Introduction to Robotics – Modeling and Control of a 4-DOF Robotic Arm

2ineddine zbou6599@gmail.com

May 15, 2025

This document summarizes the mathematical model of the PincherX-100 robot (DH, inverse DH, Jacobian, etc.). It also presents the results obtained from the simulation in RViz.

I Introduction

As part of this lab session, we focused on the study of the Pincher X100 robot, shown in Figure 1, whose DataSheet is available here: Pincher X100. The main objective of this lab is to implement theoretical concepts related to direct and inverse geometric and kinematic models of a serial manipulator, while optimizing the algorithm performing the pick-and-place task of an object whose Cartesian position is known on a horizontal table.

In the first phase of the lab, we will perform the pick-and-place task by specifying the angle ψ. In the second phase, we will develop an iterative algorithm to deduce the angle ψ from the current position and the target position.

Figure 1: Pincher X100 Robot

II Forward Geometric Model

Based on the DH convention linking the base frame to the end-effector, we can define the necessary transformations to model the manipulator's motion and calculate its configurations from Cartesian positions:

$$\begin{bmatrix} C_1 \cdot C_{234} & -C_1 \cdot S_{234} & -S_1 & C_1 \cdot (L_r \cdot C_2 + L_3 \cdot C_{23} + L_4 \cdot C_{234}) \\ S_1 \cdot C_{234} & -S_1 \cdot S_{234} & C_1 & S_1 \cdot (L_r \cdot C_2 + L_3 \cdot C_{23} + L_4 \cdot C_{234}) \\ -S_{234} & -C_{234} & 0 & -(L_r \cdot S_2 + L_3 \cdot S_{23} + L_4 \cdot S_{234}) + L_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

From the homogeneous transformation matrix, we deduce the following geometric model:

$$\begin{aligned} p_x &= (L_r \cos(q_2) + L_3 \cos(q_2+q_3) + L_4 \cos(q_2+q_3+q_4)) \cos(q_1) \\ p_y &= (L_r \cos(q_2) + L_3 \cos(q_2+q_3) + L_4 \cos(q_2+q_3+q_4)) \sin(q_1) \\ p_z &= L_1 - L_r \sin(q_2) - L_3 \sin(q_2+q_3) - L_4 \sin(q_2+q_3+q_4) \\ \psi &= q_2 + q_3 + q_4 \end{aligned} \tag{1}$$

III Inverse Geometric Model

After solving the algebraic equations from the previous part, we get:

$$\begin{aligned} q_1 &= arctan2(p_x, p_y) \text{ or } q_1 = arctan2(p_x, p_y) \pm \pi \\ U &= p_x \cos(q_1) + p_y \sin(q_1) - L_4 \cos(\psi) \\ V &= L_1 - p_z - L_4 \sin(\psi) \\ A &= 2L_rU \\ B &= 2L_rV \\ W &= U^2 + V^2 + L_r^2 - L_3^2 \\ q_2(q_1) &= \arctan2\left(BW - \epsilon A\sqrt{A^2+B^2-W^2}, AW + \epsilon B\sqrt{A^2+B^2-W^2}\right) \\ q_3(q_1,q_2) &= \arctan2\left(-US_2 + VC_2, UC_2 + VS_2 - L_r\right) \\ q_4 &= \psi - q_2 - q_3 \end{aligned} \tag{2}$$

with $\epsilon = \pm 1$

IV Kinematic Model

There are several ways to grasp an object in the robot's workspace. In this section, we will develop an iterative algorithm that will deduce the optimal angle without including it in the operational parameter vector. The Jacobian matrix J is given by:

$$\mathrm{J} = \begin{pmatrix} -(L_rC_2 + L_3C_{23} + L_4C_{234})S_1 & -(L_rS_2 + L_3S_{23} + L_4S_{234})C_1 & -(L_3S_{23} + L_4S_{234})C_1 & -L_4S_{234}C_1 \\ (L_rC_2 + L_3C_{23} + L_4C_{234})C_1 & -(L_rS_2 + L_3S_{23} + L_4S_{234})S_1 & -(L_3S_{23} + L_4S_{234})S_1 & -L_4S_{234}S_1 \\ 0 & -L_rC_2 - L_3C_{23} - L_4C_{234} & -(L_3C_{23} + L_4C_{234}) & -L_4C_{234} \end{pmatrix}$$

