University of Messina
Department of Engineering
Engineering and Computer Science

Industrial Automation and Robotics Mod. B Final Project

Robotic Arm

Lombardo Giovanni, Marchese Salvatore and Fabiano Manuel

A.Y. 2024/2025

Contents

1 Introduction

The purpose of this work is to implement and experimentally validate a small-scale, Industry-4.0 inspired material-handling cell that combines a Fischertechnik 3-DOF robotic arm, a conveyor belt, and an autonomous rail-guided cart.

The specific goals are to:

1. Design and realise an integrated control architecture able to orchestrate the conveyor, the robotic arm, and the cart by means of a Siemens SIMATIC S7-1200 PLC.
2. Implement and optimise the pick-and-place cycle so that the arm can detect an incoming object on the conveyor, grasp it, deposit it accurately on the cart, and release it at the destination station within a target cycle time.
3. Establish a real-time Wi-Fi link between the PLC and an Arduino-based cart controller.
4. Develop firmware for the Arduino cart controller that handles motion profiles and acknowledgement messages to the PLC.
5. Develop an improved digital twin of the cell in CoppeliaSim/TIA Portal, including inverse kinematics of the arm and a kinematic model of the cart.

2 Enabling technologies

This section lists all the hardware elements that make the cell work; every component is presented in its own section with a short functional description and the key reasons for its selection.

2.1 PLC Siemens SIMATIC S7-1200

The PLC used in the project is the Siemens SIMATIC S7-1200, model 1215C AC/DC/RLY (Figure 1). It serves as the central controller, managing the automation sequence of the robotic arm, the conveyor belt, and the communication interface with external devices. The chosen model offers 14 digital inputs (24V), 10 relay outputs, as well as two analog input/output channels. These specifications proved adequate for handling the I/O requirements of the system without the need for expansion modules.

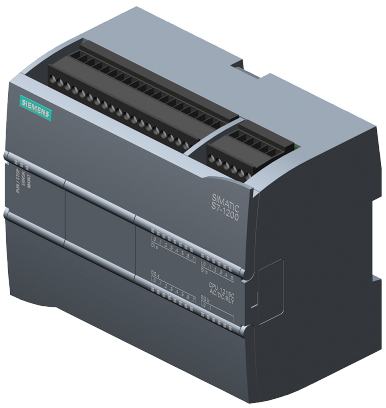

Figure 1: Siemens SIMATIC S7-1200 1215C AC/DC/RLY PLC

2.2 Fischertechnik 3D-Robot 24V

The robotic manipulator employed is the Fischertechnik 3D-Robot 24V (Figure 2), a compact educational platform offering three degrees of freedom. The structure consists of a rotational base and two linear axes: one enables vertical motion while the other controls horizontal reach. The arm terminates with a gripper, capable of opening and closing via a separate actuator.
The manipulator operates within a cylindrical workspace and features built-in position detection elements; four limit switches are used to detect the home positions of each degree of freedom, including gripper fully open, base fully clockwise, arm fully retracted, and vertical axis fully lifted. In addition, two pulse counters are integrated to measure the extent of gripper closure and the advancement of the horizontal arm. These work by registering gear rotations and converting them into digital counts.
To enhance positional feedback, the robot is also equipped with two incremental encoders (Figure 3) connected to the vertical and rotational axes. These are used in conjunction with the PLC's high-speed counters to monitor real-time displacement with higher accuracy.

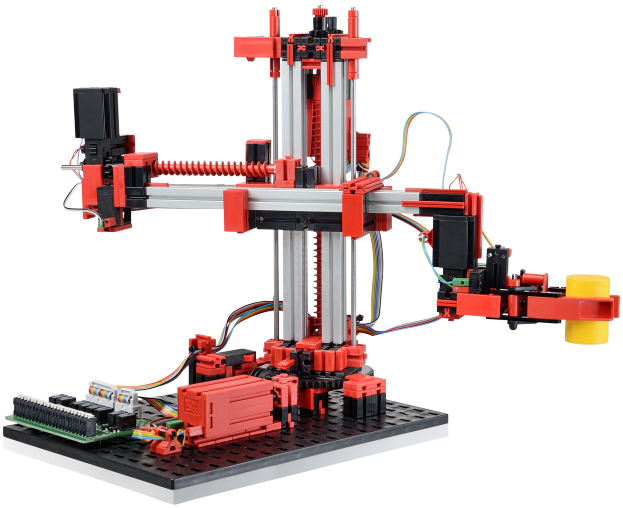

Figure 2: Fischertechnik 3D-Robot 24V

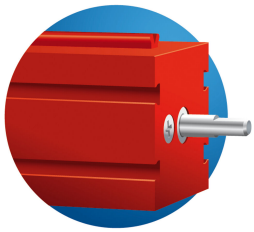

Figure 3: Motor encoders used on vertical and rotational axes

2.3 Conveyor Module

The conveyor system was repurposed from an existing prototype(Figure 4). It's powered by a DC brush motor, which operates at 3V to deliver a consistent feed rate for object transport.
To control the motion of the belt, a 24V relay (Figure 5) was introduced between the PLC and the motor. It allows the PLC to switch the 3V motor circuit without exposing the outputs to excessive current or voltage levels. Direct activation of the motor via the sensor would have been possible electrically, but it would have caused functional limitations: the motor would have restarted immediately once the sensor no longer detected an object, potentially leading

to collisions or premature motion; additionally, it would have excluded the PLC from the loop, preventing any coordinated timing or messaging toward the server.
The presence of the part at the end of the conveyor is detected using an E3F-DS10C4 photoelectric sensor (Figure 6). This cylindrical diffuse-reflection sensor outputs a digital signal when an object interrupts the reflected light beam. It's powered at 24V and connected directly to one of the PLC's digital inputs. As soon as an object is detected, the PLC halts the conveyor by switching off the relay and sends a signal over TCP to trigger the next step of the sequence.

Figure 4: Conveyor Belt repurposed and reassembled for this project

Figure 5: 24V relay used to control the motor

Figure 6: E3F-DS10C4 photoelectric sensor

2.4 Arduino-Controlled Cart

The cart is controlled by an Arduino UNO WiFi, it's driven by two continuous-rotation servos that power the left and right wheels, and has one ultrasonic sensor at the front and one at the

top, allowing the cart to recognize when it has reached the end of the rail or if the object has been loaded.

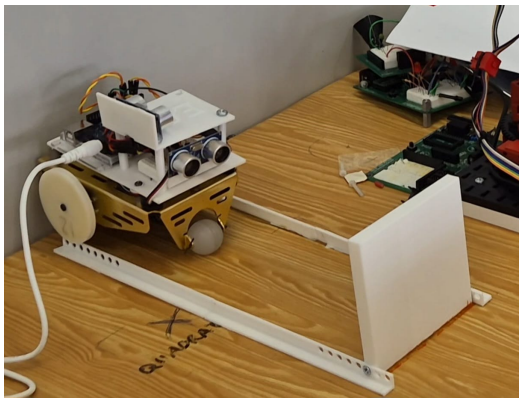

Figure 7: Arduino Cart

2.5 Laptop (Bridge Server)

A standard laptop provides the missing wireless interface between PLC and cart. The PLC is connected through an Ethernet port, while the Arduino communicates with Wi-Fi; A small relay program (Python script) listens on both interfaces and forwards messages, allowing the PLC and Arduino to communicate seamlessly even though they use different physical media.

3 Implementation Details

3.1 Ladder Logic

The following variables were defined in order to program the PLC:

Table 1: PLC variable map extracted from the Ladder program

Address	Symbolic name	Kind	Function / comment
	Digital inputs		
%I0.0	FullyOpenGripper	DI	Gripper fully open limit switch
%I0.1	GripperCounter	DI	Gripper pulse counter
%I0.2	FullBackArm	DI	Arm fully retracted limit switch
%I0.3	ArmHCounter	DI	Advancement of the horizontal axis pulse counter
%I0.4	ArmTopStop	DI	Vertical axis fully lifted limit switch
%I0.5	FullyClockwiseArm	DI	Base rotated fully clockwise limit switch
%I1.5	ObjectDetected	DI	Photo-electric sensor at conveyor exit
	Digital outputs (actuators)		
%Q0.0	GRIP_OPEN	DO	Gripper – open rotation
%Q0.1	GRIP_CLOSE	DO	Gripper – close rotation
%Q0.2	ARM_FRONT	DO	Horizontal slide – forward

(Continues on next page)

(Continued from previous page)

Address	Symbolic name	Kind	Function / comment
%Q0.3	ARM_BACK	DO	Horizontal slide – backward
%Q0.4	ARM_DOWN	DO	Vertical slide – down
%Q0.5	ARM_UP	DO	Vertical slide – up
%Q0.6	ARM_CLOCKWISE	DO	Base rotation clockwise
%Q0.7	ARM_COUNTERCW	DO	Base rotation counter-clockwise
%Q1.1	Conveyor	DO	24 V DC belt motor (active-low)
	Internal markers (flag bits)		
%M0.2	Clock_2_5Hz	M	Free-running clock (TSEND trigger)
%M1.0	FirstScan	M	TRUE only at first PLC cycle
%M1.2	AlwaysTRUE	M	Permanently forced to 1
%M6.3	Started	M	Cycle enable
%M6.4	FirstTarget	M	Seq. state 1 – arm forward and down
%M6.5	SecondTarget	M	Seq. state 2 – gripper close
%M6.6	ThirdTarget	M	Seq. state 3 – arm up, backward, and base rotates counterclockwise
%M7.3	FourthTarget	M	Seq. state 4 – arm down
%M7.4	FifthTarget	M	Seq. state 5 – release grip
%M7.5	SixthTarget	M	Seq. state 6 – arm up and return home
%M6.7	Reset	M	Request for homing
%M7.0	ForceReset	M	Reset state
%M7.1	ResetCompleted	M	Homing finished
%M8.0	CartReady	M	Set when Arduino acknowledges start
	High-speed counter / encoder words		
%MW2	RotationalCounter	MW	Base-axis encoder (HSC #257)
%MW4	VerticalCounter	MW	Vertical-axis encoder (HSC #258)
	Data blocks		
%DB1	ArmPulseCounter	DB	CTUD instance – counts motor pulses
%DB2	GripperPulseCounter	DB	CTUD instance – counts gripper pulses
%DB3	CTRL_HSC_0_DB	DB	Instance for HSC 257 and 258 setup
%DB4	TCON_DB	DB	Socket-init instance (PLC–Laptop)
%DB5	PLC_1_Connection	DB	TCP connection parameters
%DB6	TSEND_DB	DB	Send buffer (JSON *TCP_msg.Send*)
%DB7	TRCV_DB	DB	Receive buffer (JSON *TCP_msg.Receive*)

3.2 Ladder Diagram

3.2.1 Segment 1 - Sequence Controller (Finite-State Machine)

Segment 1 realises the *high-level finite-state machine (FSM)* that drives the complete pick-and-place cycle and, in addition, starts and stops the conveyor and sends the first TCP command to the cart. In Ladder this FSM is encoded with one-hot S/R markers whose bits `%M6.4 ...`

%M7.5 represent the current state. Figure 8 summarises the states, transition conditions and corresponding actuator commands.

