Project Report

Industrial Robotics 2024

Camera-based Pick-&-Place

Group

Table of Contents

1. Introduction

As automation and intelligent manufacturing continue to evolve, the need for affordable machine vision solutions has become increasingly evident. This project is dedicated to designing a robotic arm system for object grasping, utilizing monocular vision powered by the Logitech C270 webcam as the visual input source. By incorporating the advanced ArUco marker technology, the system achieves accurate pose estimation for the camera, thereby enabling dynamic vision functionality. Such a capability is vital for real-time environmental awareness and precise manipulation of objects.

To illustrate its functionality, the project targets the grasping of a 2x2 LEGO block, highlighting its effectiveness in handling small-scale objects. Additionally, the reliance on a monocular vision system significantly reduces hardware complexity, making the overall setup more cost-effective and easier to deploy.

In conclusion, this project not only highlights the role of machine vision in advancing automation but also proposes a budget-friendly and practical solution suitable for resource-constrained scenarios.

1.1. Hardware Required

1. **Computer Device**

Description: A computer capable of running Python programming.

2. **Processor**

Multi-core processors such as Intel i5, i7, or their AMD equivalents.

3. **Additional Features**

Adequate USB ports to connect peripherals like the webcam and the Arduino controller.

4. **Arduino Tinkerkit Braccio Robotic Arm**

Description: An Arduino-based robotic arm designed to perform object-grabbing tasks. Additional Requirements: Ensure a reliable power source and stable connectivity for optimal performance.

5. **Logitech C270 Webcam**

Description: A webcam used for capturing images and enabling dynamic vision processing.

6. **ArUco Markers**

Description: A set of four ArUco markers with IDs 0, 1, 2, and 3.

Printing Instructions: The markers must be printed with clarity and sized appropriately to be recognized accurately by the webcam.

7. **LEGO Brick**

Description: A red LEGO brick designated for object manipulation and testing. Quantity: One piece.

1.2. Software Required

8. **Operating System**

Supported Platforms: Compatible with modern versions of Windows, macOS, or Linux, depending on hardware requirements and library support.

9. **Python**

Required Version: Version 3.9 or newer.

10. **Development Environment**

Suggested Tools: PyCharm, VSCode, or Jupyter Notebook, ideal for development and debugging.

11. **Additional Libraries/Dependencies**

Key Libraries: OpenCV (cv2), NumPy, PySerial, and SymPy.

Installation: Use pip or other package managers for installation.

2. Objection

In our project utilizing monocular vision for a robotic arm Pick-and-Place system, the primary objective is to employ a webcam for accurately localizing objects within the robotic arm's coordinate system. This capability will allow the robotic arm to execute tasks such as object grasping and placement with high precision.

To achieve this overarching goal, the process has been divided into multiple stages, each addressing specific tasks and milestones. These include developing algorithms for detecting and recognizing objects, calibrating the camera to ensure accurate depth estimation, seamlessly integrating the vision system with the robotic arm's control framework, and performing rigorous testing to validate the system's reliability and precision in practical applications.

The following sections provide a detailed explanation of each stage, along with the specific tasks required to accomplish these goals.

2.1. Goal 1: Controlling the Robotic Arm

The first step of the project is to ensure the robotic arm can be effectively controlled. This involves writing a Python script to manage the Tinkerkit Braccio robotic arm. The process begins with connecting the robotic arm to a computer via a USB interface and identifying the COM port assigned to it. Python's pyserial library is then used to establish USB/serial communication. Once the connection is established, movement commands are sent to test the robotic arm's responsiveness. During this stage, it is also important to determine the positive rotation direction for each axis and label the axes appropriately. With these preparatory tasks completed, the project can proceed to developing the core kinematics code and implementing algorithms for object detection and pose estimation.

2.2. Goal 2: Forward and Backward Kinematics

In this project, implementing both Forward and Backward Kinematics is essential to achieving precise control of the robotic arm for grasping tasks. Forward Kinematics focuses on calculating the position of the gripper in the robotic arm's coordinate system based on the given joint angles. To achieve this, the Denavit-Hartenberg (DH)

Transformation method is utilized. This process begins with measuring and modeling the robotic arm, defining coordinate frames, and determining the DH parameters, which are then organized into a parameter table. With this table, the Forward Kinematics calculations can be coded, allowing the position of the gripper to be computed accurately.

