Hands-on Intervention

Lab #3 - Task Priority Kinematic Control (1A)

Delivered by:

Rihab Laroussi, u1100330

Supervisor:

Patryk Cieślak

Date of Submission:

09/03/2025

Table of Contents

Introduction

In kinematics, when a system has redundant degrees of freedom, it creates the challenge of controlling the motion of the manipulator to achieve multiple tasks simultaneously. The key idea is to use a control strategy that assigns priorities to tasks. Higher-priority tasks are satisfied first, while lower-priority tasks are addressed in the null space of the higher-priority tasks, ensuring they don't interfere. This approach is implemented using task-priority kinematic control.

In this lab, we explore the control of a planar 3-link manipulator using resolved-rate motion control and task-priority control. The lab consists of two exercises: the first exercise focuses on visualizing the null space motions using resolved-rate motion control and null space projection, while the second exercise implements the hierarchical control strategy for two tasks: end-effector position control and first joint position control.

Exercice 1

1. Methodology

In order to visualize the null space of the planar 3d-link manipulator, we will continue with what we built from Lab 2. Using the implemented resolved-rate motion control to move the end effector towards a desired position while generating null space motions using the null space projector. I computed the null space projector as:

$$\mathbf{P = I \, . \, J\dagger \, * \, J}$$

where J is the jacobian matrix and J† is the pseudoinverse jacobian.

An arbitrary function of time **y** was then used to generate joint velocities for null space motions using the formula:

$$\boldsymbol{\zeta = J\dagger \, (q) \, * \, xE + (\, I - J\dagger (q \,) * \, J(q)) \, * \, y}$$

The robot has three revolute joints. The Denavit-Hartenberg parameter values used in the code are:

The link offsets: d1=0, d2=0, d3=0

The link lengths: a1=0.75, a2=0.5, a3=0.5

The link twist angles: alpha1=0, alpha2=0, alpha3=0

The joint angles initial values are set as theta1=q1=0, tetha2=q2=0.6, tetha3=q3=0.3

The drawing of the robot model with its DH parameters and coordinates systems are in Figure1.