We look for the best configuration of the joint parameters vector $q = [q_1, q_2, q_3, q_4]$ that satisfies the task $x = [p_x, p_y, p_z]$, in order to reduce the distance to joint limits or approach the nominal configuration q_{nom}. The equation of the algorithm is given by:

$$\mathrm{q}_{k+1} = \mathbf{q}_k + \alpha \mathbf{J}^{\#}(\mathbf{q}_k)(\mathbf{x}_{\text{goal}} - \mathbf{x}_{\text{current}}) + \beta(\mathbf{I}_4 - \mathbf{J}^{\#}(\mathbf{q}_k)\mathbf{J}(\mathbf{q}_k))(\mathbf{q}_{\text{nom}} - \mathbf{q}_k) \tag{3}$$

α, β are small positive scalar constants to be tuned. They should generally be relatively small for stability and local validity of the Jacobian. q_k is the current value (preferably measured if available, otherwise take the last computed value).

q_{k+1} is the new value of the joint variables vector to be sent as command to the servomotors.

V Work Done

V.1 Joint Angles in Radians

According to the DH convention shown in Figure 2, we have: $q_1 = 0$, $q_2 = -\tan^{-1}\left(\frac{L_m}{L_2}\right) = -1.23$ rad, $q_3 = \tan^{-1}\left(\frac{L_m}{L_2}\right) = 1.23$ rad, $q_4 = 0$ Thus, we can write the vector $q = [0, -1.23, 1.23, 0]$

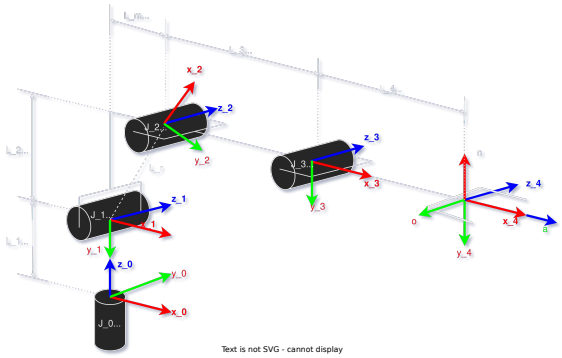

Figure 2: DH convention frames

V.2 Relationship Between Inverse Geometry Joint Configuration and Actuator Command

To compensate for the offset between the joint coordinate vector provided by the inverse geometric model and the actuator command, and to place the robot in the position shown in Figure 2, we use the following functions:

```
# From q_dh to q_actuator
def q_dh_to_q_actuator(q_dh):
    return q_dh - q_nom

# From q_actuator to q_dh
def q_actuator_to_q_dh(q_actuator):
    return  q_actuator + q_nom
```

When we send the command $q_a = [0, 0, 0, 0]$, we obtain the configuration shown in Figure 3.

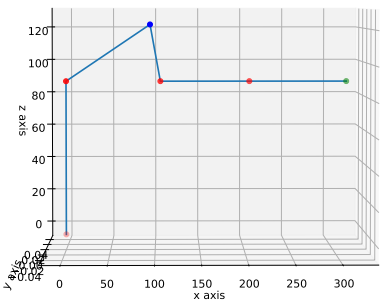

Figure 3: Robot position with command [0,0,0,0]

V.3 Homogeneous Transformation Matrix of the End-Effector

```
def matrice_t_h(q_dh):
    C1 = np.cos(q_dh[0])
    C2 = np.cos(q_dh[1])
    S1 = np.sin(q_dh[0])
    S2 = np.sin(q_dh[1])
    C23 = np.cos(q_dh[1] + q_dh[2])
    S23 = np.sin(q_dh[1] + q_dh[2])
    C234 = np.cos(q_dh[1] + q_dh[2] + q_dh[3])
    S234 = np.sin(q_dh[1] + q_dh[2] + q_dh[3])

    mat = np.array([[C1 * C234, -C1 * S234, -S1, C1
        * (Lr * C2 + L3 * C23 + L4 * C234)],
                    [S1 * C234, -S1 * S234, C1, S1 *
                        (Lr * C2 + L3 * C23 + L4 *
                        C234)],
                    [-S234, -C234, 0, -(Lr * S2 + L3
                        * S23 + L4 * S234) + L1],
                    [0, 0, 0, 1]])
    return mat
```