Backward Kinematics, on the other hand, is crucial for translating target positions and orientations into the required joint angles for the robotic arm. This is particularly important in the final stages of the project, where object detection and pose estimation are performed through image processing. The output of this process provides the object's position and rotation in the robotic arm's coordinate system. Using Backward Kinematics and the previously established DH parameters, these values are converted into the precise rotation angles needed for each joint. This step requires a thorough analysis of the robotic arm's structure to ensure accurate computation of the required joint movements, enabling the robotic arm to perform grasping tasks with precision.

2.3. Goal 3: Image Processing

Accurate image processing plays a pivotal role in this project, as it forms the foundation for detecting LEGO bricks and extracting reliable data for subsequent operations. A robust visual system is essential to minimize errors and ensure that the information used in later stages is precise and consistent. Given its importance, image processing represents one of the most resource-intensive components of the project.

To achieve high-quality results, various techniques such as image pyramids, template matching, and contour detection will be explored. These methods enable effective object identification and tracking, ensuring the system operates with precision. The OpenCV library will serve as the primary tool for implementing these image-processing techniques, offering a powerful and flexible framework to support the system's visual requirements.

2.4. Goal 4: Monocular Visual Pose Estimation

Monocular vision-based pose estimation is a critical component for enhancing the accuracy of object detection results during image processing. Achieving precise pose estimation is essential to ensure the robotic arm can perform reliable and accurate

grasping tasks. For this project, the goal is to minimize errors in pose estimation while also maintaining a cost-effective and efficient system.

Our approach emphasizes the use of dynamic vision to reduce the need for rigidly fixed camera setups. Unlike the conventional eye-to-hand method, we plan to leverage ArUco markers to facilitate the transformation of pixel coordinates into the robotic arm's coordinate system. This approach simplifies the setup process, as placing ArUco markers and fine-tuning parameters in the code is far more practical than physically fixing a monocular camera in a static position.

In summary, this goal focuses on implementing monocular vision-based pose estimation using an unfixed camera, aiming to balance simplicity, cost-efficiency, and precision in the system's design.

2.5. Goal 5: Testing and Evaluation

Testing and evaluation are critical to ensuring the reliability and functionality of this low-cost robotic arm grasping system. Through comprehensive testing, we aim to identify potential limitations, refine performance, and optimize the overall system. Spot testing will be conducted to analyze the robotic arm's basic errors and assess its fundamental performance. Additionally, the arm's error behavior across different positions will be evaluated, enabling us to optimize the code and compensate for hardware inaccuracies using software-based adjustments. Each component of the program will undergo unit testing to assess its individual error performance and facilitate targeted optimizations. Following this, integration testing will be carried out to evaluate the system as a whole, with a particular focus on determining the success rate of grasping LEGO bricks. This process will provide valuable insights for refining the system's design and functionality while also highlighting areas for future development and improvement.

3. State of the Art

3.1. Object Detection Recognition- SSD

SSD (Single Shot Multibox Detector) is a groundbreaking deep learning-based algorithm for object detection that strikes an effective balance between speed and accuracy, making it ideal for real-time applications. Introduced as a one-stage detection model, SSD avoids the computational overhead of traditional two-stage approaches like Faster R-CNN by directly predicting object classes and bounding boxes in a single forward pass through the network.

Core Principles

The foundation of SSD lies in its ability to detect objects at multiple scales using a series of convolutional feature maps. Starting with a base network (commonly VGG-16 in the original SSD), SSD adds additional convolutional layers to produce feature maps of progressively smaller spatial resolution. These feature maps capture objects at different scales, with larger objects being detected in lower-resolution maps and smaller objects in higher-resolution maps.

Anchor boxes (default boxes) with varying aspect ratios are defined for each feature map, ensuring that objects of different shapes and orientations can be detected. For every anchor box, SSD predicts both the class probabilities and bounding box offsets, enabling it to localize and classify objects efficiently. This multiscale approach is one of SSD's standout features, allowing it to detect objects ranging from small to large in size.

Another innovation in SSD is its use of a loss function that combines localization loss (smooth L1 loss) and confidence loss (softmax loss), ensuring accurate predictions for both bounding boxes and class probabilities.

Advantages

SSD offers several advantages over other detection methods:

1. **Speed**: By eliminating the need for region proposal generation, SSD processes images in a single pass, making it significantly faster than two-stage methods like Faster R-CNN. This makes SSD particularly suitable for real-time applications.