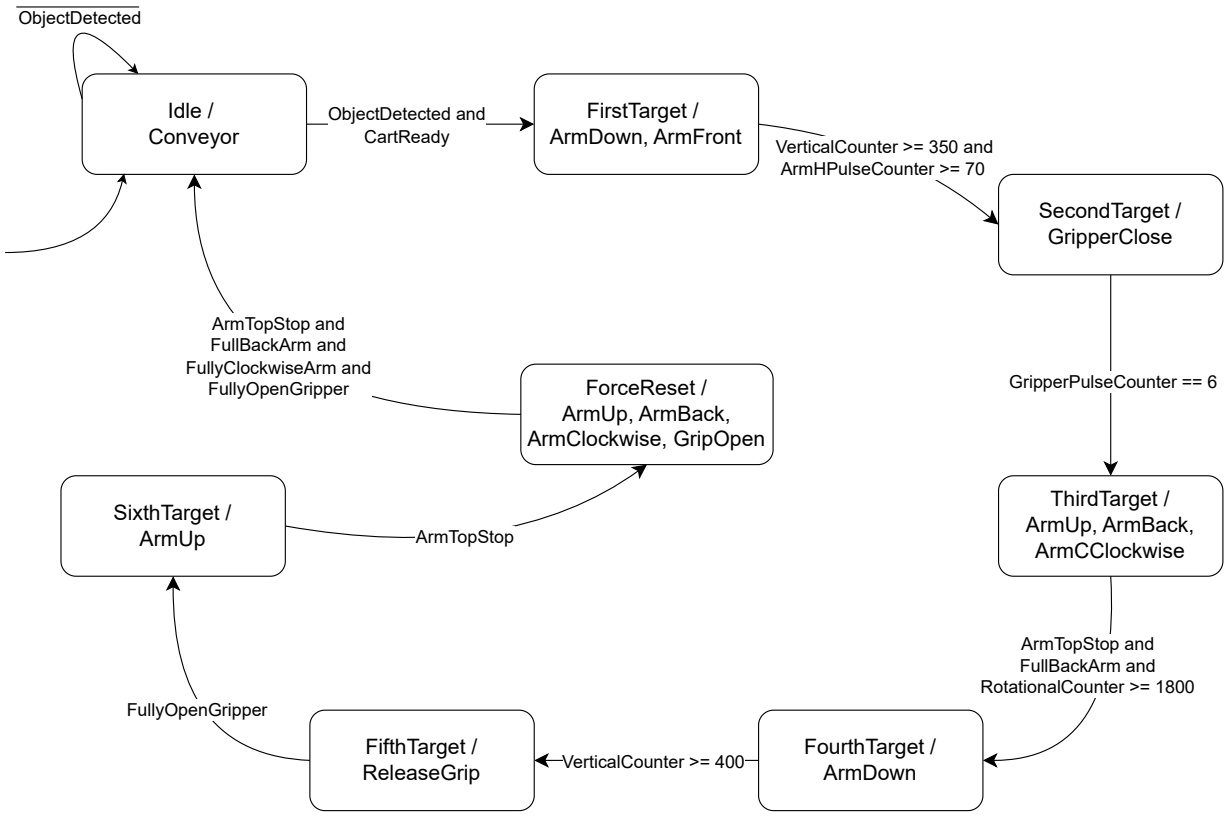

Figure 8: Finite-State Machine implemented in Segment 1

Conveyor & Object Detection

- **Rung 1** keeps the belt motor (`%Q1.1 Conveyor`) energised (because it's active-low) while the cycle hasn't been *Started* yet (`%M6.3`) or the photo sensor (`%I1.5 ObjectDetected`) hasn't detected the object yet.
- When the object is detected, the belt stops with the object exactly at the pick position.

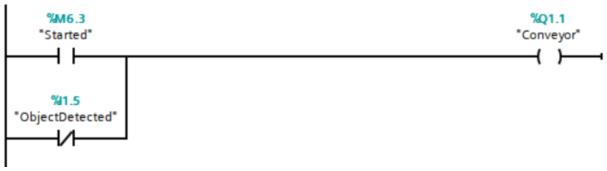

Figure 9: Rung 1 - Conveyor & Object Detection

Cart Start Command Immediately after the rising edge of `%I1.5`, **Rung 2** executes `MOVE 1 → TCP_msg.Send`. Because the `TSEND` block in Segment 5 transmits every 400 ms, the value `1` reaches the Arduino cart in the next frame, commanding it to depart.

FSM Initialisation **Rung 3** sets `%M6.4 FirstTarget` only when `%M7.0 ForceReset` is low, the system is not *Started* yet, and the cart has acknowledged readiness (`%M8.0 CartReady`).

Figure 10: Rung 2 - Cart Start Command

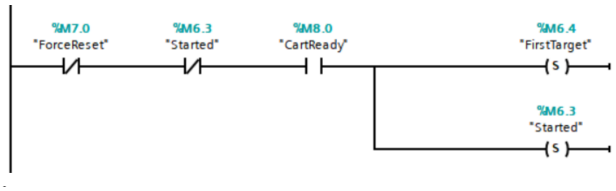

Figure 11: Rung 3 - FSM Initialisation

Motion Phases Driven by Counters Each state bit is paired with a numeric comparison: either the incremental encoder words (`MW2`, `MW4`) or the CTUD counters (*ArmPulseCounter*, *GripperPulseCounter*). When the comparison becomes false, a parallel branch issues `Set NextTarget / Reset CurrentTarget`, advancing the FSM.

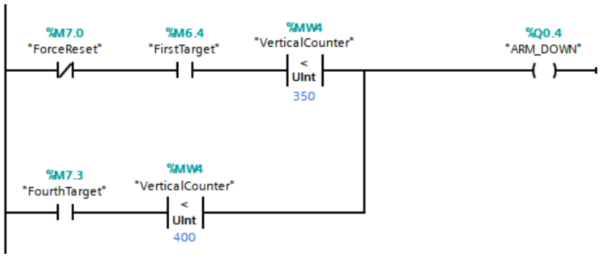

Figure 12: Rung 4 - Arm down during FirstTarget and FourthTarget

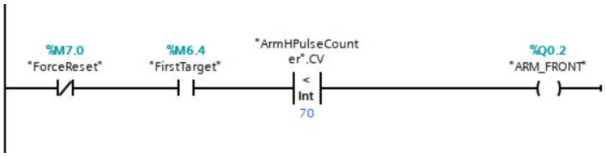

Figure 13: Rung 5 - Arm goes front during FirstTarget

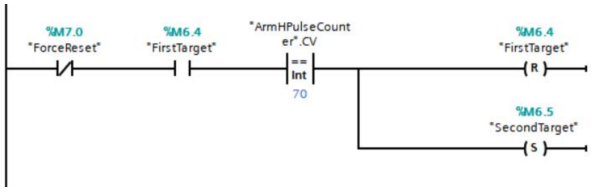

Figure 14: Rung 6 - Transition from FirstTarget to SecondTarget

Figure 15: Rung 7 - Grip closes during SecondTarget

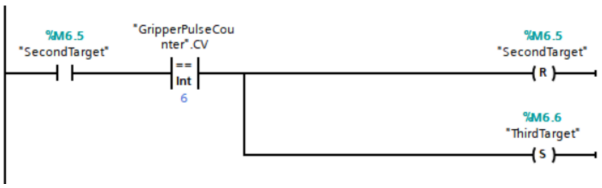

Figure 16: Rung 8 - Transition from SecondTarget to ThirdTarget

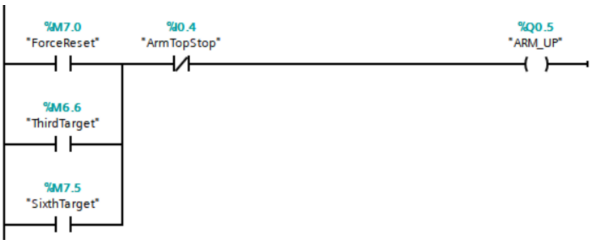

Figure 17: Rung 9 - Arm goes up until the end during Reset, ThirdTarget and SixthTarget

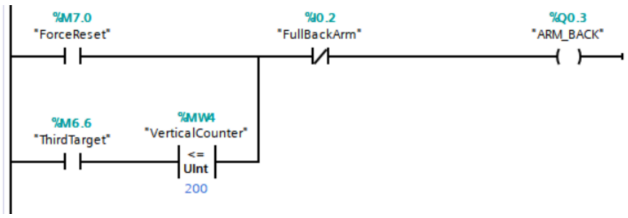

Figure 18: Rung 10 - Arm goes back until the end during Reset and ThirdTarget

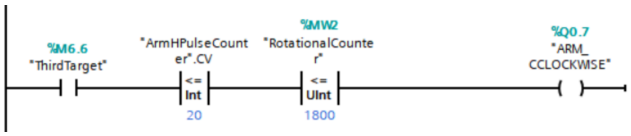

Figure 19: Rung 11 - Arm rotates counterclockwise during ThirdTarget

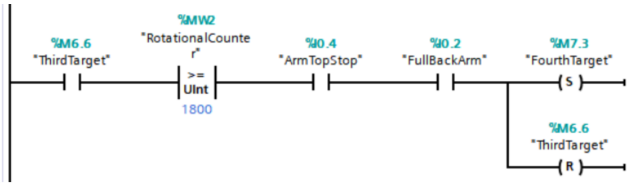

Figure 20: Rung 12 - Transition from ThirdTarget to FourthTarget

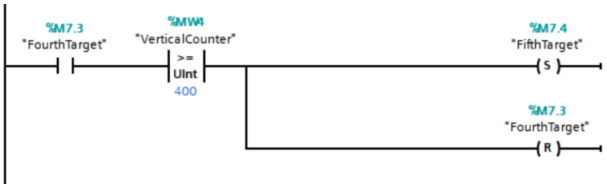

Figure 21: Rung 13 - Transition from FourthTarget to FifthTarget

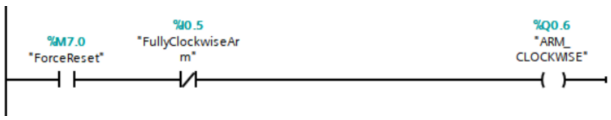

Figure 22: Rung 14 - Arm rotates clockwise until the end during Reset

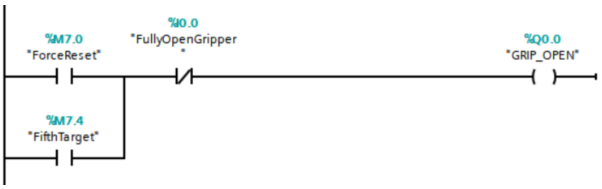

Figure 23: Rung 15 - Gripper fully opens during Reset and FifthTarget

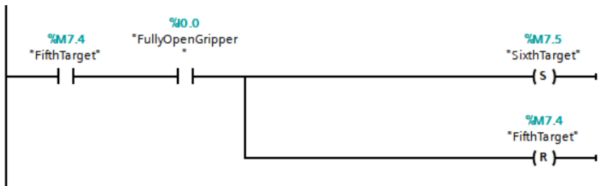

Figure 24: Rung 16 - Transition from FifthTarget to SixthTarget

Return and Reset After *SixthTarget*, the ForceReset routine starts, and the arm returns to its mechanical home.

