Students:
Loc Thanh Pham [u1992429@campus.udg.edu]

Lab4: Task-Priority kinematic control

Introduction

This lab session in focused on Task-Priority algorithm in its recursive form, to allow for an arbitrary hierarchy of tasks. The implementation is split into two parts. The first part is the definition of different tasks as Python classes and the second part is the recursive TP itself, with a simulation on a 3-link planar manipulator.

Methodology

1 Exercise 01

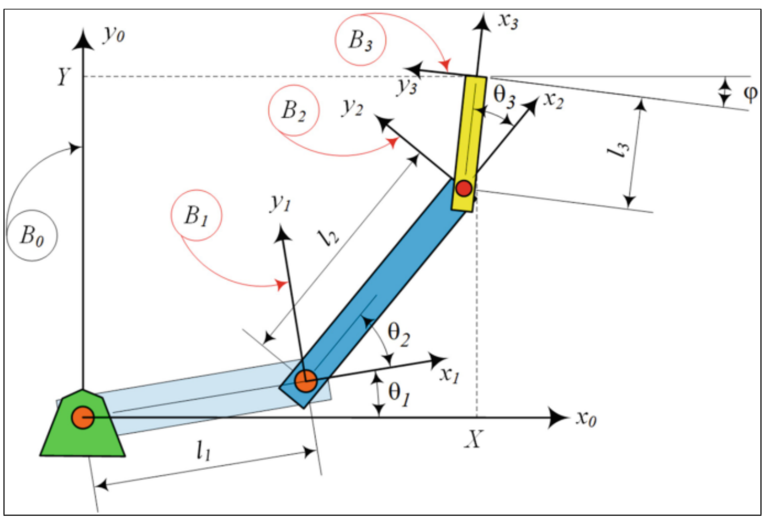


Figure 1: Three links manipulator model, including DH parameters and coordinate systems

Link	**d**	θ	a	α	**Home**
1	0	θ_1	a_1	0	0
2	0	θ_2	a_2	0	0
3	0	θ_3	a_3	0	0

Table 1: Denavit-Hartenberg table of the three links manipulator

Denavit-Hartenberg formulation which is used to compute transformation matrix between coordinate systems:

$$T_n^{n-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) & 0 & 0 \\ \sin(\theta_i) & \cos(\theta_i) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha_i) & -\sin(\alpha_i) & 0 \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

$$T_n^0(q) = T_1^0 T_2^1 T_3^2 \dots T_n^{n-1} = \begin{bmatrix} R_n^0(q) & O_n^0(q) \\ 0 & 1 \end{bmatrix} \tag{2}$$

Implementation on python code:

1.1 Definition of parameters of the simulation

Listing 1: **Robot definition (3 revolute joint planar manipulator)**

```
# Robot definition (3 revolute joint planar manipulator)
d = np.zeros(3)                       # displacement along Z-axis
q = np.array([0.2, 0.5, 0.2])    # rotation around Z-axis (theta)
a = np.array([0.75, 0.5, 0.3])  # displacement along X-axis
alpha = np.zeros(3)                   # rotation around X-axis
revolute = [True, True, True]    # flags specifying the type of joints
n_DoF = len(revolute)                 # Number of Degree of Freedom
dq_max = np.array([3, 3, 3]) # The maximum joint velocity limit
# Setting desired position of end-effector to the current one
sigma_d = np.array([1.0, 1.0])
```

1.2 Implementation of subclasses of the base Task class

Listing 2: **Position of the end-effector (2D)**

```
'''
    Subclass of Task, representing the 2D position task.
'''
class Position2D(Task):
    def __init__(self, name, desired, robot: Manipulator):
        super().__init__(name, desired, robot)
        self.J = np.zeros((2, robot.getDOF()))
        self.err = np.zeros((2, 1))

    def update(self, robot: Manipulator, dt: float, newDesired):
```

```python
        DoF = robot.getDOF()
        # Update Jacobean matrix - task Jacobian
        self.J  = robot.getLinkJacobean(self.link_index)[[0,1]].reshape(self.task_dim,DoF)
        # Update task error
        self.err = self.getDesired() - robot.getJointPos2D(self.link_index)
        # Compute feed-forward velocity
        self.ffVel = (newDesired - self.getDesired()) / dt
        # Set new desired
        self.setDesired(newDesired)
        return True
```