2. **Multiscale Detection**: The use of multiple feature maps allows SSD to handle objects of varying sizes effectively.

3. **Efficiency**: SSD achieves high accuracy while maintaining a relatively lightweight architecture, making it deployable on devices with limited computational power.

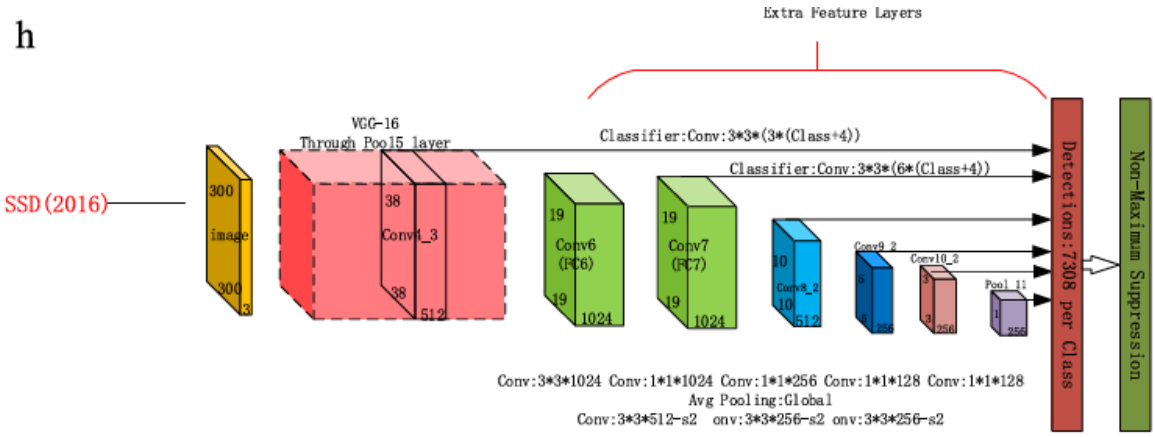
Applications

Autonomous Driving: SSD's speed and ability to detect objects of varying sizes make it a natural fit for real-time applications like detecting pedestrians, vehicles, and traffic signs in self-driving cars.

Robotic Grasping: In industrial settings, SSD can assist robotic arms in recognizing and handling small or irregularly shaped objects, such as tools or parts.

Surveillance Systems: SSD's real-time capabilities allow it to monitor dynamic scenes, making it effective for security and wildlife monitoring.

Embedded Devices: Its lightweight architecture allows SSD to be deployed on drones, mobile devices, and IoT systems for object detection tasks.



3.2. Grasping and Pose Estimation-DGGN

The **Deep Geometry-Aware Grasping Network (DGGN)** is a state-of-the-art model designed to address the challenges of robotic grasping in complex and unstructured environments. It incorporates geometric understanding to enhance grasping precision, particularly for objects with irregular shapes or diverse orientations. By integrating geometry-awareness into the learning process, DGGN significantly improves the reliability and efficiency of grasp generation, making it an ideal choice for real-world applications.

Core Principles

DGGN operates by leveraging **RGB-D** input (color and depth data) to simultaneously reconstruct 3D object geometry and predict optimal grasp configurations. The network is divided into two stages:

1. **3D Shape Reconstruction and Representation**:

Using the RGB-D data, DGGN generates a 3D representation of the object by reconstructing its geometric features. This step is crucial for capturing spatial relationships, surface curvatures, and depth information, which are essential for determining stable grasp points.

2. **Geometry-Aware Grasp Prediction**:

The reconstructed geometry is processed to learn grasp representations. DGGN uses deep learning to analyze the contact regions, force balance, and stability of the grasp. By embedding geometry-aware features, the model ensures that the predicted grasp aligns with the object's physical characteristics, minimizing slippage or failure.

Innovative Features

1. **Geometry-Aware Learning**: Unlike traditional approaches that rely solely on visual features, DGGN integrates geometric understanding into the prediction process. This enhances the system's ability to handle objects with complex shapes.

2. **Robustness to Occlusions**: By utilizing depth data, DGGN can infer hidden object regions, making it effective in cluttered or partially obstructed environments.

3. **Multi-Modal Data Fusion**: The model combines RGB and depth data, achieving a richer understanding of the object's physical properties.

Applications

Industrial Automation: DGGN is particularly suited for grasping tasks in manufacturing, such as handling tools or components with irregular geometries.