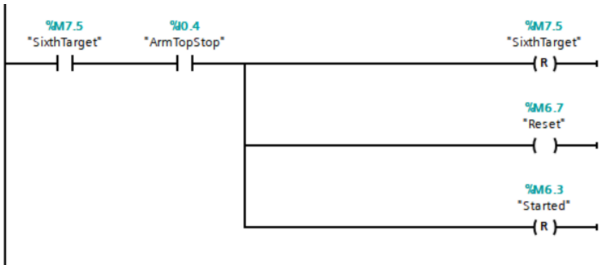

Figure 25: Rung 17 - Return and Reset

Design Rationale

- One-hot encoding prevents dual-coil conflicts: only one motion output is active at any instant.
- The handshake bit `CartReady` ensures the robot never rotates over an empty rail.
- Adding new phases requires only one extra marker and a comparison rung, keeping the program readable and extendable.

3.2.2 Segment 2 - Reset and Homing Logic

Segment 2 guarantees that the manipulator starts every new cycle from a known, reproducible reference pose.

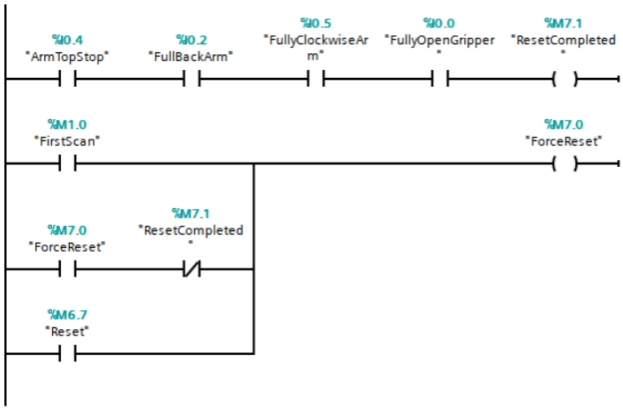

Figure 26: Segment 2 - Reset and Homing Logic

Homing condition The four limit switches are wired *in series*. The contact chain therefore evaluates to `TRUE` *only* when the vertical slide is at top, the horizontal slide at back, the base axis is clockwise at its datum, and the gripper is fully open.

ForceReset `%M7.0 ForceReset` flag is set active at power-up (`FirstScan=1`), and if the Reset routine is started after the SixthTarget. We introduced two distinct marker bits because, in Ladder logic, an address must not be driven by coils in two separate rungs: if one rung energises the coil, any subsequent rung that contains the same coil will overwrite it, typically forcing the value back to 0. In our case the "first-scan" reset rung would continuously clear the bit, so a single marker could never stay high long enough to let the homing sequence run. By separating the functions: Reset as a one-scan pulse and ForceReset as a sustained latch, we avoid that conflict and keep each rung responsible for just one coil.

Design Notes

- All interlocks are expressed with *direct* limit-switch feedback, avoiding timer-based assumptions about travel time.
- Because the four switches are wired in series, any mechanical fault (e.g. a stuck axis) blocks the chain and prevents the FSM in Segment 1 from starting.
- The logic is independent of the encoder counters; once homing is complete, other segments may safely zero the counters or preset them to reference offsets.

3.2.3 Segment 3 - Pulse-Counting for Slide and Gripper

Segment 3 converts the *drive pulses* of the horizontal slide and the gripper into numerical position feedback that the state machine in Segment 1 can test against fixed thresholds. This is done with two standard **CTUD** (Count-up/Count-down) function blocks stored in instance data blocks `DB1` and `DB2`. No timers are involved: the distance traveled depends only on the number of electrical pulses delivered to the motors.

Rung structure

1. **One-shot edge detection** Every transition of the coils from 0 to 1 drives exactly one count on the `CU` and `CD` pins.

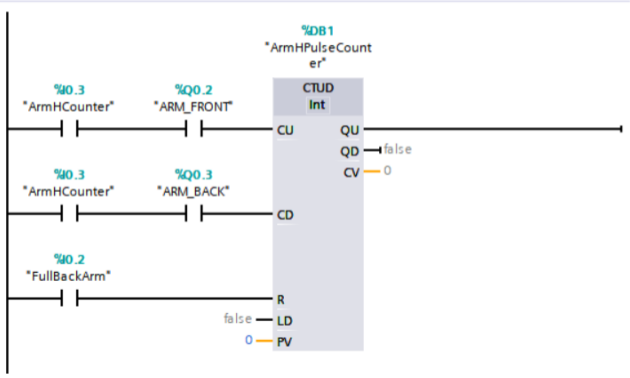

Figure 27: Horizontal Pulse Counter

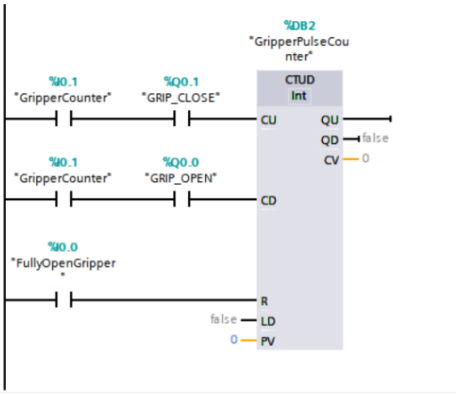

Figure 28: Gripper Pulse Counter

2. **Reset logic** The `R` input of both counters is wired to `%I0.2 FullBackArm` *or* `%I0.0 FullyOpenGripper`. Thus every homing clears the counts so that the next cycle starts from a clean zero reference.

How Segment 3 integrates with other logic

- Segment 1 reads the live `CV` values to decide when to transition from one target to the next. Because the counters are synchronous with the motor coils, motion stops exactly on the desired pulse count.
- During Reset routine, in Segment 2, the Slide reaches its end and the gripper is fully open, thus resetting its counters.

Design rationale

- A dedicated counter per axis keeps the program readable: thresholds such as "70 pulses" or "6 pulses" appear explicitly in the ladder rung, making the mechanical tuning transparent to anyone reading the code.
- Resetting via `Reset / ForceReset` guarantees that stale counts cannot propagate into a new cycle after a manual abort.

3.2.4 Segment 4 - Encoder Acquisition via High-Speed Counters

Where Segment 3 counts each time a gear involved in the corresponding movement rotate at a certain angle, Segment 4 captures the actual *feedback* coming from the two incremental encoders mounted on the vertical slide and on the base-rotation axis. Both devices are wired to the high-speed counter module integrated in the S7-1200 CPU. Figure 29 and 30 shows three ladder networks that deal with the two incremental encoders mounted on the cell:

1. *rotational encoder* on the base axis, wired to HSC #257;
2. *vertical-slide encoder* on the Z-axis, wired to HSC #258.

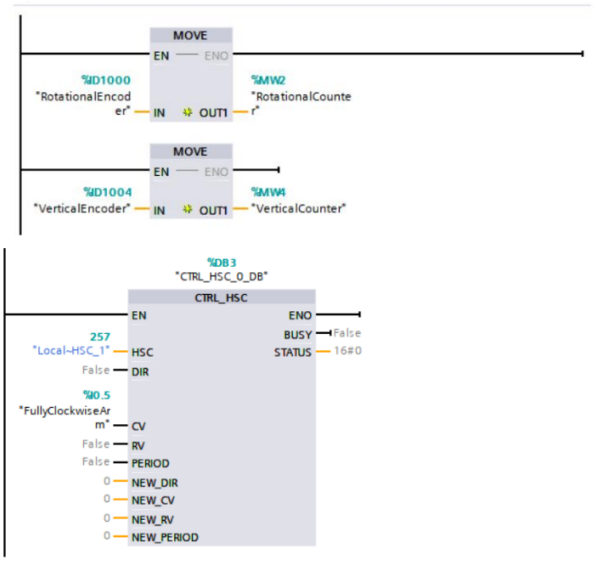

Figure 29: Rotational Encoder

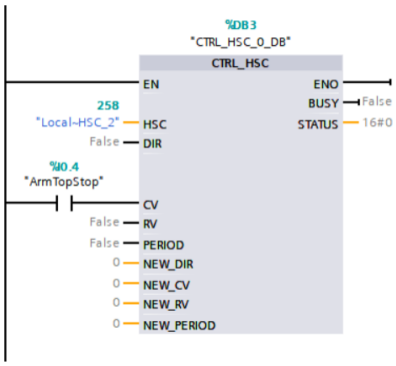

Figure 30: Vertical Encoder

The purpose of Segment 4 is two-fold: (a) mirror the raw 32-bit counts into ordinary data words that other segments can test easily, and (b) keep each high-speed counter automatically zero-referenced whenever its home sensor is reached.

Automatic re-zero of the axis While reading the sensors and holding the values the program simply overwrites its count with 0 whenever the `ArmTopStop` or `FullyClockwiseArm` switches are active. The reset happens in one scan and the counter resumes immediately.

How Segment 4 is used elsewhere

- *Segment 1* reads `MW4 < 200` to decide when the slide is clear of the conveyor (*ThirdTarget*) and `MW2 < 1800` to decide when the base rotation is complete (*FourthTarget*).

Design rationale

- Copying the system registers into normal data words (`MW2`, `MW4`) isolates the rest of the program from the special `ID` address space and improves readability.
- Using the built-in HSC channels avoids extra hardware; the two calls to `CTRL_HSC` add less than 25 µs to the cyclic time.

3.3 Segment 5 - TCP Socket Communication with the Laptop Bridge

Segment 5 implements the PLC side of the bidirectional TCP link that joins the S7-1200, the laptop bridge, and indirectly, the Arduino cart. The networks, reproduced in Fig. 31, revolve around Siemens' open IE communication blocks `TCON`, `TSEND` and `TRCV`. Table 2 summaries the three calls.

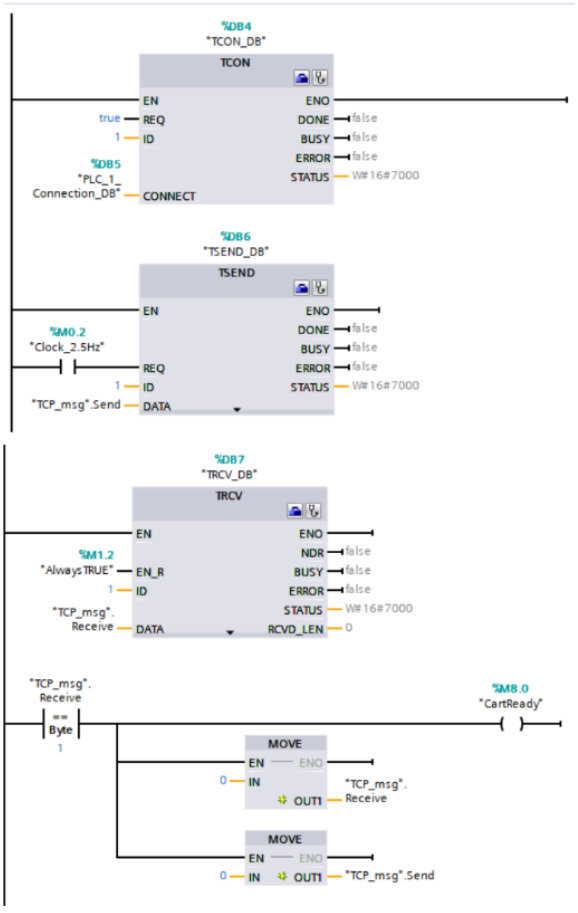

Figure 31: Segment 5

Table 2: Open-socket blocks used in Segment 5

Block	Instance DB	Key inputs	Role in the ladder
TCON	DB4	EN=1, REQ=1, ID=#1, CON_PAR=%DB5	Executed every scan; maintains a passive-open TCP socket to the laptop. Status 16#7000 confirms "connection ok".
TSEND	DB6	EN=1, REQ=%M0.2 Clock_2.5Hz, ID=#1, DATA="TCP_msg".Send	Every 400 ms (2.5 Hz) transmits the 1-byte command buffer towards the laptop; clears its own REQ automatically.
TRCV	DB7	EN=%M1.2 AlwaysTRUE, EN_R=1, ID=#1, DATA="TCP_msg".Receive	Runs continuously; when a frame arrives it copies the payload into *TCP_msg.Receive* and raises NDR for one scan.