Listing 3: **Orientation of the end-effector (2D)**

```python
'''
    Subclass of Task, representing the 2D orientation task.
'''
class Orientation2D(Task):
    def __init__(self, name, desired, robot: Manipulator):
        super().__init__(name, desired, robot)
        self.J = np.zeros((1, robot.getDOF()))
        self.err = np.zeros((1, 1))

    def update(self, robot: Manipulator, dt: float, newDesired):
        DoF = robot.getDOF()
        # Update Jacobean matrix - task Jacobian
        self.J  = robot.getLinkJacobean(self.link_index)[[5]].reshape(self.task_dim,DoF)
        # Update task error
        self.err = self.getDesired() - robot.getJointOrientation2D(self.link_index)
        # Compute feed-forward velocity
        self.ffVel = (newDesired - self.getDesired()) / dt
        # Set new desired
        self.setDesired(newDesired)
        return True
```

Listing 4: **Configuration of end-effector (2D)**

```python
'''
    Subclass of Task, representing the 2D configuration task.
'''
class Configuration2D(Task):
    def __init__(self, name, desired, robot: Manipulator):
```

```python
        super().__init__(name, desired, robot)
        self.J = np.zeros((6, robot.getDOF()))                               # Initialize with proper dimensions
        self.err = np.zeros((6, 1))                                           # Initialize with proper dimensions

    def update(self, robot: Manipulator):
        DoF = robot.getDOF()
        # Update Jacobean matrix - task Jacobian
        self.J  = robot.getLinkJacobean(self.link_index)[[0,1,5]].reshape(self.task_dim,DoF)
        # Update task error
        self.err = self.getDesired() - robot.getJointConfiguration2D(self.link_index)
        # Compute feed-forward velocity
        self.ffVel = (newDesired - self.getDesired()) / dt
        # Set new desired
        self.setDesired(newDesired)
        return True
```

1.3 Implementation of the recursive formulation of the Task-Priority algorithm

Listing 5: **Simulation Loop**

```python
# Simulation loop
def simulate(t):
    global tasks
    global robot
    global PPx, PPy
    global time, N_iter, Tt

    ### Recursive Task-Priority algorithm
    # Initialize null-space projector
    P   = np.eye(n_DoF, n_DoF)
    # Initialize output vector (joint velocity)
    dq  = np.zeros((n_DoF, 1))
    # Loop over tasks
    for i in range(len(tasks)):
        # Update task state
        tasks[i].update(robot)
        # Compute augmented Jacobian
        Jbar    = tasks[i].J @ P
        # Compute task velocity
        # Accumulate velocity
```

```
        dq       = dq + DLS(Jbar, 0.1) @ (tasks[i].err - tasks[i].J @ dq)
        # Update null-space projector
        P        = P - DLS(Jbar, 0.001) @ Jbar
    ###
    # 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])

    return line, path, point
```

1.4 Definition of different task hierarchies (lists of tasks)

Listing 6: **Robot definition (3 revolute joint planar manipulator)**

```
tasks = [
Position2D("End-effector position", np.array([0.0, 1.2]).reshape(2,1),
    robot),
Position2D("Joint 1 position", np.array([0.0, 0.2]).reshape(2,1), robot),
Orientation2D("End-effector orientation", np.array(0.0), robot),
Configuration2D("End-effector configuration", np.array([0.0, 1.2, 0.0]).
    reshape(3,1), robot)
]
```

1.5 Random desired end-effector position

Listing 7: **Init Function**

```
# Simulation initialization
def init():
    global tasks, N_iter
    line.set_data([], [])
    path.set_data([], [])
    point.set_data([], [])
    q1.set_data([], [])

    # Set random new desired position
```

```
    theta_rand = 2 * math.pi * np.random.rand()
    length_rand = np.random.uniform(1.0, sum(a[0:3]))
    tasks = [
        Position2D("End-effector position", np.array([length_rand * np.cos(theta_rand), length_rand * np.sin(theta_rand)]).reshape(2,1), robot),
        # Position2D("Joint 1 position", np.array([0.0, 0.2]).reshape(2,1), robot)
        # Orientation2D("End-effector orientation", np.array(0.0), robot)
        # Configuration2D("End-effector configuration", np.block([np.array([length_rand * np.cos(theta_rand), length_rand * np.sin(theta_rand)]), 0.0]).reshape(3,1), robot),
    ]
    # Set number of iteration
    N_iter += 1

    return line, path, point, q1
```

2 Exercise 02

The goal of this exercise is to extend the code of Exercise 1, adding new features that allow for more flexible task definition. These features include: link selection for position and orientation tasks, gain matrices (with associated weighted DLS implementation) and the feed-forward velocity component (tracking).

