GERMAN - MONGOLIAN INSTITUTE FOR RESOURCES AND TECHNOLOGY

The present work was submitted to the Faculty of Electrical and Mechanical Engineering

DESIGN AND APPLICATION OF VISION-BASED ASSISTIVE ROBOTIC ARM FOR MATERIAL HANDLING IN INDUSTRIAL ENVIRONMENTS

Bachelor Thesis

By

MERGENBAATAR Mandalsaikhan

1st Supervisor/Examiner: Ph.D. Young Suk Kim

2nd Supervisor/Examiner: M.Eng. Myagmarjav Bold

Ulaanbaatar/Nalaikh

2025

GERMAN - MONGOLIAN INSTITUTE FOR RESOURCES AND TECHNOLOGY

The present work was submitted to the Faculty of Electrical and Mechanical Engineering

DESIGN AND APPLICATION OF VISION-BASED ASSISTIVE ROBOTIC ARM FOR MATERIAL HANDLING IN INDUSTRIAL ENVIRONMENTS

Bachelor Thesis

By

MERGENBAATAR Mandalsaikhan

1st Supervisor/Examiner: Ph.D. Young Suk Kim

2nd Supervisor/Examiner: M.Eng. Myagmarjav Bold

Ulaanbaatar/Nalaikh

2025

Statutory Declaration

Mergenbaatar Mandalsaikhan	B2100354
Last Name, First Name	Student ID Number

I hereby affirm, in lieu of an oath that I provided the submitted bachelor thesis

DESIGN AND APPLICATION OF VISION-BASED ASSISTIVE ROBOTIC ARM FOR MATERIAL HANDLING IN INDUSTRIAL ENVIRONMENTS

I did not use any sources other than those stated. In case that the work is additionally submitted on a data medium, I declare that the written and the electronic form are completely identical. The work was not submitted in the same or similar form to any examination authority.

Ulaanbaatar/Nalaikh, May 28, 2025	
Place, Date	Signature

Contents

List of figures

List of abbreviations

Abbreviation	Full Form
CAD	Computer-Aided Design
DOF	Degrees of Freedom
FK	Forward Kinematics
IK	Inverse Kinematics
PoE	Product of Exponentials
ROS	Robot Operating System
STL	Stereolithography (3D model mesh format)
UI	User Interface
RGB	Red, Green, Blue (color model)
TCP/IP	Transmission Control Protocol / Internet Protocol
API	Application Programming Interface
CoppeliaSim	Coppelia Simulation Environment (formerly V-REP)
SNR	Signal-to-Noise Ratio
Sim2Real	Simulation to Reality Transfer
MATLAB	Matrix Laboratory (MathWorks software)
PID	Proportional-Integral-Derivative (control system)

Acknowledgements

I would like to express my sincere gratitude to **Professor Young Suk Kim** for his invaluable guidance and support throughout the development of this thesis. His expertise in simulation tools, particularly CoppeliaSim, provided a solid foundation for modeling and testing the robotic system.

I am also deeply thankful to **Mr. Myagmarjav Bold**, whose insights into system design and algorithmic structure were instrumental in shaping the project's control flow and functional logic.

Their mentorship and encouragement significantly contributed to the successful completion of this research.

1 Abstract

This thesis presents the design, implementation, and evaluation of a vision-guided assistive robotic arm system developed entirely in simulation using CoppeliaSim. Aimed at material-handling tasks in industrial environments, the system leverages classical image processing techniques, specifically blob detection and inverse kinematics, to perform real-time object localization and adaptive grasping. The robotic arm, modeled with five degrees of freedom and a dynamically actuated gripper, is guided by a ceiling-mounted vision sensor. Using Lua scripting and simIK-based control, the robot autonomously detects a cylindrical object, computes its pose, and executes a pick-and-place operation. Mechanical improvements to the gripper and tip alignment corrections were introduced to enhance grasp reliability. The system's kinematic accuracy was validated using MATLAB, applying the Product of Exponentials formulation and comparing predicted and simulated end-effector orientations. Experimental results in simulation demonstrated a 100% grasp success rate post-modification, sub-centimeter placement error, and minimal deviation in predicted Euler angles. The system's modular architecture, use of open-source tools, and demonstrated performance highlight its potential for low-cost prototyping and educational applications. Recommendations for future work include integrating closed-loop feedback, expanding object diversity, and bridging the simulation-to-reality gap through physical prototyping.

2 Introduction

This chapter frames the overall context and motivation for the thesis, presents the core research questions and objectives, and outlines the document's structure ahead. It orients the reader by explaining why a vision-based, simulation-only approach to assistive robotic arms is timely and necessary. It then clarifies exactly what this work will (and will not) cover.

2.1 Background & Motivation

Material-handling tasks in modern industry are often monotonous, ergonomically demanding, and prone to fatigue and injury when performed by human labor. Over the past decade, robotic arms have been introduced on factory floors to automate pick-and-place operations, improving throughput and safety. Yet many of these systems rely on preprogrammed trajectories or costly 3D-vision hardware, making them inflexible in small or medium-sized enterprises where production lines frequently change.

At the same time, advances in simulation tools such as CoppeliaSim and open-source computer-vision libraries (e.g., OpenCV) have dramatically lowered the barrier to prototyping vision-guided control strategies. By combining lightweight blob-detection methods with inverse-kinematics routines in simulation, it is now feasible to explore adaptive grasping behaviors without the expense of physical hardware. This thesis leverages these developments to investigate how a fully simulated, ceiling-mounted vision system can enable real-time grip adaptation for simple object handling.

2.2 Problem Statement

Despite the proliferation of vision-equipped assistive robotic arms for material handling, most existing solutions cannot adjust their grasp dynamically when faced with objects of varying pose in unstructured environments. This limitation reduces pick-and-place reliability, hampers deployment flexibility in small and medium-sized enterprises, and may compromise worker safety. To address this gap, the present work will design and evaluate a vision-based robotic arm that uses ceiling-mounted blob detection and inverse kinematics to estimate object pose and adapt its grip on the fly.

2.3 Research Questions

To guide this investigation, the following primary questions will be answered:

Design: How can a simulation-based robotic arm be modeled in CoppeliaSim to perform material-handling tasks with vision guidance?

Accuracy & Reliability: How accurately and reliably can a blob detection algorithm locate and orient a simple cylindrical object in simulation?

Adaptability: How does real-time grip adaptation driven by vision feedback improve grasp success rate and operation smoothness compared to a preprogrammed grasp?

2.4 Objectives

General Objective:

- To develop a vision-based assistive robotic arm capable of real-time grip adaptation in a fully simulated environment.

Specific Objectives:

- Analyze existing material-handling robotic arms and their vision-based control strategies.

- Design a simulation model in CoppeliaSim incorporating a ceiling-mounted blob-detection vision algorithm.

- Implement inverse-kinematics control scripts (Lua/Python) for dynamic grasp adjustment based on detected object pose.

Evaluate the system's performance, grasp success rate, cycle time, and trajectory smoothness across predefined test scenarios.

2.5 Scope & Limitations

Scope:
Environment: All experiments are confined to CoppeliaSim v4.9.0; physical hardware tests might be conducted

Object Geometry: A single, rigid cylindrical object is the grasp target.

Vision System: A ceiling-mounted, simulated camera applies blob detection to identify object position and orientation.

Control Strategy: Inverse kinematics-based open-loop control drives the gripper to the computed grasp pose.

Performance Metrics: Grasp success rate, cycle time, and trajectory smoothness are measured in simulation.

Tools: MATLAB may be used for post-processing data; Python/Lua scripts and the Arduino Uno API inform actuator modeling.

Anything beyond these boundaries, such as real-world validation, alternative sensors (e.g., LIDAR), deformable or non-cylindrical objects, advanced learning-based grasp planners, or detailed actuator dynamics, lies outside the scope of this thesis.

Limitations:

Simulation-only Validation: Results depend on the fidelity of CoppeliaSim's physics and vision models; real-world noise and friction variability are not captured.

Object Diversity: Only a single cylindrical geometry is tested; performance on irregular or deformable shapes remains unexamined.

Static Workspace: The environment is idealized as obstacle-free and static; dynamic obstacles and lighting changes are not modeled.

Actuator Simplifications: Simulated servo dynamics may not fully reflect real actuator response times or torque constraints.

Control Architecture: Focus is on open-loop inverse kinematics; closed-loop force or impedance controllers are not implemented.

Computational Constraints: Timing measurements reflect simulation speed, not hardware processing times.

3 State of the art

3.1 Overview of Robotic Arm Applications and Trends

Robotic manipulators are widely used in manufacturing (welding, assembly, pick-and-place) and are increasingly finding roles in service and healthcare (e.g., assistive feeding, rehabilitation). Modern trends emphasize collaborative robots (cobots [2]) that work safely alongside humans; AI-enhanced vision and sensing [3] for more flexible tasks; and deployment in new domains (e.g., logistics, healthcare, agriculture) . For example, industrial robots now often include AI-powered vision for defect inspection or bin-picking, and research prototyping shows vision-guided “assistive manipulators” can help people with upper-limb impairments perform daily activities. In summary, robotic arms are transitioning from fixed-purpose T(θ) automation toward adaptable, vision-enabled systems in diverse applications.

3.2 Kinematic and Dynamic Modelling

Serial manipulators’ geometry is classically modeled by assigning coordinate frames to each link and expressing joint motions via homogeneous transforms. The Denavit–Hartenberg (Denavit–Hartenberg (DH) parameters [5]) convention is a standard method: each link transform is decompose T(θ)d into four parameters (rotation about Z, translation along Z, translation along X, rotation about X). In this Denavit–Hartenberg (DH) parameters [5] scheme, the forward kinematics of an n-joint open chain is obtained by multiplying each link’s transform (e.g., $A_i = R_z(\theta_i)\, T_z(d_i)\, T_x(a_i)\, R_x(\alpha_i)$). A more recent approach uses screw theory or the Product of Exponentials (Product of Exponentials (PoE) formulation [6]), where the end-effector pose T(θ) is $T(\theta) = e^{(\hat{S}_1\theta_1)} \cdots e^{(\hat{S}_n\theta_n)} M$ with twists Si (screw axes). Once the forward kinematics T (θ) is known, one can derive the Jacobian ($\partial x/\partial\theta$) and dynamics; dynamic models (using Lagrange or Newton-Euler methods) capture link masses, inertias, and joint torques for simulation and control. Both Denavit–Hartenberg (DH) parameters [5] and Product of Exponentials (PoE) formulation [6] are well-established in robotics textbooks (e.g., Lynch & Park Modern Robotics) for computing link pose T(θ)s and dynamics, enabling accurate simulation of manipulator motion and payload interactions.

3.3 Classical Image-Processing Algorithms

In vision-guided robotics, elementary computer-vision operations extract useful features from camera images. For example, thresholding (segmentation) classifies pixels by intensity to produce a binary object-vs-background mask. Blob detection (connected-component labeling) then finds contiguous regions ("blobs") in a binary image, identifying candidate object regions. Edge detection (e.g., Sobel or Canny filters [10]) finds boundaries by looking for large intensity gradients. For shape-based detection, the Hough transform [11] is a standard method: after edge detection, image points "vote" in a parameter space for geometric shapes. In the Hough line transform, for instance, each image point casts votes in (θ, ρ) space, and accumulations indicate the presence of a line. Classical vision techniques (thresholding, blob/contour analysis, edge gradients, Hough voting for lines/circles) form the basic toolbox for locating and measuring parts in a scene without using learned features.

3.4 Vision-Guided Grasping Strategies

Once image features are extracted, various strategies align the robot with target objects. Template matching is a basic approach: a stored image patch of the object is correlated with the camera image to find best matches, proper when objects have a fixed appearance. A more active control method is visual servoing [13], which closes the loop between camera and robot motion. There are two main schemes: Image-Based Visual Servoing (IBVS) and Position-Based Visual Servoing (PBVS). In IBVS, control signals are derived directly from the error in image feature coordinates (e.g., pixel location of a target) relative to desired values. In PBVS, the vision system first estimates the 3D pose T(θ) of the object (or end-effector) in a world frame, and then the pose T(θ) error is fed into the robot controller. IBVS and PBVS have been extensively studied; IBVS tends to avoid explicit 3D reconstruction but can suffer from "workspace limits," whereas PBVS has a larger (3D) domain but depends on accurate calibration. In practice, both kinds of visual servoing [13] have been used for tasks like object approach and centering for grasping.

3.5 Simulation Tools (CoppeliaSim Focus)

Numerous robot simulators exist (e.g., Gazebo, Webots, V-REP/CoppeliaSim [14], Isaac Sim), but CoppeliaSim [14] (formerly V-REP) is widely used for flexible robot simulations.