CartReady protocol (last rung in Fig. 31)

1. A comparator "== Byte" checks whether the *first* byte of TCP_msg.Receive equals 1.
2. If the PLC receives 1, it means that the cart is ready in its starting position, coil %M8.0 CartReady is set; this bit is the prerequisite for entering *FirstTarget* in Segment 1 (see Sect. 3.2.1).
3. In the same rung two MOVE 0 instructions overwrite both the receive and the send buffers, thereby:
   - acknowledging the message (buffer no longer equal to 1),
   - preventing the next TSEND cycle from re-sending old data.

Timing considerations The 2.5 Hz trigger (%M0.2) limits outbound traffic to one frame every 400ms. Inbound frames are accepted as soon as they arrive because TRCV is permanently enabled.

3.4 Arduino Firmware

The sketch Cart.ino is responsible for every on-board function of the rail-guided cart: Wi-Fi connectivity, reception of the start command, closed-loop motion, ultrasonic ranging and the automatic return travel after the workpiece has been deposited.

3.4.1 Overall structure

- **Libraries** – WiFiNINA.h implements the TCP client; Servo.h drives the two continuous-rotation servos used as wheel motors.
- **Global resources** (only the most relevant ones are listed here):

  `ssid, pass`
  : credentials of the laptop access-point.

  `server, port`
  : IP address and socket used by the Python relay.

`motor1`, `motor2`
: `Servo` objects (left / right wheel).

`TRG_PIN_1`, `ECHO_PIN_1`
: front ultrasonic sensor.

`TRG_PIN_2`, `ECHO_PIN_2`
: top ultrasonic sensor.

- **State machine** – declared as an `enum CartState` (Listing 1); the current state is stored in the variable `cartState`.

```
enum CartState {
  WAIT_FOR_START,   // awaiting '1' byte from PLC
  MOVING_TO_TARGET, // travelling to pick-up position
  WAITING_FOR_OBJECT, // stationary, waiting for piece
  RETURNING,        // back to home rail
  IDLE              // final state after RETURNING
};
```

Listing 1: High-level states of the cart firmware

3.4.2 Communication protocol

1. `setup()` joins the Wi-Fi network and issues a `client.connect(server, port)` towards the laptop.
2. In every call to `loop()` the sketch polls `client.available()`. When a byte is present it is read, and if it is equal to 0x01 it changes state from `WAIT_FOR_START` to `MOVING_TO_TARGET`.
3. After the cart has reached the home position, the Arduino sends the message 'Target'; otherwise, it continuously sends 'Arduin' at each cycle.

3.4.3 Motion control

Both servos are driven by 50 Hz PWM signals generated by `Servo::write()`. Three helper functions encapsulate the duty settings:

`moveForward()`	⇒	L 70 µs / R 110 µs
`moveBackward()`	⇒	L 110 µs / R 70 µs
`stopMotors()`	⇒	L 90 µs / R 92 µs (neutral)

These values were empirically calibrated so that the cart drives straight and stops when necessary.

3.4.4 Ultrasonic guidance

Front and top sensors share the same timing logic:

- Each ultrasonic sensor is fired with a 10 µs HIGH pulse. A timeout of 100 ms prevents lock-up if no echo is received.
- Echo duration is captured in two `volatile` variables inside `attachInterrupt()` ISRs.
- In state `MOVING_TO_TARGET` the cart stops when the *front* distance falls below the threshold. It then enters `WAITING_FOR_OBJECT`.
- In `WAITING_FOR_OBJECT` the *top* sensor waits for the freshly dropped part to be detected, delays 8 s to let the robot clear, then commands `moveBackward()` and switches to `RETURNING`.

3.4.5 Full code

```
#include <SPI.h>
#include <WiFiNINA.h>
#include <Servo.h>

// WiFi credentials
char ssid[] = "Zancle E-Drive";
char pass[] = "7EC446D837";

// Server info
char server[] = "192.168.218.122"; // IP of PC/server
int port = 5001;

WiFiClient client;

// Servo motors
Servo motor1;
Servo motor2;

// Sensor pins
#define TRG_PIN_1 2
#define TRG_PIN_2 5
#define ECHO_PIN_1 3
#define ECHO_PIN_2 6

// Flags and timing for sensors
volatile uint8_t trig_flag_1 = 0;
volatile uint8_t echo_flag_1 = 0;
volatile uint8_t trig_flag_2 = 0;
volatile uint8_t echo_flag_2 = 0;
volatile unsigned long echoStart_1 = 0;
volatile unsigned long echoEnd_1 = 0;
volatile unsigned long echoStart_2 = 0;
volatile unsigned long echoEnd_2 = 0;

unsigned long duration_1 = 0;
unsigned long duration_2 = 0;
float distance_1 = 0;
float distance_2 = 0;


unsigned long lastTrigTime_1 = 0;
unsigned long lastTrigTime_2 = 0;
const unsigned long echoTimeout = 100; // ms

// State machine for cart behavior
enum CartState {
  WAIT_FOR_START,
  MOVING_TO_TARGET,
  WAITING_FOR_OBJECT,
  RETURNING,
  IDLE
};

CartState cartState = WAIT_FOR_START;
unsigned long returnStart = 0; // For timing return movement

// Interrupt Service Routines
void echo1() {
  switch (digitalRead(ECHO_PIN_1)) {
    case HIGH:
      echoStart_1 = micros();
      break;
    case LOW:
      echoEnd_1 = micros();
      echo_flag_1 = 1;
  }
}

void echo2() {
  switch (digitalRead(ECHO_PIN_2)) {
```

```
    case HIGH:
      echoStart_2 = micros();
      break;
    case LOW:
      echoEnd_2 = micros();
      echo_flag_2 = 1;
  }
}

void setup() {
  Serial.begin(9600);
  while (!Serial);

  // Motor pins
  motor1.attach(9);
  motor2.attach(10);

  // Sensor pin modes
  pinMode(TRG_PIN_1, OUTPUT);
  pinMode(TRG_PIN_2, OUTPUT);
  pinMode(ECHO_PIN_1, INPUT);
  pinMode(ECHO_PIN_2, INPUT);

  // Built-in LED
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, LOW);

  // Attach interrupts
  attachInterrupt(digitalPinToInterrupt(ECHO_PIN_1), echo1, CHANGE);
  attachInterrupt(digitalPinToInterrupt(ECHO_PIN_2), echo2, CHANGE);

  // WiFi connection
  int status = WiFi.begin(ssid, pass);
  Serial.print("Connecting");
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("\nConnected to WiFi");

  if (client.connect(server, port)) {
    Serial.println("Connected to server!");
  } else {
    Serial.println("Connection to server failed");
  }

  stopMotors(); // Ensure motors are stopped at startup
}

void loop() {
  // Reconnect if client disconnects
  if (!client.connected()) {
    Serial.println("Disconnected. Reconnecting...");
    client.connect(server, port);
    delay(500);
    return;
  }


  // Handle server message (start signal)
  if (client.available()) {
    Serial.println("Start signal received");
    client.read(); // Clear message
    digitalWrite(LED_BUILTIN, HIGH);
    cartState = MOVING_TO_TARGET;
  }

  Serial.println(cartState);
  // State machine logic
  switch (cartState) {
    case WAIT_FOR_START:
      stopMotors();
      break;
```

```
case MOVING_TO_TARGET:
  moveForward();
  // Trigger ultrasonic sensors
  if (!trig_flag_1) {
    digitalWrite(TRG_PIN_1, LOW);
    delayMicroseconds(2);
    digitalWrite(TRG_PIN_1, HIGH);
    delayMicroseconds(10);
    digitalWrite(TRG_PIN_1, LOW);
    trig_flag_1 = 1;
    lastTrigTime_1 = millis();
  }

  // Process echo sensor 1 (target)
  if (echo_flag_1) {
    Serial.println("Echo1");
    duration_1 = echoEnd_1 - echoStart_1;
    distance_1 = 0.0343 * (duration_1 / 2);
    echo_flag_1 = 0;
    trig_flag_1 = 0;
    if (distance_1 < 5) {
      Serial.println("Arrived to target");
      client.write("Target");
      stopMotors();
      cartState = WAITING_FOR_OBJECT;
    }
  }

  // Timeout for sensor 1
  if (trig_flag_1 && (millis() - lastTrigTime_1 >= echoTimeout)) {
    Serial.println("Sensor1 timeout");
    trig_flag_1 = 0;
    echo_flag_1 = 0;
  }



  break;

case WAITING_FOR_OBJECT:
  if (!trig_flag_2) {
    digitalWrite(TRG_PIN_2, LOW);
    delayMicroseconds(2);
    digitalWrite(TRG_PIN_2, HIGH);
    delayMicroseconds(10);
    digitalWrite(TRG_PIN_2, LOW);
    trig_flag_2 = 1;
    lastTrigTime_2 = millis();
  }



  // Process echo sensor 2 (object)
  if (echo_flag_2) {
    Serial.println("Echo2");
    duration_2 = echoEnd_2 - echoStart_2;
    distance_2 = 0.0343 * (duration_2 / 2);
    echo_flag_2 = 0;
    trig_flag_2 = 0;
    if (distance_2 < 5) {
    Serial.println("Object placed, returning");
    delay(8000);
    moveBackward();
    returnStart = millis(); // Start return timer
    cartState = RETURNING;
  }
  }

  // Timeout for sensor 2
  if (trig_flag_2 && (millis() - lastTrigTime_2 >= echoTimeout)) {
    Serial.println("Sensor2 timeout");
    trig_flag_2 = 0;
```

```
      echo_flag_2 = 0;
    }

    break;

  case RETURNING:
    if (millis() - returnStart >= 2000) {
      Serial.println("Return complete. Stopping.");
      stopMotors();
      digitalWrite(LED_BUILTIN, LOW);
      cartState = IDLE;
    }
    break;

  case IDLE:
    // Reset to initial state
    cartState = WAIT_FOR_START;
    break;
 }

 delay(10); // Reduce loop frequency

 client.write("Arduin");
}

// -----------------
// Helper Functions
// -----------------

void moveForward() {
 motor1.write(70); // Calibrate this if needed
 motor2.write(110);
}

void moveBackward() {
 motor1.write(110); // Calibrate this if needed
 motor2.write(70);
}

void stopMotors() {
 motor1.write(90); // Use calibrated stop value if necessary
 motor2.write(92);
}
```

Listing 2: Arduino's firmware

3.5 Laptop relay service

The file `main.py` mplements a lightweight bridge between the PLC and the Arduino cart. Communication is purely byte-oriented: the value `0x01` means "start", while the ASCII string `"Target"` means "cart in position".

