HANDS-ON INTERVENTION

Lab # 1

Delivered by:

Muhammad Faran Akram (u1999088)

Worked with:

Solomon Chibuzo Nwafor (u1999124)

Supervisor:

Patryk Cieślak

Date of Submission:

26/02/2024

Table of Contents

1. Introduction:

In this lab, We implemented several helper functions to perform a kinematic simulation of a planar robotic manipulator and resolved motion control. This includes implementing Denavit-Hartenberg (DH) formulation, computing Jacobians as the building blocks of different solution methods like **Pseudo-inverse, Jacobian transpose** and **Damped least-squares** method for resolved rate motion control. Moreover, The implementation takes advantage of the feedback control loop to improve the accuracy and responsiveness of the system.

The detailed implementation, results and explanations are provided in the following sections.

2. Part 1(Kinematics):

In this section, We implemented helper functions ***DH()*** *and* ***kinematics().*** These functions provide the required functionality to move robotic arm based on kinematics.

2.1. Denavit-Hartenberg (DH):

This implemented function which computes the transformation matrix using Denavit-Hartenberg parameters. DH parameters are given below:

Parameters	**Symbol**	**Description**	**Units**
Link Offset	*d*	Distance along the previous z-axis from the previous x-axis to the current x-axis (translation along z).	Length (meters)
Joint Angle	θ	Angle about the previous z-axis from the previous x-axis to the current x-axis (rotation about z).	Angle (radians)
Link Length	a	Distance along the current x-axis from the previous z-axis to the current z-axis (translation along x).	Length (meters)
Link Twist	α	Angle about the current x-axis from the previous z-axis to the current z-axis (rotation about x).	Angle (radians)

The transformation matrix T, can be found by sequentially applying translation and rotation operations according to the standard DH convention.

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

The implementation is given below:

```
def DH(d, theta, a, alpha):

    # T_z(d) : Translation along z-axis
    M1 = block_diag(np.identity(2), np.identity(2))
    M1[2, 3] = d

    # R_z(theta) : Rotation about z-axis
    M2 = block_diag(np.array([[np.cos(theta), -np.sin(theta)],
                              [np.sin(theta),  np.cos(theta)]]), 1, 1)

    # T_x(a)  : Translation along x-axis
    M3 = block_diag(np.identity(2), np.identity(2))
    M3[0, -1] = a

    # R_x(alpha) : Rotation about x-axis
    M4 = block_diag(1, np.array([[np.cos(alpha), -np.sin(alpha)],
                                 [np.sin(alpha),  np.cos(alpha)]]), 1)

    # Combine all transformations T = T_z(d) · R_z(theta) · T_x(a) · R_x(alpha)
    T = M1 @ M2 @ M3 @ M4
    return T
```

2.2. Kinematics:

The function ***kinematics()*** creates the final transformation list, a chain of transformations from vectors of Denavit-Hartenberg parameters.The aim of this function is to compute the forward kinematics of a robotic manipulator using the Denavit-Hartenberg (DH) convention. Total transformation from 0^{th} frame to n^{th} frame is given by:

$$T_n^{\ 0} = T_1^{\ 0}.T_2^{\ 1}...T_n^{\ n-1}$$

```
def kinematics(d, theta, a, alpha):
  T = [np.eye(4)] # Base transformation
   # For each set of DH parameters:
   # 1. Compute the DH transformation matrix.
   # 2. Compute the resulting accumulated transformation from the base frame.
   # 3. Append the computed transformation to T.
   for d_i,theta_i,a_i,alpha_i in zip(d,theta,a,alpha): #iterating over the vectors
      T.append(T[-1]@DH(d_i,theta_i,a_i,alpha_i))
      # Appending a new transformation after multiplying it with the last transformation.
   return T
```

2.3. Graphs:

To get the desired graph, We changed the rate of change of joint velocities and made the revolute joint 2 move faster than the first revolute joint.

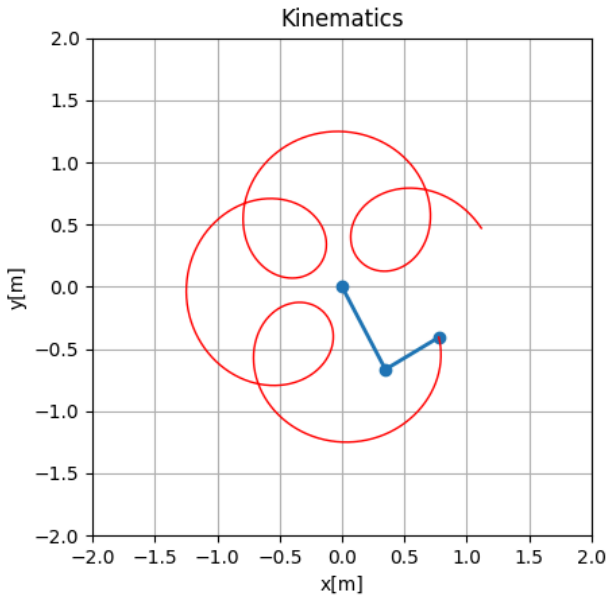

Figure 1. Robotic arm trajectory with different rate of change of velocities.

As we changed the joint velocity of joint 2 more than joint 1, therefore the graph below shows a rapid increase in velocity of joint 2 than joint 1.