Logistics and Warehousing: The network can grasp and manipulate diverse items, from small packages to complexly shaped products, in dynamic environments.

Assistive Robotics: DGGN enhances service robots by enabling them to handle everyday objects with high precision, such as kitchen utensils or medical instruments.

Recycling and Sorting: Its ability to deal with varied shapes and materials makes it ideal for tasks such as sorting waste or recyclable items.

Reference: Bai, Qiang, et al. "Object detection recognition and robot grasping based on machine learning: A survey." IEEE access 8 (2020): 181855-181879.

4. Requirements

4.1.1. Image Processing Part

The following points should be noticed for ensuring high success rate of object detection and location.

1. In the first step when calculating the projection matrix, it is important to ensure all four ArUco markers are successfully detected. And their real coordinates must be accurately measured and inputted into the program to ensure precise calibration and projection.

2. Ensure the ArUco markers are clean and fat, as well as no reflective light.

3. Once the projection matrix has been successfully calculated, it is crucial not to move the camera.

4. It is important to maintain sufficient lighting throughout this experiment, and avoid shadows.

4.1.2. Kinematics part

The following points should be noticed to ensure a high success rate in grabbing the object at the specified position.

1. **Robot Arm Calibration**:

The robot arm needs to be properly calibrated to compensate for any inherent angular offsets, ensuring precise positioning and movement.

2. **Trajectory Planning**:

The trajectory of the robot arm as it moves toward the destination should be thoroughly considered. Since direct control of the trajectory is not feasible, placing intermediate waypoints is essential. These waypoints help to guide the arm smoothly and accurately, minimizing sudden deviations and potential collisions with the object

3. **Speed Optimization**:

The speed of the robot arm must be carefully controlled to balance efficiency and accuracy. Excessive speed may lead to overshooting or unstable grasping, while a speed that is too slow can reduce operational efficiency.

5. Implementation/Major Steps

5.1. Kinematic part

In the kinematics section of our project, we need to develop a program that calculates the joint angles of the robotic arm based on the coordinates and rotation angle of a LEGO brick within a coordinate system centered on the robotic arm. The goal is to accurately achieve the grasping of the LEGO brick. The coordinate axes of the defined robotic arm coordinate system are illustrated in the following diagram.

Inverse Kinematic:

For Inverse Kinematic, we have implemented a function $\boldsymbol{BraccioInverse}(w)$ to compute joint angles based on the given target position, which is represented by w. With the application of inverse kinematic principles, the function returns joint angles as a vector q. Below, we will provide a brief explanation of the theoretical foundations used for the calculation.

Input and Output:

Tool configuration vector w:

$$w(q) = \begin{pmatrix} x \\ y \\ z \\ Rot.around\ x_0 \\ Rot.around\ y_0 \\ Rot.around\ z_0 \end{pmatrix}$$

Vector of joint angles q:

$$q = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \end{pmatrix} = \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \\ \theta_5 \end{pmatrix}$$

Calculation:

The base's rotation angle q_1:

$$q_1 = atan2(w_2, w_1)$$

The total tool pitch angle q_{234}:

$$q_{234} = atan2(-w_4 \cos q_1 - w_5 \sin q_1 , -w_6)$$

2 intermediate variables b_1 **and** b_2:

$$b_1 = w_1 \cos q_1 + w_2 \sin q_1 - a_4 \cos q_{234} + d_5 \sin q_{234}$$

$$b_2 = d_1 - a_4 \sin q_{234} - d_5 \cos q_{234} - w_3$$

$$b^2 = {b_1}^2 + {b_2}^2$$

Elbow joint angle q_3 ($\pm$ takes into account the elbow up + and the elbow down -):

$$q_3 = \pm \operatorname{acos} \frac{b^2 - {a_2}^2 - {a_3}^2}{2\, a_2\, a_3}$$

Shoulder-joint angle q_2:

$$q_2 = atan\big((a_2 + a_3 \cos q_3) b_2 - a_3 b_1 \sin q_3 , (a_2 + a_3 \cos q_3) b_1 + a_3 b_2 \sin q_3\big)$$

