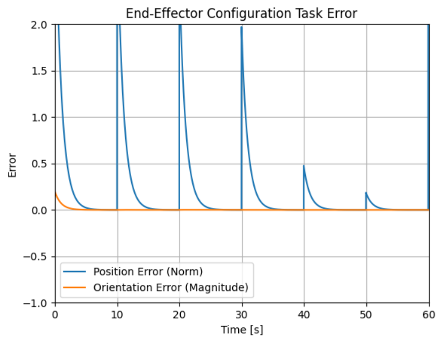


(b) Evolution of the TP control error

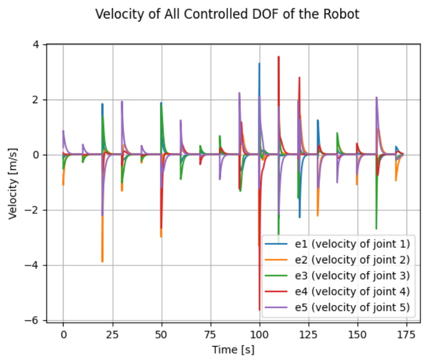


(c) Evolution of velocity

Figure 3: Results of Exercise 2 - W1

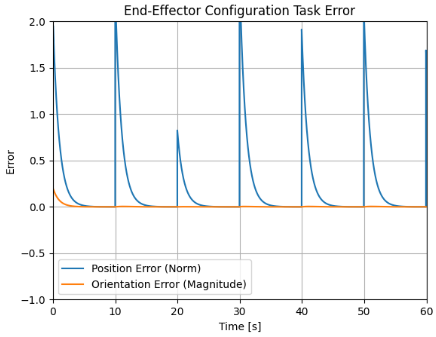


(a) Simulation of the mobile manipulator

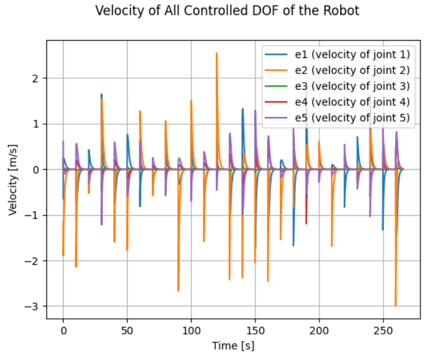


(b) Evolution of the TP control error

Figure 4: Results of Exercise 2 - W2



(a) Simulation of the mobile manipulator



(b) Evolution of the TP control error

Figure 5: Results of Exercise 2 - W3

The error in the end-effector position and orientation decreases over time as the robot moves towards the desired position. The three subplots in the three figures show the norm of the position error and the norm of the orientation error.
As evident from the plots above, the WDLS algorithm effectively adjusts these velocities to reduce the task error. The influence of the weight matrix is apparent, as higher weights on specific joints led to larger velocity updates for those joints, ensuring their greater contribution to the task.

2.3 Exercice3

In the final exercise, I compared three methods of updating the robot's position and orientation as it moves along a path. These methods are as follows:

1. **Move then Rotate**: The robot first moves forward along its heading by a distance d, where $d = v \cdot dt$, and then rotates around its axis by an angle θ. The equations for this method are:

$$x = x + v \cdot dt \cdot \cos(\theta)$$

$$y = y + v \cdot dt \cdot \sin(\theta)$$

$$\theta = \theta + \omega \cdot dt$$

2. **Rotate then Move**: The robot first rotates around its axis and then moves forward along its new heading. The equations for this method are:

$$\theta = \theta + \omega \cdot dt$$

$$x = x + v \cdot dt \cdot \cos(\theta)$$

$$y = y + v \cdot dt \cdot \sin(\theta)$$

3. **Move and Rotate**: The robot moves forward while simultaneously rotating, effectively following an arc. The equations for this method are:

$$x = x - R \cdot \sin(\theta) + R \cdot \sin(\omega \cdot dt + \theta)$$

$$y = y - R \cdot \cos(\theta) + R \cdot \cos(\omega \cdot dt + \theta)$$

$$\theta = \theta + \omega \cdot dt$$

   where $R = \frac{v}{\omega}$.