Table 3: Network end-points handled by `main.py`

Device	IP → Laptop	TCP port
Arduino UNO WiFi Rev. 2	`192.168.4.101`	5001
PLC (S7-1200)	`192.168.0.10`	2001

3.5.1 Thread model

1. *Main thread*: spawns two listener threads, one per port.
2. *Arduino thread* — executes `handle_arduino()`:

- if the global flag `start_cart = 1` it sends the single byte `0x01` to the cart and resets the flag;
- upon receiving exactly the string `"Target"` it sets `arrived_to_target = 1`.

3. *PLC thread* — executes `handle_plc()`:
   - if `arrived_to_target = 1` it transmits `0x01` to the PLC and clears the flag;
   - otherwise it waits for a single byte from the PLC; if that byte equals `0x01` it sets `start_cart = 1`.

The entire exchange is driven by two Boolean globals; no locks are needed because each variable is written by only one thread and read by the other, ensuring thread-safe behaviour on Python.

3.5.2 Sequence of events

Step 1: The PLC detects the part at the conveyor and sends `0x01`.

Step 2: The PLC thread sets `start_cart` (**flag A**).

Step 3: The Arduino thread sees flag A, sends `0x01` to the cart, then clears flag A.

Step 4: When the cart reaches the pick-up point it replies `"Target"`. The Arduino thread sets `arrived_to_target` (**flag B**).

Step 5: The next PLC-thread iteration transmits `0x01` to the PLC, clears flag B, and the robot sequence starts.

3.5.3 Full code

```
import socket
import threading

#Impostazioni porte
ARDUINO_PORT = 5001
PLC_PORT = 2001
SERVER_IP = '0.0.0.0' # ascolta su tutte le interfacce

client_socket_arduino = None
addr_arduino = None
client_socket_PLC = None
addr_plc = None
arrived_to_target = 0

start_cart = 0

def handle_arduino(client_socket, addr, dispositivo):
    global start_cart, arrived_to_target
    print(f"[{dispositivo}] Connection from {addr}")
    while True:
        #if an objet was detected send command to arduino to start cart
        if start_cart:
            print("Starting cart")
            client_socket.sendall(bytes(1))
            start_cart = 0

        data = client_socket.recv(6)
        #if received data from Arduino this means the cart is ready, so set arrived_to_target flag
        if data.decode()=="Target" and not arrived_to_target:
            print("Cart Ready!")
            arrived_to_target = 1
```

```
def handle_plc(client_socket, addr, dispositivo):
    global start_cart, arrived_to_target
    print(f"[{dispositivo}] Connection from {addr}")
    while True:

            #if the cart arrived to target, send trigger to PLC to start the arm
            if arrived_to_target:
                print("Starting arm")
                data = bytes([1])
                client_socket.sendall(data)
                arrived_to_target = 0
            else:
                data = client_socket.recv(1)
                data = data[0] #convert to int
                if data==1 and not start_cart:
                    #If received data from PLC, this means an object is detected, so set start_cart flag
                    print("Object detected")
                    start_cart=1



def avvia_listener_arduino(porta, dispositivo):
    global client_socket_arduino, addr_arduino
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server:
        server.bind((SERVER_IP, porta))
        server.listen()
        print(f"[{dispositivo}] Waiting on port {porta}")
        while True:
            client_socket_arduino, addr_arduino = server.accept()
            threading.Thread(target=handle_arduino, args=(client_socket_arduino, addr_arduino, dispositivo)).start()

def avvia_listener_plc(porta, dispositivo):
    global client_socket_plc, addr_plc
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server:
        server.bind((SERVER_IP, porta))
        server.listen()
        print(f"[{dispositivo}] Waiting on port {porta}")
        while True:
            client_socket_plc, addr_plc = server.accept()
            threading.Thread(target=handle_plc, args=(client_socket_plc, addr_plc, dispositivo)).start()



def main():
    threading.Thread(target=avvia_listener_arduino, args=(ARDUINO_PORT, "ARDUINO")).start()
    threading.Thread(target=avvia_listener_plc, args=(PLC_PORT, "PLC")).start()

if __name__ == "__main__":
    main()
```

Listing 3: Relay server's code

4 CoppeliaSim

CoppeliaSim, formerly known as V-REP, is a robot simulator used in industry, education, and research. It was used to simulate the behavior of the real robot considering a more complex environment and also by computing **reverse kinematic**.
The software already includes a list of available robot models, but to make the simulation more similar to the real world, a reproduction of the Fischertechnik robot was used. The robot model was taken from the project of Catalfamo, Morabito, Nocera, Sebbio to which the gripper was added.

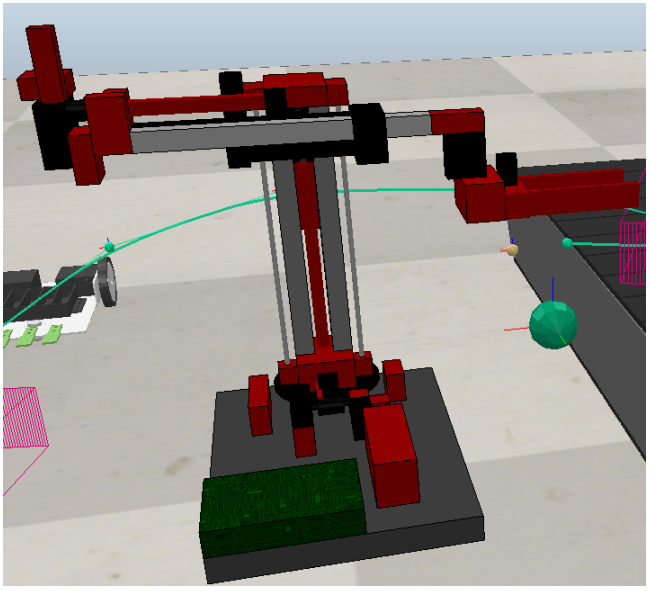

Figure 32: Screenshot of the robot model in coppelia

4.1 Robot model

The model is a complete reproduction of the original robot, including joints and other stuff such as motors and sensors. In particular, there are one revolute joint and two prismatic joints. Some changes to the value ranges of the joints were performed in order to make Inverse Kinematic solver correctly work. In addition, the original model did not include the gripper, and so for this project it was decided to add it. In particular, the gripper has been designed as composed of two revolute joints controlling the opening and closing movement of two "fingers" (Figure 33). The gripper is not included in inverse kinematic, so the joints have been set to dynamic mode, and the opening and closing movements are controlled through scripts.

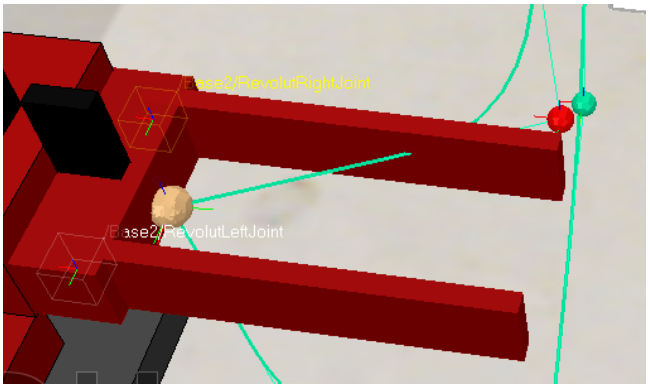

Figure 33: Gripper

4.2 Testing environment

CoppeliaSim gives the possibility to create very complex environments. Thanks to this, it has been possible to reproduce a working experiment in which an object, moving on a conveyor belt, is moved by a robot arm on a cart and then moved towards another robot arm which takes and places it on a table (Figure 34).
To reproduce all these steps, the environment is composed of the following elements:

- **x2 Fischertechnik robots**, placed in different positions. These are responsible for moving the object from the conveyor to the cart, and from the cart to the table.
- **x1 conveyor belt**, this is used to move the object towards the first arm.
- **x4 proximity sensors**, these are placed respectively: at the end of the conveyor to detect an object, near the first arm to detect the presence of the cart, above the cart to detect the presence of an object, near the second arm to detect the presence of the cart.
- **x1 cart**, the cart will move forward and back to carry the object from one arm to the other.
- **x6 paths**, used to move the target object for inverse kinematics and correctly move the robots.
- **1x cuboid**, the object moved between the arms.
- **x1 custom table**, an ad-hoc created model of a table to place object carried by the second arm.

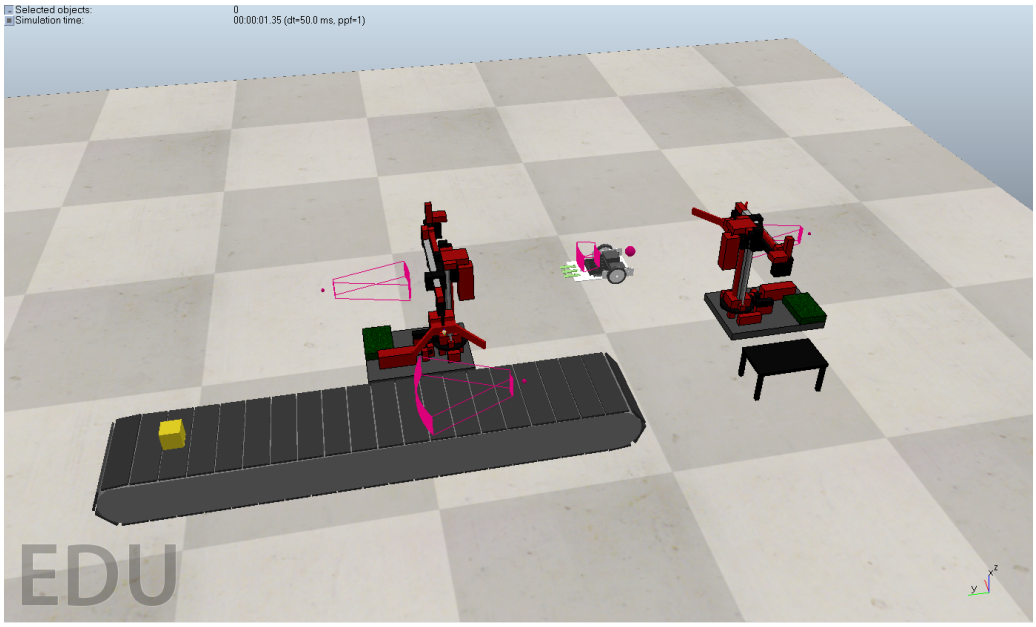

Figure 34: Overview of the simulated environment

Proximity sensors play an important role because they represent triggers for initiating actions. These are the working steps of the experiment:

1. At the beginning, an object is placed on the conveyor and starts moving towards the first sensor.
2. When the object is detected by the first sensor, it will start the motion of the cart by moving its joints. The cart will go forward until it will reach the second sensor.
3. This will stop the cart and start the first arm motion to carry the object.
4. When the object will be placed on the cart, the sensor on top of the cart will detect it and will start moving back the cart to the second arm.
5. Here, the fourth sensor will detect the presence of the cart and trigger the second arm to take the object and place it on the table.