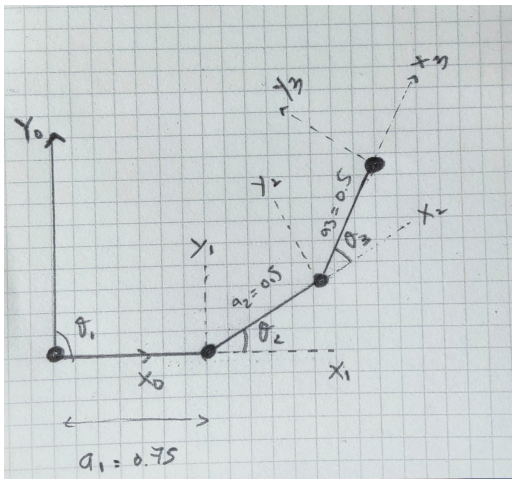


Figure 1: A simple representation of the robot configuration

2. Code

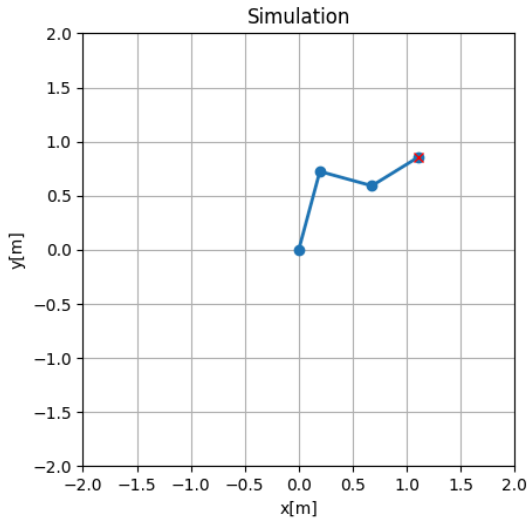
----------Exercice1.py----------

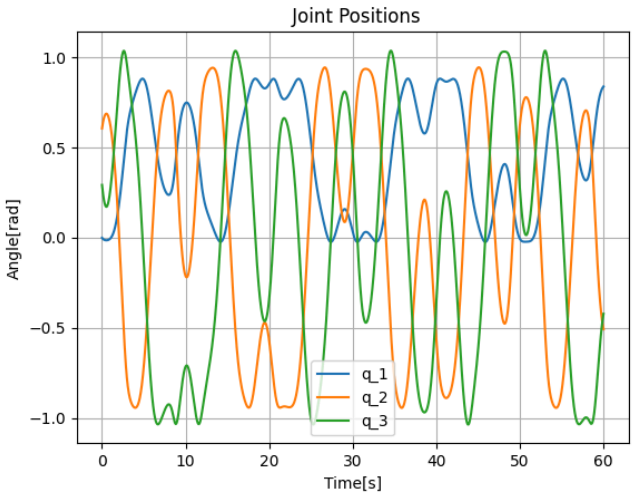
```
# Import necessary libraries
from Common import * # Includes numpy import
import matplotlib.pyplot as plt
import matplotlib.animation as anim
from scipy.linalg import pinv

# Robot definition (3 revolute joint planar manipulator)
d = np.zeros(3)                 # displacement along Z-axis
q = np.array([0, 0.6, 0.3]).reshape(3,1)   # rotation around Z-axis (theta)
alpha = np.zeros(3)                     # displacement along X-axis
a = np.array([0.75, 0.5, 0.5])          # rotation around X-axis
revolute = [True, True, True]           # flags specifying the type of joints
K = np.diag([1, 1])                     # gain

# Setting desired position of end-effector to the current one
T = kinematics(d, q.flatten(), a, alpha) # flatten() needed if q defined as column vector !
sigma_d = T[-1][0:2,3].reshape(2,1)

# Simulation params
dt = 1.0/60.0
Tt = 10 # Total simulation time
tt = np.arange(0, Tt, dt) # Simulation time vector

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

q1 = []
q2 = []
q3 = []

time = []

# Simulation initialization
def init():
    line.set_data([], [])
    path.set_data([], [])
    point.set_data([], [])
    return line, path, point
```

```python
# Simulation loop
def simulate(t):
    global q, a, d, alpha, revolute, sigma_d
    global PPx, PPy

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

    # Update control

    sigma =  T[-1][0:2, 3].reshape(2,1)    # Current position of the end-effector
    err = sigma_d - sigma    # Error in position
    Jbar = J[:2, :]     # Task Jacobian       select only the linear velocities
    Jbar_inv = np.linalg.pinv(Jbar)
    I = np.eye(3)
    P = I - (Jbar_inv @ Jbar)        # Null space projector   P = I - J'+ J
    y = np.array([[np.sin(t), np.cos(t), np.sin(t)]]).T     # Arbitrary joint velocity
    dq = (Jbar_inv @ (K @ err)) + (P @ y)                    # Control signal
    q = q + dt * dq # Simulation update

    # Update drawing
    PP = robotPoints2D(T)
    line.set_data(PP[0,:], PP[1,:])
    PPx.append(PP[0,-1])
    PPy.append(PP[1,-1])
    path.set_data(PPx, PPy)
    point.set_data(sigma_d[0], sigma_d[1])

    q1.append(q[0])
    q2.append(q[1])
    q3.append(q[2])
    time.append(t)

    return line, path, point


# Run simulation
animation = anim.FuncAnimation(fig, simulate, np.arange(0, 60, dt),
                               interval=10, blit=True, init_func=init, repeat=False)
plt.show()

# Joint positions plot
fig = plt.figure()
ax = fig.add_subplot()
ax.set_title(" Joint Positions")
ax.set_xlabel("Time[s]")
ax.set_ylabel("Angle[rad]")
ax.grid()
plt.plot(time, q1, label="q_1")
plt.plot(time, q2, label="q_2")
plt.plot(time, q3, label="q_3")
ax.legend()
plt.show()
```

3. Results

The results of the first exercice are depicted in ***Figure2.*** As we can see in the simulation on the left, the motion of the robot's structure on a 2D plane targeting the end-effector's desired position is visualized, and on the right, the graph presents the evolution of joint positions over time.





