Robot Arm Bonus Report

Abeer Khaled Ahmed 8026
Hesham Mohamed 7924
Ahmed Abdulaal 8541

June 14, 2025

Contents

1 Introduction

Our project this term for the Industrial Robotics course was mainly the reverse engineering of a 5 degree-of-freedom robot arm, where we initiated a complete study of all the possible motions that could be performed by the robot, while also conducting a full study of its behavior based on the DH table and the implementation of both forward and reverse kinematics.

2 Robotic Arm Design

To begin the robotic arm project, we reverse-engineered a 5-DOF robotic arm provided by our college. The process included physically examining the arm to understand its structure and servo motor placement, measuring with rulers, and manually disassembling components. To ensure accuracy, we also reassembled the joints to center the servos and verify joint limits.

The SolidWorks modeling phase involved designing each part based on the obtained measurements, with extra attention given to the gripper due to its intricate gear system. This effort allowed us to achieve an accurate digital twin of the robotic arm.

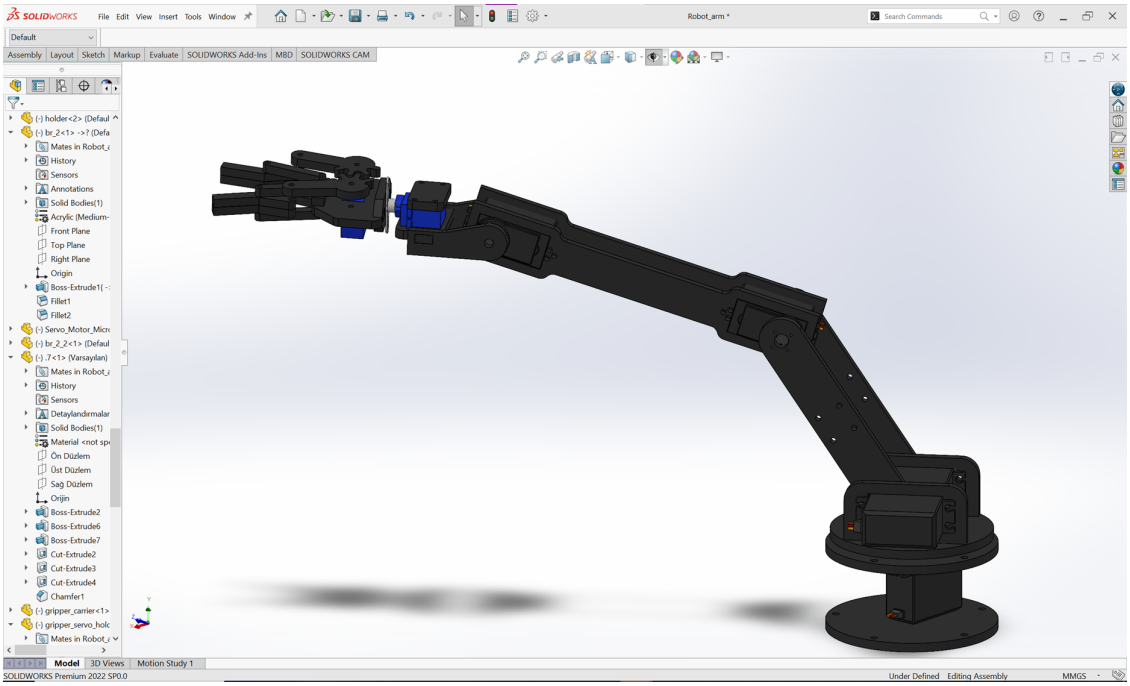

Figure 2.1: Mechanical Design of Robotic arm

The components used are:
- 5 Servo motors FS5115M
- 2 SG-90 Servo Motor
- 5V/10A DC Power Supply
- ESP32 Microcontroller

3 Matlab

3.1 DH-Table

The DH table was achieved by identifying a co-ordinate axis system for each joint of motion in the robot arm, and utilizing the lengths of links provided to the nearest millimeter. On matlab, we used Peter-Corke built in library for any further calculation or studying of the robot.

j	theta	d	a	alpha	offset
1	q1	22	17.29	1.5708	0
2	q2	0	110	0	0
3	q3	0	170	0	0
4	q4	0	0	1.5708	0
5	q5	162	24.24	0	0

Figure 3.1.1: DH-Table

3.2 Workspace

The workspace is implemented using a simple nested for loop code, assuming that the robot can move freely without identifying the ground plate as an obstacle, to understand the full capability of the robot