These operations are repeated every time an object is placed on the conveyor. Trigger events are handled using scripts; more details are given later.

4.3 Inverse kinematics

The inverse kinematics problem consists of determining the joint variables corresponding to a given end-effector position and orientation. The solution to this problem is of fundamental importance for transforming the motion specifications, assigned to the end-effector in the operational space, into the corresponding joint space motions that allow the execution of the desired movement. In some cases, but not all, there exist analytical solutions to inverse kinematic problems. One such example is for a 6-Degrees-of-Freedom (DoF) robot (for example, 6 revolute joints) moving in 3D space (with 3 position degrees of freedom, and 3 rotational degrees of freedom). An analytic solution to an inverse kinematics problem is a closed-form expression that takes the end-effector pose as input and returns the joint positions as output,

$$q = f(x)$$

. Analytical inverse kinematics solvers can be significantly faster than numerical solvers and provide multiple, but only finitely many solutions, for a given end-effector pose.

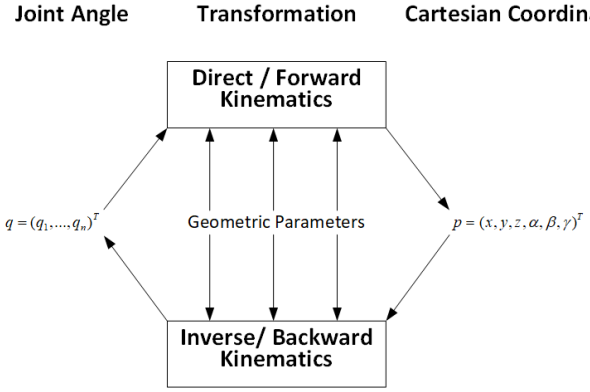

Figure 35: Kinematics

4.3.1 Inverse kinematics in CoppeliaSim

CoppeliaSim provides an easy way to solve inverse kinematics problems, allowing a robot to move to a desired position. The tool provided by CoppeliaSim has evolved over different software versions. For this project, CoppeliaSim 4.9 was used.
The steps used to solve inverse kinematics problem are:

1. Load or create a robot model. Note: if the model is custom, it's necessary to set the robot base as "model" from properties.

2. Create two dummies objects, usually called **"Tip"** and **"Target"**. The tip has to be placed on the end-effector, and the target in an arbitrary position. These objects represent the core of CoppeliaSim's inverse kinematics. Indeed, the software will try to find a way, by solving inverse kinematics problem, to move the robot in such a way that the tip reaches the target. So, by correctly placing the Tip and then moving the target on specific positions, it will be possible to move end-effector (by moving all the joints of the robot) in the position specified by the target.

3. CoppeliaSim needs to know which are the joints included in the chain, so, it's necessary to create a joint group. First, it's necessary to select the joints included in the chain by clicking on them and holding CTRL button, then from: Modules->Kinematics->Define joint group and choosing a name, it will be possible to create the chain. Note: the joints of the gripper

don't have to be included, because the inverse kinematics is used to move the end-effector itself.

4. The next step is to generate the script that will perform inverse kinematics. Click on Modules->Kinematics->Inverse kinematics generator and the menu on Figure 36 will appear. From this menu, it is necessary to select the base of the model (it was defined as

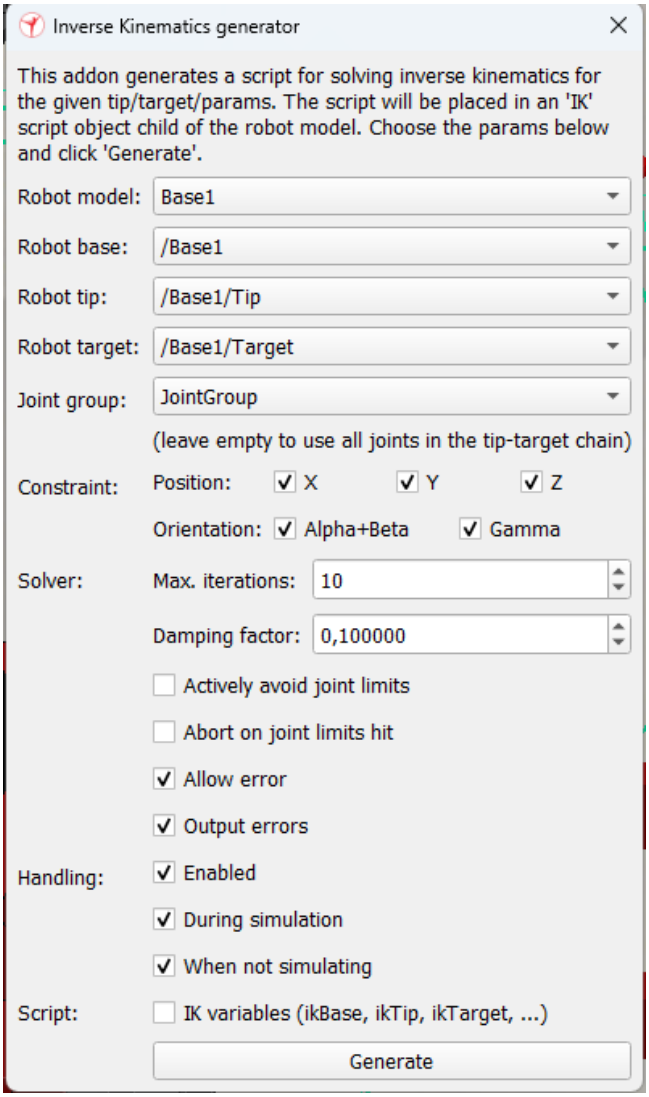

Figure 36: Inverse kinematics generator option menu

"model" in the first step), the base, the Tip object, the Target object, and the JointGroup defined in the previous step. Note: Tip and Target objects have to be child of the base. Then, it is possible to set some parameters such as constrain, max-iterations and damping factor and some options for the script. After clicking on "generate" a new Lua script named "IK" will appear. This is the script responsible for solving inverse kinematics of the robot and the selected joints.

5. Now by moving the target in different positions it will be possible to see that the end effector will follow it accordingly.

4.4 Scripts

CoppeliaSim gives the possibility to simulate our system by easily dragging and dropping elements in the environment, however, the most important logic that drives our application is contained

within the scripts. CoppeliaSim gives the possibility to add threaded or non-threaded scripts in Python or Lua language. For this project, only non-threaded Lua scripts have been used. Every script can be associated to an object, a model, or to the entire scene, and they are usually used for specific actions. For this project 6 scripts have been created:

- One script associated to the conveyor belt. It is used to control the cart start and to stop and restart the conveyor.
- One IK script associated to the first robot. It is used to solve inverse kinematics for the first robot arm.
- One script associated to the target of the first arm robot. It is used to move the target of the first robot along the first three paths.
- One IK script associated to the second robot. It is used to solve inverse kinematics for the second robot arm.
- One script associated to the target of the second robot arm. It is used to move the target of the second robot along the last three paths.
- One generator script associated to the global scene. It is used to generate a cuboid on the conveyor.

4.4.1 Conveyor belt script

```
function sysCall_init()
    sim = require('sim')

    conveyor1 = sim.getObject('/conveyor1')

    leftJointCart=sim.getObject("/DynamicLeftJoint")
    rightJointCart=sim.getObject("/DynamicRightJoint")

    sensor1 = sim.getObject('/sensor1')
    sensor2 = sim.getObject('/sensor2')
    sensor3 = sim.getObject('/sensor3')
    cartSensor = sim.getObject('/cartSensor')
    previousT = 0
    -- do some initialization here
end

function sysCall_actuation()
    -- put your actuation code here
    res1, dist1, point1, obj1, n1 = sim.readProximitySensor(sensor1)
    res2, dist2, point2, obj2, n2 = sim.readProximitySensor(sensor2)
    resc, distc, pointc, objc, nc = sim.readProximitySensor(cartSensor)
    res3, dist2, point2, obj2, n2 = sim.readProximitySensor(sensor3)

    --if an object on conveyor is detected
    if res1==1 then
        print("Object detected, moving cart")
        sim.writeCustomTableData(conveyor1, '__ctrl__', {['vel'] = 0.0})
        sim.setJointTargetVelocity(leftJointCart,2.0)
        sim.setJointTargetVelocity(rightJointCart,2.0)
    end

    --if the cart arrived to target and the object has not been placed
    -- or if the object is at second arm and there is not an object on conveyor
    if (res2==1 and resc == 0) or (res3 == 1 and res1 == 0) then
        print("Stopping cart")
        sim.setJointTargetVelocity(leftJointCart,0)
        sim.setJointTargetVelocity(rightJointCart,0)
        previousT = sim.getSimulationTime()
    end
    if resc == 1 and res3 == 0 then
```

```
        t = sim.getSimulationTime()
        if t - previousT > 5 then
            print("Cart going back")
            sim.setJointTargetVelocity(leftJointCart,-2.0)
            sim.setJointTargetVelocity(rightJointCart,-2.0)
        end

    end


end

function sysCall_sensing()
    -- put your sensing code here

end

function sysCall_cleanup()
    -- do some clean-up here
end

-- See the user manual or the available code snippets for additional callback functions and details
```

By using the function *readProximitySensor* it is possible the read the state of all the four sensors. In particular, when the first return value can be used to know if the sensors have detected something. When *res1* is set to 1, it means that sensor1 has detected the object on the conveyor and so it can stop the conveyor and start the cart. When *res2* is equal to 1 and *resc* is equal to 0 or *res3* is equal to 1 and *res1* is equal to 0 (meaning that the cart is ready to pick up an object, but the object has not been placed yet, or that the cart arrived to the second arm and there is not any object detected on conveyor), stop the cart. If instead, *resc* is equal to 1 and *res3* is equal to 0 (meaning that an object has been placed on the cart but the cart is not near the second arm), wait 5 simulation time units and then move back the cart.