2.1. Manipulator with end-effector goal

2.2. Positions of robot's joints

Figure 2: Null space motions simulation

The end-effector moves toward the target position smoothly, while the joints move in a way that combines resolved-rate motion control -to move the end-effector- and null space motions -to reconfigure the joints without affecting the end-effector-. As a result, the joint angles evolve over time, with the first joint q1 changes slightly; whereas, the second and third joint are constantly oscillating due to null space motions.

Exercice 2

1. Methodology

In this exercise, I implemented a Task-Priority (TP) control algorithm for the planar 3-link manipulator with two tasks: end-effector position control and first joint position control. The TP algorithm ensures that the higher-priority task is satisfied first, and the lower-priority task is satisfied as much as possible without interfering with the higher-priority task.

Initially, the task errors were represented as the difference between the desired and current task variables and computed as:

End-Effector Position Error: $\tilde{\sigma}_p = \sigma_{p,d} - \sigma_p$

Joint Position Error: $\tilde{\sigma}_{ji} = \sigma_{ji,d} - \sigma_{ji}$

Then, the Jocobian matrices that relate the joint velocities to the task-space velocities are as follow:

End-Effector Position Jacobian: Jp maps the joint velocities to the end-effector's linear velocity in the xy plane.

$$\mathbf{J}_p = \mathbf{J}_p(\mathbf{q}) \in \mathbb{R}^{2\times 3}$$

Joint Position Jacobian: J1 maps the joint velocities to the velocity of the first joint.

$$J_{ji} = [1, 0, 0] \in \mathbb{R}^{1\times 3}$$

To compute TP, I use the null space projection method. The null space projector ensures that any motion in the null space of task 2 does not interfere with task 1, The null space projector is computed as used previously:

$$P = I - J^{\dagger}J$$

The joint velocity required to minimize the higher-priority task error is calculated using the damped least squares pseudoinverse of the Jacobian, then the total joint velocity that combines the velocities for both tasks is computed using the formula:

$$\dot{\xi}_i = \dot{\xi}_{i-1} + J_i^{\dagger}(\dot{x}_i - J_i\dot{\xi}_{i-1})$$

The output velocity derived from the TP algorithm is modified at every time step to ensure it remains within the boundaries of the maximum joint velocity limit. This is done by scaling the joint velocities if their norm exceeds the specified limit as shown below.