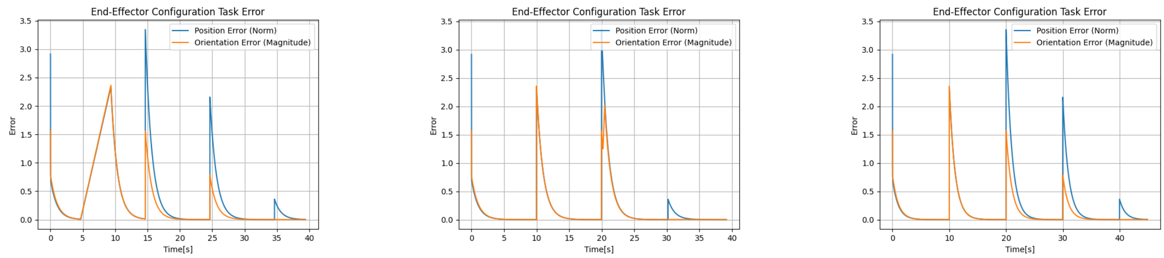
Each method was implemented to update the robot's kinematics differently in order to assess their impact on the resulting motion and error. In this program, instead of using random desired end-effector configurations, selected vectors were used:

$$[1, 1.5, \pi],\ [-0.5, -1.5, \frac{\pi}{2}],\ [1.6, -1, \frac{\pi}{4}],\ [1.8, -0.7, \frac{\pi}{4}].$$

The task defined is the same as the previous exercise: the End-Effector Configuration Task. The simulation loop updates the robot's state at each timestep based on the chosen mobile base integration method. The joint velocities are computed using the Weighted DLS algorithm, which I modified to use:

$$W = \texttt{np.eye(5)}.$$

Results



(a) Move and then Rotate (b) Rotate and then Move (c) Move and Rotate together

Figure 6: Results of Exercise 3

The task error, including position and orientation, was tracked over time for each method. The simulation compared a basic, straight-line update of the mobile base to the correct method, which considers the robot moving in an arc. The plots show how the errors change over time.

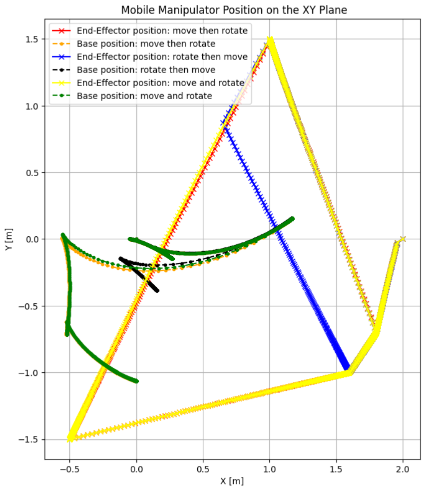


Figure 7: Mobile base position and the end-effector position

Figure 7 displays the robot's path on the x-y plane, with axes labeled "Y[m]" and "X[m]." It includes four different tracks showing how the end effector and base moved in three scenarios: "move then rotate," "rotate then move," and "move and rotate," which represent moving forward then turning, turning first, and moving while turning at the same time.

3 Code

3.1 Exercice1.py

```
from lab6_robotics import * # Includes numpy import
from lab4_robotics import *
import matplotlib.pyplot as plt
import matplotlib.patches as patch
import matplotlib.animation as anim
import matplotlib.transforms as trans

# 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 = MobileManipulator(d, theta, a, alpha, revolute) # Manipulator object


# Task definition

tasks = [ JointLimits("Joint 1 limits", 2, np.array([-0.5, 0.5]), np.array([0.03, 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]')
rectangle = patch.Rectangle((-0.25, -0.15), 0.5, 0.3, color='blue', alpha=0.3)
veh = ax.add_patch(rectangle)
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 = [] # Time points for plotting
# Simulation initialization
def init():
    global tasks, i, time
    line.set_data([], [])
    path.set_data([], [])
    point.set_data([], [])
    # Set a new desired position for the end-effector
    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

   # Initialize null-space projector
    P = np.eye(5)
    # Initialize output vector (joint velocity)
    dq = np.zeros([5,1])
    # 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
    # -- Manipulator links
    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])
    # -- Mobile base
    eta = robot.getBasePose()
    veh.set_transform(trans.Affine2D().rotate(eta[2,0]) + trans.Affine2D().translate(eta
    [0,0], eta[1,0]) + ax.transData)

    time.append(t+i)
    return line, veh, 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()
```