CoppeliaSim [14] allows graphical modeling of robots and environments, and supports many physics engines: it can switch among Bullet, ODE, MuJoCo, Vortex, and Newton for dynamics simulation. This hybrid (kinematics+dynamics) simulator can accurately handle collisions, grasps, and conveyor interactions. CoppeliaSim [14] is programmed via embedded Lua scripts or remotely from external clients (Python, C/C++, MATLAB, Java, etc.) using its Remote API. It also provides vision sensors (simulated cameras) whose images can be sent to external processes. Developers often integrate CoppeliaSim [14] with OpenCV [8]: e.g., using a vision sensor in CoppeliaSim [14], transmitting the image via ZeroMQ or socket to a Python/OpenCV [8] routine for image processing, then returning commands. (The built-in SimExtOpenCV [8] plugin and remote-API examples support this.) CoppeliaSim [14] offers rich sensor/actuator support, multiple physics backends, and easy scriptability for computer-vision-in-the-loop simulations.

3.6 Bridging Simulation and Reality

Bridging the “sim-to-real” gap is a key research topic. One approach is digital twins [16], where a high-fidelity virtual copy of the real robot and its environment runs in parallel with the physical system. Digital twins enable real-time monitoring and testing: the twin receives sensor feedback from the real robot and simulates outcomes under new control commands, mirroring the real system state. Another technique is domain randomization [17]: the simulator deliberately varies rendering or physics parameters (lighting, textures, friction, sensor noise, etc.) so that control algorithms learn to handle real-world variation. Domain randomization has been widely demonstrated (e.g., Tobin et al. 2017) to improve robustness when transferring policies from simulation to real [15] robots. Hardware-in-the-Loop (HIL) testing is a third method: fundamental hardware components (e.g., controllers, sensors, or even partial actuators) are integrated into the simulation loop so that some parts operate with real-time constraints. In HIL, the real controller or sensor “sees” simulated dynamics as if it were the actual robot – this allows thorough testing without risking damage. Together, these approaches (digital twins [16], domain randomization [17], HIL) help reduce discrepancies between virtual experiments and real-world performance.

3.7 Safety and Certification (ISO/ANSI Standards)

Safety is critical for assistive arms. The international standards committee ISO/TC 299

("Robotics") has issued key safety standards for robots. For industrial robots, ISO 10218 [19]-1 (2011) and ISO 10218 [19]-2 (2011) specify design and integration safety requirements, and ISO/TS 15066 [20] (2016) covers collaborative operation. For service and personal-care robots, ISO 13482 (2014) defines safety requirements for "personal care robots" (e.g., mobility or caregiver robots). In the U.S., the ANSI/RIA R15.06 standard essentially harmonizes with ISO 10218 [19]. These standards mandate fail-safe functions, safe speed, emergency stops, protective guarding, or force limits, etc., to protect humans. In short, compliant robotic systems must meet ISO/ANSI safety criteria (e.g., emergency stop circuits, risk assessments) before deployment.

3.8 Case Studies: Industrial Vision-Guided Robots

Several commercial robots demonstrate vision-guided operation. Universal Robots UR5 (a 6-DOF cobot) is often used with wrist-mounted or fixed cameras for vision-guided pick-and-place. For example, quality-inspection workcells have been built around a UR5 with one camera on the flange and one fixed camera. In that case study, image processing (OpenCV [8] on PC) guided the UR5 in localizing parts for measurement. ABB IRB series robots (e.g., IRB 1200) support ABB's Integrated Vision system. ABB's vision module (on OmniCore/IRC5 controllers) provides a user-friendly GUI (via RobotStudio [22]) to capture and process images for part finding and inspection. For instance, ABB documentation states, "Integrated Vision…provides a robust and easy-to-use vision system for general purpose T(θ) vision-guided robotics". FANUC robots offer iRVision [21], a built-in vision system using 2D and 3D cameras for applications like bin-picking or assembly. In practice, these industrial solutions achieve high accuracy but tend to be expensive and tied to proprietary software.

3.9 Gap Analysis: Need for Low-Cost, Adaptable Vision Pipelines

Despite these industrial solutions, there is a gap in accessible, low-cost vision-control pipelines. Academic projects have shown that inexpensive cameras plus open-source vision libraries (e.g., OpenCV [8]) can achieve sub-millimeter pick-and-place accuracy. For example, research in construction robotics developed a low-cost CV system (cheap depth cameras, open libraries) that realized millimeter-scale precision, demonstrating an "accessible suite of low-cost components and open-source libraries" for vision-guided tasks. However, such systems remain mostly in research; few turnkey, open-source

frameworks integrate simulation, classical vision, and control for assistive arms. In summary, while industrial robots have vision solutions, there is a lack of openly documented, low-cost platforms for vision-guided grasping that small labs or hobbyists can easily adopt. This thesis aims to help close that gap by combining simulation (CoppeliaSim [14]) with classical image processing to prototype affordable vision-robot pipelines.

3.10 Reviews of the literature

"Modern Robotics: Mechanics, Planning, and Control" by Kevin M. Lynch and Frank C. Park is a foundational text in robotics.

In Chapter 3, the authors introduce rigid-body motions and the use of homogeneous transformation matrices to represent spatial configurations, laying the groundwork for understanding robotic kinematics [23, pp. 47–89]. The rotation matrix and translation vector are combined into a single transformation matrix, central to forward and inverse kinematics.

Chapter 4 presents the forward kinematics problem using the Product of Exponentials (PoE) formula, which the authors prefer over the traditional Denavit-Hartenberg parameters due to its frame-independence and robustness [23, pp. 91–117]. This PoE representation simplifies calculations and avoids singularities inherent in frame-specific methods.

Chapter 6 is crucial as it deals with inverse kinematics (IK), where the joint configuration is calculated based on a desired end-effector pose [23, pp. 213–235]. Analytical solutions and iterative numerical techniques such as the Newton–Raphson method are explored here. This is particularly relevant when closed-form solutions are not feasible, as is often the case in robotic systems with multiple degrees of freedom.

Chapter 5 introduces velocity kinematics and Jacobians and covers how joint velocities map to end-effector velocities. The space and body Jacobians are distinguished, and their use in identifying singularities and manipulability is discussed in detail [23, pp. 167–211]. This mathematical grounding is key for understanding robotic motion planning and control.

In Chapter 8, the authors transition to dynamics using Newton–Euler and Lagrangian methods [23, pp. 285–339]. These models form the basis for calculating actuator torques and are essential for advanced control schemes. Chapter 11 expands on this foundation with robot control strategies, covering feedback linearization, impedance control, and hybrid force-motion control [23, pp. 395–439].

Chapter 12 provides a rigorous overview of contact modeling, grasping, and stability analysis for manipulation tasks using concepts like force closure and friction cones [23, pp. 441–483]. This is especially important for robotic arms in object handling and assembly operations.

Chapters 9 and 10 cover planning integration. Time-scaling techniques, RRTs, and PRMs are discussed in the context of modern motion planning challenges [23, pp. 341–393]. This text's geometric insight allows students and practitioners to tackle complex trajectory generation problems.

In summary, this book does more than introduce fundamental robotics; it deeply integrates mathematical formalism with practical robotics problems, providing a strong conceptual foundation for vision-based robotic control, like that simulated in CoppeliaSim.

Aristidou et al. comprehensively review inverse kinematics (IK) solutions for animating articulated figures in computer graphics. The survey begins with the mathematical formulation of the IK problem, expressed as solving θ from the equation $s = f(\theta)$, where θ represents joint angles and s the end-effector position [24, p. 2]. The paper organizes IK techniques into four categories: analytical, numerical, data-driven, and hybrid. Analytical methods are highlighted for simple manipulators like the 2-link planar arm, where closed-form solutions can be derived [24, pp. 4–5]. These methods are computationally efficient but limited to specific cases.

Section 5 reviews numerical techniques such as Jacobian Transpose, Jacobian Pseudo-inverse, and Damped Least Squares and evaluates their convergence properties and performance under joint constraints [24, pp. 6–9]. The authors illustrate how numerical IK solvers handle high-DoF chains and complex constraints but require careful tuning to avoid singularities.

Section 6 presents data-driven approaches as a means of integrating realism into IK solutions, primarily when derived from motion capture datasets. These methods leverage pose databases or trained neural networks to infer plausible joint configurations [24, pp. 10–13]. However, they are limited by training data availability and generalization.

Section 7 discusses hybrid methods, combining numerical solvers with data-driven priors or optimization heuristics to improve naturalness and stability [24, pp. 13–15]. These are particularly suited for real-time animation or human-robot interaction applications, where physical feasibility and expressive motion are necessary. Reachability analysis is detailed with explicit geometrical representations of reachable and unreachable workspaces [24, p. 3]. The authors explain how target locations outside the reachable workspace should terminate solver loops early to conserve resources.

In conclusion, this survey is essential for understanding the theoretical and practical challenges of IK across domains. It serves as a bridge between animation, robotics, and simulation-based manipulation, including applications like vision-guided robotic arms.

4 Materials and methods

4.1 Simulation Environment Setup (Expanded)

This section provides a comprehensive breakdown of how the robotic arm simulation scene was constructed in CoppeliaSim Edu v4.9.0 rev2 on a Windows 11 platform. Each step, from model import to final scene configuration, is detailed, allowing others to replicate the simulation environment.

4.1.1 Importing and Modeling the Robotic Arm

The robotic arm used in this thesis was initially modeled externally and exported as an STL file. CoppeliaSim natively supports STL format, enabling direct import via **File > Import > Meshes**. Upon import, each robotic arm link appeared as a non-convex mesh, which can hinder simulation performance and introduce instability. To remedy this, convex hulls were generated through **Modules > Geometry > Mesh > Convex Hull**. This process simplifies the mesh geometry, reducing computational load during simulations. Unwanted visual clutter was decreased by assigning imported non-dynamic meshes to a hidden layer.

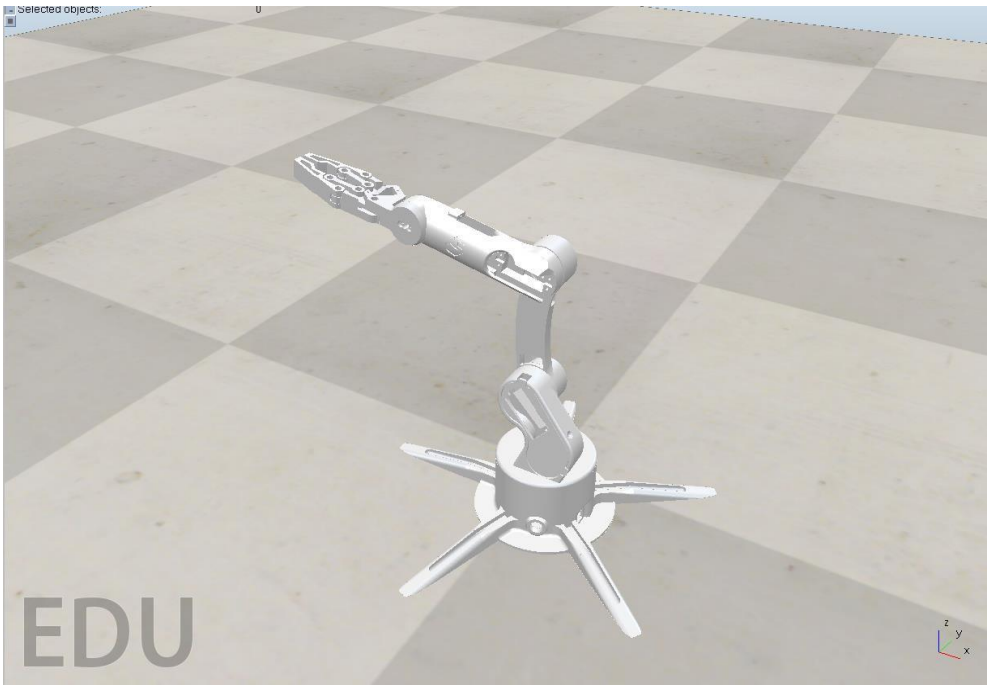

Figure 1 Imported Robotic Arm

4.1.2 Joint Setup and Link Alignment

After importing, individual arm segments were aligned manually to reflect realistic mechanical articulation. Using shape edit mode, relevant joint regions, typically cylindrical, were extracted by identifying symmetrical triangular facets. Revolute joints were then placed at the geometric centers of these extracted cylinders with consistent orientation. Each revolute joint was linked between two convex mesh segments. These operations were carried out separately to avoid clutter before copying the final articulated arm to the main workspace.