Implementation on python code:

2.1 Manipulator class

Listing 8: **Get the transformation for a selected link**

```
'''
    Method that return the transformation of a selected link

    Argument:
    link_index (integer):

    Returns:
    (np.array((4,4))): transformation matric of the selected link
'''
def getLinkTranform(self, link_index: int):
    return self.T[link_index]
```

Listing 9: **Get the Jacobian for a selected link**

```
'''
    Method that return the Jacobean for a selected link

    Argument:
    link_index (integer):

    Returns:
    (np.array((6,DoF))): Jacobean matric of the selected link
'''
def getLinkJacobean(self, link_index):
    # Transformation matrix from base to each link, until get the
   selected link
    TT = []
    for i in range(link_index+1):
        TT.append(self.getLinkTranform(i))
    # return Jacobean matrix for selected link
    return jacobian(TT, self.revolute)
```

2.2 Task class

Listing 10: **Methods work with the Gain Matrix and Feed-Forward Velocity**

```
'''
    Method returning the gain matrix (K).
'''
def getGainMatrix(self):
    return self.K
'''
    Method setting the gain matrix (K).
'''
def setGainMatrix(self, K):
    self.K = K
    return True

 '''
    Method returning the feed-forward velocity.
'''
def getFFVelocity(self):
    return self.ffVel
'''
    Method setting the gain matrix (K).
'''
def setFFVelocity(self, ffVel):
```

```
    self.ffVel = ffVel
    return True
```

2.3 Task class with task index

Listing 11: **Init Function**

```
'''
    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, robot: Manipulator):
        self.name = name # task title
        self.sigma_d = desired # desired sigma
        self.task_dim = np.shape(desired)[0] # Task dimension
        self.J = np.zeros((self.task_dim, robot.getDOF()))
        self.err = np.zeros((self.task_dim, 1))
        # Get joint number of task from name of the task
        self.link_index = self.name_to_link_index(name, robot.getDOF())

        self.K = np.eye(self.task_dim,self.task_dim) * 1
        self.ffVel = np.eye(self.task_dim,1)
    '''
        Method getting link index.

        Arguments:
        name: string
        DoF

        Return:
        link_index: integer
    '''
    def name_to_link_index(self, name: str, DoF):
        if name.split()[0] == "End-effector":
            # End-effector task
            return DoF
        else:
```

```
            # Joint i position task
            return int(name.split()[1])
```

2.4 Desired trajectory

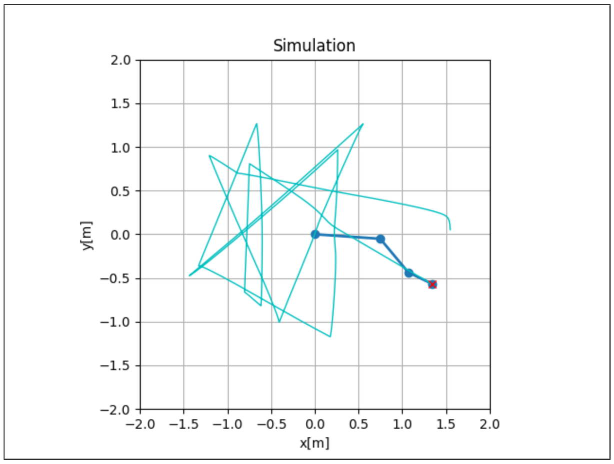
Listing 12: **Desired Trajectory**

```
    r_origin    = 1.0
    r_A         = 0.0
    r_freq      = 5.0
    pos_ref     = 0.5
    r           = r_origin + r_A * np.sin(r_freq*t)
    newDesired = [(np.array([np.cos(pos_ref*t) * r, np.sin(pos_ref*t) * r
   ]).reshape(2,1)), np.array([0.0])]
```