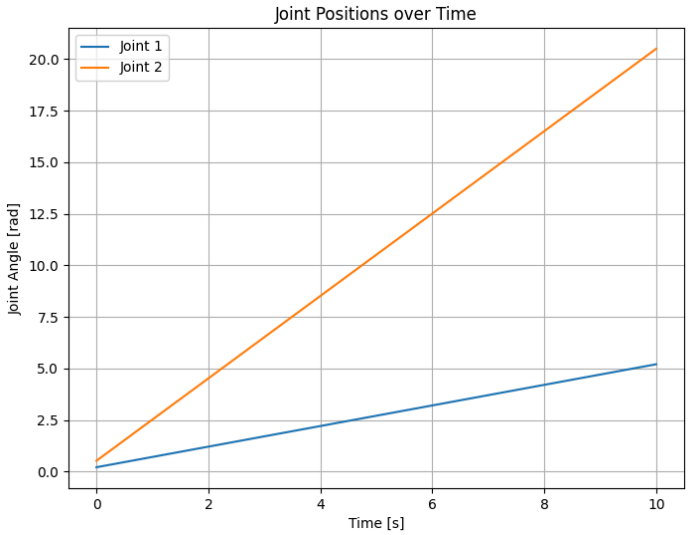

Figure 2. Joint positions over time of robotic arm.

3. Part 2 (Resolved Rate Kinematics):

In this section, We implemented three solutions of resolved rate kinematics named as; **Jacobian Transpose**, **Pseudoinverse Jacobian** and **Damped Least Square** (***DLS***) solution with feedback control loop. All of the aforementioned solutions leverage the helper functions explained in subsections below. The implemented equation for joint velocities is given below:

$$\zeta = J^{-1} x_E$$

3.1. Control:

In simulate() function we implemented the feedback loop for our system using following equations:

$$\sigma_E + K\tilde{o}_E$$
$$\tilde{o}_E = \sigma_{E,d} - \sigma_E$$

Where,

- $\tilde{o}_E$ *is feedback action*
- *K is the Gain.*

```
def simulate(t,controller_solution):
  global d, q, a, alpha, revolute, sigma_d
  global PPx, PPy

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

  # Update control
  #extracting the robot pos in 2D plane for sigma
  P = robotPoints2D(T)
  P_sigma = [P[0, -1], P[1, -1]]
  sigma = np.array(P_sigma)      # Position of the end-effector
  err =  sigma_d - sigma       # Control error  ~sigma_E = sigma_E,d - sigma_E
  print(err)
  delta_f = K @ err #feedback action   df = K x ~sigma_E
  # X_dot_E = sigma_dot_E + delta_f  # No compensation sigma_dot_E ->0
  dq =control(controller_solution, J[0:2, :]) @ delta_f  # vel_vect = control_sol x delta_f
  q += dt * dq # velocity to position
  update_error_norm(q,err, controller_solution)
  # Update drawing
  line.set_data(P[0,:], P[1,:])
  PPx.append(P[0,-1])
  PPy.append(P[1,-1])
  path.set_data(PPx, PPy)
  point.set_data(sigma_d[0], sigma_d[1])

  return line, path, point
```

3.2. Jacobian:

The geometrical Jacobian represents the relationship between joint velocities and end-effector velocities.. The purpose of this function is to compute the Jacobian matrix, which serves as a mapping between the joint velocities and the end-effector velocities. In other words, it maps how changes in joint angles or displacements (depending on the joint type) affect the linear and angular velocity of the end-effector.

```
def jacobian(T, revolute):
   J = []
   O_n = T[-1][:3,-1]  # points of last transform
   for Ti,rev_flag in zip(T,revolute):
      Ri = Ti[:3,:3]  #Rotation matrix 3x3 matrix
      Oi = Ti[:3,-1] #Origin vector
      zi = Ri[:3,-1]  #z-vector
```

```
        J.append(np.vstack([(np.cross(rev_flag*zi,(O_n-Oi)) +
                             (1-rev_flag)*zi).reshape(3,1),(rev_flag*zi).reshape(3,1)]))
        # For revolute joints (rev_flag=1):
        #   J_i =[ [ z_i x (O_n - O_i) ],
        #          [       z_i         ]]
        # For prismatic joints (rev_flag=0):
        #   J_i =[ [ z_i ],
        #          [ 0 ]]
    J = np.hstack(J) #stacking jacobians

    return J
```

3.3. Damped Least-Squares (DLS):

The function ***DLS()*** implements the Damped Least Squares (DLS) method to compute a DLS pseudo-inverse solution for resolved-rate motion control. The function modifies the inverse computation by adding a small damping factor ($\lambda^2 I$). The equation implemented is as follows:

$$A_{DLS} = A^T(AA^T + \lambda^2 I)^{-1}$$

- λ *is small regularization factor*
- *A is the original Jacobian Matrix.*

DLS method is used where there is a high chance of facing singularity. The DLS method helps improve numerical stability in the presence of singularities by regularizing the inversion process.. A higher damping value results in smoother but less precise solutions and A lower damping value gives solutions closer to the true pseudo-inverse but may still be unstable near singularities.

```
def DLS(A, damping):
    # Compute the DLS pseudo-inverse using the formula:
    # A_dls = A^T * (A*A^T + λ^2*I)^(-1)
    DLS = A.T@np.linalg.inv(A@A.T+(damping**2)*np.identity(2))

    return DLS
```