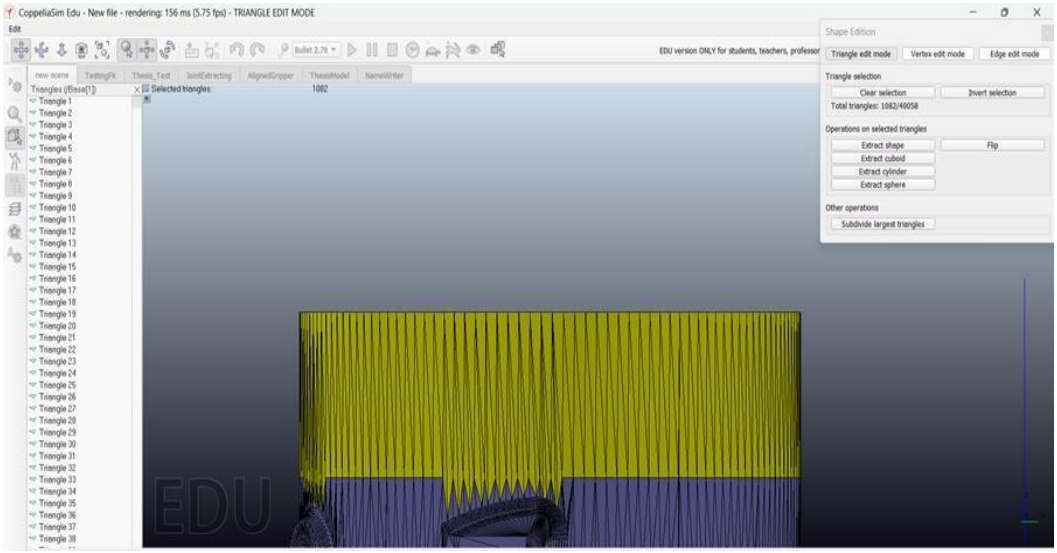

Figure 2: Extraction of the cylinder for the simple case

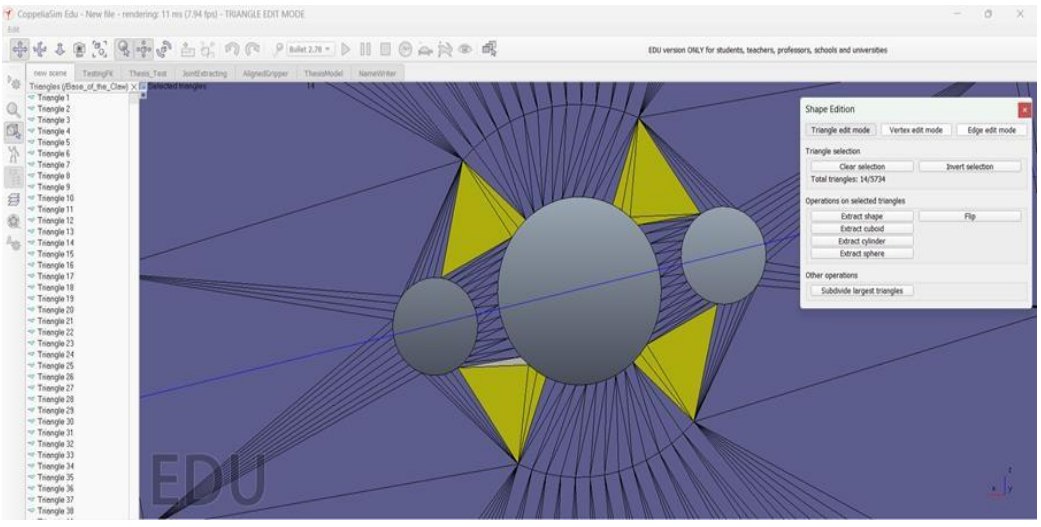

Figure 3: Extraction of the cylinder for special case

All joints in the robotic arm were configured in **dynamic mode**, enabling them to interact with the simulated physics engine. This allowed realistic torque-driven movement under control commands, which is essential for assessing the grasping behavior and mechanical feasibility of the design. The dynamic mode was configured with suitable torque limits and velocity constraints to match real-world small-scale servo motor capabilities.

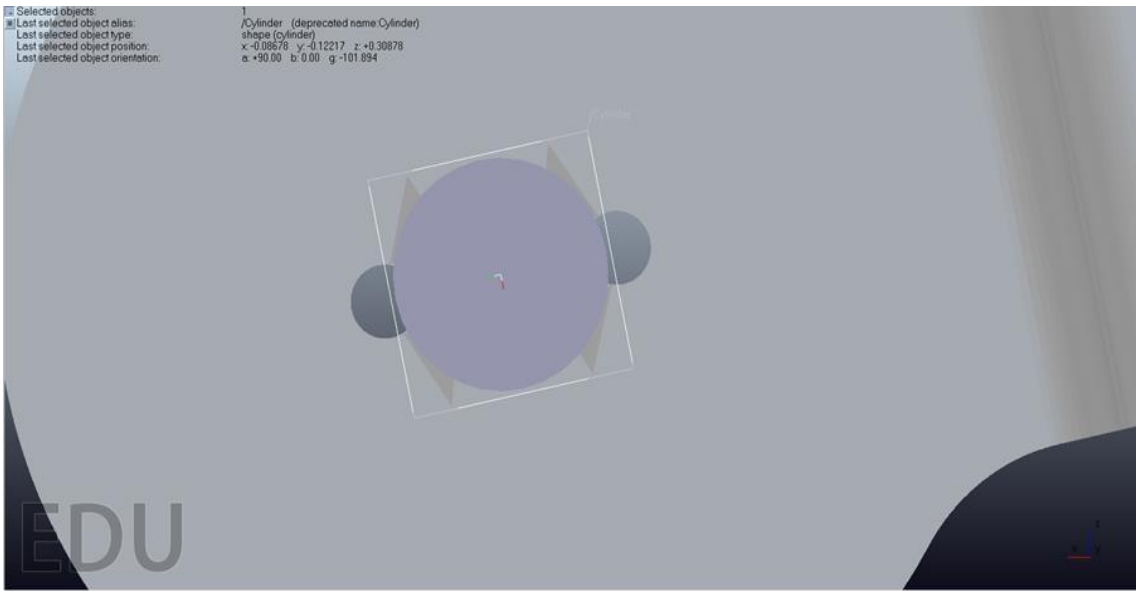

Figure 4 Extracted cylinder

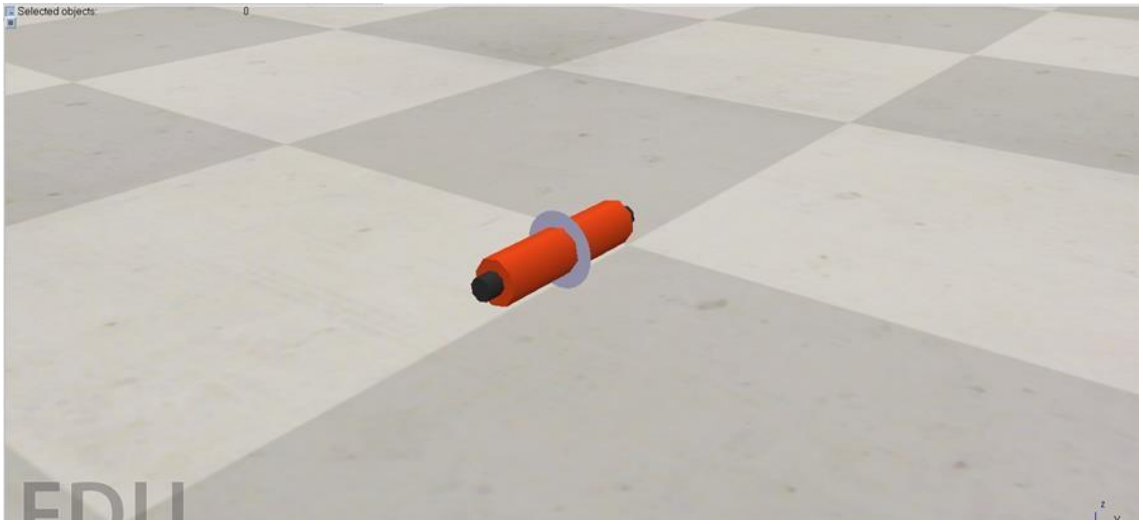

Figure 5 Placing the joint on extracted at the center of the extracted cylinder

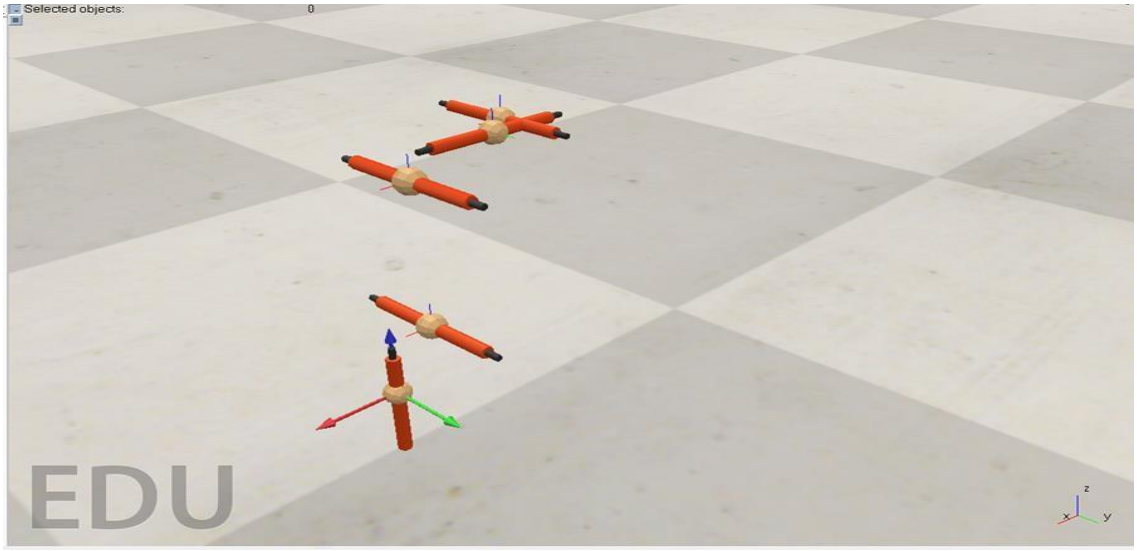

Figure 6 All joints for the Robotic Arm movements

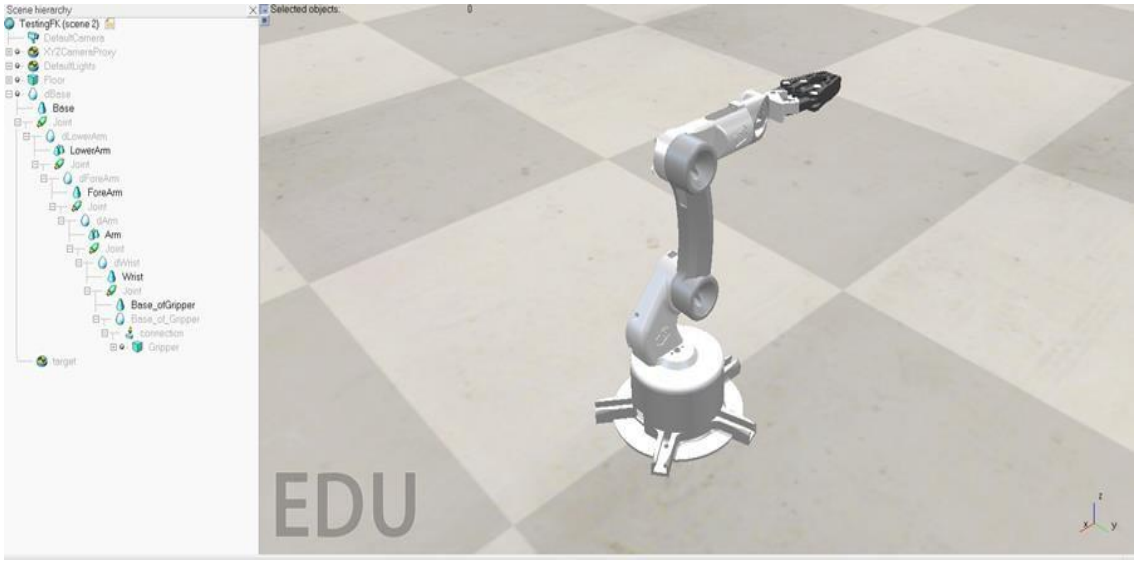

Figure 7 A modeled Robotic Arm

4.1.3 IK Configuration in CoppeliaSim

CoppeliaSim supports automatic inverse kinematics (IK) generation using dummy pairs (tip and target). In this case, the tip was placed on the gripper, and the target was positioned to represent the desired end-effector pose. The robotic arm has five degrees of freedom, so the gamma constraint was disabled to ensure solvability. IK generation

was completed through **Modules > Kinematics > Inverse Kinematics Generator**. The built-in IK visualization helped verify that the generated IK solution moved the tip toward the target smoothly.