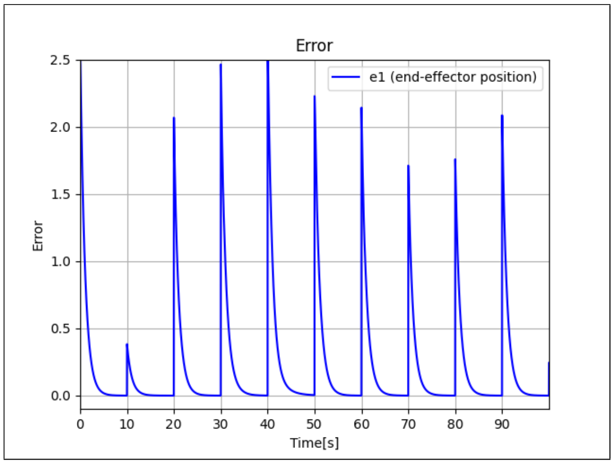
Result

3 Exercise 1:

3.1 One task -> 1: end-effector position



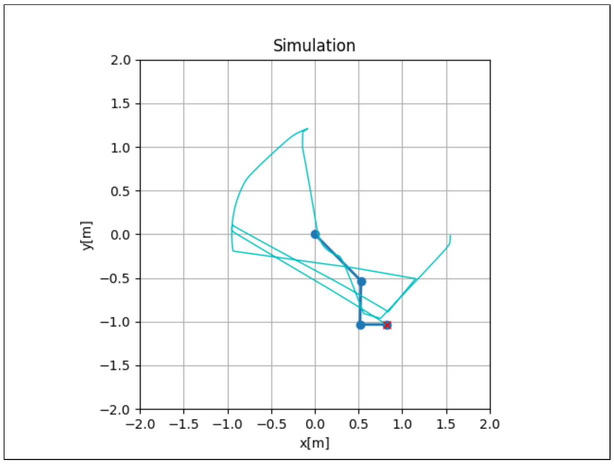
(a) Simulation of the manipulator



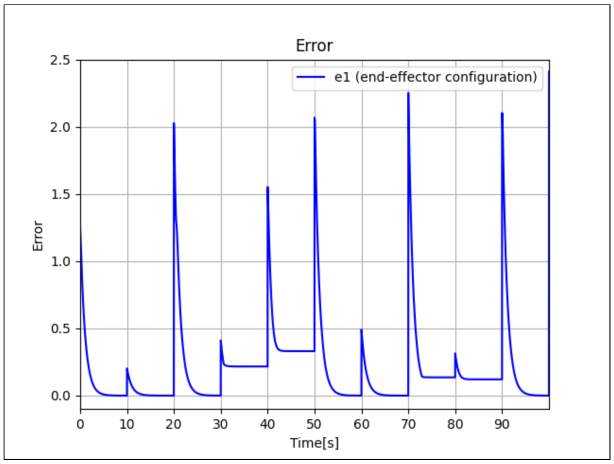
(b) Evolution of the TP control errors

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

3.2 One task -> 1: end-effector configuration



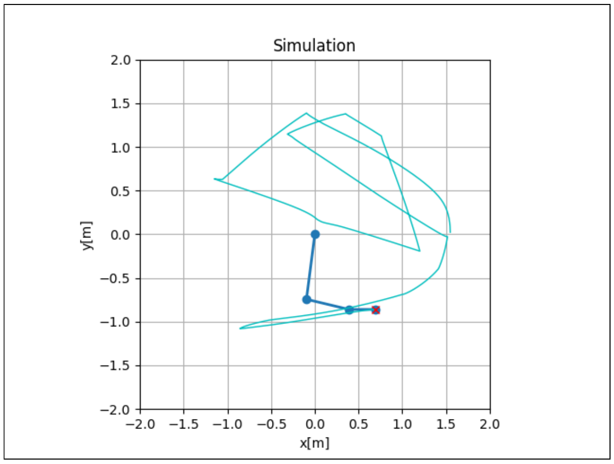
(a) Simulation of the manipulator



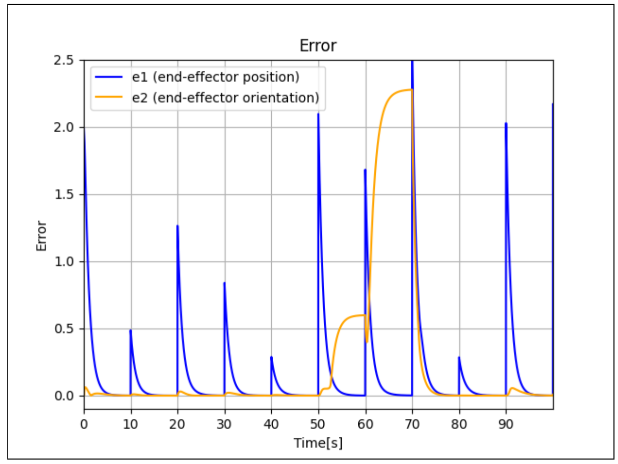
(b) Evolution of the TP control errors

Figure 3: One task -> 1: end-effector configuration (Position: Random; Heading: 0.0 deg)