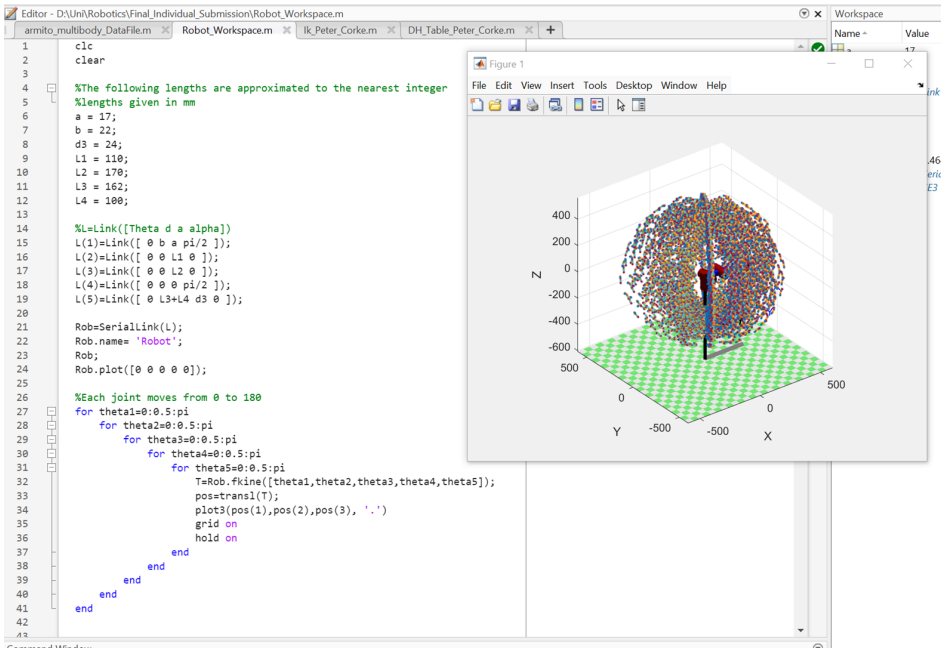

Figure 3.2.2: Workspace

The following is the workspace after including the obstacles, which in our case represent the wooden plate carrying the robotic arm

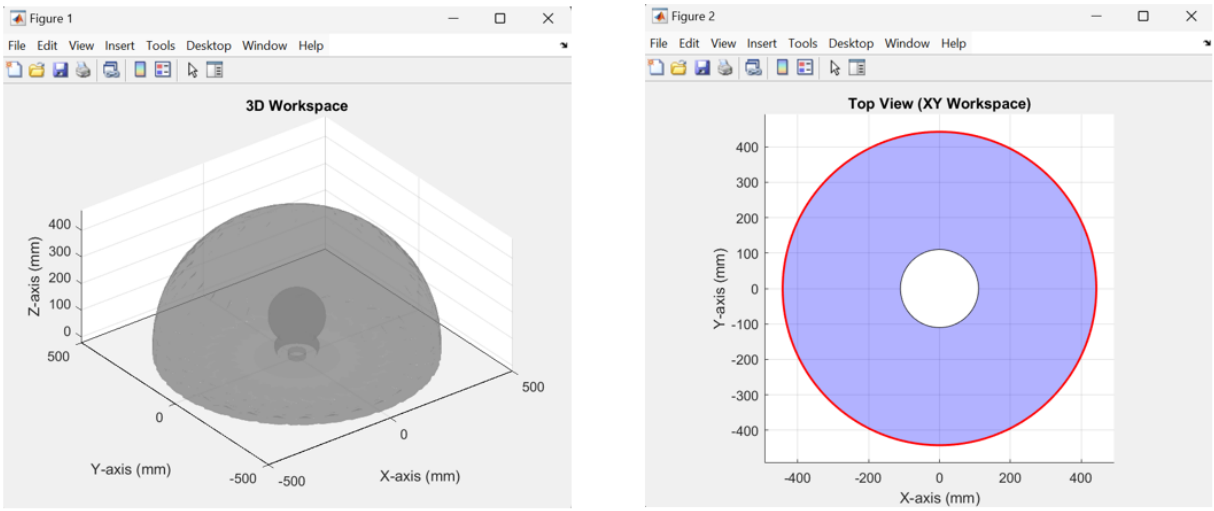

Figure 3.2.3: Workspace after acknowledging surroundings

3.3 Simscape model

One of the methods we used to study the robotic arm was Simscape multibody.. In the following model, we implemented forward kinematics, inverse kinematics, and Jacobian. We implemented a square trajectory using the trapezoidal trajectory planning. We also implemented each of these individually before combining them into one model to further study the robot and understand its motion in addition to understanding the basics of Robotics. The multibody was implemented after creating the solidworks using a plugin to convert the files, then multibody tree was implemented to be used in the model.