3.3.1. Plot:

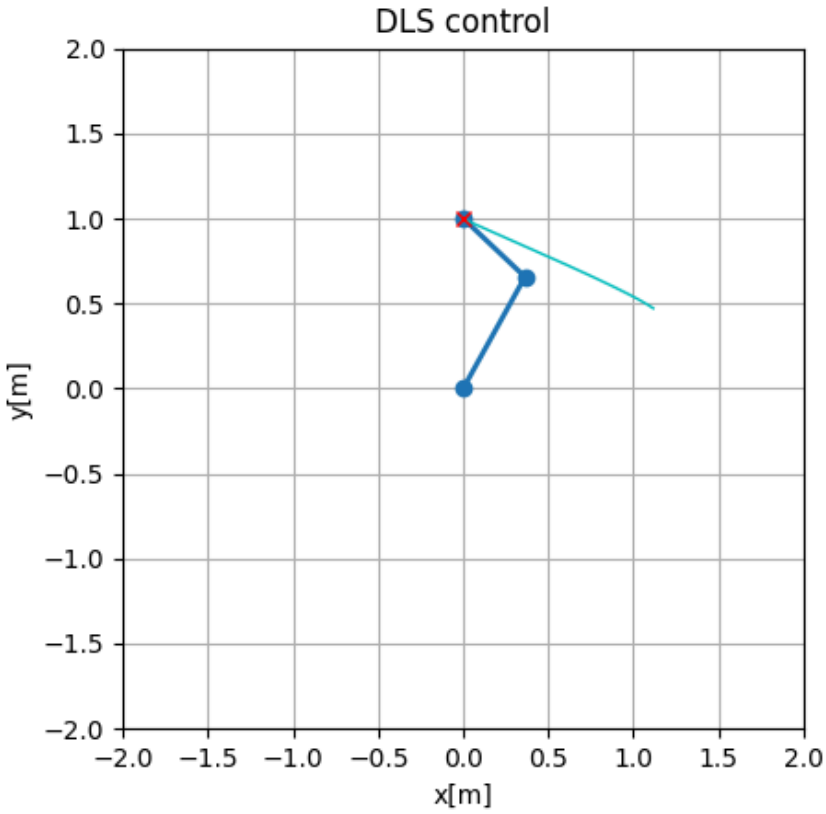

Figure 3. Simulation using DLS method.

3.4. Pseudoinverse:

To avoid unpredictable behavior of the robotic arm near singularities, the Jacobian matrix should not become rank-deficient. At singular configurations, the Jacobian loses full row rank, meaning that some rows become linearly dependent or approach zero, leading to loss of 1 DoF. To handle this issue, the pseudoinverse of the Jacobian is used, providing a least-squares solution that minimizes control errors and helps manage singularities more robustly. However, near singularities, even the pseudoinverse can produce large joint velocities as shown in Figure 7.

$$\zeta = J^{+} x_{E}$$

Where,

- $J^{+}\ is\ Pseudo - Inverse$

3.4.1. Plot:

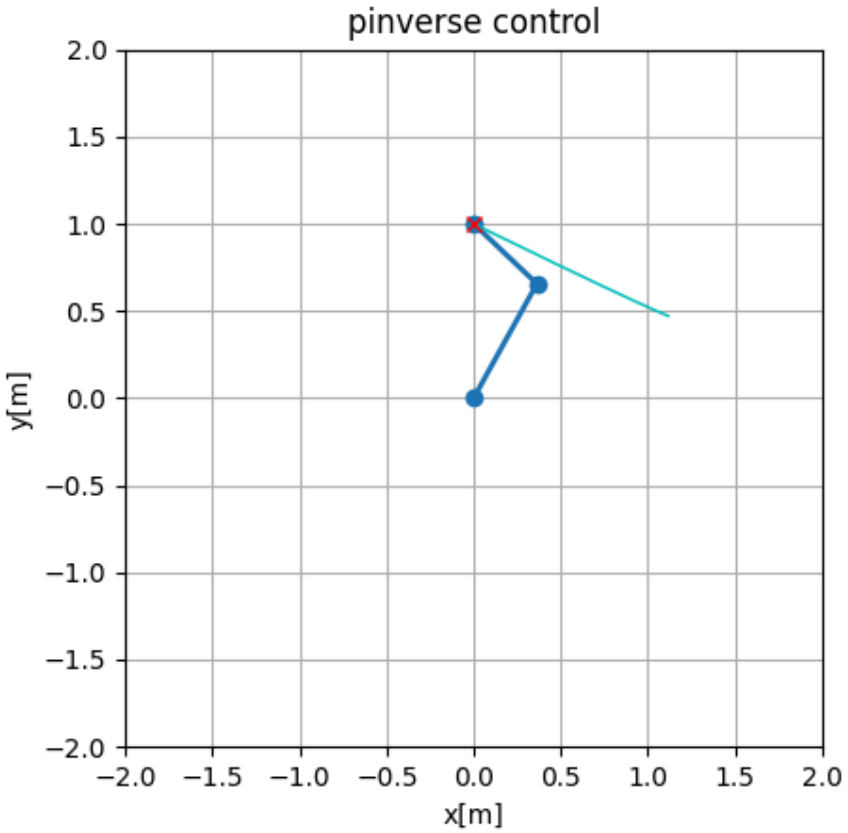

Figure 4. Simulation using Pseudoinverse method.