3.2 Exercice2.py

```
from lab6_robotics import * # Includes numpy import
from lab4_robotics import *
from Common import *
import matplotlib.pyplot as plt
import matplotlib.patches as patch
import matplotlib.animation as anim
import matplotlib.transforms as trans
```

```python

# 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 = MobileManipulator(d, theta, a, alpha, revolute) # Manipulator object


# Task definition

tasks = [ Configuration2D("End-Effector Configuration", np.array([1, 0.5, 0]).reshape(3,
    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]')
rectangle = patch.Rectangle((-0.25, -0.15), 0.5, 0.3, color='blue', alpha=0.3)
veh = ax.add_patch(rectangle)
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 = []
velocities = []
# Simulation initialization
def init():
    global tasks, i
    line.set_data([], [])
    path.set_data([], [])
    point.set_data([], [])
    # Set a new desired position for the end-effector
    tasks[-1].setDesired(np.array([np.random.uniform(-1.5, 1.5), np.random.uniform(-1.5,
     1.5), 0.2]).reshape(3, 1))
    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, i

    ### Recursive Task-Priority algorithm
```

```

    # Initialize null-space projector
     P = np.eye(5)
     # Initialize output vector (joint velocity)
     dq = np.zeros((5,1))
     # 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
             W = 0.2*np.diag([2,2,3,1,4])
             #W = 0.2*np.diag([5,5,1,1,1])  # More emphasis on the base (first 2 DOFs)
             #W = 0.2*np.diag([1,1,5,5,5])  # More emphasis on the manipulator joints
             dq_acc = Weighted_DLS(J_bar, 0.1, W) @ ((task.getError()) - (J @ dq))
             # Accumulate velocity
             dq += dq_acc
             # Update null-space projector
             P = P -np.linalg.pinv(J_bar) @ J_bar
             velocities.append(dq)


     # Update robot
     robot.update(dq, dt)

     # Update drawing
     # -- Manipulator links
     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])
     # -- Mobile base
     eta = robot.getBasePose()
     veh.set_transform(trans.Affine2D().rotate(eta[2,0]) + trans.Affine2D().translate(eta
     [0,0], eta[1,0]) + ax.transData)

     time.append(t+i)
     return line, veh, 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("End-Effector Configuration Task Error")
ax.set_xlabel("Time[s]")
```

```python
ax.set_ylabel("Error / Joint Position")
ax.grid()

plt.plot(time, tasks[0].error[0], label="Position Error (Norm)")
plt.plot(time, tasks[0].error[1], label="Orientation Error (Magnitude)")

# Add title, axis labels, and legend
ax.set_title("End-Effector Configuration Task Error")
ax.set_xlabel("Time [s]")
ax.set_ylabel("Error")
ax.legend()
plt.show()

# Plot for velocities of all controlled DOF of the robot
fig, ax2 = plt.subplots()
fig.suptitle('Velocity of All Controlled DOF of the Robot')
ax2.set_xlabel('Time [s]')
ax2.set_ylabel('Velocity [m/s]')
ax2.plot(time, [v[0, 0] for v in velocities], label='e1 (velocity of joint 1)')
ax2.plot(time, [v[1, 0] for v in velocities], label='e2 (velocity of joint 2)')
ax2.plot(time, [v[2, 0] for v in velocities], label='e3 (velocity of joint 3)')
ax2.plot(time, [v[3, 0] for v in velocities], label='e4 (velocity of joint 4)')
ax2.plot(time, [v[4, 0] for v in velocities], label='e5 (velocity of joint 5)')
ax2.legend()
ax2.grid()
plt.show()
```