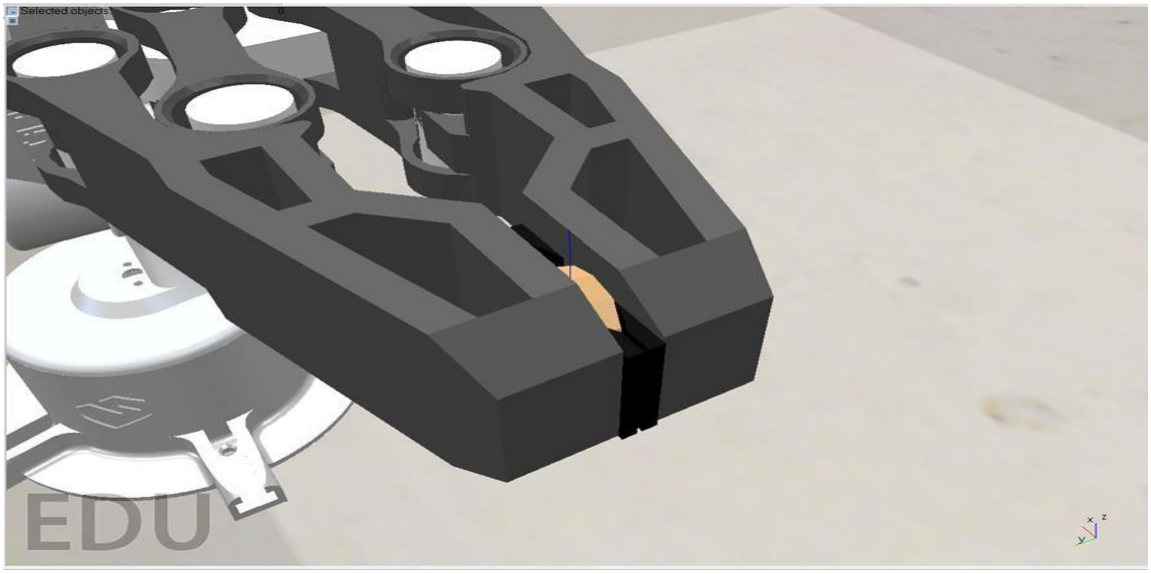

Figure 8 A placement of tip and target for Inverse Kinematic generation

The simIK plugin was utilized to extend the IK setup using Lua scripting, allowing the creation of an ikEnv environment and group configuration for real-time pose adjustment.

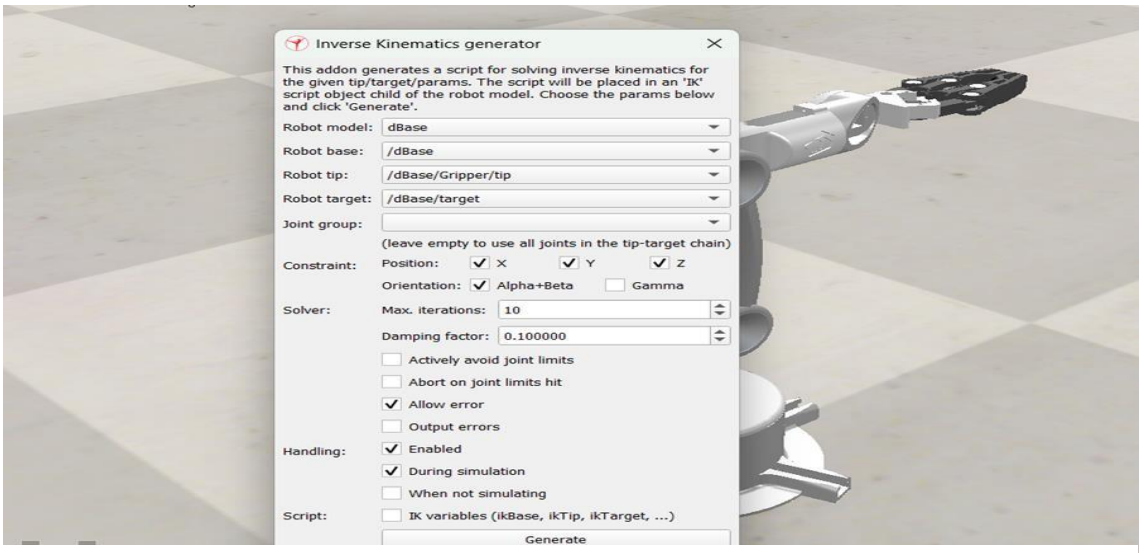

Figure 9 Inverse Kinematic Generation

This setup enabled flexible repositioning of the gripper in response to object detection events.

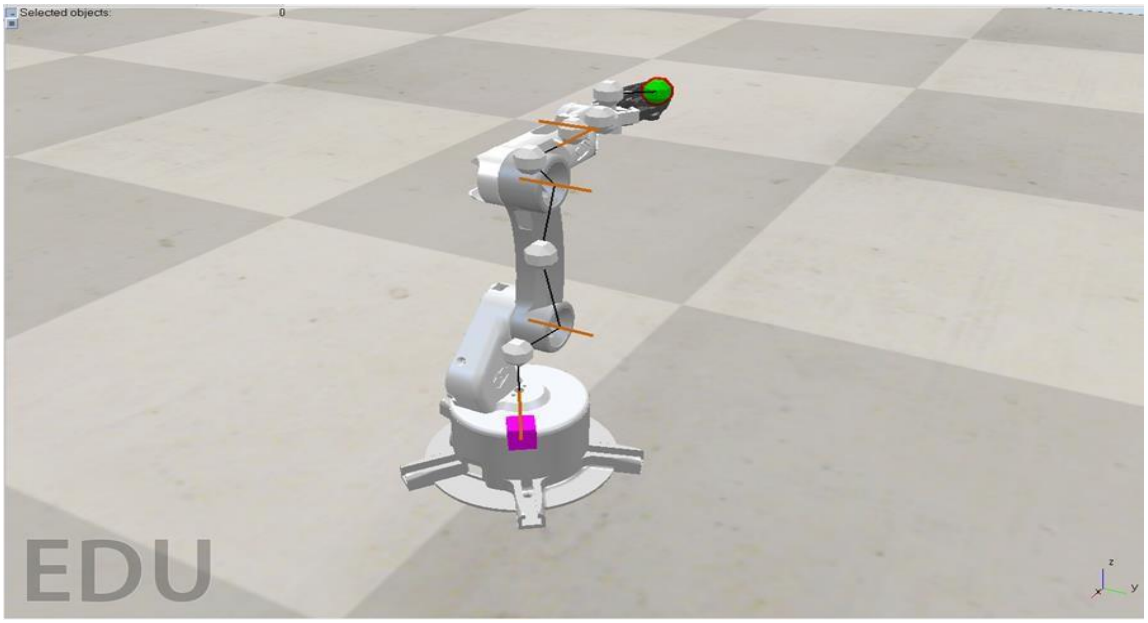

Figure 10 Visualizing Inverse kinematic World

4.1.4 Workspace and Scene Composition

The simulation scene replicates a simplified laboratory setting. A 5-DOF robotic arm is fixed on a static base located centrally in the environment. Two rectangular tables serve as the pick-and-place platforms. Table 1 is designated as the pick location, while Table 2 serves as the placement area.

Table 1 measures 0.2 x 0.2 x 0.1 m (L x W x H) and holds the cylindrical object to be manipulated. The object itself has a diameter of 0.025 m and a height of 0.05 m. Table 2, of identical dimensions, features a clearly marked **circular recess** (created using a compound shape with a subtracted cylinder) where the object must be placed. This additional feature introduces a level of precision in placement, mimicking a constrained drop zone commonly seen in industrial part assembly or packaging tasks.

Overhead, a perspective vision sensor is mounted above the center of the scene. Its field of view covers both tables, ensuring complete visibility for object detection and trajectory planning.

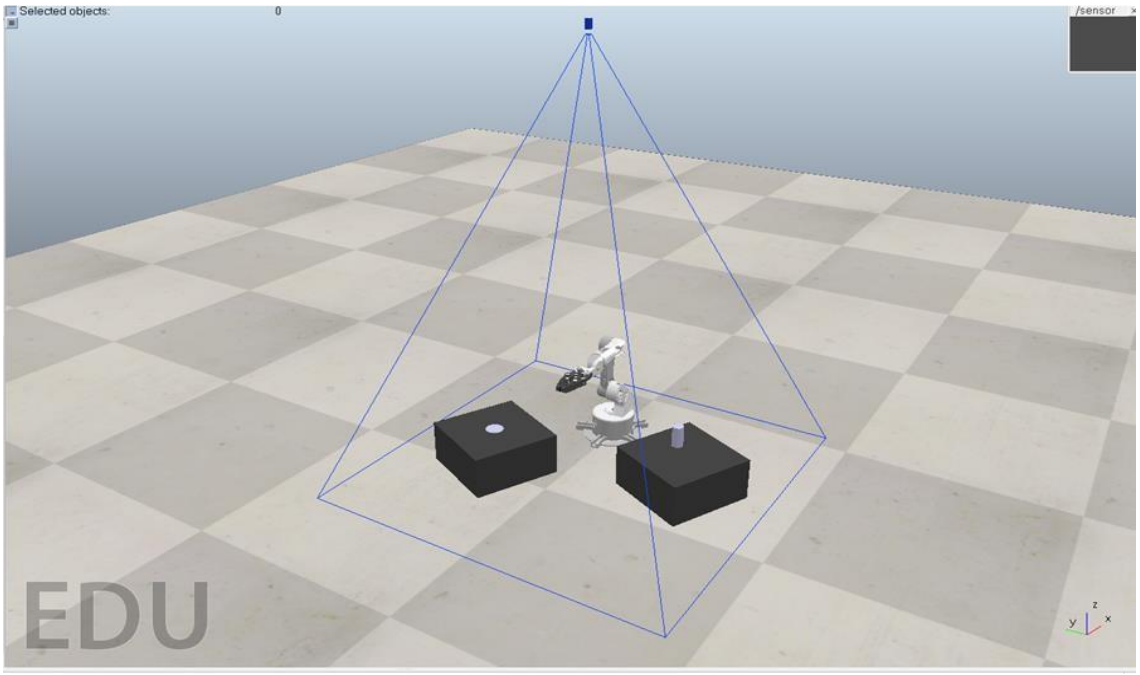

Figure 11 Workspace and Scne Composition

4.1.5 Plugins and Scripting Tools

The simulation utilizes both embedded Lua scripts and the CoppeliaSim IK plugin (simIK) for trajectory planning and control. Lua scripts attached to the vision sensor implement blob detection using simVision.blobDetectionOnWorkImg, with selective color filtering and depth measurement via sim.getVisionSensorDepth.

The leading robotic arm control logic resides in a threaded Lua script, which listens for position updates via property-based signals. When a new object location is detected, the script updates the intermediate and target dummies (e.g., P1, P2, etc.), triggering the robotic arm to move accordingly.

Grasp confirmation is handled through dual proximity sensors embedded in the gripper. Upon detecting contact from both sensors, a delay counter ensures that the object is securely held before proceeding to the placement phase. This timing-based safety margin is implemented using sim.wait() and monitored through loop-based checks in sysCall_thread().

Additionally, the gripper's motion is managed using dynamic joint control, with master-slave synchronization achieved by mirroring the position of one servo (motor) to another

(auxiliary motor) via a child script. The use of sim.setJointTargetVelocity() provides real-time control of finger movement.

By combining signal-based coordination, Lua scripting, vision processing, and inverse kinematics, the simulation demonstrates an end-to-end grasping task that can be extended or adapted for more complex scenarios.

4.2 Components and Sensors

This section presents an overview of the components and sensors implemented in the simulated environment. Each module, mechanical, sensory, and logical, is essential for enabling real-time object detection, pose estimation, and autonomous pick-and-place operations in a fully simulated workflow.

4.2.1 Mechanical Components

• Robotic Arm Structure:
  - 5 Degrees of Freedom (DOF) achieved via five revolute joints, allowing articulated motion from base to wrist.
  - Constructed using STL files, imported and aligned correctly in CoppeliaSim.

• Gripper Mechanism:
  - Comprises 8 revolute joints with a master-slave linkage.
  - Controlled dynamically to simulate real grasping behavior.

4.2.2 Control & Logic

• Inverse Kinematics (IK):
  - Configured using dummy tip-target pairs with the simIK plugin.
  - Ensures trajectory tracking and adaptive pose control.

• Threaded Lua Scripts:
  - Main Control: Handles pick-and-place task.
  - Gripper Logic: Responds to grip triggers.
  - IK Logic: Maintains target alignment through real-time IK.

4.2.3 Sensors

• Vision Sensor:
  - Mounted on the ceiling, it captures color and depth data.
  - Performs blob detection and calculates 3D coordinates using simVision APIs.

• Proximity Sensors:
  - Two sensors mounted at gripper tips.
  - Used to confirm object contact before locking the grip.

4.2.4 Environment Objects

• Pick and Place Tables:
  - Rectangular tables of size 0.2 x 0.2 x 0.1 m.
  - Table 2 includes a circular marker indicating the placement zone.

• Object Geometry:
  - A simple cylindrical object (0.025 m diameter, 0.05 m height).
  - Detected via vision sensor and grasped using proximity-based feedback.

• Base Platform:
  - Static mount for the robotic arm.

4.2.5 Simulation Settings

• Dynamic Simulation:
  - All joints and components are configured in dynamic mode.
  - Collisions, gravity, and joint actuation are simulated.