3.5. Jacobian transpose:

Instead of inverting the Jacobian matrix, this method uses transpose jacobian, providing a computationally efficient alternative. This method avoids direct singularity issues but it does not always yield the most accurate solution. As a result, it offers a trade-off between computational speed and accuracy.

$$\zeta = J^T x_E$$

3.5.1. Plot:

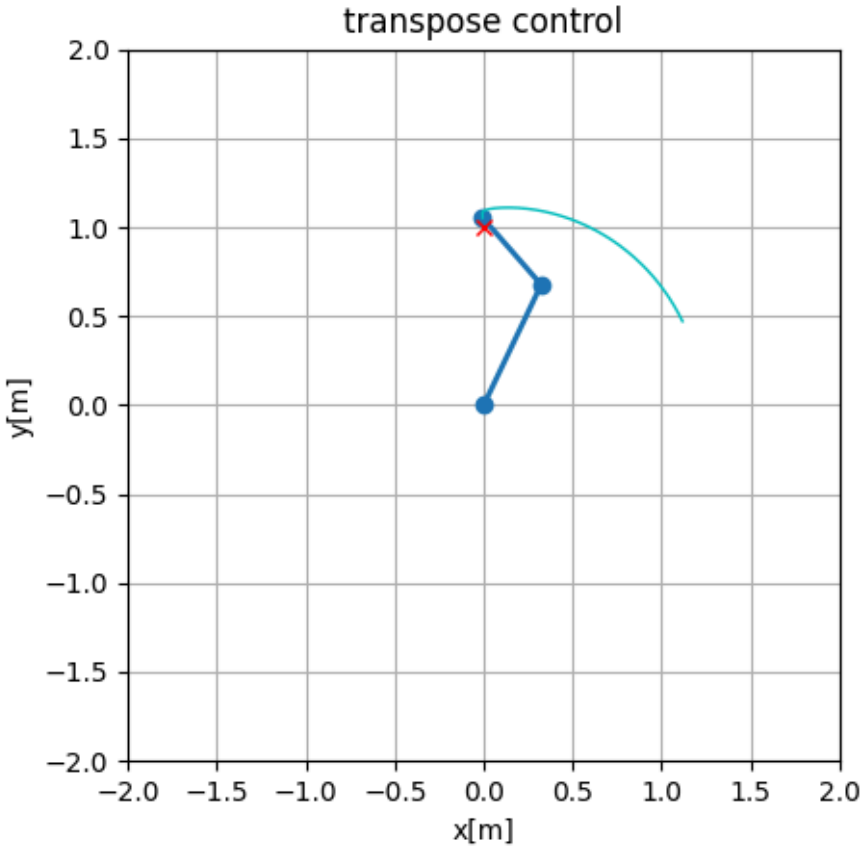

Figure 5. Simulation using Pseudoinverse method.

3.6. Plot Function:

To plot the evolution of control error, we implemented the following function which loads the error files, saves errors during simulation, and plot it against time.

```
def plot_control_error(transpose_file, pinverse_file, DLS_file, simulation_time=10,
fps=60):
    # Loading  saved control error norms with exception handling
    try:
        transpose_errors = np.load(transpose_file)
    except FileNotFoundError:
        print(f"File {transpose_file} not found\n. Please run controller_solution
'transpose' to generate file.")
        exit(0)
    try:
        pinverse_errors = np.load(pinverse_file)
    except FileNotFoundError:
```

```
        print(f"File {pinverse_file} not found\n. Please run controller_solution 'pinverse' to generate file.")
        exit(0)
    try:
        DLS_errors = np.load(DLS_file)
    except FileNotFoundError:
        print(f"File {DLS_file} not found\n. Please run controller_solution 'DLS' to generate file.")
        exit(0)


    # print(transpose_errors)
    # Expected number of time steps
    expected_length = int(simulation_time * fps)

    # Find the minimum length among all error arrays to avoid mismatches
    min_length = min(len(transpose_errors), len(pinverse_errors), len(DLS_errors), expected_length)

    # Trimming arrays to have the same length
    transpose_errors = transpose_errors[:min_length]
    pinverse_errors = pinverse_errors[:min_length]
    DLS_errors = DLS_errors[:min_length]

    # Generate time array dynamically
    tt = np.arange(0, min_length) / fps  # Converts indices to time in seconds

    # Plotting
    plt.figure(figsize=(8, 6))
    plt.plot(tt, transpose_errors, label='Transpose Method')
    plt.plot(tt, pinverse_errors, label='PInverse Method')
    plt.plot(tt, DLS_errors, label='DLS Method')

    plt.xlabel('Time [s]')
    plt.ylabel('Control Error Norm')
    plt.title('Evolution of Control Error Norm over Time')
    plt.legend()
    plt.savefig("plot.png")
    plt.grid(True)
    plt.show()
```