$$s = \max_{i\in 1\ldots n}\left(\frac{|\zeta_i|}{\zeta_{i,\max}}\right)$$

$$\textbf{if} \quad s > 1$$

$$\textbf{return} \quad \frac{\zeta}{s}$$

2. Code

----------Exercice2.py----------

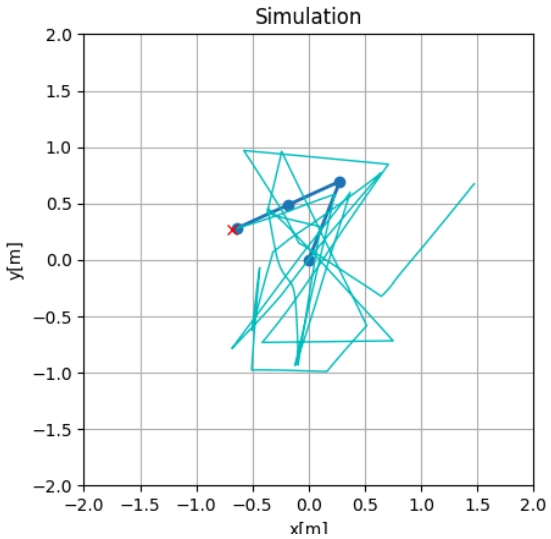
```python
# Import necessary libraries
from Common import * # Includes numpy import
import matplotlib.pyplot as plt
import matplotlib.animation as anim

# Robot definition (3 revolute joint planar manipulator)
d = np.zeros(3) # displacement along Z-axis
q = np.array([0, 0.6, 0.3]).reshape(3,1) # rotation around Z-axis (theta)
alpha = np.zeros(3) # displacement along X-axis
a = np.array([0.75, 0.5, 0.5]) # rotation around X-axis
revolute = [True, True, True] # flags spcifying the type of joints

Task_Priority =  'Task EE'
#Task_Priority = 'Task Joint1'

# Desired values of task variables
sigma1_d = np.array([0.0, 1.0]).reshape(2,1) # Position of the end-effector
sigma2_d = np.array([[0.0]]) # Position of joint 1

dq12_max = 1      # maximum allowed joint velocity vector

# Simulation params
dt = 1.0/60.0
Tt = 10 # Total simulation time
tt = np.arange(0, Tt, dt) # Simulation time vector
time = []         # List to store all time steps
i = 0             # variable used to store the last recorded time step

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

e1 = []     # error for Task1
e2 = []     # error for Task2

# Simulation initialization
def init():
    global sigma1_d, i
    line.set_data([], [])
    path.set_data([], [])
    point.set_data([], [])
    if time:
       i = time[-1]
    else: i = 0
    sigma1_d = np.random.uniform(-1, 1, size=(2, 1))
    return line, path, point

# Simulation loop
def simulate(t):
    global q, a, d, alpha, revolute, sigma1_d, sigma2_d
    global PPx, PPy
    # Update robot
    T = kinematics(d, q.flatten(), a, alpha)
    J = jacobian(T, revolute)

    # Update control
    if Task_Priority == 'Task EE':
        # TASK 1
        sigma1 = T[-1][:2,3].reshape(2,1) # Current position of the end-effector
        err1 = sigma1_d - sigma1 # Error in Cartesian position
        J1 = J[:2, :] # Jacobian of the first task
        Jbar_inv = np.linalg.pinv(J1)
        I = np.eye(3)
        P1 = I - (Jbar_inv @ J1) # Null space projector
        # TASK 2
        sigma2 = q[0] # Current position of joint 1
        err2 = sigma2_d - sigma2 # Error in joint position
        J2 = np.array([1, 0, 0]).reshape(1,3) # Jacobian of the second task
        J2bar = J2 @ P1 # Augmented Jacobian
        # Combining tasks
        dq1 = (DLS(J1,0.1) @ err1).reshape(3, 1) # Velocity for the first task
        dq12 = dq1 + (DLS(J2bar, 0.1)) @ (err2 - J2 @ dq1) # Velocity for both tasks
```

```
else :
    # TASK 1
    sigma2 = q[0] # Current position of the end-effector
    err2 = sigma2_d - sigma2 # Error in Cartesian position
    J1 = np.array([1, 0, 0]).reshape(1,3) # Jacobian of the first task
    Jbar_inv = np.linalg.pinv(J1)
    I = np.eye(3)
    P1 = I - (Jbar_inv @ J1) # Null space projector
    # TASK 2
    sigma1 = T[-1][:2,3].reshape(2,1) # Current position of joint 1
    err1 = sigma1_d - sigma1 # Error in joint position
    J2 = J[:2, :] # Jacobian of the second task
    J2bar = J2 @ P1 # Augmented Jacobian
    # Combining tasks
    dq1 = (DLS(J1,0.1) @ err2).reshape(3, 1) # Velocity for the first task
    dq12 = dq1 + (DLS(J2bar, 0.1)) @ (err1 - J2 @ dq1) # Velocity for both tasks

# Make sure the output velocity does not exceed the maximum joint velocity limit
s = np.max(dq12/dq12_max)   # s is the maximum ratio
if s > 1:
    dq12 = dq12/s

q = q + dq12 * dt # Simulation update

# Update drawing
PP = robotPoints2D(T)
line.set_data(PP[0,:], PP[1,:])
PPx.append(PP[0,-1])
PPy.append(PP[1,-1])
path.set_data(PPx, PPy)
point.set_data(sigma1_d[0], sigma1_d[1])



e1.append(np.linalg.norm(err1))      # store the history of error magnitude for task1
e2.append(np.linalg.norm(err2))      # store the history of error magnitude for task2

# Store total timestamp of the simulation
time.append(t + i)

return line, path, point
```