V.4 Inverse Geometric Model

Using the equations from the inverse geometric model section, we can write the function that converts an operational parameter vector $[p_x, p_y, p_z, \psi]$ into a joint parameter vector $[q_1, q_2, q_3, q_4]$. The multiplicity of solutions is due to the values of parameter q_1, which can take several values, and also due to the sign of ϵ. The best solution is selected using a cost function, which returns the configuration closest to q_{nom}.

```
def convert_param_op_to_qdh(param_op):
    px, py, pz, phi = param_op

    # Calculation of q1
    q1 = [np.arctan2(py, px), np.arctan2(py, px) +
        np.pi, np.arctan2(py, px) - np.pi]

    # Initialization of variables
    solutions_qdh = np.empty((0, 4))  # Empty matrix
        to store the solutions
    epsilon = np.array([-1, 1])         # Possible
        values for epsilon

    for j in q1:
        # Calculation of q2
        U = px * np.cos(j) + py * np.sin(j) - L4 *
            np.cos(phi)
        V = L1 - pz - L4 * np.sin(phi)
        A = 2 * Lr * U
        B = 2 * Lr * V
        W = U ** 2 + V ** 2 + Lr ** 2 - L3 ** 2
        value = A ** 2 + B ** 2 - W ** 2

        if value >= 0:
            for epsilon_value in epsilon:
                q2 = np.arctan2(B * W -
                    epsilon_value * A *
                    np.sqrt(value),
                                A * W +
                                    epsilon_value *
                                    B *
                                    np.sqrt(value))

                # Calculation of q3
                S2 = np.sin(q2)
                C2 = np.cos(q2)
                q3 = np.arctan2(-U * S2 + V * C2, U
                    * C2 + V * S2 - Lr)

                # Calculation of q4
```

```
                q4 = phi - q2 - q3

                # Stack the solution into the
                    solutions_qdh array
                solutions_qdh =
                    np.vstack([solutions_qdh, [j,
                    q2, q3, q4]])

    # Check if solutions were found
    if solutions_qdh.size > 0:
        print(f"The best solution for q vector is:
            {coast_function(q_nom, solutions_qdh)}")
        return coast_function(q_nom, solutions_qdh)
    else:
        print("There is no solution for this
            position")
        return np.empty((0, 4))  # Empty array
            compatible with np.array
```

V.5 Cost Function

```
    def coast_function(qnom, q):
        # Calculate the squared differences between
            qnom and all rows of q
        distances = np.linalg.norm(q - qnom, axis=1)
        # Return the configuration that minimizes
            the distance
        idx_min = np.argmin(distances)
        return np.array(q[idx_min])  # Return the
            closest configuration
```

The existence of a solution depends on the term $\sqrt{A^2 + B^2 - W^2}$. If this term is positive, then a solution exists. However, the existence of a solution does not necessarily mean that it is feasible — the mechanical constraints of the robot must also be considered.

To determine the robot's workspace, two approaches are possible: - via the kinematic model of the robot (we adopt a simplifying assumption), - or via an algorithm that explores the workspace.

Given the complexity of the kinematic equations, we will restrict ourselves to a 2D workspace. To do this, we eliminate q_1 by setting $p_y = 0$ and fix the angle ψ.

We start from the base equation:

$$A^2 + B^2 - W^2 = 4L_r^2 \cdot (U^2 + V^2)$$

Where:

$$W^2 = (U^2 + V^2)^2 + (L_r^2 - L_3^2)^2 - 2 \cdot (U^2 + V^2) \cdot (L_r^2 - L_3^2)$$

Let:

$$X = U^2 + V^2, \quad b = 2L_r^2 + 2 \cdot L_3, \quad c = -(L_r^2 - L_3^2)^2$$

The equation becomes:

$$A^2 + B^2 - W^2 = -X^2 + bX + c$$

With $\Delta > 0$, where $\Delta = b^2 + 4c$, we deduce:

$$X = \left\{(L_r + L_3)^2, (L_3 - L_r)^2\right\}$$

Thus, the constraints on X are:

$$(L_3 - L_r)^2 < X < (L_3 + L_r)^2$$

By introducing the expressions of U and V:

$$U = p_x - L_4, \quad V = L_1 - p_z$$

We get:

$$X = U^2 + V^2 = (p_x - L_4)^2 + (L_1 - p_z)^2$$

Finally, the constraints become:

$$(L_3 - L_r)^2 < (p_x - L_4)^2 + (p_z - L_1)^2 < (L_3 + L_r)^2$$

We conclude that the workspace is shaped like a torus, qualitatively illustrated in Figure 4.