4. Error Plot:

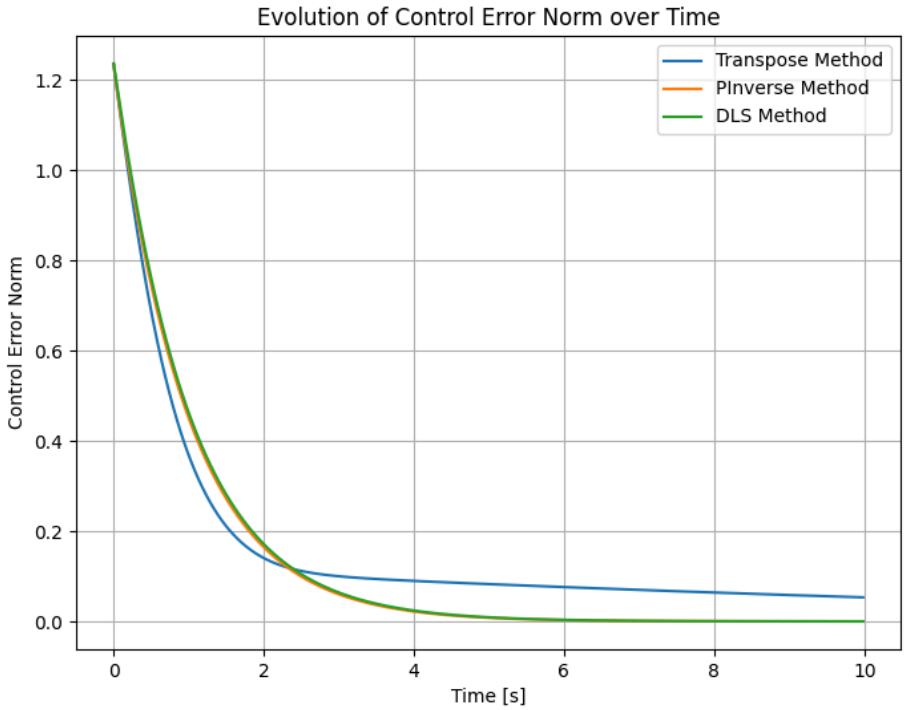

Figure 6. Simulation using Pseudoinverse method.

From the graph above, It is evident that the ***Jacobian Transpose*** method compromises the accuracy of the solution for faster computation. For the transpose method, The control error plateau early.

For ***Pseudo-inverse***, The control error goes around zero and plateau which shows it provides an accurate solution but sometimes at singularities, Pseudo-inverse can give unstable solutions.

The ***DLS*** method, Unlike the methods stated above, doesn't suffer from unstable solutions (mostly) or low accuracy. Hence proving to be more robust than the other two.

5. Questions:

5.1. What are the advantages and disadvantages of using kinematic control in robotic systems?

Ans:

Advantages	Disadvantages
Kinematic control enables accurate positioning of the robot's end-effector by directly computing joint parameters.	Kinematic control does not account for the robot's dynamics, such as inertia, friction, and external forces.
Kinematic control is computationally less intensive	Accurate kinematic model is needed
Easy to design and Implement.	May encounter singularities that prevent smooth motion
The reduced computational burden allows for high control loop frequency.	It is unsuitable for high-speed tasks, hence less robust.

5.2. Give examples of control algorithms that may be used in the robot's hardware to follow the desired velocities of the robot's joints, being the output of the resolved-rate motion control algorithm.

Ans: There are multiple control algos. which can be used for desired velocities of the robot's joints. Some of them are listed below.

- Proportional-Integral-Derivative (PID) Control: PID adjusts motor input based on the error between desired and actual velocity using feedback loop.
- Sliding Mode Control (SMC): The method that forces system states to "slide" along a predefined trajectory.
- Adaptive Control: This method controls parameters dynamically to handle model uncertainties and external disturbances.
- Impedance Control: This method controls motion based on interaction forces.

6. Singularities:

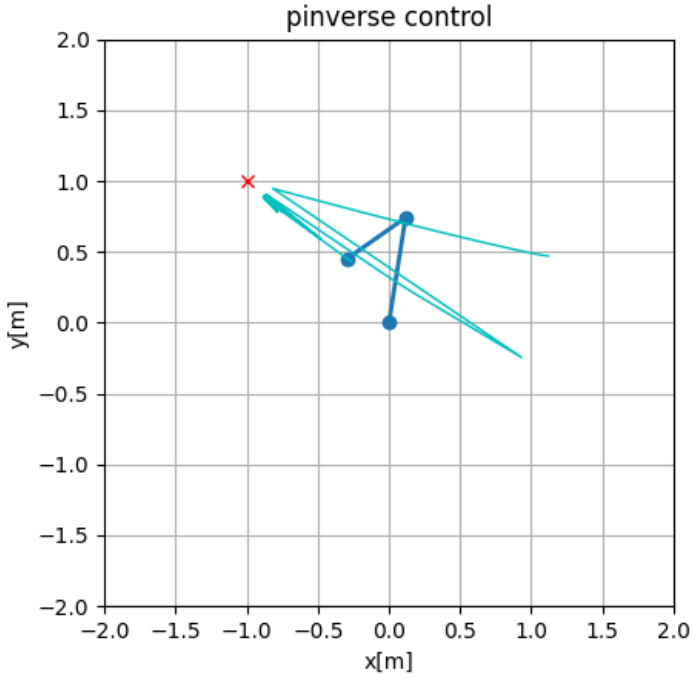

Figure 7. Simulation using Pseudoinverse method.