3.3 Exercice3.py

```python
from lab6_robotics import * # Includes numpy import
from lab4_robotics import *
from Common import *
import matplotlib.pyplot as plt
import matplotlib.patches as patch
import matplotlib.animation as anim
import matplotlib.transforms as trans

# 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 = MobileManipulator(d, theta, a, alpha, revolute, mode = 'move') # Manipulator
    object

# Configuration list for the end-effector task
configuration_list = [np.array([1, 1.5, np.pi]).reshape(3, 1),
                np.array([-0.5, -1.5, np.pi/2]).reshape(3, 1),
                np.array([1.6, -1, np.pi/4]).reshape(3, 1),
                np.array([1.8, -0.7, np.pi/4]).reshape(3, 1)]
configuration_idx = 0

# Task definition
tasks = [ Configuration2D("End-Effector Position", np.array([1, 1, 0]).reshape(3, 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]')
rectangle = patch.Rectangle((-0.25, -0.15), 0.5, 0.3, color='blue', alpha=0.3)
veh = ax.add_patch(rectangle)
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 = []
velocities = []                    # to store the joint velocities to be used in the second plot.
EE_pos=[[],[]]                     # to store the ee position of each joint to be used in the second plot.
base_pos=[[],[]]                   # to store the base position to be used in the second plot.

# Simulation initialization
def init():
    global tasks, i
    global configuration_idx
    line.set_data([], [])
    path.set_data([], [])
    point.set_data([], [])
    tasks[-1].setDesired(configuration_list[configuration_idx % len(configuration_list)])
    configuration_idx += 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, 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)
        if task.isActive():
            # Compute augmented Jacobian
            J = task.getJacobian()
            J_bar = J @ P
            # Compute task velocity
            W = np.eye(robot.getDOF())
            dq_acc = Weighted_DLS(J_bar, 0.1, W) @ ((task.getError()) - (J @ dq))
            # Accumulate velocity
            dq += dq_acc
            # Update null-space projector
            P = P -np.linalg.pinv(J_bar) @ J_bar
            velocities.append(dq)

            EE_pos[0].append(robot.getEETransform()[0, 3])
            EE_pos[1].append(robot.getEETransform()[1, 3])
            base_pos[0].append(robot.getBasePose()[0, 0])
            base_pos[1].append(robot.getBasePose()[1, 0])


    # Update robot
    robot.update(dq, dt)

    # Update drawing
    # -- Manipulator links
    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])
    # -- Mobile base
    eta = robot.getBasePose()
    veh.set_transform(trans.Affine2D().rotate(eta[2,0]) + trans.Affine2D().translate(eta[0,0], eta[1,0]) + ax.transData)

    time.append(t+i)
    return line, veh, path, point

# Run simulation
animation = anim.FuncAnimation(
    fig, simulate, np.arange(0, 10, dt),
    interval=10, blit=True, init_func=init, repeat=True # <--- Don't repeat
)
plt.show()

fig_joint = plt.figure()
ax = fig_joint.add_subplot(111, autoscale_on=True)
ax.set_title("End-Effector Configuration Task Error")
ax.set_xlabel("Time[s]")
ax.set_ylabel("Error")
ax.grid()

```

```python
ax.plot(time, tasks[0].error[0], label="Position Error (Norm)")
ax.plot(time, tasks[0].error[1], label="Orientation Error (Magnitude)")
ax.legend()
plt.show()

# Save EE_pos and base_pos to file
np.save('move then rotate.npy',[EE_pos,base_pos])
#np.save('rotate then move.npy',[EE_pos,base_pos])
#np.save('move and rotate.npy',[EE_pos,base_pos])
```

3.4 final_plot.py

```python
import numpy as np
import matplotlib.pyplot as plt

# Load saved trajectory data
# Each file contains: [end_effector_xy, base_xy], each as (x_list, y_list)
forward_then_rotate = np.load("move then rotate.npy", allow_pickle=True)
rotate_then_forward = np.load("rotate then move.npy", allow_pickle=True)
move_and_rotate = np.load("move and rotate.npy", allow_pickle=True)

# Unpack data for each method
ee_FR_x, ee_FR_y = forward_then_rotate[0]
base_FR_x, base_FR_y = forward_then_rotate[1]

ee_RF_x, ee_RF_y = rotate_then_forward[0]
base_RF_x, base_RF_y = rotate_then_forward[1]

ee_ARC_x, ee_ARC_y = move_and_rotate[0]
base_ARC_x, base_ARC_y = move_and_rotate[1]

# Plotting
fig, ax = plt.subplots()
ax.set_title(" Mobile Manipulator Position on the XY Plane")
ax.set_xlabel("X [m]")
ax.set_ylabel("Y [m]")
ax.set_aspect('equal')
ax.grid(True)

# Forward then Rotate
ax.plot(ee_FR_x, ee_FR_y, 'x-', color='red', label='End-Effector position: move then rotate')
ax.plot(base_FR_x, base_FR_y, '.--', color='orange', label='Base position: move then rotate')

# Rotate then Forward
ax.plot(ee_RF_x, ee_RF_y, 'x-', color='blue', label='End-Effector position: rotate then move')
ax.plot(base_RF_x, base_RF_y, '.--', color='black', label='Base position: rotate then move')

# Arc (Simultaneous motion)
ax.plot(ee_ARC_x, ee_ARC_y, 'x-', color='yellow', label='End-Effector position: move and rotate')
ax.plot(base_ARC_x, base_ARC_y, '.--', color='green', label='Base position: move and rotate')
```

```

# Add legend
ax.legend(loc='best')

# Save plot for report
plt.savefig("xy_trajectories_comparison.png", dpi=300)

# Show plot
plt.show()
```