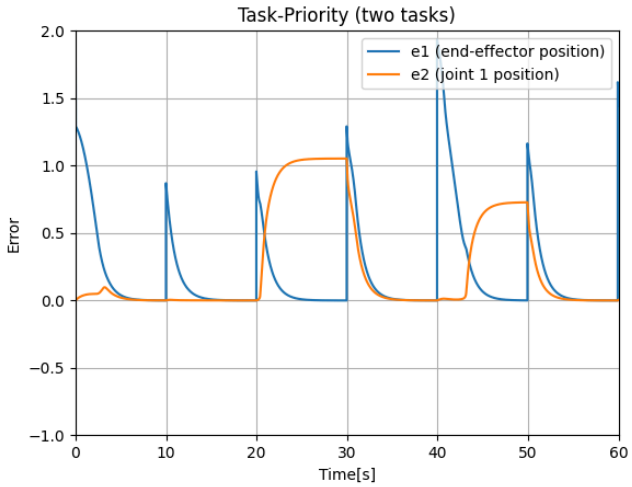
```
# Post simulation plotting
fig_joint = plt.figure()
ax = fig_joint.add_subplot(111, autoscale_on=False, xlim=(0, 60), ylim=(-1, 2))
ax.set_title("Task-Priority (two tasks)")
ax.set_xlabel("Time[s]")
ax.set_ylabel("Error")
ax.grid()
plt.plot(time, e1, label="e1 (end-effector position)")
plt.plot(time, e2, label="e2 (joint 1 position)")
ax.legend(loc='upper right')
plt.show()
```

3. Results

The results of the second exercice are depicted in ***Figure3*** and ***Figure4***. As we can see, in the simulation on the left, the motion of the robot structure and on the right, the graph presents the progression of the norm of control errors for the tasks over time.