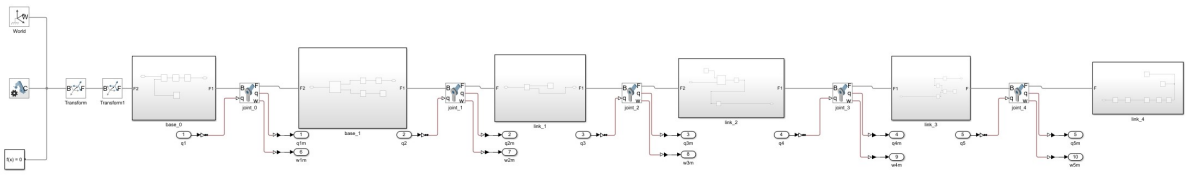

Figure 3.3.4: Simscape Multibody Robot Blocks

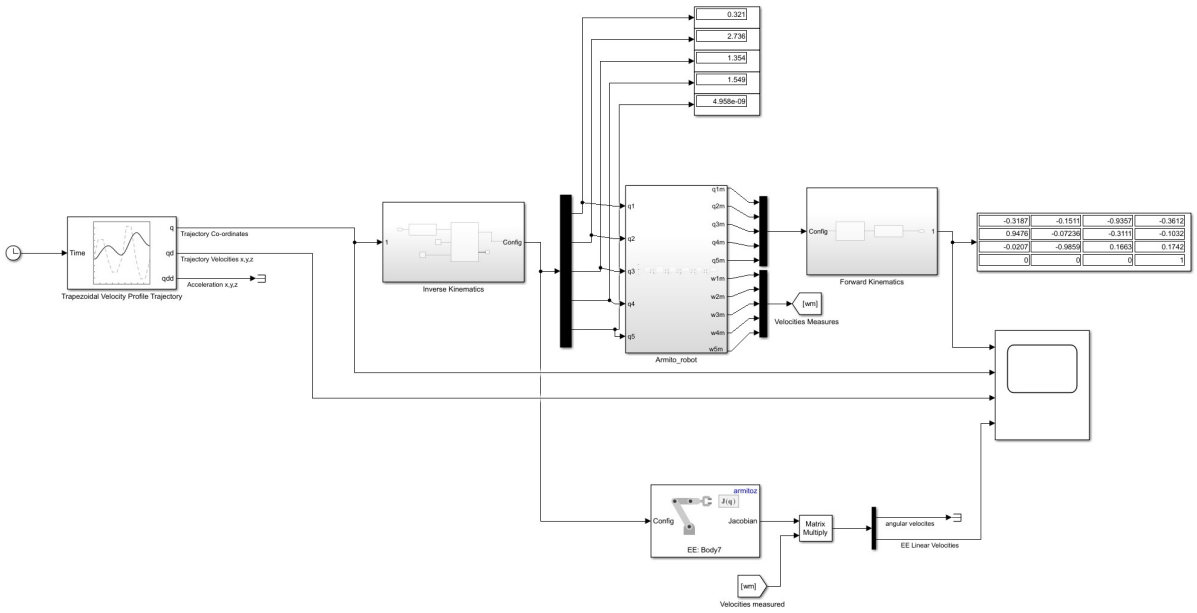

Figure 3.3.5: Simscape Multibody Full Model

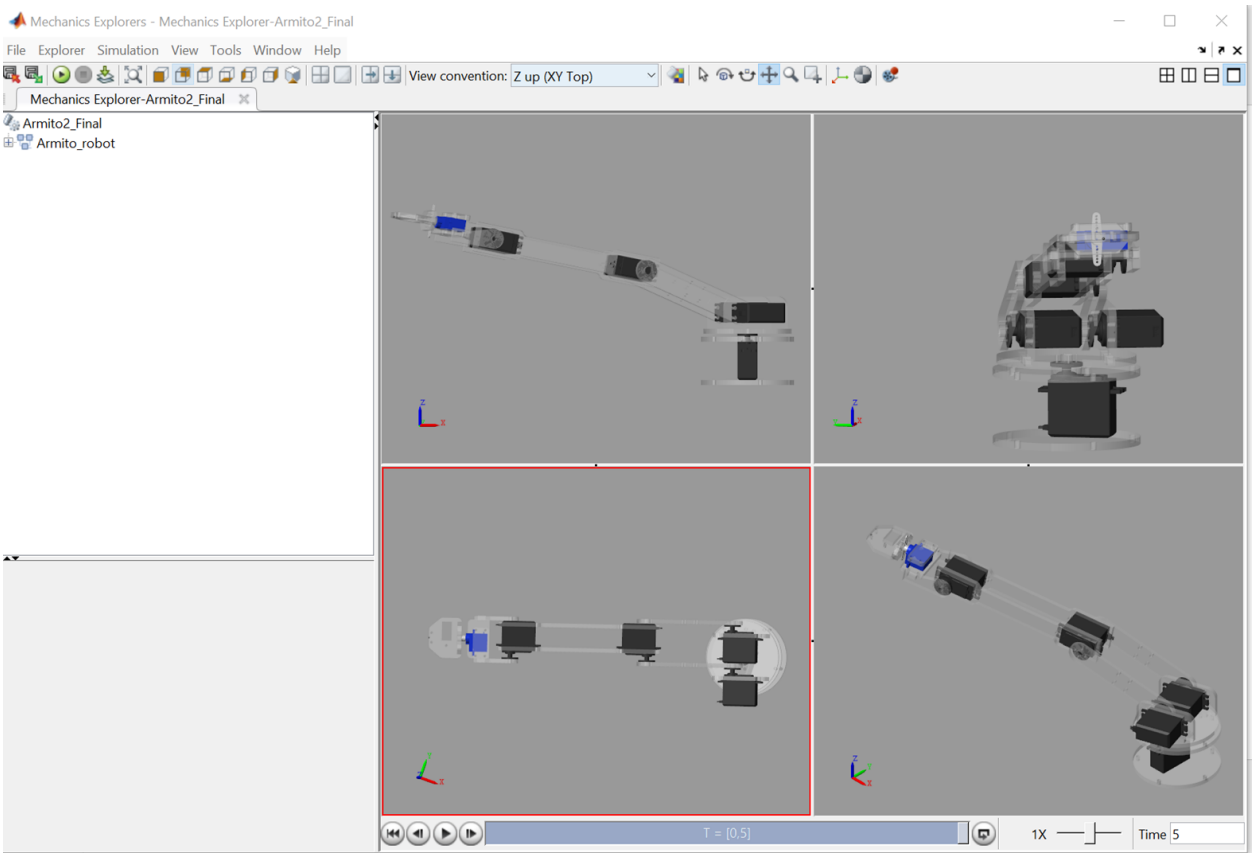

Figure 3.3.6: Robot Visualization in Matlab

4 ROS 2 Simulation Workflow: Robotic Arm with SolidWorks URDF

4.1 Setup Environment

We used ROS 2 Jazzy to control and manage the robotic arm. For simulation, we worked with Gazebo Harmonic using gz sim, which helped us test the robot's movement in a virtual world. The robot model was exported from SolidWorks using a special ROS plugin that kept the correct joint structure. We also followed a Z-up coordinate system in both the design and simulation to keep everything aligned.

4.2 Coordinate Frame Handling

In this step, we ensured that all coordinate frames between SolidWorks, ROS 2, and Gazebo were consistent to avoid any discrepancies in joint orientation or motion behavior. Each link and joint in the robotic arm was carefully assigned a unique TF (transform) frame name in the URDF. These TF frames helped establish the spatial relationship between parts of the robot and were critical for both visualization and control.

We used the robot_state_publisher node in ROS 2 to broadcast the TF tree, which allowed the rest of the ROS ecosystem (e.g., RViz, controllers) to understand the robot's kinematic structure in real time. During the SolidWorks export, care was taken to match the parent-child frame hierarchy as expected in the ROS convention (base link to end-effector).

4.3 Rviz

RViz was used to visualize the robot's joint positions and coordinate frames in real time. By loading the URDF model into RViz, we could verify that the joint hierarchy, transforms, and motions were correctly implemented. It also helped us debug issues related to robot orientation, joint limits, and frame alignment during early stages of integration.

4.4 Gazebo

Gazebo Harmonic was used to simulate the robotic arm's physical behavior in a 3D environment. After importing the URDF model, we validated the robot's structure, motion range, and collision dynamics under gravity. This allowed us to test different movement strategies and verify the inverse kinematics before deploying them to the real hardware.