3.5 Lab6_robotics.py

```
from lab4_robotics import *

class MobileManipulator:  #integrate MBK (x,y,theta) + Mk (q1,q2,q3) + CSK (6DOF)
    '''
        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):  # add mode variable for exercise 3
        self.d = d
        self.theta = theta
        self.a = a
        self.alpha = alpha
        self.revolute = revolute
        self.revoluteExt = [True,False] + self.revolute    # List of joint types extended with base joints
        self.r = 0.24             # Distance from robot centre to manipulator base
        self.dof = len(self.revoluteExt) # Number of DOF of the system
        self.q = np.zeros((len(self.revolute),1)) # Vector of joint positions (manipulator)
        self.eta = np.zeros((3,1)) # Vector of base pose (position & orientation)
        #self.mode = mode
        self.update(np.zeros((self.dof,1)), 0.0) # Initialise robot state


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

        Arguments:
        dQ (Numpy array): a column vector of quasi velocities
        dt (double): sampling time
    '''
    def update(self, dQ, dt):
        # Update manipulator
        self.q += dQ[2:, 0].reshape(-1,1) * dt
        for i in range(len(self.revolute)):
            if self.revolute[i]:
```

```
                self.theta[i] = self.q[i]   # revolute joint angle
            else:
                self.d[i] = self.q[i]   # prismatic joint displacement

        #---------------exercice1&2----------------
        # Update mobile base pose
        self.eta[2,0] += dQ[0,0] * dt
        self.eta[0,0] += dQ[1,0]* dt*np.cos(self.eta[2,0])
        self.eta[1,0] += dQ[1,0]* dt*np.sin(self.eta[2,0])

        # Base kinematics
        Tb =   np.array([[np.cos(self.eta[2,0]), -np.sin(self.eta[2,0]),   0, self.eta[0,0]],
                        [np.sin(self.eta[2,0]),  np.cos(self.eta[2,0]),    0, self.eta[1,0]],
                        [0,                0,                           1, 0],
                        [0,                0,                           0, 1]
                        ])
        '''
        #---------------exercice3----------------
        # Base update
        d = dQ[1, 0] * dt
        theta = dQ[0, 0] * dt
        x = self.eta[0, 0]
        y = self.eta[1, 0]
        yaw = self.eta[2, 0]

        if self.mode == 'rotate':  # First rotate, then move
            yaw += theta
            x += d * np.cos(yaw)
            y += d * np.sin(yaw)

        elif self.mode == 'move':  # First move, then rotate
            x += d * np.cos(yaw)
            y += d * np.sin(yaw)
            yaw += theta

        else:  # Simultaneous motion
            if abs(dQ[0, 0]) < 1e-5:  # Prevent division by 0
                x += d * np.cos(yaw)
                y += d * np.sin(yaw)
            else:
                r = dQ[1, 0] / dQ[0, 0]
                x += r * (np.sin(yaw + theta) - np.sin(yaw))
                y += -r * (np.cos(yaw + theta) - np.cos(yaw))
            yaw += theta

        self.eta[0, 0] = x
        self.eta[1, 0] = y
        self.eta[2, 0] = yaw

        # Transformation of the MB to Manipulator

        T = np.array([[1,0,0,x],
                      [0,1,0,y],
                      [0,0,1,0],
```

```
                       [0,0,0,1]])
        R = np.array([[np.cos(yaw),-np.sin(yaw),0,0],
                       [np.sin(yaw),np.cos(yaw),0,0],
                       [0,0,1,0],
                       [0,0,0,1]])

        Tb = T @ R  # Base-to-world transform'''

        self.theta[0]+=-np.pi/2

        # Combined system kinematics (DH parameters extended with base DOF)
        dExt = np.concatenate([np.array([  0 , self.r   ]), self.d])
        thetaExt = np.concatenate([np.array([ np.pi/2   ,0     ]), self.theta])
        aExt = np.concatenate([np.array([ 0    , 0    ]), self.a])
        alphaExt = np.concatenate([np.array([  np.pi/2    , -np.pi/2     ]), self.alpha])

        self.T = kinematics(dExt, thetaExt, aExt, alphaExt, Tb)

    '''
        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.revoluteExt)

    '''
        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,0]


    def getBasePose(self):
        return self.eta

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