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



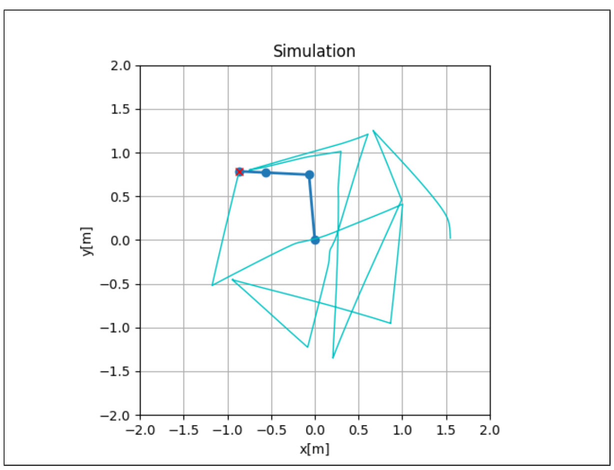
(a) Simulation of the manipulator



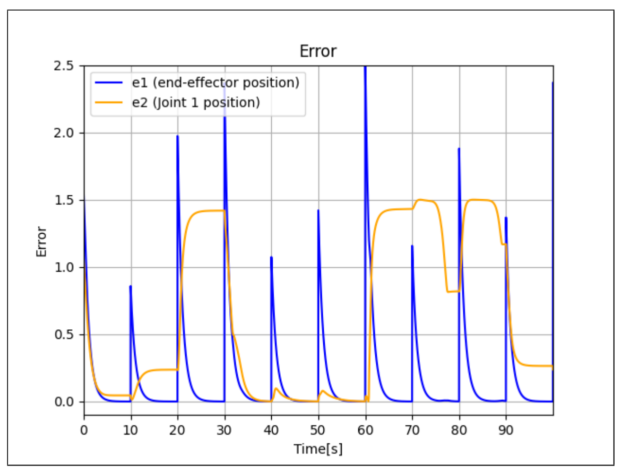
(b) Evolution of the TP control errors

Figure 4: Two tasks -> 1: end-effector position, 2: end-effector orientation (Position: Random; Heading: 0.0 deg)

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



(a) Simulation of the manipulator

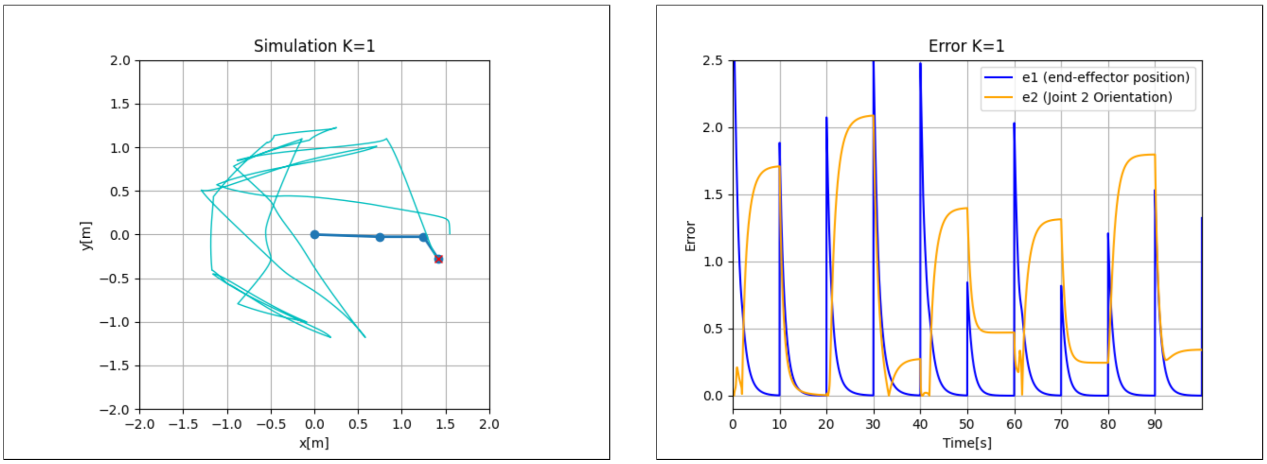


(b) Evolution of the TP control errors

Figure 5: Two tasks -> 1: end-effector position, 2: joint 1 position (End-Effector Position: Random; Joint 1 Position: [0.0, 0.75])