Tool pitch angle q_4:

$$q_4 = q_{234} - q_2 - q_3$$

Tool roll angle q_5:

$$q_5 = \pi \ln({w_4}^2 + {w_5}^2 + {w_6}^2)$$

for the **5-axis** articulated robot, the **approach vector** has to be given in the form:

$$w = \binom{w_p}{w_o}\left(\exp\left(\frac{q_n}{\pi}\right)^{p} r_3\right)$$

The corresponding **Python code** for the calculation steps above is as follows:

```
def BraccioInverse(w):
    # w is a vector: (x, y, z, Rot.around x0, Rot.around y0, Rot.around z0)

    # define q as the returned vector of the function: (q1, q2, q3, q4, q5)
    q = Matrix([0, 0, 0, 0, 0, 0])

    # atan2(y,x) function is the 4-quadrant variant for the arcus-tangent function
    q[1] = atan2(w[2], w[1])

    # q234 = q[2] + q[3] + q[4] = w[4]
    q234 = atan2(-w[4] * cos(q[1]) - w[5] * sin(q[1]), -w[6])

    # 2 intermediate variables are introduced: b1 and b2
    b1 = w[1] * cos(q[1]) + w[2] * sin(q[1]) - a4 * cos(q234) + d5 * sin(q234)
    b2 = d1 - a4 * sin(q234) - d5 * cos(q234) - w[3]
    bb = (b1**2) + (b2**2)

    q[3] = acos((bb - a2**2 - a3**2) / (2 * a2 * a3))
    q[2] = atan2((a2 + a3 * cos(q[3])) * b2 - a3 * b1 * sin(q[3]),
                 (a2 + a3 * cos(q[3])) * b1 + a3 * b2 * sin(q[3]))
    q[4] = q234 - q[2] - q[3]

    q[5] = q[1] + pi * log(sqrt(w[4]**2 + w[5]**2 + w[6]**2))
    q[5] = q[5] % pi

    # Offset
    # q[1] = q[1] + 0
    q[2] = q[2] + pi # - 8 / 180 * pi
    q[3] = q[3] + pi / 2 # - 7 / 180 * pi
    q[4] = q[4] + pi / 2 # + 3 / 180 * pi
    # q[5] = q[5] + 0

    return q

```

Robot Control:

In order to transmit the calculated joint angles to the robotic arm for execution, we need to implement a function that transforms the angles array into a command string formatted for serial communication. This section of the code integrates the angles into a string, which is then sent via the serial port to the robotic arm.

```
def Braccio_toString(q):
    q_degree = N(radian_to_degree(q))
    flag = True

    for angle in q_degree:
        # use "try" to keep the code running
        try:
            if angle < 0 or angle > 180:
                return None
```

```
        except:
            return None

    command = "P" + str(int(q_degree[1])) + ", " \
          + str(int(q_degree[2])) + ", " \
          + str(int(q_degree[3])) + ", " \
          + str(int(q_degree[4])) + ", " \
          + str(int(q_degree[5]))

    return command

```

Visualization of Range:

We iterate through a range of x and y coordinates to evaluate the workspace of the robot's inverse kinematics algorithm. The goal is to determine whether the BraccioInverse function can generate valid outputs for various x and y inputs. Two configurations are analyzed: **one at the target position** $(x, y, 0)$ **and another slightly above it at** $(x, y, 20)$. This approach assesses whether the robotic arm can reach a position above the target and subsequently move downward to grasp an object at the destination. If both function outputs are valid, the corresponding coordinates are recorded for further visualization and analysis.