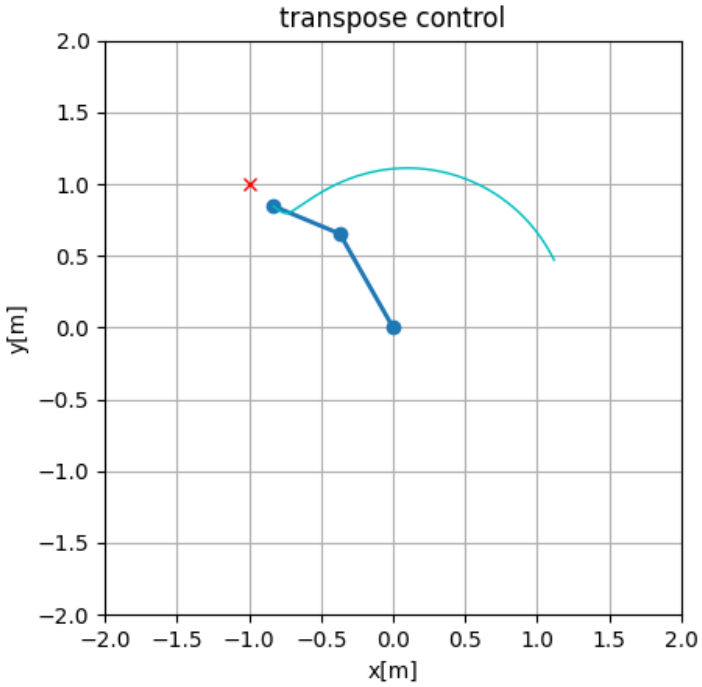

Figure 8. Simulation using Jacobian Transpose method.

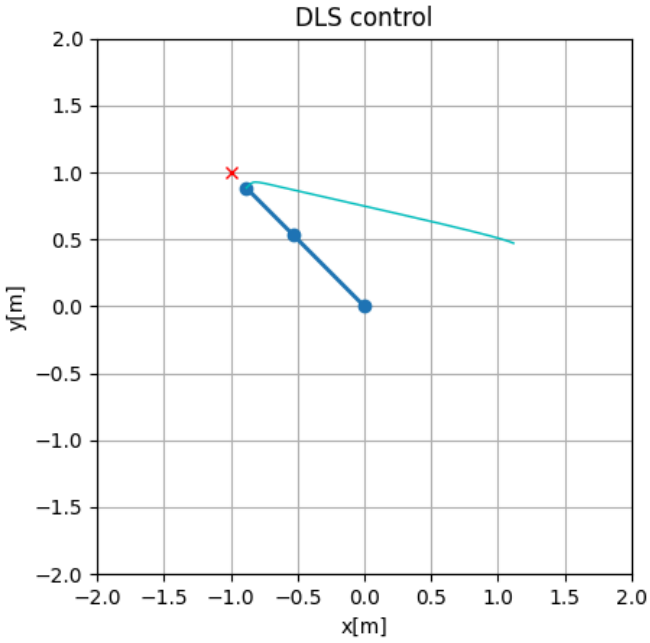

Figure 9. Simulation using DLS method.

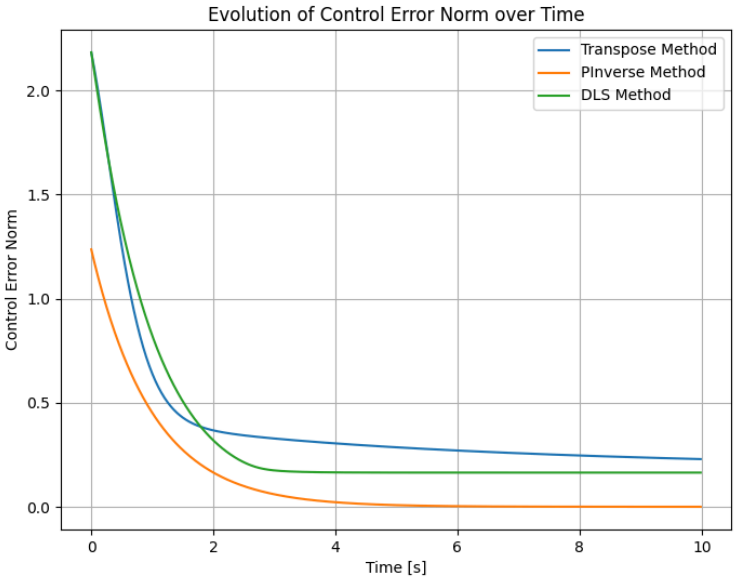

Figure 10. Simulation using Pseudoinverse method.