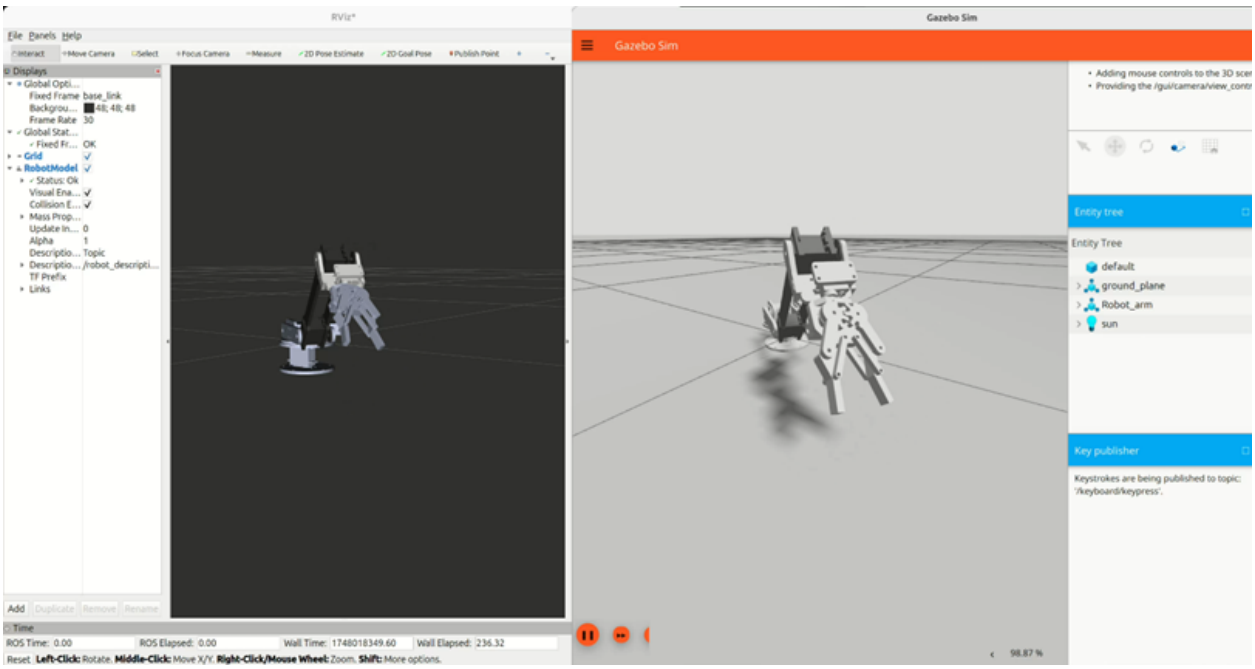

Figure 4.4.1: RVIZ - Gazebo Simulation

5 Codes

5.1 Forward Kinematics Code

The ManualJointController is a ROS 2 Python node designed to sequentially and safely test the motion of a 5-degree-of-freedom (DOF) robotic arm. It operates in two phases: first holding a fixed "vertical" configuration for 20 seconds, and then cycling through each joint individually with smooth, oscillating motion. This setup provides a practical and low-risk way to observe and verify the mechanical and software behavior of each joint, making it particularly useful for early-stage testing and calibration.

In the first phase, the arm is held in a predefined upright pose by continuously publishing joint positions that align the shoulder, elbow, and wrist vertically. This ensures the robot starts from a stable and known configuration. In the second phase, each joint is animated one at a time using a sinusoidal wave pattern that causes it to oscillate ± 0.5 radians around its resting value. The sine wave ensures smooth acceleration and deceleration, reducing mechanical stress and making the motion easy to observe.

The code uses ROS 2 publishers to send Float64 position commands to each joint's control topic at a steady 20 Hz rate. All joints are published to simultaneously, but only one joint is actively animated during a given time window. The others remain in the fixed "stand pose." A clamping mechanism ensures no joint command exceeds the typical safe servo range of 0 to radians.

This controller serves as a manual verification tool for checking servo response, joint integration, and motion isolation in simulation or on physical hardware. It does not perform inverse kinematics or Cartesian path planning, but provides foundational control for validating joint-level actuation before higher-level behaviors (like trajectory tracking or IK) are introduced.

```python
def run_forever(self):
    warmup_time = 20.0        # Stand time before motion
    joint_duration = 3.0      # Each joint moves for 3 seconds
    rate_hz = 20
    period = 1.0 / rate_hz

    while rclpy.ok():
        t = time.time() - self.start_time

        if t < warmup_time:
            # Hold vertical posture
            for i, angle in enumerate(self.stand_pose):
                msg = Float64()
                msg.data = float(angle)
                self.joint_pubs[i].publish(msg)

            self.get_logger().info(f"Holding vertical pose: {int(warmup_time - t)}s remaining...")
        else:
            # Animate one joint at a time
            motion_time = t - warmup_time
            joint_index = int(motion_time // joint_duration) % 5
            local_t = motion_time % joint_duration

            # Create new joint angles list
            angles = self.stand_pose[:]

            # Sine wave motion: ±0.5 rad around center
            if joint_index == 2:
                center = math.pi / 2  # elbow vertical
            else:
                center = self.stand_pose[joint_index]

            animated_angle = center + 0.5 * math.sin(2 * math.pi * 0.5 * local_t)
            animated_angle = max(0.0, min(math.pi, animated_angle))  # Clamp to [0, π]
            angles[joint_index] = animated_angle

            for i, angle in enumerate(angles):
                msg = Float64()
                msg.data = float(angle)
                self.joint_pubs[i].publish(msg)

            self.get_logger().info(
                f"Joint {joint_index} moving → " +
                ", ".join(f"{a:.2f}" for a in angles)
            )
```

Figure 5.1.1: Forward kinematics Sine wave

5.2 Main code

This ROS 2-based Python script implements a Cartesian motion controller for a 5-DOF robotic arm, enabling the robot to follow a predefined square trajectory in 3D space. The node, named CartesianSquareController, computes joint-space commands based on Cartesian target points using inverse kinematics, and publishes these commands in real time to the respective joint controllers.

The system is structured to simulate realistic robotic arm control by integrating key robotics principles, including:
- Geometric Inverse kinematics
- Workspace constraint validation

- Path interpolation implemented in the Cartesian space
- Analytical Jacobian matrix computation
- Real-time joint angle publishing

5.2.1 __init__(self)