```python
        return self.dof

    ###
    def getLinkJacobian(self, link):
        return jacobianLink(self.T, self.revoluteExt, link)

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

3.6 Common.py

```python
import numpy as np # Import Numpy

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
    '''
    # 1. Build matrices representing elementary transformations (based on input parameters).
    # T1: Translation along Z-axis by d
    Tz = np.array([[1, 0, 0, 0],
                   [0, 1, 0, 0],
                   [0, 0, 1, d],
                   [0, 0, 0, 1]])
    # R1: Rotation around Z-axis by theta
    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]])
    # T2: Translation along X-axis by a
    Tx = np.array([[1, 0, 0, a],
                   [0, 1, 0, 0],
                   [0, 0, 1, 0],
                   [0, 0, 0, 1]])
    # R2: Rotation around X-axis by alpha
    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]])
    # 2. Multiply matrices in the correct order (result in T).
    T = Tz @ Rz @ Tx @ Rx

    return T

def kinematics(d, theta, a, alpha,Tb):  # have the table as the input , call DH
    '''
        Functions builds a list of transformation matrices, for a kinematic chain,
        descried by a given set of Denavit-Hartenberg parameters.
        All transformations are computed from the base frame.

```

```python
        Arguments:
        d (list of double): list of displacements along Z-axis
        theta (list of double): list of rotations around Z-axis
        a (list of double): list of displacements along X-axis
        alpha (list of double): list of rotations around X-axis

        Returns:
        (list of Numpy array): list of transformations along the kinematic chain (from the base frame)
    '''
    #T = [np.eye(4)]
    T = [Tb]  # Base transformation m ,next transformstion is end from first link ot the base , second transformation is end from second link to the base
    # For each set of DH parameters:
    for i in range(len(d)):
        # 1. Compute the DH transformation matrix for the current joint.
        # 2. Compute the resulting accumulated transformation from the base frame.
        T_accumulated = T[-1] @ DH(d[i], theta[i], a[i], alpha[i])
        # 3. Append the computed transformation to T.

        T.append(T_accumulated)

    return T


# Inverse kinematics
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
    '''
    # 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
    On = np.array([T[-1][:3, 3]]).T  # End-effector's origin (position)
    Z = np.array([[0, 0, 1]]).T  # Z-axis of the base frame
    # 2. For each joint of the robot
    for i in range(len(T)-1):
        #   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
```

```python

        # 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, (On - 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

# Damped Least-Squares
def DLS(A, 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
    '''

    # Create an identity matrix with dimensions matching A @ A.T
    I = np.eye((A @ A.T).shape[0])
     # Compute the DLS
    DLS = A.T @ np.linalg.inv(A @ A.T + damping**2 * I)
    return DLS

def Weighted_DLS(A, damping, W):
    '''
    Function computes the damped least-squares (DLS) solution to the
    matrix inverse problem, incorporating weights for each DOF.

    Arguments:
    A (Numpy array): Task Jacobian (m x n matrix).
    damping (float): Damping factor (regularization term).
    W (Numpy array): Diagonal weighting matrix (n x n).

    Returns:
    (Numpy array): Weighted DLS solution.
    '''
    # Ensure W is a diagonal matrix
    if len(W.shape) == 1:  # If W is a 1D array, convert to diagonal matrix
        W = np.diag(W)

    # Invert W
    W_inv = np.linalg.inv(W)

```

```
    # Compute Weighted DLS
    identity = np.identity(A.shape[0])  # Ensure identity matches DOF
    term1 = W_inv @ A.T
    term2 = A @ term1 + (damping ** 2) * identity
    return term1 @ np.linalg.inv(term2)


# Extract characteristic points of a robot projected on X-Y plane
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
```

4 Conclusion

This lab successfully implemented a mobile manipulator class, incorporated the weighted Damped Least Squares algorithm, and examined simulation errors caused by simplified base kinematics updates. The results clearly demonstrate how weighting influences motion distribution across the base and manipulator, and highlight the importance of accurate kinematic modeling for achieving reliable task execution in mobile manipulators.