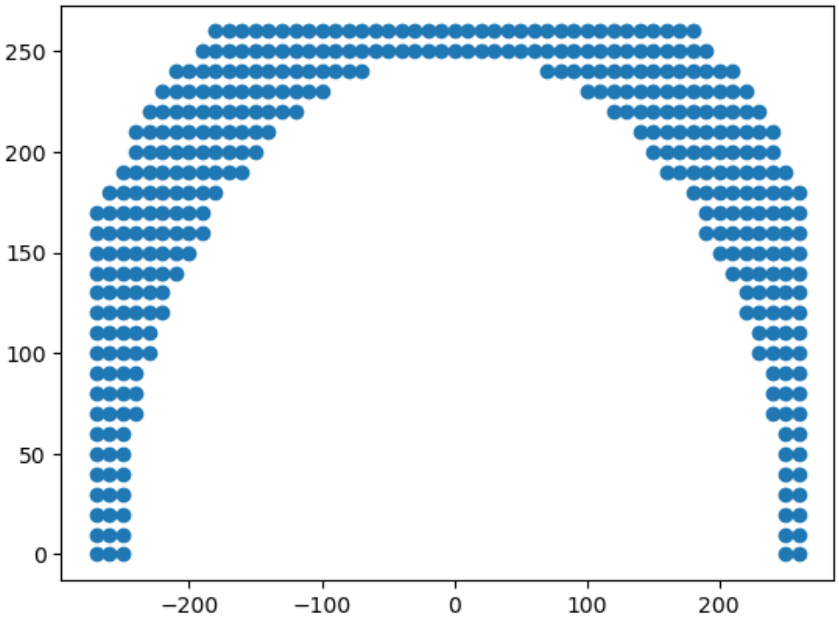
```
print("Start iteration")
posx = []
posy = []
for x in range(-270, 270, 10):
    for y in range(0, +270, 10):
        w = Matrix([0, x, y, 0, 0, 0, -exp(0)])
        w_above = Matrix([0, x, y, 20, 0, 0, -exp(0)])
        angles = Braccio_toString(BraccioInverse(w))
        angles_above = Braccio_toString(BraccioInverse(w_above))
        print("x=" + str(x) + " " + "y=" + str(y), end = "\n")
        if angles and angles_above:
            posx.append(x)
            posy.append(y)

            print(": " + angles, end = "\n")
        else:
            print(": ---")
print("Finished")
```

Using $matplotlib.pyplot$, the scatter fucntion is employed to plot the stored x and y coordinates ($posx$ and $posy$) on a 2D graph. This visualization helps to analyze the distribution of reachable points and provides insights into the operational range of the robotic arm.

```
import numpy as np
import matplotlib.pyplot as plt
plt.scatter(posx,posy)
```

Result of the Visualization:



5.2. LEGO localization

We use ArUco markers to obtain accurate 2D localization of LEGO bricks. The ArUco markers are placed at the four corners of the operational space. The goal is to identify these markers and use their positions to compute the homograph matrix H. We need to calculate the real world coordinates of the four ArUco markers in terms of the robot arm and feed to the homograph matrix calculating function. For a 2D transformation, the formula for mapping image coordinates (x,y) to world coordinates (X,Y) is:

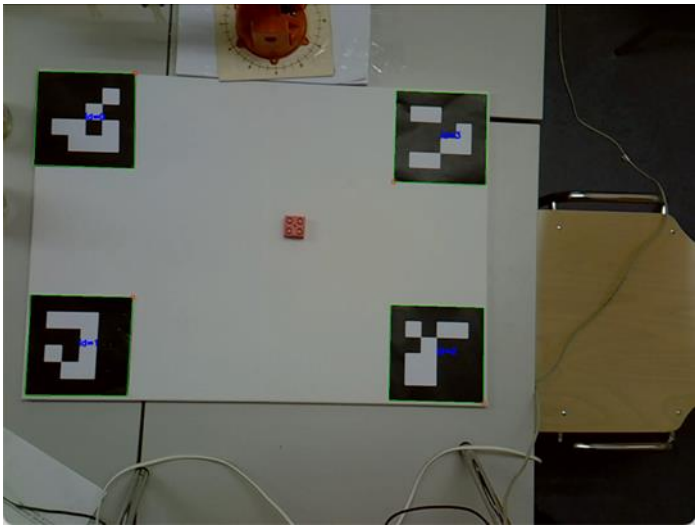
$$\begin{matrix} X \\ Y \\ 1 \end{matrix} = H \cdot \begin{matrix} x \\ y \\ 1 \end{matrix}$$

Once the homograph matrix is computed, it can be used to map the 2D coordinates of the detected LEGO brick to real-world coordinates.

5.3. Image processing part

5.3.1. Set Up

System Layout



Camera View

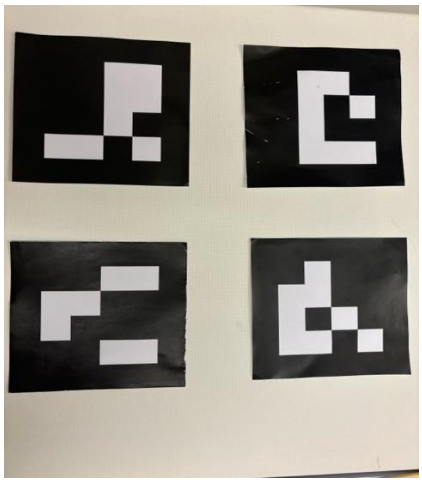
5.3.2. Image Acquisition

Image Acquisition is done by OpenCV VideoCapture function.

```
1. video_capture = cv2.VideoCapture(0)
2.     if not video_capture.isOpened():
3.         print("Unable to open the camera")
4.         return
```