4.4.2 IK script

```
sim = require 'sim'
simIK = require 'simIK'

function sysCall_init()
    self = sim.getObject '.'

    simBase = sim.getReferencedHandle(self, 'ik.base')
    simTip = sim.getReferencedHandle(self, 'ik.tip')
    simTarget = sim.getReferencedHandle(self, 'ik.target')
    jointGroup = sim.getReferencedHandle(self, 'jointGroup')
    simJoints = sim.getReferencedHandles(jointGroup)

    enabled = true
    handleWhenSimulationRunning = true
    handleWhenSimulationStopped = false
    dampingFactor = 0.100000
    maxIterations = 1000
    if dampingFactor > 0 then
        method = simIK.method_damped_least_squares
    else
        method = simIK.method_pseudo_inverse
    end
    constraint = simIK.constraint_position
    ikOptions = {
        syncWorlds = true,
        allowError = true,
    }
    ikEnv = simIK.createEnvironment()
    ikGroup = simIK.createGroup(ikEnv)
    simIK.setGroupCalculation(ikEnv, ikGroup, method, dampingFactor, maxIterations)
```

```
    _, ikHandleMap, simHandleMap = simIK.addElementFromScene(ikEnv, ikGroup, simBase, simTip, simTarget, constraint
        )

    if jointGroup then
        -- disable joints not part of the joint group
        ikJoints = {}
        for i, ikJoint in ipairs(simIK.getGroupJoints(ikEnv, ikGroup)) do
            if table.find(simJoints, simHandleMap[ikJoint]) then
                table.insert(ikJoints, ikJoint)
            else
                simIK.setJointMode(ikEnv, ikJoint, simIK.jointmode_passive)
            end
        end
    else
        ikJoints = simIK.getGroupJoints(ikEnv, ikGroup)
    end
end

function sysCall_actuation()
    if enabled and handleWhenSimulationRunning then
        handleIk()
    end
end

function sysCall_nonSimulation()

end

function sysCall_cleanup()
    simIK.eraseEnvironment(ikEnv)
end

function handleIk()
    local result, failureReason = simIK.handleGroup(ikEnv, ikGroup, ikOptions)
    if result ~= simIK.result_success then
        sim.addLog(sim.verbosity_errors, 'IK failed: ' .. simIK.getFailureDescription(failureReason))
    end
end

function getEnvironment()
    return ikEnv
end

function getGroup()
    return ikGroup
end

function getElement()
    return ikElement
end

function getBase()
    return simBase
end

function getTip()
    return simTip
end

function getTarget()
    return simTarget
end

function getIkJoints()
    return ikJoints
end

function getIkConfig()
    return map(partial(simIK.getJointPosition, ikEnv), getIkJoints())
end

function setIkConfig(cfg)
    foreach(partial(simIK.setJointPosition, ikEnv), getIkJoints(), cfg)
```

```
end

function getEnabled()
    return enabled
end

function setEnabled(b)
    enabled = not not b
end

function getHandleWhenSimulationRunning()
    return handleWhenSimulationRunning
end

function setHandleWhenSimulationRunning(b)
    handleWhenSimulationRunning = not not b
end

function getHandleWhenSimulationStopped()
    return handleWhenSimulationStopped
end

function setHandleWhenSimulationStopped(b)
    handleWhenSimulationStopped = not not b
end

function mapToIk(simHandle)
    return ikHandleMap[simHandle]
end

function mapToSim(ikHandle)
    return simHandleMap[ikHandle]
end
```

This is the automatic generated code by the inverse kinematic plugin. The script initialize the solver using the parameters specified in the Figure 36 and actuates it. The scripts for the robots are identical.

4.4.3 First robot target script

```
local objectToFollowPath = -1
local path1 = -1
local path2 = -1
local path3 = -1
local path1Positions = {}
local path1Quaternions = {}
local path2Positions = {}
local path2Quaternions = {}
local path3Positions = {}
local path3Quaternions = {}
local path1Lengths = {}
local path2Lengths = {}
local path3Lengths = {}
local total1Length = 0
local total2Length = 0
local total3Length = 0
local velocity = 0.08 -- m/s
local posAlongPath1 = 0
local posAlongPath2 = 0
local posAlongPath3 = 0
local previousSimulationTime = 0
local path1Completed = false
local path2Completed = false
local path3Completed = false
local startPath2 = false
local startPath3 = false
local cuboid
local gripperBase
local reachedPoint6 = false
local start = 0
```

```
function sysCall_init()
    sim = require('sim')
    objectToFollowPath = sim.getObject('/Target')
    path1 = sim.getObject('/Path1')
    path2 = sim.getObject('/Path2')
    path3 = sim.getObject('/Path3')
    sensor2 = sim.getObject('/sensor2')
    rightfinger = sim.getObject('/RevolutRightJoint')
    leftfinger = sim.getObject('/RevolutLeftJoint')

    gripperBase = sim.getObject('/Base1/Gripper')
    conveyor1 = sim.getObject('/conveyor1')


    if objectToFollowPath == -1 or path1 == -1 or path2 == -1 or path3 == -1 then
        sim.addLog(sim.verbosity_errors, "Error: Could not find '/Target' or '/Path1' or '/Path2' or '/Path3'
            objects.")
        sim.pauseSimulation()
        return
    end

    local path1Data = sim.unpackDoubleTable(sim.getBufferProperty(path1, 'customData.PATH'))
    local path2Data = sim.unpackDoubleTable(sim.getBufferProperty(path2, 'customData.PATH'))
    local path3Data = sim.unpackDoubleTable(sim.getBufferProperty(path3, 'customData.PATH'))
    local m1 = Matrix(#path1Data // 7, 7, path1Data)
    local m2 = Matrix(#path2Data // 7, 7, path2Data)
    local m3 = Matrix(#path3Data // 7, 7, path3Data)
    path1Positions = m1:slice(1, 1, m1:rows(), 3):data()
    path1Quaternions = m1:slice(1, 4, m1:rows(), 7):data()
    path2Positions = m2:slice(1, 1, m2:rows(), 3):data()
    path2Quaternions = m2:slice(1, 4, m2:rows(), 7):data()
    path3Positions = m3:slice(1, 1, m3:rows(), 3):data()
    path3Quaternions = m3:slice(1, 4, m3:rows(), 7):data()

    path1Lengths, total1Length = sim.getPathLengths(path1Positions, 3)
    path2Lengths, total2Length = sim.getPathLengths(path2Positions, 3)
    path3Lengths, total3Length = sim.getPathLengths(path3Positions, 3)

    previousSimulationTime = 0 -- Force zero time delta at start

    startPath2 = false
    startPath3 = false
    path1Completed = false -- Ensure path is not marked complete at start
    path2Completed = false
    path3Completed = false
    start = 0
end

function sysCall_actuation()
    res2, dist2, point2, obj2, n2 = sim.readProximitySensor(sensor2)


    --Starts only if the cart is arrived
    if start == 0 and res2 == 1 then
        --Reset flags
        path3Completed = false
        path2Completed = false
        path1Completed = false
        startPath2 = false
        startPath3 = false
        previousSimulationTime = 0 -- Force zero time delta at start
        posAlongPath1 = 0
        posAlongPath2 = 0
        posAlongPath3 = 0
        cuboid = sim.getObject('/Cuboid')
        start = 1
    end
    if start == 1 then
        if previousSimulationTime == 0 then
            previousSimulationTime = sim.getSimulationTime()
            return
        end
```

```
local t = sim.getSimulationTime()
local deltaTime = t - previousSimulationTime

-- If the path is already completed, do nothing more
if path3Completed then
    start = 0
    return
end

if startPath2 == true then
    posAlongPath2 = posAlongPath2 + velocity * deltaTime

    -- Check if the end of the path has been reached or exceeded
    if posAlongPath2 >= total2Length then
        posAlongPath2 = total2Length -- Clamp to the end of the path
        path2Completed = true    -- Mark path as completed
        --Open gripper and leave object
        sim.setObjectParent(cuboid, -1, true)
        sim.setObjectInt32Param(cuboid, sim.shapeintparam_static, 0)
        sim.setObjectInt32Param(cuboid, sim.shapeintparam_respondable, 1)
        sim.setJointTargetPosition(rightfinger, 0.61)
        sim.setJointTargetPosition(leftfinger, -0.61)
        --Restart conveyor
        sim.writeCustomTableData(conveyor1, '__ctrl__', {['vel'] = 0.1})
        startPath2 = false
        startPath3 = true
    end

    -- Interpolate position and quaternion along the path
    local pos = sim.getPathInterpolatedConfig(path2Positions, path2Lengths, posAlongPath2)
    local quat = sim.getPathInterpolatedConfig(path2Quaternions, path2Lengths, posAlongPath2, nil, {2, 2, 2,
          2})

    -- Set the position and quaternion of the object
    sim.setObjectPosition(objectToFollowPath, path2, pos)
    sim.setObjectQuaternion(objectToFollowPath, path2, quat)

elseif startPath3 == true then
    posAlongPath3 = posAlongPath3 + velocity * deltaTime

    -- Check if the end of the path has been reached or exceeded
    if posAlongPath3 >= total3Length then
        posAlongPath3 = total3Length -- Clamp to the end of the path
        path3Completed = true    -- Mark path as completed
    end

    -- Interpolate position and quaternion along the path
    local pos = sim.getPathInterpolatedConfig(path3Positions, path3Lengths, posAlongPath3)
    local quat = sim.getPathInterpolatedConfig(path3Quaternions, path3Lengths, posAlongPath3, nil, {2, 2, 2,
          2})

    -- Set the position and quaternion of the object
    sim.setObjectPosition(objectToFollowPath, path3, pos)
    sim.setObjectQuaternion(objectToFollowPath, path3, quat)


else
    --follow Path1
    posAlongPath1 = posAlongPath1 + velocity * deltaTime

    -- Check if the end of the path has been reached or exceeded
    if posAlongPath1 >= total1Length then
        posAlongPath1 = total1Length -- Clamp to the end of the path
        path1Completed = true    -- Mark path as completed
        sim.setObjectParent(cuboid, gripperBase, true)
        sim.setJointTargetPosition(rightfinger,0.08)
        sim.setJointTargetPosition(leftfinger, -0.08)
        sim.setObjectInt32Param(cuboid, sim.shapeintparam_static, 1)
        sim.setObjectInt32Param(cuboid, sim.shapeintparam_respondable, 0)
        startPath2 = true
    end
```

```
            -- Interpolate position and quaternion along the path
            local pos = sim.getPathInterpolatedConfig(path1Positions, path1Lengths, posAlongPath1)
            local quat = sim.getPathInterpolatedConfig(path1Quaternions, path1Lengths, posAlongPath1, nil, {2, 2, 2,
                  2})

            -- Set the position and quaternion of the object
            sim.setObjectPosition(objectToFollowPath, path1, pos)
            sim.setObjectQuaternion(objectToFollowPath, path1, quat)

        end
        previousSimulationTime = t -- Update previous time for next step delta calculation
    end
end

function sysCall_cleanup()

end
```

This Lua script is designed to control the movement of the target object along the first three predefined paths in sequence. The code is designed to begin once the object is detected on conveyor. When the sensor is triggered and the system is not already running, the script resets various state variables and sets the movement process into motion. The target then follows Path1 first. As it moves along the path, its position and orientation are updated in real time based on the current simulation time and a fixed movement velocity. When Path1 is completed, the cuboid is attached to the gripper, the gripper fingers close to grasp it, and the object becomes static and non-respondable in the simulation. This transition also initiates movement along Path 2 by setting corresponding flag.
Upon completing Path 2, the object is released from the gripper and made dynamic again. The gripper opens, and the object is no longer parented to the gripper. The script also restarts the conveyor belt. After this, movement continues along Path3, used to reset the position of the arm. Once Path3 is completed, the start flag is reset to make the arm ready for a new cycle.