7. Appendix:

kinematics.py

```
# Import necessary libraries
from lab2_robotics import * # Import our library (includes Numpy)
import matplotlib.pyplot as plt
import matplotlib.animation as anim
from plot import *
# Robot definition (planar 2 link manipulator)
d = np.zeros(2)           # displacement along Z-axis
q = np.array([0.2, 0.5])  # rotation around Z-axis (theta)
a = np.array([0.75, 0.5]) # displacement along X-axis
alpha = np.zeros(2)       # rotation around X-axis

# Simulation params
dt = 0.01 # Sampling time
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('Kinematics')
ax.set_xlabel('x[m]')
ax.set_ylabel('y[m]')
ax.set_aspect('equal')
ax.grid()
line, = ax.plot([], [], 'o-', lw=2) # Robot structure
path, = ax.plot([], [], 'r-', lw=1) # End-effector path
joint_pos = []
# Memory
PPx = []
PPy = []

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

# Simulation loop
def simulate(t):
   global d, q, a, alpha
   global PPx, PPy, joint_pos

   # Update robot
   T = kinematics(d, q, a, alpha)
   dq = np.array([0.5,2.0])#np.ones(2) # Define how joint velocity changes with time!
   q = q + dt * dq
```

```
    joint_pos.append(q)
    # 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)

    return line, path

# Run simulation
animation = anim.FuncAnimation(fig, simulate, tt,
                               interval=1, blit=True, init_func=init, repeat=False)
plt.show()
joint_pos_file = "plot_data/joint_pose.npy"
np.save(joint_pos_file, joint_pos)
joint_pos = np.load(joint_pos_file)

# Load the joint positions from the saved file
joint_pos_data = np.load(joint_pos_file)
# Convert to a numpy array
joint_pos_data = np.array(joint_pos_data.tolist())

# Create a time vector corresponding to the simulation time
time_vector = np.linspace(0, Tt, len(joint_pos_data))

# Plot the joint positions over time
plt.figure(figsize=(8, 6))
plt.plot(time_vector, joint_pos_data[:, 0], label='Joint 1')
plt.plot(time_vector, joint_pos_data[:, 1], label='Joint 2')
plt.xlabel('Time [s]')
plt.ylabel('Joint Angle [rad]')
plt.title('Joint Positions over Time')
plt.legend()
plt.grid(True)
plt.show()
```

resolved_rate_kin.py

```
# Import necessary libraries
from lab2_robotics import * # Includes numpy import
import matplotlib.pyplot as plt
import matplotlib.animation as anim


# Robot definition
d = np.zeros(2)           # displacement along Z-axis
q = np.array([0.2, 0.5])  # rotation around Z-axis (theta)
a = np.array([0.75, 0.5]) # displacement along X-axis
alpha = np.zeros(2)       # rotation around X-axis
```

```
revolute = [True, True]
sigma_d = np.array([0,1]) # goal position
K = np.diag([1, 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()
line, = ax.plot([], [], 'o-', lw=2) # Robot structure
path, = ax.plot([], [], 'c-', lw=1) # End-effector path
point, = ax.plot([], [], 'rx') # Target
PPx = []
PPy = []

# Lists to store control error norms
transpose_errors, pinverse_errors, DLS_errors = [], [], []
joint_pos = []

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

##making the controls for the three control solutions
def control(type: str, J: np.ndarray, lambda_: float = 0.1) -> np.ndarray:
   #dictionary of the three controls
   methods = {
       "transpose": lambda J: J.T,
       "pinverse": np.linalg.pinv,
       "DLS": lambda J: DLS(J, lambda_)
   }

   if type in methods:
       return methods[type](J)
   else:
       raise ValueError(f"Invalid controller type '{type}'. Choose from
{list(methods.keys())}.")

def plot_control_error(transpose_file, pinverse_file, DLS_file, simulation_time=10, fps=60):
   # Loading  saved control error norms with exception handling
   try:
       transpose_errors = np.load(transpose_file)
   except FileNotFoundError:
```

```
        print(f"File {transpose_file} not found\n. Please run controller_solution 'transpose'
to generate file.")
        exit(0)
    try:
        pinverse_errors = np.load(pinverse_file)
    except FileNotFoundError:
        print(f"File {pinverse_file} not found\n. Please run controller_solution 'pinverse' to
generate file.")
        exit(0)
    try:
        DLS_errors = np.load(DLS_file)
    except FileNotFoundError:
        print(f"File {DLS_file} not found\n. Please run controller_solution 'DLS' to generate
file.")
        exit(0)


    # print(transpose_errors)
    # Expected number of time steps
    expected_length = int(simulation_time * fps)

    # Find the minimum length among all error arrays to avoid mismatches
    min_length = min(len(transpose_errors), len(pinverse_errors), len(DLS_errors),
expected_length)

    # Trimming arrays to have the same length
    transpose_errors = transpose_errors[:min_length]
    pinverse_errors = pinverse_errors[:min_length]
    DLS_errors = DLS_errors[:min_length]

    # Generate time array dynamically
    tt = np.arange(0, min_length) / fps  # Converts indices to time in seconds

    # Plotting
    plt.figure(figsize=(8, 6))
    plt.plot(tt, transpose_errors, label='Transpose Method')
    plt.plot(tt, pinverse_errors, label='PInverse Method')
    plt.plot(tt, DLS_errors, label='DLS Method')

    plt.xlabel('Time [s]')
    plt.ylabel('Control Error Norm')
    plt.title('Evolution of Control Error Norm over Time')
    plt.legend()
    plt.savefig("plot.png")
    plt.grid(True)
    plt.show()

# --- Helper Function for Error Norm ---
def update_error_norm(pose,err,controller_solution):
    """Computes the error norm and appends it to the corresponding list based on the
```

```
controller type."""
   error_norm = np.linalg.norm(err)

   error_dict = {
       "transpose": transpose_errors,
       "pinverse": pinverse_errors,
       "DLS": DLS_errors,
       "joint_pose": joint_pos,
   }

   if controller_solution in error_dict:
       error_dict[controller_solution].append(error_norm)
       error_dict["joint_pose"].append(pose)
   else:
       raise ValueError(f"Invalid controller type '{controller_solution}'. Choose from
{list(error_dict.keys())}.")

def show_n_save_plots():
   """saves the joint position"""
   plot_control_error('transpose_errors.npy', 'pinverse_errors.npy', 'DLS_errors.npy')


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

   # Update robot
   T = kinematics(d, q, a, alpha)
   J = jacobian(T, revolute) # Implement!
   # Update control
   #extracting the robot pos in 2D plane for sigma
   P = robotPoints2D(T)
   P_sigma = [P[0, -1], P[1, -1]]
   sigma = np.array(P_sigma)      # Position of the end-effector
   err =  sigma_d - sigma        # Control error  ~sigma_E = sigma_E,d - sigma_E
   print(err)
   delta_f = K @ err #feedback action   df = K x ~sigma_E
   # X_dot_E = sigma_dot_E + delta_f  # No compensation sigma_dot_E ->0
   dq =control(controller_solution, J[0:2, :]) @ delta_f  # vel_vect = control_sol x delta_f
   q += dt * dq # velocity to position
   update_error_norm(q,err, controller_solution)

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