5.3.3. ArUco Marker Detection

The objective is to calculate the actual coordinates of a brick using some reference object to avoid influence of camera location, so we do not need fix camera location. We choose to use ArUco markers because they are widely used in computer vision task for precise position tracking and calibration. These markers are square, binary patterns that can be easily detected and uniquely identified in an image. By placing ArUco markers in a known spatial arrangement, their real-world positions can be used to compute a transformation matrix that relates camera pixel coordinates to real coordinates in space.



ArUco Markers

OpenCV offers various functions related to ArUco marker, we use cv2.aruco.detectMarters function to detect four markers and recalculate the position of center point of ArUco Markers.

```
1.       # Detect ArUco markers
 2.          parameters = cv2.aruco.DetectorParameters()
 3.          aruco_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_4X4_50)
 4.          corners, ids, rejectedImgPoints = cv2.aruco.detectMarkers(gray, aruco_dict,
parameters=parameters)
 5.          frame = cv2.aruco.drawDetectedMarkers(frame, corners, ids)
 6.          # If markers are detected
 7.          if len(corners) == 4:
 8.              frame = cv2.aruco.drawDetectedMarkers(frame, corners, ids)
 9.
10.              for i in range(len(ids)):
11.                  if ids[i][0] > 3:  # Only consider markers with IDs 0-3
12.                      continue
13.                  else:
14.                      # Calculate the center point of the ArUco marker
15.                      moment = cv2.moments(corners[i][0])
16.                      if moment["m00"] != 0:
17.                          cX = int(moment["m10"] / moment["m00"])
18.                          cY = int(moment["m01"] / moment["m00"])
19.                          aruco_centers_dict[ids[i][0]] = (cX, cY)
20.
```

```
        # Get sorted ArUco marker centers
        aruco_centers = [aruco_centers_dict[i] for i in sorted(aruco_centers_dict)]
```

5.3.4. LEGO Brick Detection

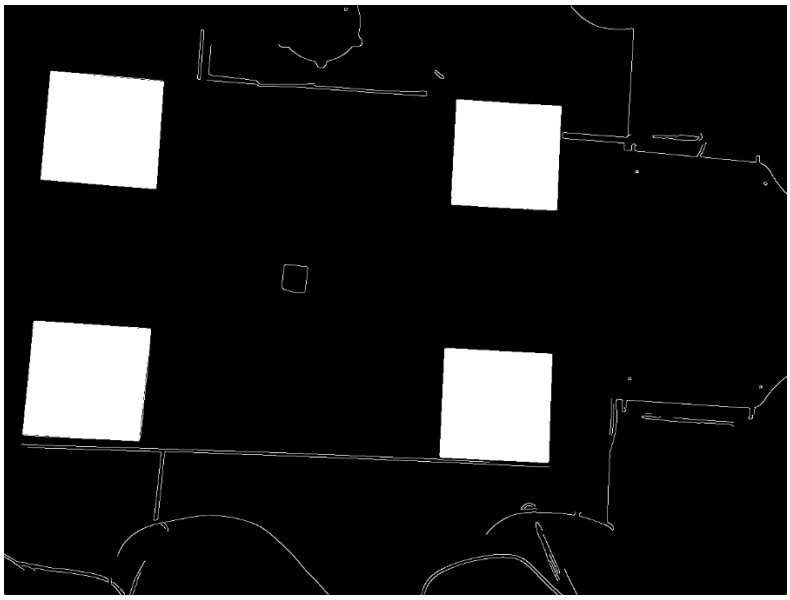
After calculating the transformation matrix, the next step involves detecting the LEGO brick to obtain the target coordinates for grasping. Given that the LEGO brick has a fixed shape, we opted to use edge detection as the primary method for identifying its position. Following a review of relevant literature, we selected the Canny edge detection algorithm due to its robustness and precision. The goal of our detection process is to identify the geometric center of the LEGO brick, which serves as the target for the grasping operation.

Canny edge detection involves several steps. The first step is noise reduction. Since edge detection is sensitive to noise, the algorithm begins with a Gaussian filter to smooth the image and reduce high-frequency noise. The parameter of GaussianBlur is obtained by several tests to ensure the best performace of detection the edge information with the lowest noise.