The __init__ function serves as the system initializer and is fundamental to configuring the robot's operational environment. It instantiates ROS 2 publishers for each of the five joints, enabling position commands to be sent individually via /joint_i/position_cmd topics. The function also defines the physical dimensions of the robotic arm, including the lengths of its primary links (L1, L2, L3, and L4), and computes the composite reach of the distal arm segment (L34 = L3 + L4). These parameters are critical for inverse kinematic calculations and workspace analysis. To ensure safe operation, the function defines the robot's default "stand pose" and calculates the maximum reachable workspace in cylindrical coordinates, which helps prevent trajectory commands from exceeding mechanical limits. Finally, it invokes the square path generator and interpolator to produce a series of Cartesian waypoints that the controller will follow during execution.

```python
class CartesianSquareController(Node):
    def __init__(self):
        super().__init__('cartesian_square_pid_controller')

        self.joint_pubs = [self.create_publisher(Float64, f'/joint_{i}/position_cmd', 10) for i in range(5)]
        self.joint_state_sub = self.create_subscription(JointState, '/joint_states', self.joint_state_callback, 10)

        self.current_joint_angles = [0.0] * 5
        self.joint_names = [f'joint_{i}' for i in range(5)]

        self.pid_controllers = [
            PIDController(1, 0.00001, 0.2),
            PIDController(1, 0.00001, 0.2),
            PIDController(1, 0.00001, 0.2),
            PIDController(1, 0.00001, 0.2),
            PIDController(1, 0.00001, 0.2)
        ]

        self.L1, self.L2, self.L3, self.L4 = 110.0, 170.0, 62.0, 100.0
        self.L34 = self.L3 + self.L4

        self.start_time = time.time()

        self.square_path = self.generate_square_path_xy(
            fixed_z=160.0, center_x=180.0, center_y=10.0, side_length=100.0
        )
        self.path_segments = self.generate_cartesian_segments(self.square_path, steps_per_edge=40)
        self.segment_index = 0

        self.timer_period = 0.05  # 20 Hz
        self.timer = self.create_timer(self.timer_period, self.update)

    def joint_state_callback(self, msg):
        temp = {name: pos for name, pos in zip(msg.name, msg.position)}
        for i, name in enumerate(self.joint_names):
            self.current_joint_angles[i] = temp.get(name, 0.0)

    def generate_square_path_xy(self, fixed_z, center_x, center_y, side_length):
        half = side_length / 2.0
        return [
            [center_x - half, center_y - half, fixed_z],
            [center_x + half, center_y - half, fixed_z],
            [center_x + half, center_y + half, fixed_z],
            [center_x - half, center_y + half, fixed_z],
            [center_x - half, center_y - half, fixed_z],
        ]
```

Figure 5.2.2: Code Initializations

5.2.2 PID Controller

A Proportional-Derivative (PD) controller was implemented to regulate the motion of the robotic arm. The tuning process began by setting an initial value for the proportional gain Kp to observe its effect on the system's responsiveness. Gradually increasing Kp improved the rise time and reduced steady-state error. Following this, the derivative gain Kd was introduced and incrementally adjusted to reduce overshoot and enhance damping, thereby stabilizing the system's transient response.Throughout the tuning process, the system exhibited satisfactory performance without requiring an integral component Ki, as there was no significant steady-state error or drift that warranted correction. Hence, a full PID controller was not necessary, and the controller remained a PD type with zero-state change behavior observed under the tested conditions.

```python
class PIDController:
    def __init__(self, Kp, Ki, Kd, output_limits=(-math.pi, math.pi)):
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd
        self.output_limits = output_limits
        self._integral = 0.0
        self._previous_error = 0.0

    def update(self, setpoint, current_value, dt):
        error = setpoint - current_value
        p_term = self.Kp * error
        self._integral += error * dt
        self._integral = max(min(self._integral, self.output_limits[1]), self.output_limits[0])
        i_term = self.Ki * self._integral
        derivative = (error - self._previous_error) / dt if dt > 0 else 0.0
        d_term = self.Kd * derivative
        self._previous_error = error
        output = p_term + i_term + d_term
        return max(min(output, self.output_limits[1]), self.output_limits[0])
```

Figure 5.2.3: PID Controller

5.2.3 generate_square_path_xy

generate_square_path_xy(self, fixed_z, center_x, center_y, side_length) function is responsible for defining the robot's trajectory in task (Cartesian) space by generating the four corner points of a square path in the XY plane at a constant Z elevation. Given the square's center coordinates and side length, it calculates the positions of the square's vertices and closes the loop by repeating the starting point. The output is a list of [x, y, z] coordinates that describe the high-level motion goal. This simple geometric path is often used for validation and calibration of Cartesian trajectory planning and inverse kinematics performance in robotic systems.

```python
def generate_square_path_xy(self, fixed_z, center_x, center_y, side_length):
    half = side_length / 2.0
    return [
        [center_x - half, center_y - half, fixed_z],  # Bottom-left
        [center_x + half, center_y - half, fixed_z],  # Bottom-right
        [center_x + half, center_y + half, fixed_z],  # Top-right
        [center_x - half, center_y + half, fixed_z],  # Top-left
        [center_x - half, center_y - half, fixed_z],  # Closing the loop
    ]
```

Figure 5.2.4: Square path generation function

5.2.4 generate_cartesian_segments

generate_cartesian_segments(self, path, steps_per_edge) used to convert the coarse square trajectory into a continuous and smooth motion, this function linearly interpolates between each pair of adjacent path points. For each edge of the square, it generates multiple intermediate waypoints based on a normalized blending parameter alpha. This results in a higher-resolution trajectory suitable for joint-space conversion and minimizes motion discontinuities and high joint acceleration spikes. The fine segmentation is critical for generating trapezoidal velocity profiles and for real-time controllers to follow Cartesian paths accurately with minimal overshoot or oscillation.

```python
def generate_cartesian_segments(self, path, steps_per_edge):
    segments = []
    for i in range(len(path) - 1):
        p0, p1 = path[i], path[i + 1]
        for step in range(steps_per_edge):
            alpha = step / steps_per_edge
            point = [
                (1 - alpha) * p0[j] + alpha * p1[j]
                for j in range(3)
            ]
            segments.append(point)
    return segments
```