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

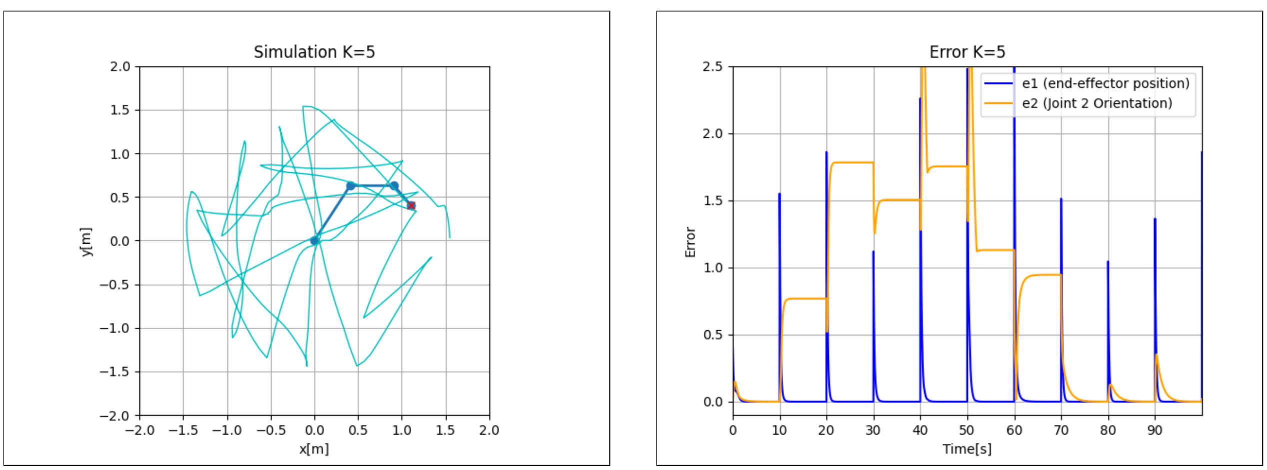
4.1 K = 1.0



(a) Simulation of the manipulator (b) Evolution of the TP control errors

Figure 6: K=1.0

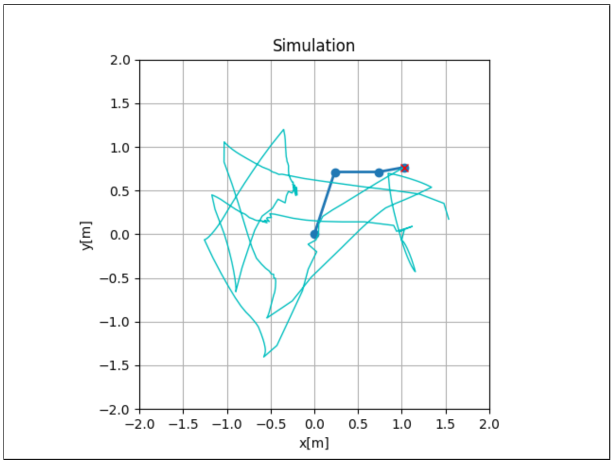
4.2 K = 5.0



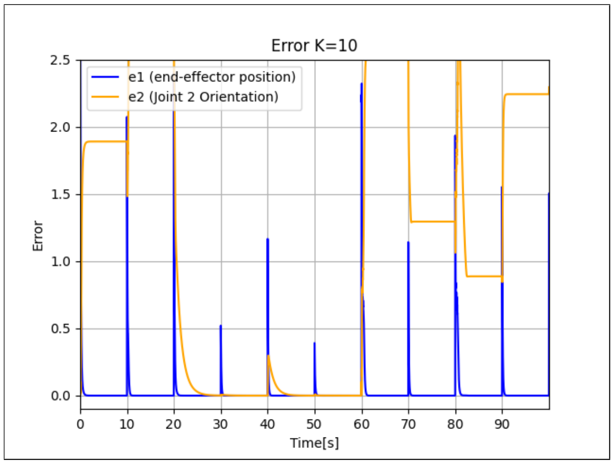
(a) Simulation of the manipulator (b) Evolution of the TP control errors