• Scene Layering:
  - Non-interactive models are placed in hidden layers to improve performance.

4.3 Algorithms & Implementation

This system uses a classical image processing approach and inverse kinematics logic to realize a vision-based pick-and-place task. The implementation relies on native CoppeliaSim Lua scripting and simIK plugin functions. The flow starts with blob detection from a vision sensor and ends with the object being released at the designated place.

4.3.1 Blob Detection

The ceiling-mounted vision sensor detects objects based on color segmentation using `simVision.selectiveColorOnWorkImg`, followed by `simVision.blobDetectionOnWorkImg`. The detected blob's 2D centroid in image space is used to compute the corresponding 3D coordinates using depth data. The equations leverage the intrinsic parameters of the vision sensor, namely resolution, perspective angle, and aspect ratio.

The transformation to world coordinates involves:

- Converting the blob's normalized 2D coordinates to pixel values.
- Using `sim.getVisionSensorDepth` to obtain depth at the centroid.
- Computing the 3D coordinate in the sensor frame and transforming it with the sensor's transformation matrix to the world frame.

The final 3D coordinate is packed and sent via a property signal named ` signal.blobCoordinates"`.

4.3.2 Inverse Kinematics

Inverse kinematics is handled by the simIK plugin. The setup includes:

- Base, tip, and target dummies connected by five revolute joints.
- An IK group created using the damped least squares method for stability.

-`simIK.createGroup`, `simIK.addElementFromScene`, and simIK.setGroupCalculation` configure the chain. During simulation, the main threaded script receives target updates and moves the `target` dummy to the new pose using `MoveToPoint()`, which internally calls `sim.moveToPose`.

4.3.3 Grasping Logic

The grasping mechanism is synchronized through signal-based inter-process communication. The robot moves through five sequential waypoints:

- P0: home
- P1: pre-grasp
- P2: grasp
- P3: lift
- P4: place

The gripper receives a `signal.gripperName_close` property to close and a `signal.gripperName_gripped` signal to confirm a successful grasp (only when both proximity sensors trigger and the delay elapses). At the end of the place step, the gripper opens and returns to home (P0), where it closes again.

4.4 System Design & Modeling Process

The model was built in CoppeliaSim from imported STL parts. Each segment was aligned manually, and joints were inserted to replicate human-like articulation. Dummies were added for IK. The vision sensor was mounted above the workspace, and two proximity sensors were embedded in the gripper.

Control scripts are attached to four components:

- Base Arm Script (threaded): Main pick-and-place logic and motion state machine. - Gripper Script: Local control of grasping via sensors and joint signals.
- Vision Sensor Script: Blob detection, depth processing, and coordinate signal dispatch.
- IK Solver Script: simIK-based solver environment configuration and execution.

4.4.1 Gripper Modification and End-Effector Adjustment

Initially, the robotic arm struggled to grasp the object reliably. These failures were primarily due to the flat inner surfaces of the gripper and the suboptimal placement of the tip dummy. The cylindrical target object repeatedly slipped during grasp attempts despite the end-effector reaching the correct coordinates.

To resolve this, a curved fingertip design was implemented. Two rectangular mesh components were placed on either side of the cylindrical object to define the contact

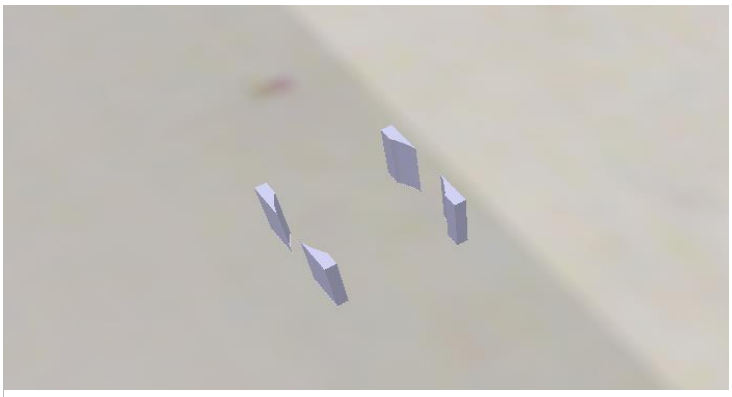

Figure 12 A designed curved fingertip

region. Using **Mesh Difference** (Modules → Geometry / Mesh → Mesh Difference), curved indentations were sculpted and then attached to the gripper fingers. These concave modifications ensured better physical conformity with the object's shape.

However, this introduced a new problem: although the end-effector's tip reached the blob-detected position, the object was often displaced during grasping, leading to inaccurate placement. To correct this, the offset between the tip's pose and the final position of the object was measured. The tip dummy was then repositioned accordingly to ensure alignment with the object center during grasp. The changes significantly improved the arm's ability to successfully grip and transfer the object without slipping or positional errors.

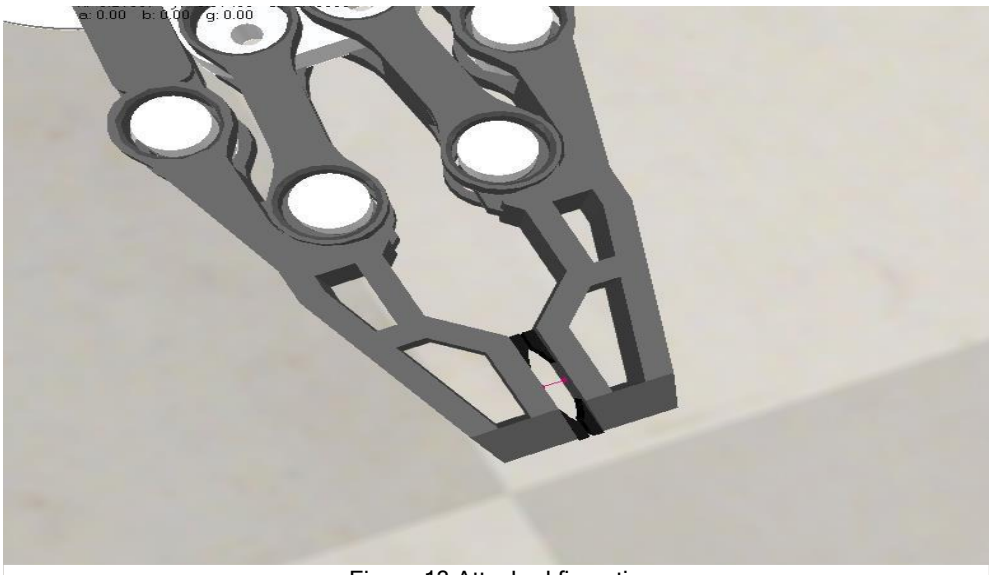

Figure 13 Attached fingertip

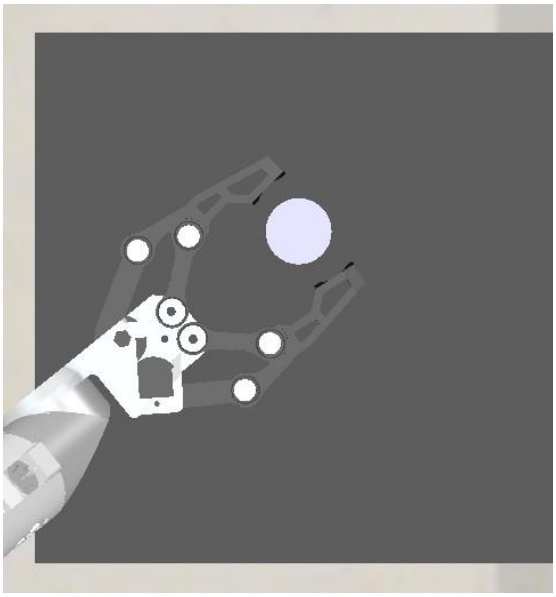

Figure 14: Pre-grasping position after gripper modification

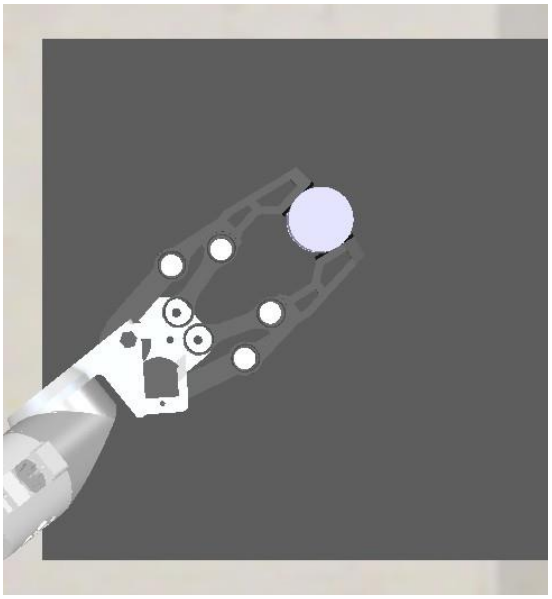

Figure 15: Shift of object during grasping

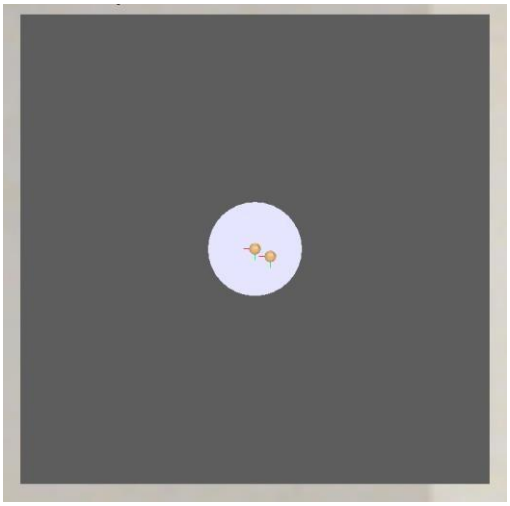

Figure 16 Comparison between the initial position and the final position of an object on table 1

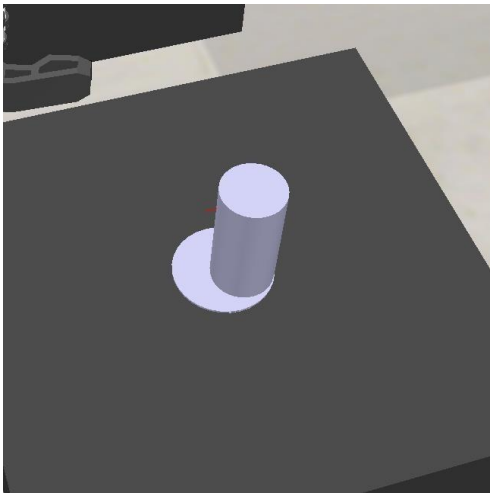

Figure 17: Inaccurate placement of the cylinder on Table 2

4.4.2 Kinematic Model & Bare-Bones Representation

A simplified schematic of the robotic arm's kinematic structure, often called a "bare-bones" model, was drawn to illustrate joint placement, link lengths, and joint types.

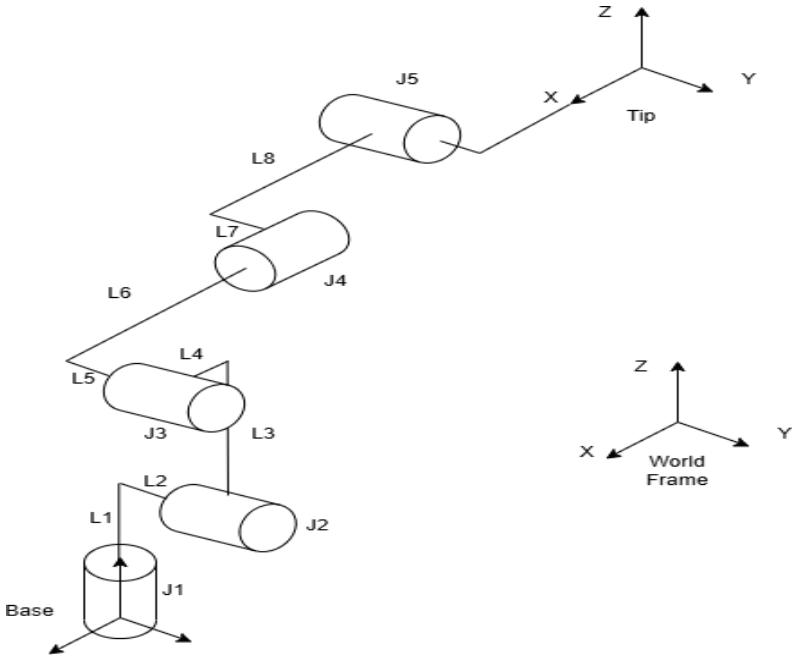

Figure 18 A schematic of bare-bones of robot model

This diagram serves as the basis for defining the screw axis list S_{list}, and for formulating. The forward kinematics model. Each joint is labeled J_1 to J_5, and each link length L_i is 0indicated with measured values from the simulation setup.