Figure 5.2.5: Cartesian path generation

5.2.5 inverse_kinematics

The inverse kinematics (IK) function is the computational heart of the Cartesian control architecture. It receives a desired end-effector position in 3D space and analytically solves for the corresponding joint angles required to reach it. The method uses a geometric approach involving the projection of the 3D point into the robot's RZ (radial and vertical) plane, followed by the application of trigonometric relations and the Law of Cosines to determine

the joint angles. Specifically, it computes the base rotation 0, shoulder 1, elbow 2, and wrist 3 such that the end-effector remains vertically oriented. Joint 4 is fixed but may represent tool or gripper alignment. The function includes workspace validation and angle clamping to ensure the solution is physically achievable and safe for the robot to execute. This IK solver assumes a non-redundant, planar elbow-down configuration, typical in fixed-base 5-DOF manipulators.

```python
def inverse_kinematics(self, position):
    x, y, z = position

    # Link lengths
    L1 = 110.0
    L2 = 170.0
    L3 = 62.0
    L4 = 100.0
    L34 = L3 + L4

    # Workspace check
    r_xy = math.hypot(x, y)
    min_r = abs(L2 - L34)
    max_r = L2 + L34
    workspace_z = L1 + (max_r / 2)
    if not (min_r <= r_xy <= max_r and abs(z - workspace_z) <= max_r):
        self.get_logger().error(f"Point {position} is outside the calculated workspace!")
        return None

    # Base joint angle (theta0)
    theta0 = math.atan2(y, x)
    z_adj = z - L1
    r = math.hypot(r_xy, z_adj)

    # Check reachability
    if r > (L2 + L34) or r < abs(L2 - L34):
        self.get_logger().warn(f"Unreachable point: {position}")
        return None

    # Elbow angle (theta2)
    cos_theta2 = (r**2 - L2**2 - L34**2) / (2 * L2 * L34)
    cos_theta2 = max(-1.0, min(1.0, cos_theta2))
    theta2 = math.acos(cos_theta2)

    # Convert to elbow-down by negating theta2
    theta2 = -theta2

    # Shoulder angle (theta1)
    k1 = L2 + L34 * math.cos(theta2)
    k2 = L34 * math.sin(theta2)
    theta1 = math.atan2(z_adj, r_xy) - math.atan2(k2, k1)

    # Wrist angle (theta3) to keep end effector vertical
    desired_wrist_angle = math.pi / 2
    theta3 = desired_wrist_angle - (theta1 + theta2)

    # Clamp to servo limits [0, π]
    theta0 = max(0.0, min(math.pi, theta0))
    theta1 = max(0.0, min(math.pi, theta1))
    theta2 = max(0.0, min(math.pi, theta2))
    theta3 = max(0.0, min(math.pi, theta3))

    return [theta0, theta1, theta2, theta3, math.pi/6]
```

Figure 5.2.6: Geometric inverse kinematics

5.2.6 compute_jacobian

The compute_jacobian function computes the analytical Jacobian matrix for the robotic arm at a given joint configuration. This 3×4 matrix linearly maps the joint velocities to the Cartesian velocities of the end-effector and is derived using partial derivatives of the forward kinematics equations with respect to each joint angle. In robotics, the Jacobian is fundamental for implementing velocity-based control schemes, Cartesian impedance control, and manipulability analysis. It also plays a crucial role in redundancy resolution, inverse differential kinematics, and detecting singularities.

```python
def compute_jacobian(self, thetas):
    theta0, theta1, theta2, theta3 = thetas[:4]
    L1 = self.L1
    L2 = self.L2
    L3 = self.L3
    L4 = self.L4

    # Forward kinematics for position
    t12 = theta1 + theta2
    t123 = theta1 + theta2 + theta3
    r = (L2 * np.cos(theta1) +
         L3 * np.cos(t12) +
         L4 * np.cos(t123))
    z = (L1 +
         L2 * np.sin(theta1) +
         L3 * np.sin(t12) +
         L4 * np.sin(t123))
    x = r * np.cos(theta0)
    y = r * np.sin(theta0)

    # Partial derivatives
    # d(x, y, z)/d(theta0)
    dx_dtheta0 = -r * np.sin(theta0)
    dy_dtheta0 =  r * np.cos(theta0)
    dz_dtheta0 = 0

    # d(x, y, z)/d(theta1)
    dr_dtheta1 = (-L2 * np.sin(theta1)
                  -L3 * np.sin(t12)
                  -L4 * np.sin(t123))
    dz_dtheta1 = (L2 * np.cos(theta1)
                  +L3 * np.cos(t12)
                  +L4 * np.cos(t123))
    dx_dtheta1 = dr_dtheta1 * np.cos(theta0)
    dy_dtheta1 = dr_dtheta1 * np.sin(theta0)

    # d(x, y, z)/d(theta2)
    dr_dtheta2 = (-L3 * np.sin(t12)
                  -L4 * np.sin(t123))
    dz_dtheta2 = (L3 * np.cos(t12)
                  +L4 * np.cos(t123))
    dx_dtheta2 = dr_dtheta2 * np.cos(theta0)
    dy_dtheta2 = dr_dtheta2 * np.sin(theta0)

    # d(x, y, z)/d(theta3)
    dr_dtheta3 = -L4 * np.sin(t123)
    dz_dtheta3 = L4 * np.cos(t123)
    dx_dtheta3 = dr_dtheta3 * np.cos(theta0)
    dy_dtheta3 = dr_dtheta3 * np.sin(theta0)

    # Assemble Jacobian
    J = np.array([
        [dx_dtheta0, dx_dtheta1, dx_dtheta2, dx_dtheta3],
        [dy_dtheta0, dy_dtheta1, dy_dtheta2, dy_dtheta3],
        [dz_dtheta0, dz_dtheta1, dz_dtheta2, dz_dtheta3]
    ])
    return J
```