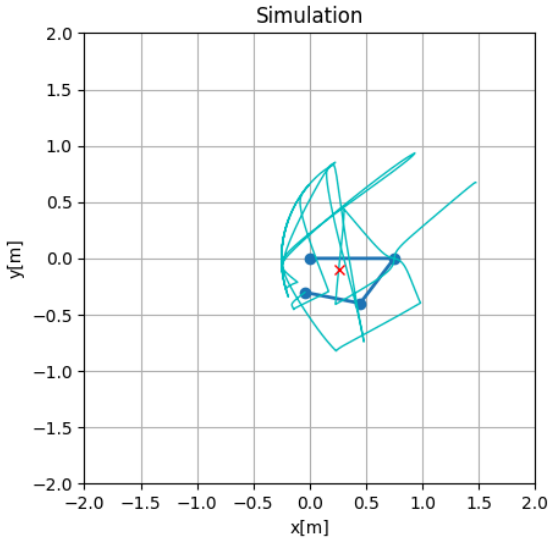


3.1. Motion of the robot

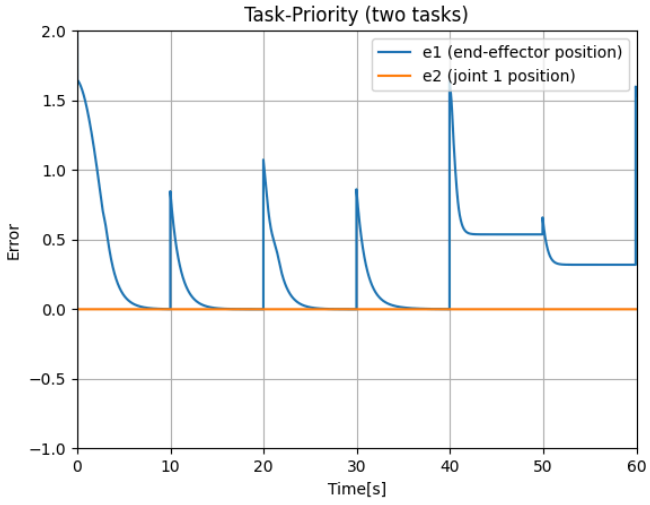


3.2 Control errors of the tasks over time

Figure3: End-Effector position task as the priority



4.1. Motion of the robot



4.2 Control errors of the tasks over time

Figure4: Joint 1 position task as the priority

Figure3 shows the case where the priority task is the end effector position. The robot will prioritize moving the EE to the desired position and the first joint will move only if it does not interfere with the first task. Every 10s, the EE is assigned a new random position and its position error **e1** decrease over time and converges to zero. However, the first joint error **e2** doesnt always converge to zero when it conflicts with Task1.

Figure4 shows the case where the priority task is the joint1 position. The robot continuously prioritizes reaching the position for joint 1 then attempts to move the end-effector to its designed location. As shown in the output, the robot doesnt always succeed in getting the EE as supposed to; thus, the error **e1** is converged to zero and remains at zero as the joint's position is maintained consistently wether the EE ends up moving or not.

Questions

1. What are the advantages and disadvantages of redundant robotic systems?

In robotics, redundant systems are systems that have more degrees of freedom than are strictly necessary to perform a gicen task.

Advantages:

- They can perform tasks in multiple configurations, enabling complex movements like maneuvering around obstacles or operating in confined spaces.
- If a joint or actuator fails, redundancy lets the system reconfigure and continue working with the remaining degrees of freedom.
- Redundancy also ensures smoother trajectories by eliminating jerky movements, which is advantageous for precision tasks like painting or welding.

Disadvantages:

- Without proper constraints, they might result in self-collisions or undesirable joint configurations.
- They require more complex and advanced algorithms.
- Since they have infinite possible solutions, solving the inverse kinematics is more computationally expensive.

2. What is the meaning and practical use of a weighting matrix W, that can be introduced in the pseudo inverse/DLS implementation?

In pseudo-inverse or damped least squares (DLS) implementations, the weighting matrix **W** is used to give preference or impose penalties on particular joints within robotic systems. It is a diagonal matrix that allocates weights to specific joints, affecting their impact on movement. It adjusts the optimization

criterion from minimizing $\| \dot{\mathbf{q}} \|_2$ (joint velocities) to minimizing $\dot{\mathbf{q}}^T \mathbf{W} \dot{\mathbf{q}}$, with higher weights imposing greater penalties on particular joints.

There are several practical uses for the weighting matrix:

- Higher weights can be assigned to joints nearing their limits to limit their motion and prevent mechanical damage.
- Joints near sensitive areas, such as those in collaborative robots, can be given weights to ensure smoother, slower movements.
- Weights can be dynamically adjusted to avoid configurations where the Jacobian loses rank, known as singularities.
- Priority can be given to joints crucial for the primary task, such as end-effector positioning, while deprioritizing others.
- The weighting matrix W can be modified in real time based on external constraints like obstacles or changes in payload.

As examples, in industrial robotics, such as a welding arm, assigning higher weights to joints with high inertia can reduce abrupt movements, ensuring precise and stable welds. When robots operate near their joint limits, adjusting weights can prevent mechanical stress, extending their lifespan. Additionally, In surgical robotics used for minimally invasive procedures, heavily penalizing joints near delicate tissues ensures smooth and safe movements, while dynamically adjusting weights to avoid singularities keeps the robot fully operational during critical moments.

Conclusion

In conclusion, throughout this lab we focused on learning how to implement task-priority control using an analytical method for a planar 3-link manipulator. The results of the first exercice demonstaretd how the robot achieves its end-effector goal while optimizing joint configurations. While in the second exercice, the TP control showed how task hierarchies influence error convergence and robot behavior. They both highlighted the importance of task prioritization and null space utilization in achieving complex robotic tasks.