4.4.4 Second robot target script

```
local objectToFollowPath = -1
local path1 = -1
local path2 = -1
local path3 = -1
local path1Positions = {}
local path1Quaternions = {}
local path2Positions = {}
local path2Quaternions = {}
local path3Positions = {}
local path3Quaternions = {}
local path1Lengths = {}
local path2Lengths = {}
local path3Lengths = {}
local total1Length = 0
local total2Length = 0
local total3Length = 0
local velocity = 0.08 -- m/s
local posAlongPath1 = 0
local posAlongPath2 = 0
local posAlongPath3 = 0
local previousSimulationTime = 0
local path1Completed = false
local path2Completed = false
local path3Completed = false
local startPath2 = false
local startPath3 = false
local cuboid
local gripperBase
local reachedPoint6 = false
local start = 0
```

```
function sysCall_init()
    sim = require('sim')
    objectToFollowPath = sim.getObject('/Base2/Target')
    path1 = sim.getObject('/Path4')
    path2 = sim.getObject('/Path5')
    path3 = sim.getObject('/Path6')
    sensor2 = sim.getObject('/sensor3')
    rightfinger = sim.getObject('/Base2/RevolutRightJoint')
    leftfinger = sim.getObject('/Base2/RevolutLeftJoint')
    gripperBase = sim.getObject('/Base2/Gripper')
    lastpointPath4 = sim.getObject('/Path4/ctrlPt[2]')
    firstpointPath5 = sim.getObject('/Path5/ctrlPt[0]')


    if objectToFollowPath == -1 or path1 == -1 or path2 == -1 or path3 == -1 then
        sim.addLog(sim.verbosity_errors, "Error: Could not find '/Target' or '/Path1' or '/Path2' or '/Path3'
            objects.")
        sim.pauseSimulation()
        return
    end

    local path1Data = sim.unpackDoubleTable(sim.getBufferProperty(path1, 'customData.PATH'))
    local path2Data = sim.unpackDoubleTable(sim.getBufferProperty(path2, 'customData.PATH'))
    local path3Data = sim.unpackDoubleTable(sim.getBufferProperty(path3, 'customData.PATH'))
    local m1 = Matrix(#path1Data // 7, 7, path1Data)
    local m2 = Matrix(#path2Data // 7, 7, path2Data)
    local m3 = Matrix(#path3Data // 7, 7, path3Data)
    path1Positions = m1:slice(1, 1, m1:rows(), 3):data()
    path1Quaternions = m1:slice(1, 4, m1:rows(), 7):data()
    path2Positions = m2:slice(1, 1, m2:rows(), 3):data()
    path2Quaternions = m2:slice(1, 4, m2:rows(), 7):data()
    path3Positions = m3:slice(1, 1, m3:rows(), 3):data()
    path3Quaternions = m3:slice(1, 4, m3:rows(), 7):data()

    path1Lengths, total1Length = sim.getPathLengths(path1Positions, 3)
    path2Lengths, total2Length = sim.getPathLengths(path2Positions, 3)
    path3Lengths, total3Length = sim.getPathLengths(path3Positions, 3)

    previousSimulationTime = 0 -- Force zero time delta at start

    startPath2 = false
    startPath3 = false
    path1Completed = false -- Ensure path is not marked complete at start
    path2Completed = false
    path3Completed = false
    start = 0
end

function sysCall_actuation()
    res2, dist2, point2, obj2, n2 = sim.readProximitySensor(sensor2)


    --Starts only if the cart is arrived
    if start == 0 and res2 == 1 then
        --Reset flags
        path3Completed = false
        path2Completed = false
        path1Completed = false
        startPath2 = false
        startPath3 = false
        previousSimulationTime = 0 -- Force zero time delta at start
        posAlongPath1 = 0
        posAlongPath2 = 0
        posAlongPath3 = 0
        cuboid = sim.getObject('/Cuboid')
        start = 1
        --Move last point of Path4 and first point of Path 5 to
        -- the position of the object
        objposition = sim.getObjectPosition(cuboid, sim.handle_world)
        objposition[3] = objposition[3] + 0.2
        sim.setObjectPosition(lastpointPath4,objposition,sim.handle_world)
        sim.setObjectPosition(firstpointPath5,objposition,sim.handle_world)
    end
```

```
if start == 1 then

    if previousSimulationTime == 0 then
        previousSimulationTime = sim.getSimulationTime()
        return
    end

    local t = sim.getSimulationTime()
    local deltaTime = t - previousSimulationTime

    -- If the path is already completed, do nothing more
    if path3Completed then
        start = 0
        return
    end

    if startPath2 == true then
        posAlongPath2 = posAlongPath2 + velocity * deltaTime

        -- Check if the end of the path has been reached or exceeded
        if posAlongPath2 >= total2Length then
            posAlongPath2 = total2Length -- Clamp to the end of the path
            path2Completed = true    -- Mark path as completed
            --Open gripper and leave object
            sim.setObjectParent(cuboid, -1, true)
            sim.setObjectInt32Param(cuboid, sim.shapeintparam_static, 0)
            sim.setObjectInt32Param(cuboid, sim.shapeintparam_respondable, 1)
            sim.setJointTargetPosition(rightfinger, 0.61)
            sim.setJointTargetPosition(leftfinger, -0.61)
            startPath2 = false
            startPath3 = true
            --Delete object
            sim.removeObjects({cuboid})
        end

        -- Interpolate position and quaternion along the path
        local pos = sim.getPathInterpolatedConfig(path2Positions, path2Lengths, posAlongPath2)
        local quat = sim.getPathInterpolatedConfig(path2Quaternions, path2Lengths, posAlongPath2, nil, {2, 2, 2,
              2})

        -- Set the position and quaternion of the object
        sim.setObjectPosition(objectToFollowPath, path2, pos)
        sim.setObjectQuaternion(objectToFollowPath, path2, quat)

    elseif startPath3 == true then
        posAlongPath3 = posAlongPath3 + velocity * deltaTime

        -- Check if the end of the path has been reached or exceeded
        if posAlongPath3 >= total3Length then
            posAlongPath3 = total3Length -- Clamp to the end of the path
            path3Completed = true    -- Mark path as completed
        end

        -- Interpolate position and quaternion along the path
        local pos = sim.getPathInterpolatedConfig(path3Positions, path3Lengths, posAlongPath3)
        local quat = sim.getPathInterpolatedConfig(path3Quaternions, path3Lengths, posAlongPath3, nil, {2, 2, 2,
              2})

        -- Set the position and quaternion of the object
        sim.setObjectPosition(objectToFollowPath, path3, pos)
        sim.setObjectQuaternion(objectToFollowPath, path3, quat)


    else
        posAlongPath1 = posAlongPath1 + velocity * deltaTime

        -- Check if the end of the path has been reached or exceeded
        if posAlongPath1 >= total1Length then
            posAlongPath1 = total1Length -- Clamp to the end of the path
            path1Completed = true    -- Mark path as completed
            sim.setObjectParent(cuboid, gripperBase, true)
            sim.setJointTargetPosition(rightfinger,0.08)
            sim.setJointTargetPosition(leftfinger, -0.08)
```

```
            sim.setObjectInt32Param(cuboid, sim.shapeintparam_static, 1)
            sim.setObjectInt32Param(cuboid, sim.shapeintparam_respondable, 0)
            startPath2 = true
        end

        -- Interpolate position and quaternion along the path
        local pos = sim.getPathInterpolatedConfig(path1Positions, path1Lengths, posAlongPath1)
        local quat = sim.getPathInterpolatedConfig(path1Quaternions, path1Lengths, posAlongPath1, nil, {2, 2, 2,
            2})

        -- Set the position and quaternion of the object
        sim.setObjectPosition(objectToFollowPath, path1, pos)
        sim.setObjectQuaternion(objectToFollowPath, path1, quat)

    end
    previousSimulationTime = t -- Update previous time for next step delta calculation
  end
end

function sysCall_cleanup()

end
```

The script for the target of the second robot is more or less identical to the first one, with exception of the name of the path objects and the fact that the last point of the first path and the first of the second one are moved accordingly to the position of the object carried by the cart. Moreover, in order to theoretically run the simulation for an undefined amount of time, every time the object is placed on the table, is also deleted. This permits us to avoid problems of too many objects accumulating.

4.5 Generator script

```
function sysCall_init()
    sim = require('sim')
    lastSpawnTime = sim.getSimulationTime()
    spawnInterval = 60.0 -- seconds
    cuboidCounter = 0
    createCuboid()
end

function sysCall_actuation()
    local currentTime = sim.getSimulationTime()
    if currentTime - lastSpawnTime >= spawnInterval then
        createCuboid()
        lastSpawnTime = currentTime
    end
end

function createCuboid()
    cuboidCounter = cuboidCounter + 1

    -- size
    local sx = 0.04
    local sy = 0.04
    local sz = 0.04

    -- position
    local pos = {0.2,-0.2,0.15}

    -- Create cuboid
    local options = 0 -- backface culling, smooth shading off
    local cuboidHandle = sim.createPrimitiveShape(sim.primitiveshape_cuboid, {sx, sy, sz}, options)

    -- Set position
    sim.setObjectPosition(cuboidHandle, -1, pos)

    -- Make dynamic and respondable
    sim.setObjectInt32Param(cuboidHandle, sim.shapeintparam_static, 0)
```

```
    sim.setObjectInt32Param(cuboidHandle, sim.shapeintparam_respondable, 1)

    -- Set random color
    sim.setShapeColor(cuboidHandle, nil, sim.colorcomponent_ambient_diffuse, {
        math.random(), math.random(), math.random()
    })

end
```

This Lua script is used to generate a cuboid object every 60 "seconds" . The function *createPrimitiveShape* is used to create the object with dimensions specified by *sx*, *sy* and *sz*. The object is placed at the position specified by the *pos* variable using *setObjectPosition* function. The object is then made dynamic and respondable using *setObjectInt32Param* and a random color is set.

5 Encountered Issues

5.1 Impossibility of reading encoders values

5.1.1 Issue

Despite previous projects used 0.4ms as sampling time for digital inputs, connected to HSC, this value showed not to be sufficient to correctly read encoders.

5.1.2 Solution

Use 0.8ms for the four encoder inputs.

5.2 Impossibility of creating data structure for PLC TCP connection

5.2.1 Issue

From previous it has been seen that TCON block was connected to a "TCON_IP_v4" database, but this kind of data was not available when creating a new database.

5.2.2 Solution

After creating the TCON block, click on it then Properties->Connection Parameters and set correct parameters. After that the database to connect to TCON block will appear in program resources.

5.3 Impossibility of using CoppeliaSim inverse kinematics problems

5.3.1 Issue

All online guides, and previous projects used a different and old way to build inverse kinematics chain and then apply the solver.

5.3.2 Solution

The solution is explained in the guide provided within the chapter "CoppeliaSim".

5.4 CoppeliaSim inverse kinematics errors

5.4.1 Issue

After setting inverse kinematics solver the tool continued to print the *stepstoobig*, *limithit* and *notwithintolerance* errors.

5.4.2 Solution

The first two problems have been solved by increasing *max. iterations*, the second one has been solved by refining joints limits.

6 Future works

The presented project represents a starting point for new possible experiments that could put more effort on the integration between real and simulated environments or that can better tune some parameters. Some possible future works includes:

- Add confidence interval in encoders readings to ensure a more accurate measure.
- Focus on adding a battery package to the cart to supply the Arduino board, sensors and motors.
- Tune joints limits in CoppeliaSim to respect real robot constraints.
- Better focus gripper functions in order to handle the object without the necessity of deactivate its physics parameters.
- Use the data of joints positions computed with inverse kinematics to control the real robot.