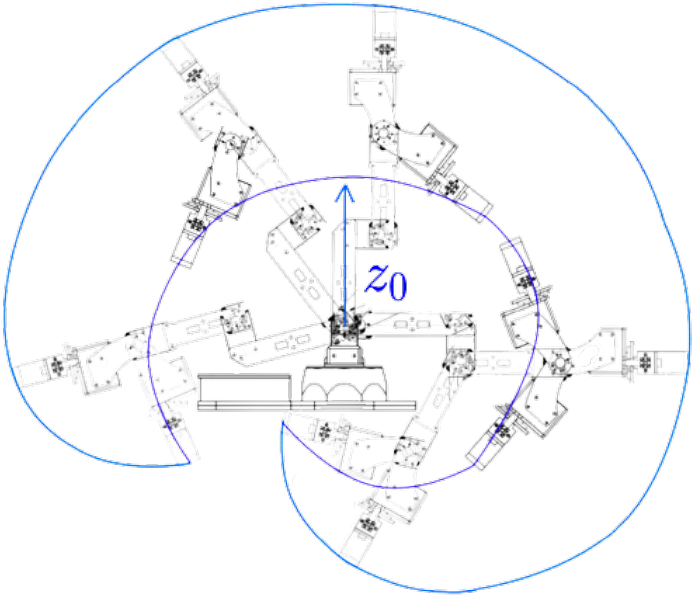

Figure 4: Approximation of the workspace of the Pincher X100 robot

V.6 Pick-and-Place Operation and Generalization

For this first case, I selected 3 objects with the following coordinates:

$$\begin{aligned} \text{object1} &= \left[\left[100, 200, 3, \frac{\pi}{4}\right], \left[100, -200, 3, \frac{\pi}{4}\right]\right] \\ \text{object2} &= \left[\left[0, 0, 250, \frac{\pi}{10}\right], \left[20, 100, 20, \frac{\pi}{2}\right]\right] \\ \text{object3} &= \left[\left[180, 200, 80, \frac{\pi}{10}\right], \left[180, -200, 80, \frac{\pi}{10}\right]\right] \end{aligned} \tag{4}$$

All of these executions were done in simulation, as working with the real robot requires special regulation of the joint angles to prevent it from losing balance due to torque. Another very important remark: always define at least one non-zero position (in x, y, or z), otherwise the robot may not reach the coordinates close to $[0, 0, 0]$.

Here is the code that performs the pick-and-place function:

```python
def main():
    bot = InterbotixManipulatorXS("px100", "arm",
        "gripper")
    # Generalization of the task on a set of objects
        to move
    object1 =
        [[100,200,3,np.pi/4],[100,-200,3,np.pi/4]]
    object2 =
        [[0,0,250,-np.pi/10],[20,100,20,np.pi/2]]
    object3 =
        [[180,200,80,np.pi/10],[180,-200,80,np.pi/10]]
    for i in [object1,object2,object3]:
        initial_position = i[0]
        second_position = i[1]
        initial_qdh_cord =
            convert_param_op_to_qdh(initial_position)
        initial_qa_cord =
            q_dh_to_q_actuator(initial_qdh_cord)
        second_qdh_cord =
            convert_param_op_to_qdh(second_position)
        second_qa_cord =
            q_dh_to_q_actuator(second_qdh_cord)
        bot.gripper.open()
        bot.arm.set_joint_positions(initial_qa_cord)
        bot.gripper.close()
        bot.arm.set_joint_positions(second_qa_cord)
        bot.gripper.open()
    bot.arm.go_to_sleep_pose()
```

The result of the simulation is shown in the video: Part1. Below are the graphs of the simulated points. "I separated the graphs for better readability."