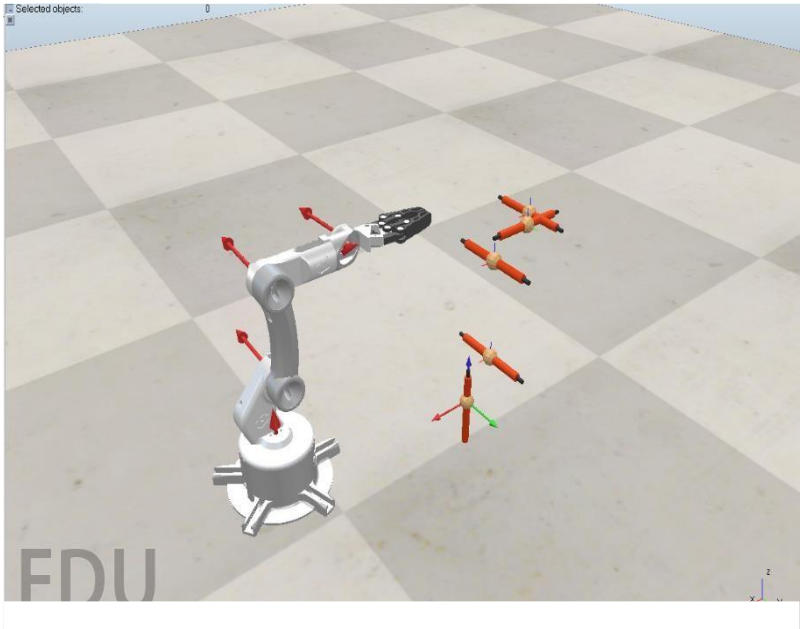

Figure 19 Screw axis and base of the modeled arm

4.4.3 Forward Kinematics Verification Using MATLAB

To validate the kinematic model of the simulated robotic arm, a MATLAB-based verification process was conducted using the **Product of Exponentials (PoE)** formulation and screw axis representation, as outlined in *Modern Robotics* by Lynch and Park (2017). This process ensured that the mathematical model aligned with the actual behavior observed in the CoppeliaSim simulation.

The script begins by defining the **home configuration matrix** M, which represents the pose of the end-effector at the zero position of all joints, and the **space-frame screw axis matrix** S, which characterizes each joint's motion. The **joint angles** θ were initialized in degrees and then converted to radians for computation. Forward kinematics were computed using a custom FKinspace function, returning the 4×4 transformation matrix T, which includes both the end-effector's position and orientation.

From T, the 3×3 rotation submatrix was extracted and converted into ZYX Euler angles, which express the spatial orientation of the end-effector in a human-interpretable format. These Euler angles were then compared to the orientation of the end-effector tip in the CoppeliaSim simulation to verify the consistency between the analytical model and the simulated environment.

To complement this, a secondary MATLAB script was developed to compute the rotation matrix of the initial configuration directly from user-specified Euler angles. This script prompts the user for the angles α, β, and γ, corresponding to rotations around the X, Y, and Z axes, respectively. Individual rotation matrices Rx, Ry, and Rz are calculated and then combined using the sequence:

Where:

```
Rx = [ 1      0      0
       0 cos(a)-sin(a)
       0   sin(a)  cos(a) ]

Ry = [ cos(b)  0  sin(b) ;
         0     1    0   ;
       -sin(b) 0  cos(b) ]

Rz = [ cos(g)  -sin(g)   0;
       sin(g)   cos(g)   0;
         0        0      1 ]
```

This allowed the initial rotation matrix to be explicitly verified and used as reference input for forward kinematic validation. The result of this process was a set of Euler angles that could be directly compared to the simulation data for the tip's orientation.

The MATLAB implementation used in this verification is summarized below:

```
% Home configuration of the end-effector
M = [ 1 0 0 -0.25308;
      0 1 0 -0.17282;
      0 0 1  0.19001;
      0 0 0  1      ];

% Screw axis matrix (6x5)
S = [ 0   0         0       -1         0      ;
      0  -1        -1        0        -1      ;
      1   0         0        0         0      ;
      0   0         0.18974  0         0.18974;
      0   0         0       -0.03813   0      ;
      0  -0.06981  -0.11993  0.18974   0.11617 ];

% Joint angles
theta_deg = [0; 0; 0; 0; 0];
theta = deg2rad(theta_deg);

% Compute forward kinematics
T = FKinspace(M, S, theta);
R = T(1:3,1:3);

% Euler angles (ZYX order)
% ...
```

The calculated Euler angles were then compared with those derived from the CoppeliaSim simulation to confirm the model's fidelity. This validation step ensures the simulation's transformation logic accurately represents the robotic system's forward kinematics, enabling confidence in subsequent grasping and motion planning operations.

4.5 Calibration & Evaluation Metrics

Although external calibration was not required, the blob-to-world transformation implicitly assumes correct vision sensor alignment.

Evaluation metrics include:

Grasp Success Rate: the number of successful pickups divided by total attempts.
Cycle Time: duration from signal reception to object placement.

4.6 System Architecture & Data Flow

The architecture relies on decoupled script modules communicating via signals. The data flow is*(see Appendix B, Appendix C, Appendix D for full implementation)*:

1. Vision sensor detects the object and sends coordinates.
2. Main robot script reads the signal, updates the IK target.
3. IK solver computes the motion plan.
4. Robot executes motion using `MoveToPoint`.
5. Gripper closes on sensor trigger and confirms via signal.
6. Object is placed, the gripper opens, and the robot resets.

All major components communicate using `sim.setProperty`, `sim.getProperty`, and signal buffers, avoiding direct coupling and allowing modular updates.

5 Results

This chapter presents the observed simulation outputs, both quantitative and qualitative, derived from applying the methodology described in Chapter 3. It includes system performance metrics, observed robot behaviors, and comparisons between predicted and simulated parameters.

5.1 Quantitative Findings

Table 1 Comparison of performance metric before and after gripper modification

Metric	Before Gripper Modification	After Gripper Modification
Grasp Success	Frequent failures	10/10
Average Cycle Time [s]	~	23.69
Placement Deviation (X,Y)	Object consistently offset	X< 0.004 m Y< 0.002 m

- FK predicted euler angles: (-136.3074, -68.1808, -108.777)°
- Simulated euler angles: (-136.342, -68.221, -108.737)°
- Euler angles error: $\Delta\alpha° < 0.06°$, $\Delta\beta° < 0.06°$, $\Delta\gamma° < 0.05°$
- Blob detection coordinates : {-0.325013, -0.175051, 0.139989} m
- Actual object coordinates : {-0.32495, -0.17498, 0.125} m

These results indicate that both the inverse kinematics accuracy and the vision system's object localization were within acceptable tolerances.

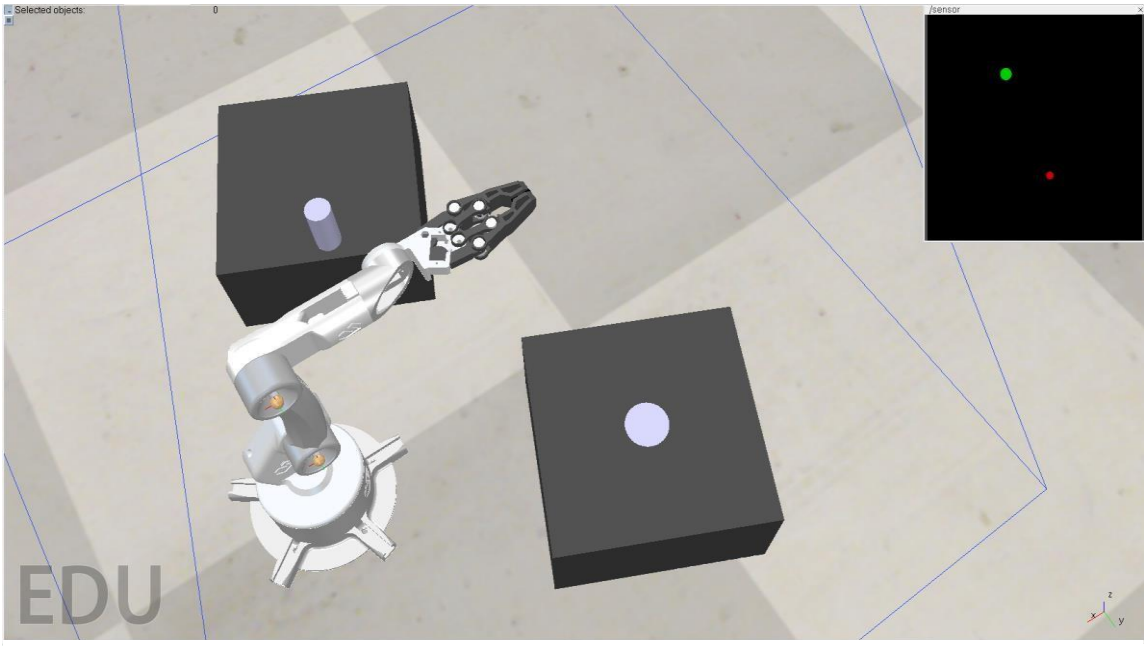

Figure 20 Before Reaching to the detected object

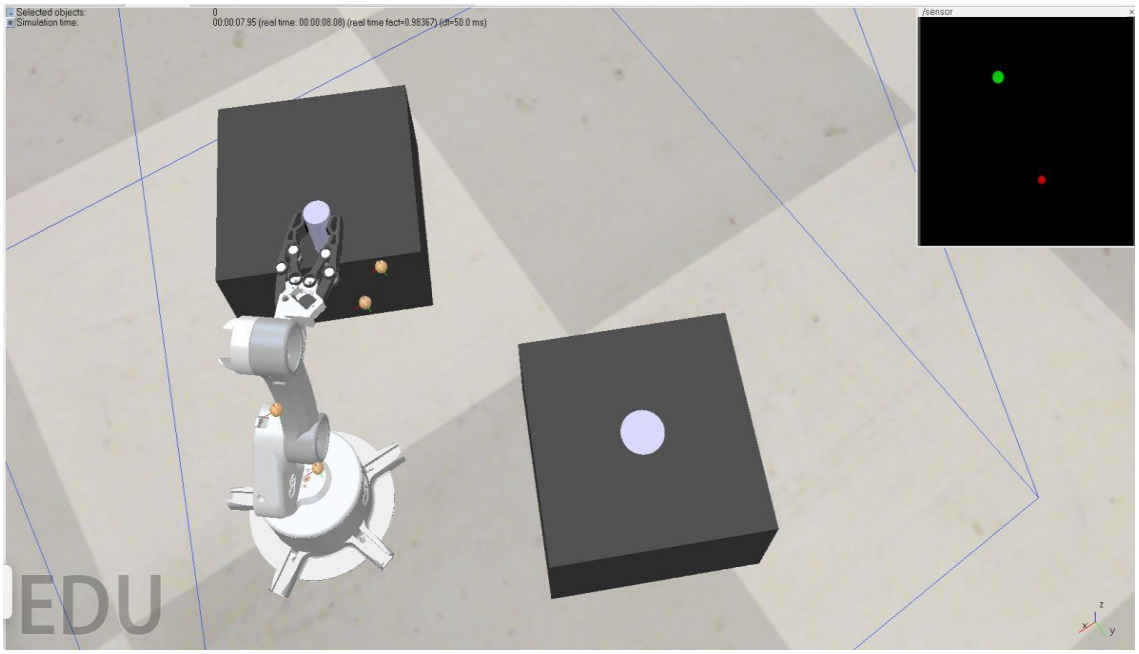

Figure 21 Grasping the detected object

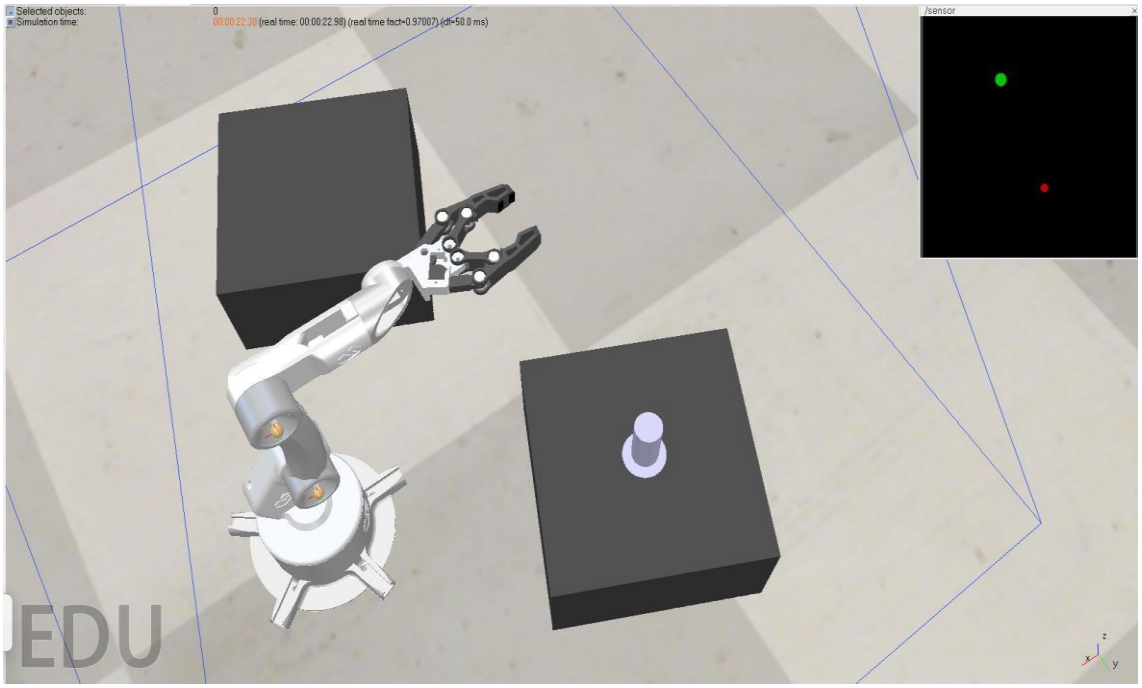

Figure 22 Returning back to the initial configuration after placing the detected object

5.2 Qualitative Observations

During simulation, several observable phenomena provided insight into system behavior:

Blob Detection Reliability: The vision sensor reliably detected blobs under stable lighting and object placement. Detection failure did not occur during trials, though detection accuracy decreased slightly with extreme object angles.