```
    return line, path, point


#this is where we select the control solution
controller_solution = "pinverse"#["transpose","pinverse","DLS"] #transpose solution, pinverse
solution and DLS
error_files = {
    "transpose": "transpose_errors.npy",
    "pinverse": "pinverse_errors.npy",
    "DLS": "DLS_errors.npy",
    "joint_pose": "joint_pose.npy"
}

# Run simulation with the current controller
ax.set_title(f'{controller_solution} control')
ax.set_xlabel('x[m]')
ax.set_ylabel('y[m]')
animation = anim.FuncAnimation(fig, simulate, np.arange(0, 10, dt),
                               interval=10, blit=True, init_func=init,
                               repeat=False, fargs=(controller_solution,))

# This call blocks until the window is closed (or the simulation ends)
plt.show()

# Save errors to file after simulation is finished
if controller_solution in error_files:
    np.save(error_files[controller_solution], eval(f"{controller_solution}_errors"))
    np.save(error_files["joint_pose"], "joint_pose")
else:
    raise ValueError(f"Invalid controller type '{controller_solution}' for saving errors.")

show_n_save_plots()
```

lab2_robotics.py

```
import numpy as np
from scipy.linalg import block_diag


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).
    # 2. Multiply matrices in the correct order (result in T).

    # Translation along z-axis
    M1 = block_diag(np.identity(2), np.identity(2))
    M1[2, 3] = d

    # Rotation about z-axis
    M2 = block_diag(np.array([[np.cos(theta), -np.sin(theta)],
                              [np.sin(theta),  np.cos(theta)]]), 1, 1)

    # Translation along x-axis
    M3 = block_diag(np.identity(2), np.identity(2))
    M3[0, -1] = a

    # Rotation about x-axis
    M4 = block_diag(1, np.array([[np.cos(alpha), -np.sin(alpha)],
                                 [np.sin(alpha),  np.cos(alpha)]]), 1)

    # Combine all transformations
    T = M1 @ M2 @ M3 @ M4
    return T

def kinematics(d, theta, a, alpha):
    '''
        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.

        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)] # Base transformation
    # For each set of DH parameters:
    # 1. Compute the DH transformation matrix.
    # 2. Compute the resulting accumulated transformation from the base frame.
    # 3. Append the computed transformation to T.
    for d_i,theta_i,a_i,alpha_i in zip(d,theta,a,alpha):
        T.append(T[-1]@DH(d_i,theta_i,a_i,alpha_i))
        #Appending a new transformation after multiplying it with the last transformation.
    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.
```

```
    J = []
    O_n = T[-1][:3,-1]  # points of last transform
    for Ti,rev_flag in zip(T,revolute):
        Ri = Ti[:3,:3]  #Rotation matrix 3x3 matrix
        Oi = Ti[:3,-1] #Origin vector
        zi = Ri[:3,-1]  #z-vector
        J.append(np.vstack([(np.cross(rev_flag*zi,(O_n-Oi)) +
                                        (1-rev_flag)*zi).reshape(3,1),(rev_flag*zi).reshape(3,1)]))
        # For revolute joints (rev_flag=1):
        #   J_i =[ [ z_i x (O_n - O_i) ],
        #          [       z_i         ]]
        # For prismatic joints (rev_flag=0):
        #   J_i =[ [ z_i ],
        #          [ 0 ]]
    J = np.hstack(J) #stacking jacobians

    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
    '''
    # A_dls = A^T * (A*A^T + λ^2*I)^(-1)
    DLS = A.T@np.linalg.inv(A@A.T+(damping**2)*np.identity(2))

    return DLS # Implement the formula to compute the DLS of matrix A.

# 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
```