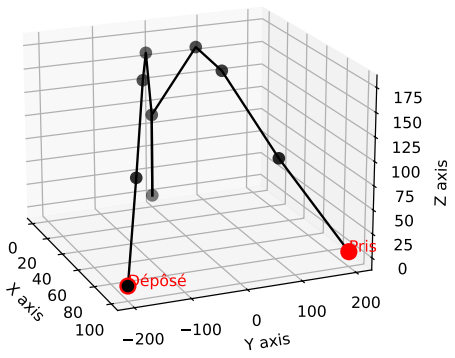

Figure 5: Simulation result for object – 1

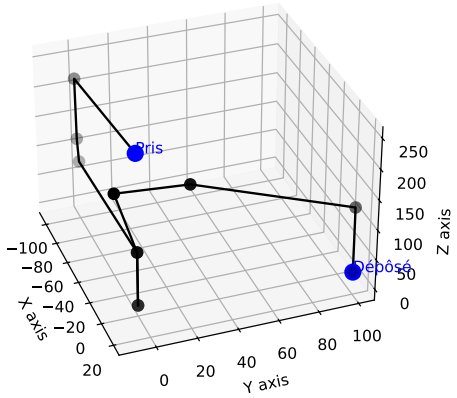

Figure 6: Simulation result for object – 2

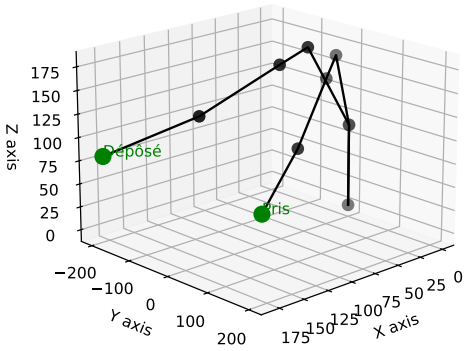

Figure 7: Simulation result for object – 3

V.7 The Manipulator Jacobian

```python
def jacobienne(q):
    C1 = np.cos(q[0])
    C2 = np.cos(q[1])
    S1 = np.sin(q[0])
    S2 = np.sin(q[1])
    C23 = np.cos(q[1] + q[2])
    S23 = np.sin(q[1] + q[2])
    C234 = np.cos(q[1] + q[2] + q[3])
    S234 = np.sin(q[1] + q[2] + q[3])

    J = np.array([[-(Lr * C2 + L3 * C23 + L4 * C234)
        * S1, -(Lr * S2 + L3 * S23 + L4 * S234) * C1,
                -(L3 * S23 + L4 * S234) * C1, -L4
                    * S234 * C1],
               [(Lr * C2 + L3 * C23 + L4 * C234)
                   * C1, -(Lr * S2 + L3 * S23 +
                   L4 * S234) * S1,
                -(L3 * S23 + L4 * S234) * S1, -L4
                    * S234 * S1],
               [0, -Lr * C2 - L3 * C23 - L4 *
                   C234, -L3 * C23 - L4 * C234,
                   -L4 * C234]])

    return J
```

V.8 Iterative Algorithm

As previously discussed, there are multiple ways to grasp an object in the robot's workspace. The goal of this section is to code an algorithm that finds the optimal angle ψ to reach a given position, based on the end-effector's current pose at time $i-1$ and the desired configuration q_{nom}.

To test this algorithm's performance, we perform two tests: - The first on q_{nom} - The second on a different target point.

The idea is to let the algorithm **estimate the ideal angle ψ**. For the first test, we use the configuration $[100, 200, 3, \frac{\pi}{4}]$ and let the algorithm compute the optimal angle. It won't necessarily be exactly $\frac{\pi}{4}$, since that was chosen arbitrarily, but it should be close.