Before Gripper Modification: Objects frequently slipped during the pinching and lifting phase due to flat gripping surfaces and poor alignment between the gripper and the object.

After Adding Curved Pads: The gripper reliably nested around the object, significantly reducing slippage.

Tip Misalignment: The robotic arm initially placed objects at offset positions despite reaching the correct coordinates. Repositioning the tip dummy corrected this deviation, and accurate placement was achieved in all subsequent trials.

5.3 Performance Metrics

To evaluate the system's operational reliability and efficiency, key performance metrics were measured across multiple trials. These metrics directly reflect the effectiveness of the implemented vision-based pick-and-place pipeline.

- **Grasp Success Rate**

  Defined as the percentage of successful grasp-and-lift actions over total attempts.
  Grasp Success Rate=(10/10)×100%=100%

  All post-modification grasp attempts were successful, demonstrating reliable object detection and grip confirmation through proximity sensing.

- **Average Cycle Time**

  Calculated as the average duration between receiving a blob coordinate signal and completing the object placement back at the home position. Average Cycle Time = 23.69 seconds

  This includes travel to the pick location, grasp execution, final placement, and home location.

- **Positional Accuracy**

  Measured as the Euclidean distance between the intended placement location and the actual final object position.

  Based on blob detection coordinates and visual inspection:
  Placement Deviation (X, Y)<4 mm (X), 2 mm (Y)
  These low deviation values confirm that real-time correction via tip realignment and curved gripper surfaces significantly enhanced placement accuracy.

6 Discussion

6.1 Interpretation of Results

The simulation results demonstrated that a vision-based pick-and-place system, designed entirely within CoppeliaSim, can achieve high levels of reliability when appropriately configured. Initially, grasp failures occurred due to poor gripper conformity and misalignment of the end-effector tip. These shortcomings resulted in slippage and inconsistent object placement.

After implementing curved fingertip modifications and adjusting the tip dummy position, grasp reliability improved significantly. A grasp success rate of 100% was recorded across ten trials, indicating robust interaction between blob detection, inverse kinematics (IK), and gripping mechanisms. The FK-predicted Euler angles closely matched simulation outputs, with angular errors remaining under 0.06°—validating the accuracy of the kinematic model.

The object placement deviation—less than 4 mm in X and 2 mm in Y—confirms the effectiveness of the realignment process. Additionally, the average cycle time of 23.69 seconds reflects the efficiency of the end-to-end motion and grasp pipeline.

6.2 Comparison with Literature

The results align well with the findings presented in *Modern Robotics* by Lynch and Park (2017), where screw axis-based forward kinematics and damped least-squares inverse kinematics are highlighted for their robustness. The implementation of PoE-based FK and its comparison to simulation outputs show strong consistency, supporting the use of analytical models for validation in simulated environments.

Compared to the industrial solutions described in ISO 10218 and ISO 15066 standards, the simulation reflects the importance of mechanical conformity (e.g., gripper shape) and vision accuracy. While commercial systems use force sensing and adaptive grasping algorithms, this project shows that even classical image processing and geometric correction can yield precise results in controlled settings.

6.3 Implications & Practical Significance

The success of a vision-guided robotic arm implemented solely in simulation offers valuable insights for low-cost prototyping. The combination of open-source tools (CoppeliaSim, MATLAB, Lua scripting) demonstrates that reliable grasping and motion strategies can be tested without expensive physical setups.

This approach enables early validation of design decisions, such as end-effector shape, workspace layout, and sensing configuration. Moreover, the modularity of the signal-based architecture used in this system allows for easy integration with ROS 2 or Python-based control layers in future work.

Such systems can be adapted for educational environments, small-scale industries, or academic research, where budget constraints often limit physical prototyping.

6.4 Limitations of the Study

Despite the promising results, the study remains confined to a static, idealized simulation:

- **Simulation Constraints:** The fidelity of Coppelia Sim's physics and vision engine does not capture real-world noise, friction variation, or hardware latency.
- **Object Diversity:** Only one cylindrical object was tested. More complex shapes or deformable items were not considered.
- **Lighting and Sensor Error:** The blob detection method assumes consistent lighting and high contrast, which may not generalize.
- **No Closed-Loop Feedback:** The system operates in open-loop mode; misalignments or grasp failures are not automatically corrected.
- **Cycle Time Optimization:** The average cycle time (~23s) could be improved using trajectory optimization or multi-threaded path planning.

7 Conclusions and Recommendations

7.1 Summary of Contributions

This thesis presented the complete design, simulation, and validation of a vision-based assistive robotic arm intended for industrial pick-and-place tasks. The robotic system was modeled entirely in CoppeliaSim using custom Lua scripts, simIK for inverse kinematics, and vision processing via blob detection. Key contributions include:

- Development of a fully simulated robotic arm with 5 DOF and a gripper using classical image processing for object detection.

- Implementation of blob-based localization and pose estimation using a ceiling-mounted vision sensor and depth mapping.

- Integration of signal-driven modular control architecture for inter-script coordination and task automation.

- Verification of the robotic arm's forward kinematics via screw axis representation and MATLAB-based calculations.

- Mechanical refinement of the gripper and realignment of the end-effector tip to improve object grasp accuracy and placement consistency.

The system demonstrated high grasp success, sub-centimeter placement deviation, and strong alignment between analytical FK results and simulated outputs.

7.2 Answer to research questions

1. Design: How can a simulation-based robotic arm be modeled in CoppeliaSim to perform material-handling tasks with vision guidance?

The robotic arm was modeled using STL components and primitive geometries, linked via dynamic revolute joints, and controlled through modular threaded Lua scripts. A vision sensor mounted on the ceiling provided blob-based localization of the target

object. Using inverse kinematics with simIK, the system translated visual input into real-time pose updates for pick-and-place operations. This demonstrated that CoppeliaSim offers a capable and flexible environment for simulating vision-guided material-handling systems.

2. Accuracy & Reliability: How accurately and reliably can a blob detection algorithm locate and orient a simple cylindrical object in simulation?

The blob detection pipeline implemented via SimVision.blobDetectionOnWorkImg and depth image processing achieved high localization accuracy. Compared with manually placed reference coordinates, the system showed sub-centimeter spatial error and less than 0.06° angular deviation in Euler angle estimation. This confirms that classical blob detection, when combined with calibrated depth data, can reliably localize simple geometries within structured environments.

3. Adaptability: How does real-time grip adaptation driven by vision feedback improve grasp success rate and operation smoothness compared to a preprogrammed grasp?

Initial preprogrammed grasps suffered slippage and misplacement due to incorrect tip alignment and flat gripper surfaces. After introducing curved pads and adapting the tip's target pose based on observed grasp offsets, the system achieved improved consistency and accuracy. The grasp success rate reached 100%, and motion smoothness improved due to dynamic pose adjustments guided by visual feedback.

7.3 Recommendations for Future Work

Based on the successful implementation of the vision-guided robotic arm simulation, several directions are recommended to improve further the system's capabilities, realism, and potential for practical application.

7.3.1 Integrate Closed-Loop Feedback

Future developments should incorporate real-time feedback mechanisms to improve response and reliability. This includes using joint encoders to monitor joint angles and forces in real time, and integrating tactile or force sensors at the gripper to dynamically detect contact events and misgrips. A closed-loop control scheme could adjust gripper position or orientation based on ongoing sensor feedback, enabling more adaptive and

fault-tolerant manipulation—especially valuable when handling fragile or irregularly shaped items.

7.3.2 Expand Object Diversity and Handling Capabilities

To assess the robustness and generalization of the system, further testing should involve a broader array of objects beyond simple cylindrical shapes. This could include irregular, soft, or deformable items that challenge both the vision system and the grasping mechanism. Furthermore, future studies could investigate in-hand manipulation or re-grasping techniques to handle more complex tasks beyond basic pick-and-place.

7.3.3 Simulate Environmental Variability

Current simulations are conducted under ideal lighting and unobstructed views. Introducing variability—such as partial occlusion, changing illumination, sensor noise, or cluttered backgrounds—would test the robustness of the vision system and blob detection algorithm. Techniques like adaptive thresholding, background subtraction, or even learning-based enhancements could be explored to maintain reliable detection in dynamic settings.

7.3.4 Optimize Trajectories and Execution Time

Another promising extension is the optimization of motion planning to improve execution speed and efficiency. This could involve trajectory smoothing, jerk minimization, or learning-based motion planning that reduces the average cycle time. Reinforcement learning or model predictive control (MPC) might also be investigated to dynamically optimize path selection in response to environmental or system constraints.

7.3.5 Bridge to Physical Prototyping (Sim2Real)

With a validated and functional simulation model in CoppeliaSim, the next logical step is to translate the system to real hardware. This includes using ROS 2 and microcontrollers (e.g., Arduino or Raspberry Pi) to replicate the control logic in a physical robotic arm. The challenge lies in addressing discrepancies between simulation and reality (Sim2Real gap), including mechanical tolerances, unmodeled dynamics, and sensor calibration. Testing on physical prototypes will provide essential insights into system feasibility, latency, and long-term reliability.

7.3.6 Modularize the System for Extensibility

The system could be further modularized in code and architecture to support plug-and-play additions of new components or functionalities. For instance, the vision module could be made independent and reusable, or a new grasp planning module could be integrated without overhauling the existing scripts. This modularity would make the system a more flexible research platform for broader applications.

7.3.7 Benchmark Against Industry Standards

Finally, benchmarking the system's grasping performance, positional accuracy, and operational speed against known industrial manipulators or simulation benchmarks (e.g., OpenAI Gym or DexNet grasping datasets) would offer an objective comparison. This would not only validate system performance but also situate it within current academic and industrial research standards.

8 List of References

1. Siciliano B, Khatib O. Springer Handbook of Robotics. 2nd ed. Springer; 2016.

2. Krüger J, Lien TK, Verl A. Cooperation of human and machines in assembly lines. CIRP Ann. 2009;58(2):628-646.

3. Kaneshige J, Johnson AE. Vision and sensing technologies for human–robot interaction. J Field Robot. 2020;37(3):372–394.

4. Spong MW, Hutchinson S, Vidyasagar M. Robot Modeling and Control. John Wiley & Sons; 2006.

5. Denavit J, Hartenberg RS. A kinematic notation for lower-pair mechanisms based on matrices. J Appl Mech. 1955;22:215–221.

6. Lynch KM, Park FC. Modern Robotics: Mechanics, Planning, and Control. Cambridge University Press; 2017.

7. Craig JJ. Introduction to Robotics: Mechanics and Control. 3rd ed. Pearson Prentice Hall; 2005.

8. Bradski G. The OpenCV Library. Dr Dobb's Journal of Software Tools. 2000.

9. Suzuki S, Be K. Topological structural analysis of digitized binary images by border following. CVGIP. 1985;30(1):32–46.

10. Canny J. A computational approach to edge detection. IEEE Trans Pattern Anal Mach Intell. 1986;8(6):679–698.

11. Duda RO, Hart PE. Use of the Hough transformation to detect lines and curves in pictures. Commun ACM. 1972;15(1):11–15.

12. Brunelli R. Template Matching Techniques in Computer Vision: Theory and Practice. Wiley; 2009.

13. Hutchinson S, Hager GD, Corke PI. A tutorial on visual servo control. IEEE Trans Robot Autom. 1996;12(5):651–670.

14. Rohmer E, Singh SPN, Freese M. V-REP: A versatile and scalable robot simulation framework. IEEE/RSJ IROS. 2013:1321–1326.

15. Koos S, Mouret JB, Doncieux S. The transferability approach: Crossing the reality gap in evolutionary robotics. IEEE Trans Evol Comput. 2013;17(1):122–145.