Figure 5.2.7: Jacobian Function

5.2.7 run_forever

This function constitutes the core control loop and governs the execution of the entire trajectory. It begins with a warm-up phase where the robot holds a neutral standby configuration (stand pose) for 10 seconds, typically used to stabilize the robot and give the user time to observe its initial state. Thereafter, it iterates over each waypoint in the Cartesian path and calls the inverse kinematics function to obtain joint configurations. If a valid solution is found, the corresponding joint commands are published simultaneously to all actuators. The loop runs at a fixed update rate defined by the period, which corresponds to 20 Hz, ensuring smooth and deterministic motion. This real-time behavior is essential for synchronized joint control in trajectory execution and prevents mechanical resonance or unsafe movements.

```python
def run_forever(self):
    warmup_time = 10.0
    rate_hz = 20
    period = 1.0 / rate_hz

    while rclpy.ok():
        t = time.time() - self.start_time

        if t < warmup_time:
            for i, angle in enumerate(self.stand_pose):
                msg = Float64()
                msg.data = angle
                self.joint_pubs[i].publish(msg)
            self.get_logger().info(f"Holding start pose for {int(warmup_time - t)}s")
        else:
            if self.segment_index >= len(self.path_segments):
                self.segment_index = 0

            pos = self.path_segments[self.segment_index]
            angles = self.inverse_kinematics(pos)
            if angles:
                for i, angle in enumerate(angles):
                    msg = Float64()
                    msg.data = angle
                    self.joint_pubs[i].publish(msg)

                self.get_logger().info(
                    f"Segment {self.segment_index}/{len(self.path_segments)} → Pos {pos} → " +
                    ", ".join(f"{math.degrees(a):.1f}°" for a in angles)
                )

            self.segment_index += 1

        time.sleep(period)
```

Figure 5.2.8: Run forever function

5.2.8 PID Controller

6 micro-ROS Implementation on ESP32

To enable real-time control and communication between the robot arm's embedded controller and the ROS 2 ecosystem, I implemented micro-ROS on an ESP32 microcontroller. micro-ROS extends the capabilities of ROS 2 to resource-constrained devices

6.1 Setup

The system was built using the ESP-IDF toolchain along with the micro-ROS build system. Communication was established between the ESP32 (acting as the micro-ROS client) and the host PC (running the micro-ROS agent) over USB.

6.2 Client-Server Communication

The ESP32 runs the micro-ROS client, while the host machine runs the micro-ROS agent, which bridges the micro-ROS nodes on the microcontroller to the ROS 2 DDS network on the PC. This allows seamless message exchange between the two systems.

6.3 Node Structure on ESP32

A micro-ROS node is initialized on the ESP32 to publish joint states and receive joint position commands.

6.4 Callback System

The system uses non-blocking timers and callback functions to handle incoming joint position messages. Each callback updates the PWM signal sent to its corresponding servo, allowing real-time movement of the robotic arm.

6.5 System Behavior

The ESP32 remains in a loop running, checking for new ROS messages and invoking the appropriate callback. Each callback writes the new position to the corresponding servo with minimal delay. In case of initialization or transport failure, an error loop detaches all servos and halts operation.

7 Future Work

- Computer vision could be integrated to be ability to perform more advanced autonomous tasks and have better avoidance to collisions
- Jacobian groundwork laid in code could be used to : Implement Cartesian velocity tracking, Add compliance or force control or Build a more advanced motion planner.

8 Conclusion

This project successfully combined mechanical design, simulation, and real-time control to build a functional 5-DOF robotic arm. Starting with reverse engineering and SolidWorks modeling, we created an accurate digital twin of the robot, complete with joint coordinate systems for seamless ROS 2 integration. Using ROS 2 Jazzy and Gazebo Harmonic, we simulated the robot's motion and verified its kinematics. Control logic was developed through

custom Python scripts that allowed for manual input, trajectory following, and hardware testing. Finally, the ESP32 microcontroller running Micro-ROS enabled smooth communication with the robot hardware, translating ROS 2 commands into servo movements. This end-to-end workflow demonstrates a complete robotics pipeline from design to execution.