Here is the algorithm code:

```
def iterative_algorithm(q_initial, x_but,\
tolerance=1e-11, max_iterations=100, qnom=None):
    # Parameters
    alpha = 0.3
    beta = 0.3

    # Initialization
    x_courant = matrice_t_h(q_initial)  # Current
        position extracted from the homogeneous
        matrix
    qk0 = q_initial
    J = jacobian(q_initial)  # Initial Jacobian

    # Check the shape of the Jacobian (should be of
        size 3x4)
    if np.shape(J) == (3, 4):
        print("The Jacobian has the correct shape.")
    else:
        print(f"The Jacobian does not have the
            correct shape, current shape: {J.shape}")

    J = J.astype(np.float64)

    # Iterative algorithm
    for i in range(max_iterations):
        # Inverse of the Jacobian
        J_inverse = np.linalg.pinv(J)  # Use the
            pseudo-inverse to avoid singularities

        # Update calculation
        term1 = alpha * np.dot(J_inverse, (x_but -
            x_courant))
        term2 = beta *
            np.dot((np.eye(len(q_initial)) -
            np.dot(J_inverse, J)), (qnom - qk0) if
            qnom is not None else np.zeros_like(qk0))
        qk1 = qk0 + term1 + term2

        # Check stop criterion
        if np.linalg.norm(qk0 - qk1) < tolerance:
            print(f"Stop criterion reached after
                {i+1} iterations.")
            break

        # Update
        qk0 = qk1

        # Calculate the new Jacobian
        J = jacobian(qk0)
        J = J.astype(np.float64)

        # Update the current position
        x_courant = matrice_t_h(qk0)

    # Final calculation
    phi = np.sum(qk1[1:])  # Sum of angles q2, q3, q4
    px, py, pz = x_but
    U = px * np.cos(qk1[0]) + py * np.sin(qk1[0]) -
        L4 * np.cos(phi)
    V = L1 - pz - L4 * np.sin(phi)
    A = 2 * Lr * U
    B = 2 * Lr * V
    W = U**2 + V**2 + Lr**2 - L3**2
    D = A**2 + B**2 - W**2

    # Return the solution or None if impossible
    if D >= 0:
        print(f"The best coordination is: {qk1} \t
            for the next position {x_but} \n"
              f"Which corresponds in real parameters
                  to: {np.insert(matrice_t_h(qk1),
                  len(matrice_t_h(qk1)), phi)}")
        return qk1
    else:
        print("No valid solution was found.")
        return None
```

```
x_but  = [100,200,3,np.pi/4]
q_but  =
    convert_param_op_to_qdh(q_actuator_to_q_dh(x_but))
iterative_algorithm(q_but, np.array([100,200,3]),
    qnom=q_nom)
```

The function `iterative_algorithm` contains `print` statements that display live coordinate updates. It returns the result as follows:

```
The Jacobian has the correct shape.
Stop criterion reached after 35 iterations.
The best coordination is: [ 1.10714871 -0.68382443
    1.45740347  0.10058747]
for the next position [100 200   3]
Which corresponds in real parameters to: [
    99.99999953 199.9999944    3.00000576
    0.87416651]
```

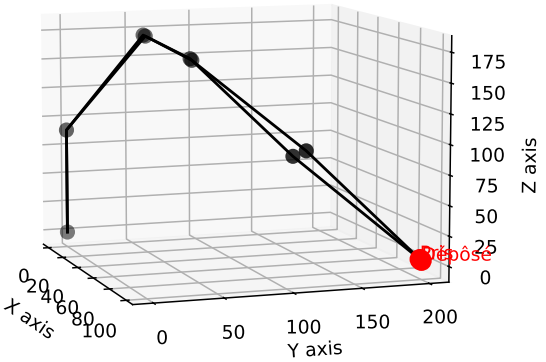

Figure 8: Simulation result – 1

We obtained an angle of `0.874 rad`, which is relatively close to $\frac{\pi}{4}$.

Next, we conduct a second test using the configuration q_{nom}.

```
x_test = np.array([Lm + L3 + L4,0,L1 + L2,-1])
q_test =
    convert_param_op_to_qdh(q_actuator_to_q_dh(x_test))
iterative_algorithm(q_test,np.array([Lm + L3 +
    L4,0,L1 + L2]) , qnom=q_nom)
```

And here is the result of that test:

```
The best solution for q vector is: [-0.00505783
    -0.48857186  0.90243115 -1.41385929]
The Jacobian has the correct shape.
Stop criterion reached after 47 iterations.
The best coordination is: [-2.48477522e-10
    -1.23412154e+00  1.23412164e+00 -1.86862729e-07]
          for the next position [244.     0.
    189.54]
Which corresponds in real parameters to: [
    2.43999997e+02 -6.06285147e-08  1.89539999e+02
    -7.97795007e-08]
```