16. Gabor T. Modeling and simulation of cyber-physical systems using digital twins. IEEE Access. 2017;5:2050–2063.

17. Tobin J, Fong R, Ray A, et al. Domain randomization for transferring deep neural networks from simulation to the real world. IROS. 2017:23–30.

18. Elsayed A, Nahavandi S, Kouzani A. Hardware-in-the-loop simulation for robotic systems. IEEE/ASME Trans Mechatron. 2009;14(2):240–249.

19. ISO 10218-1:2011. Robots and robotic devices – Safety requirements for industrial robots – Part 1: Robots.

20. ISO/TS 15066:2016. Robots and robotic devices – Collaborative robots.

21. FANUC America Corporation. iRVision System Manual. FANUC Robotics; 2021.

22. ABB Robotics. RobotStudio: Integrated Vision Manual. ABB; 2020.

23. Lynch KM, Park FC. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press; 2017. Available from: https://modernrobotics.org/

24. Aristidou A, Lasenby J, Chrysanthou Y, Shamir A. Inverse Kinematics Techniques in Computer Graphics: A Survey. *Comput Graph Forum*. 2017;00(00):1–24. doi:10.1111/cgf.13310

9 Appendices

Appendix A: MATLAB Code for Forward Kinematics

```
clc
clear
close all

% Home configuration of the end-effector (4x4 homogeneous matrix)
M = [1.0000    0.0000    0.0018 -0.25308;
    -0.0000    1.0000         0 -0.17282;
    -0.0018   -0.0000    1.0000  0.19001;
     0         0         0       1]
% Screw axis list (6x6 matrix — 5 joints)
S = [ 0  0  0 -1  0;
      0 -1 -1  0 -1;
      1  0  0  0  0;
      0     0       0.18974  0        0.18974;
      0     0       0       -0.03813  0      ;
      0  -0.06981  -0.11993  0.18974  0.11617]

% Joint angles (in degrees) and conversion to radians
theta_deg = [10;20;30;40;50]
theta = deg2rad(theta_deg);

% Compute forward kinematics
T = FKinspace(M, S, theta);

% Extract rotation matrix from transformation
R = T(1:3, 1:3)

% Convert rotation matrix to Euler angles (ZYX order)
if abs(R(1,3)) ~= 1
    theta_y = asin(R(1,3));
    theta_x = atan2(-R(2,3), R(3,3));
    theta_z = atan2(-R(1,2), R(1,1));
else
    theta_y = sign(R(1,3)) * (pi/2);
    theta_x = atan2(R(2,1), R(2,2));
    theta_z = 0;
end

% Display Euler angles in degrees
euler = rad2deg([theta_x; theta_y; theta_z])



function T = FKinspace(M, Slist, thetalist)


T = M;
for i = length(thetalist): -1: 1
    T = MatrixExp6(VecTose3(Slist(:, i) * thetalist(i))) * T;
end
end
```

Appendix B: Blob Detection Lua Script

```
local sim = require 'sim'
local simVision = require 'simVision'

local mainScriptTargetHandle = -1
local sensor = -1
local xAngle, yAngle = 0, 0
local resX, resY = 0, 0
local lastSignalState = false
local detectionSent = false

function sysCall_init()
    sensor = sim.getObject('/sensor')
    xAngle                  =                sim.getObjectFloatParam(sensor,
sim.visionfloatparam_perspective_angle)
    resX = sim.getObjectInt32Param(sensor, sim.visionintparam_resolution_x)
    resY = sim.getObjectInt32Param(sensor, sim.visionintparam_resolution_y)

    if not resX or not resY or resX <= 0 or resY <= 0 then return end

    yAngle = xAngle
    local ratio = resX / resY
    if ratio > 1 then
        yAngle = 2 * math.atan(math.tan(xAngle / 2) / ratio)
    else
        xAngle = 2 * math.atan(math.tan(yAngle / 2) * ratio)
    end

    mainScriptTargetHandle = sim.getObject('/dBase')
end

function sysCall_vision(inData)
    local retVal = {trigger=false, packedPackets={}}

    simVision.sensorImgToWorkImg(inData.handle)
```

```
    simVision.selectiveColorOnWorkImg(inData.handle, {0.85, 0.85, 1}, {0.05,
0.05, 0.05}, true, true, false)
    local                trig,                packedPacket                =
simVision.blobDetectionOnWorkImg(inData.handle, 0.1, 0.0, true)
    if trig then retVal.trigger = true end
    if  packedPacket  then  table.insert(retVal.packedPackets,  packedPacket)
end
    simVision.workImgToSensorImg(inData.handle)


    return retVal
end


function sysCall_sensing()
    if not resX or not resY or resX <= 0 or resY <= 0 or detectionSent then
return end


    local res, _, packet2 = sim.handleVisionSensor(sensor)
    if res < 0 or not packet2 or #packet2 == 0 or mainScriptTargetHandle ==
-1 then return end


    local blobCnt, valCnt = packet2[1], packet2[2]
    if blobCnt > 0 and 2 + valCnt * 0 + 6 <= #packet2 then
        local blobX = packet2[2 + valCnt * 0 + 3]
        local blobY = packet2[2 + valCnt * 0 + 4]


        local depthData = sim.getVisionSensorDepth(sensor, 1, {
            1 + math.floor(blobX * (resX - 0.99)),
            1 + math.floor(blobY * (resY - 0.99))
        }, {1, 1})


        if depthData then
            local depthTable = sim.unpackFloatTable(depthData)
            if depthTable and #depthTable > 0 and depthTable[1] > 0 then
                local depth = depthTable[1] + 0.01
                local coord = {
                    depth * math.tan(xAngle * 0.5) * (0.5 - blobX) / 0.5,
                    depth * math.tan(yAngle * 0.5) * (blobY - 0.5) / 0.5,
                    depth
```

```
                }
                local  m  = sim.getObjectMatrix(sensor,  -1)
                local worldCoord = sim.multiplyVector(m, coord)
                local packedCoords = sim.packTable(worldCoord)
                sim.setBufferProperty(mainScriptTargetHandle,
"signal.blobCoordinates", packedCoords)
                detectionSent = true
                lastSignalState = true
            end
        end
    end
end

function sysCall_cleanup()
    if mainScriptTargetHandle ~= -1 then
        sim.removeProperty(mainScriptTargetHandle,
'signal.blobCoordinates')
    end
end
```

Appendix C: Arm Base Script

```
local sim = require 'sim'

function sysCall_init()
    gripper     = sim.getObjectHandle('/Gripper')
    gripperName = sim.getObjectAlias(gripper, 4)
    target      = sim.getObjectHandle('/target')
    robotBase   = sim.getObjectHandle('/dBase')
    P0 = sim.getObjectHandle('/P0')
    P1 = sim.getObjectHandle('/P1')
    P2 = sim.getObjectHandle('/P2')
    P3 = sim.getObjectHandle('/P3')
    P4 = sim.getObjectHandle('/P4')

    motionState = 'IDLE'
    actionDone  = false
    gripTime    = 0.5
```

```
end


function sysCall_thread()
    while sim.getSimulationState() ~= sim.simulation_advancing_abouttostop
do
        if motionState == 'IDLE' then
            local       buf       =       sim.getBufferProperty(robotBase,
'signal.blobCoordinates', {noError = true})
            if buf then
                local c = sim.unpackTable(buf)
                if c and #c == 3 then
                    sim.setObjectPosition(P2, -1, c)
                    sim.setObjectOrientation(P2, -1, {0, 0, 0})
                    sim.setObjectPosition(P1,  -1,  {c[1]  +  0.025,  c[2],
c[3]})
                    sim.setObjectOrientation(P1, -1, {0, 0, 0})
                    motionState = 'GOTO_P1'; actionDone = false
                end
            end

        elseif motionState == 'GOTO_P1' and not actionDone then
            MoveToPoint(target, P1, 0.1)
            actionDone = true; motionState = 'GOTO_P2'; actionDone = false

        elseif motionState == 'GOTO_P2' and not actionDone then
            MoveToPoint(target, P2, 0.1)
            actionDone = true; motionState = 'GRASP'; actionDone = false

        elseif motionState == 'GRASP' and not actionDone then
            sim.setIntProperty(sim.handle_scene,
'signal.'..gripperName..'_close', 1)
            while                       sim.getIntProperty(sim.handle_scene,
'signal.'..gripperName..'_gripped', {noError = true}) ~= 1 do
                sim.step()
            end
            sim.wait(gripTime)
            actionDone = true; motionState = 'GOTO_P3'; actionDone = false
```

```
        elseif motionState == 'GOTO_P3' and not actionDone then
            MoveToPoint(target, P3, 0.1)
            actionDone = true; motionState = 'GOTO_P4'; actionDone = false

        elseif motionState == 'GOTO_P4' and not actionDone then
            MoveToPoint(target, P4, 0.1)
            actionDone = true; motionState = 'RELEASE'; actionDone = false

        elseif motionState == 'RELEASE' and not actionDone then
            if                          sim.getIntProperty(sim.handle_scene,
'signal.'..gripperName..'_close', {noError = true}) == 1 then
                sim.removeProperty(sim.handle_scene,
'signal.'..gripperName..'_close')
            end
            sim.step()
            sim.wait(0.5)
            actionDone = true; motionState = 'GOTO_P0'; actionDone = false

        elseif motionState == 'GOTO_P0' and not actionDone then
            MoveToPoint(target, P0, 0.1)
            sim.wait(0.5)
            actionDone = true; motionState = 'CLOSE_AT_HOME'; actionDone =
false

        elseif motionState == 'CLOSE_AT_HOME' and not actionDone then
            sim.setIntProperty(sim.handle_scene,
'signal.'..gripperName..'_close', 1)
            sim.wait(1.5)
            actionDone = true; motionState = 'END'; actionDone = false

        elseif motionState == 'END' then
            sim.stopSimulation()
        end

        sim.step()
    end
end
```

```
function MoveToPoint(obj, tgt, maxVel)
    local pose = sim.getObjectPose(tgt)
    sim.moveToPose{
        object     = obj,
        targetPose = pose,
        maxVel     = {maxVel},
        maxAccel   = {maxVel},
        maxJerk    = {maxVel},
        metric     = {1, 1, 1, 0.1},
    }
end

function sysCall_cleanup()
    for _, name in ipairs{
        'signal.'..gripperName..'_close',
        'signal.'..gripperName..'_gripped'
    } do
        if sim.getIntProperty(sim.handle_scene, name, {noError = true}) then
            sim.removeProperty(sim.handle_scene, name)
        end
    end
end
```

Appendix D: Gripper Script

```
local sim = require 'sim'

function sysCall_init()
    local gripperObj = sim.getObjectHandle('/dBase/Gripper')
    gripperName      = sim.getObjectAlias(gripperObj, 4)

    leftSensor   = sim.getObjectHandle('/dBase/Gripper/leftSensor')
    rightSensor  = sim.getObjectHandle('/dBase/Gripper/rightSensor')
    motor        = sim.getObjectHandle('/dBase/Gripper/rightJoint1')
    auxMotor     = sim.getObjectHandle('/dBase/Gripper/leftJoint1')

    maxV            = 60 * math.pi / 180
```

```
    postSenseDelay  = 0.1
    delayCounter    = -1
    closed          = false
end

function sysCall_sensing()
    sim.handleProximitySensor(leftSensor)
    sim.handleProximitySensor(rightSensor)
end

function sysCall_actuation()
    local       closing       =       sim.getIntProperty(sim.handle_scene,
'signal.'..gripperName..'_close', {noError=true}) or 0

    if closing == 1 then
        if not closed then
            sim.setJointTargetVelocity(motor, -maxV)

            local lres = sim.readProximitySensor(leftSensor)
            local rres = sim.readProximitySensor(rightSensor)
            if lres == 1 and rres == 1 then
                if delayCounter < 0 then
                    delayCounter = postSenseDelay
                else
                    delayCounter          =          delayCounter          -
sim.getSimulationTimeStep()
                    if delayCounter <= 0 then
                        sim.setJointTargetVelocity(motor, 0)
                        closed = true
                        sim.setIntProperty(sim.handle_scene,
'signal.'..gripperName..'_gripped', 1)
                    end
                end
            else
                delayCounter = -1
            end
        else
            sim.setJointTargetVelocity(motor, 0)
```

```
        end
    else
        if closed then
            closed = false
            sim.removeProperty(sim.handle_scene,
'signal.'..gripperName..'_gripped')
        end
        delayCounter = -1
        sim.setJointTargetVelocity(motor, maxV)
    end
end

function sysCall_joint(inData)
    if inData.handle == auxMotor then
        if closed then
            return { vel = 0, force = inData.maxForce }
        end
        local mp  = sim.getJointPosition(motor)
        local err = -mp - inData.pos
        local   v       =   math.max(-inData.maxVel,   math.min(err   *   20,
inData.maxVel))
        return { vel = v, force = inData.maxForce }
    end
end
```