Figure 7: K=5.0

4.3 K = 10.0



(a) Simulation of the manipulator



(b) Evolution of the TP control errors

Figure 8: K=10.0

4.4 Tracking

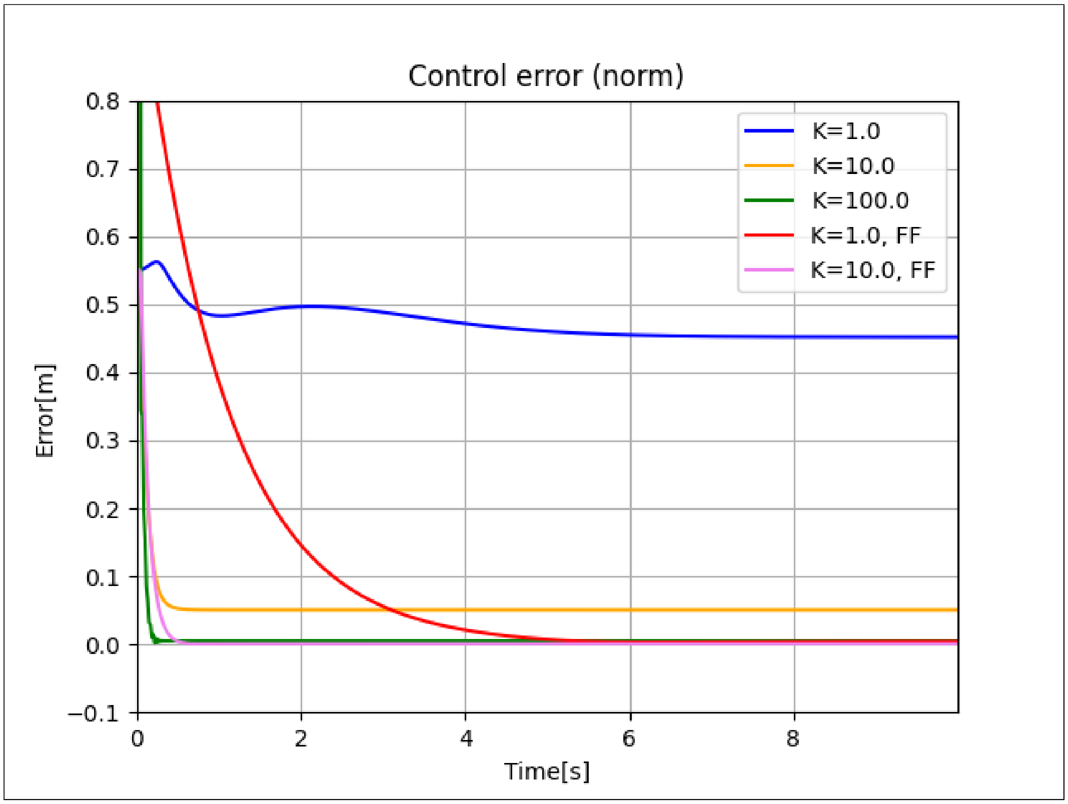


Figure 9: Error analysis: Evolution of the TP control errors

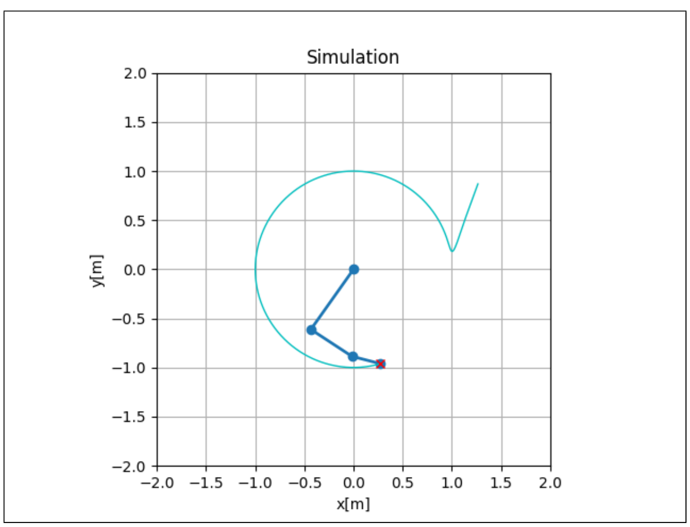


Figure 10: Simulation of the manipulator