From these results, we observe that the computed angle is `-1.86862729e-07 rad`, which is very close to `0 rad`, confirming the reliability of the algorithm.

```
def main():

    bot = InterbotixManipulatorXS("px100", "arm",
        "gripper")
    bot.gripper.open()

    object1 = [[100,250,100],[90,-100,10]]
    object2 = [[20,0,250],[150,50,10]]
    object3 = [[150,0,80],[40,-100,-30]]
    q_initial = q_nom
    for i in np.array([object1,object2,object3]):

        qdh1 = iterative_algorithm( q_initial ,i[0],
            qnom=q_nom)
        qa1 = q_dh_to_q_actuator(qdh1)
        bot.arm.set_joint_positions(qa1)
        bot.gripper.close()
        q_initial = qdh1

        qdh2 = iterative_algorithm( q_initial ,i[1],
            qnom=q_nom)
        qa2 = q_dh_to_q_actuator(qdh2)
        bot.arm.set_joint_positions(qa2)
        bot.gripper.open()
        q_initial = qdh2

    bot.arm.go_to_sleep_pose()
```

The following graphs clearly illustrate the reliability of the algorithm.

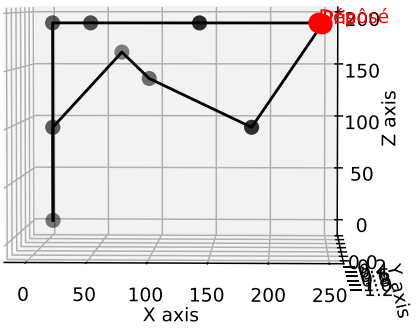

Figure 9: Simulation result – 2

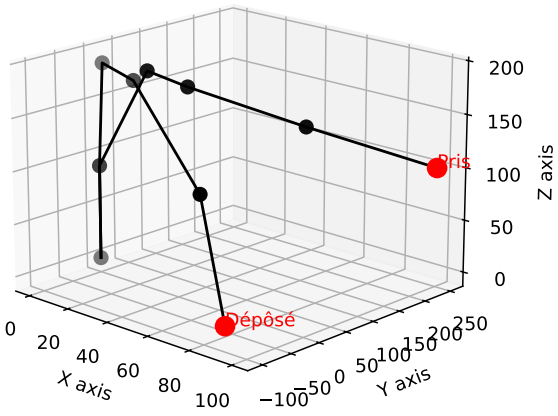

Figure 10: Simulation result for object – 1

V.9 Object Pick-and-Place

In this section, you can monitor the coordinates generated by the iterative algorithm in real-time through the `cmd` window. Additionally, you can view the simulation results in the video: Part2.

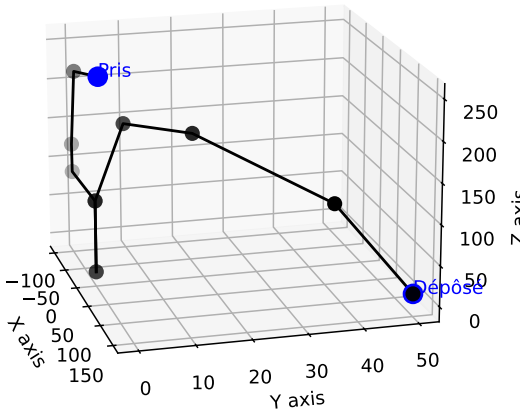

Figure 11: Simulation result for object – 2

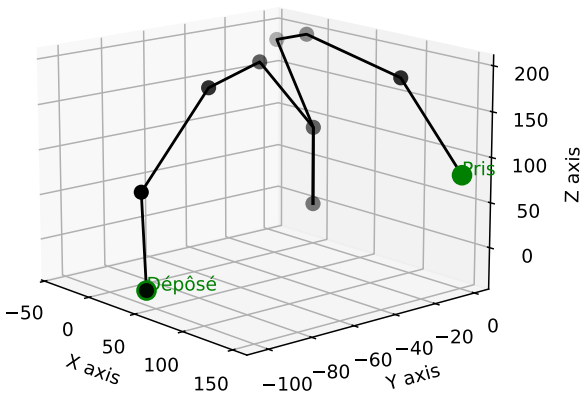

Figure 12: Simulation result for object – 3

VI Conclusion

In conclusion, inverse kinematics using the DH model (qDH) is fast and straightforward, making it ideal for obtaining unique solutions. However, it is limited when dealing with multiple or undefined configurations. On the other hand, the iterative method based on the Jacobian offers greater flexibility and allows for the optimization of secondary tasks, such as moving closer to a nominal configuration, while still fulfilling the main task. Nevertheless, it is slower and requires precise tuning of parameters to ensure stability and convergence. The choice between both methods therefore depends on the requirements for speed, simplicity, or flexibility.

The objective of this lab session was successfully achieved.