Following is the code realization.

```
gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
blurred_frame = cv2.GaussianBlur(gray_frame, (7, 7), 2.0)
edges = cv2.Canny(blurred_frame, threshold1=50, threshold2=150)
```

Following is our result of edge detection:



Then we need to identify the LEGO brick. We try to find rectangular objects (which typically correspond to LEGO bricks) within the video feed. we extract the contours using cv2.findContours(). We then approximate each contour to a polygon using

cv2.approxPolyDP(). To ensure that the detected contours are likely LEGO bricks, we filter out contours that are too small or not rectangular in shape. We calculate the aspect ratio of the bounding box around each contour, which should typically range from 0.5 to 2.0 for LEGO bricks. Additionally, the contour area is compared to a threshold to avoid detecting irrelevant small objects. Once a valid LEGO brick is detected, we calculate its geometric center using image moments. This center can then be used to compute the real-world coordinates of the LEGO brick, which can be useful for tasks like object manipulation or positioning. Following is the code realization:

```
for contour in contours:
            if is_not_grey(frame, contour):
                epsilon = 0.02 * cv2.arcLength(contour, True)
                approx = cv2.approxPolyDP(contour, epsilon, True)

                if len(approx) == 4 and cv2.isContourConvex(approx):
                    x, y, w, h = cv2.boundingRect(approx)
                    aspect_ratio = float(w) / h
                    area = cv2.contourArea(contour)
                    is_aruco = any(cv2.pointPolygonTest(corner.reshape(
                        4, 2), (x + w // 2, y + h // 2), False) >= 0 for corner in corners)

                    if not is_aruco and 0.5 <= aspect_ratio <= 2.0 and area > max_area and
area > MIN_AREA_THRESHOLD:
                        largest_square = approx
                        max_area = area
                        approx_3d = [calculate_real_coordinates(
                            point[0]) for point in approx]

        if largest_square is not None:
            cv2.drawContours(frame, [largest_square], -1, (0, 255, 0), 3)

            # Calculate the geometric center
            M = cv2.moments(largest_square)
            if M["m00"] != 0:
                cx = int(M["m10"] / M["m00"])
                cy = int(M["m01"] / M["m00"])
                cv2.circle(frame, (cx, cy), 5, (0, 0, 255), -1)

                # Convert the geometric center to world coordinates
                center_3d = calculate_real_coordinates((cx, cy))
                # print("Real coordinates:")
                # print(center_3d[0], center_3d[1])

                # Accumulate the 3D center coordinates and angles
                center_3d_accumulated += center_3d[:2]

                approx_3d = np.array(approx_3d, dtype=np.float32)

                # Calculate vectors and project onto the x-y plane
                vectors = approx_3d - center_3d
                vectors[:, 2] = 0

                # Calculate angles of each vector relative to the x-axis
                angles = np.arctan2(vectors[:, 1], vectors[:, 0]) * 180 / np.pi
                sum_angle = np.sum(angles)

                # Normalize angles to [0, 360]
                while sum_angle < 0:
                    sum_angle += 360
                while sum_angle > 360:
                    sum_angle -= 360
```

```
52.
53.                 angles_accumulated += sum_angle / 4
54.
55.                 # Increment frame count
56.                 frame_count += 1
57.
58.                 # When 10 frames are accumulated, calculate and display the average
59.                 if frame_count == max_frames:
60.                     avg_center_3d = center_3d_accumulated / max_frames
61.                     avg_angle = angles_accumulated / max_frames
62.
63.                     # print("Average real coordinates (x, y):", avg_center_3d)
64.                     # print("Average angle:", avg_angle)
65.
66.                     # Reset accumulators
67.                     center_3d_accumulated = np.zeros(2)
68.                     angles_accumulated = 0
69.                     frame_count = 0
70.                     return avg_center_3d, avg_angle
```

5.3.5. LEGO Bricks and ArUco Markers Distinguishing

One challenge for our LEGO bricks detection is distinguishing ArUco markers and LEGO bricks. ArUco markers are often present in robotic vision tasks for localization purposes. Since LEGO bricks and ArUco markers may appear similar in shape, we exclude any contour that overlaps with an ArUco marker. This is done by checking if the centroid of the bounding box lies inside any of the detected ArUco marker regions.

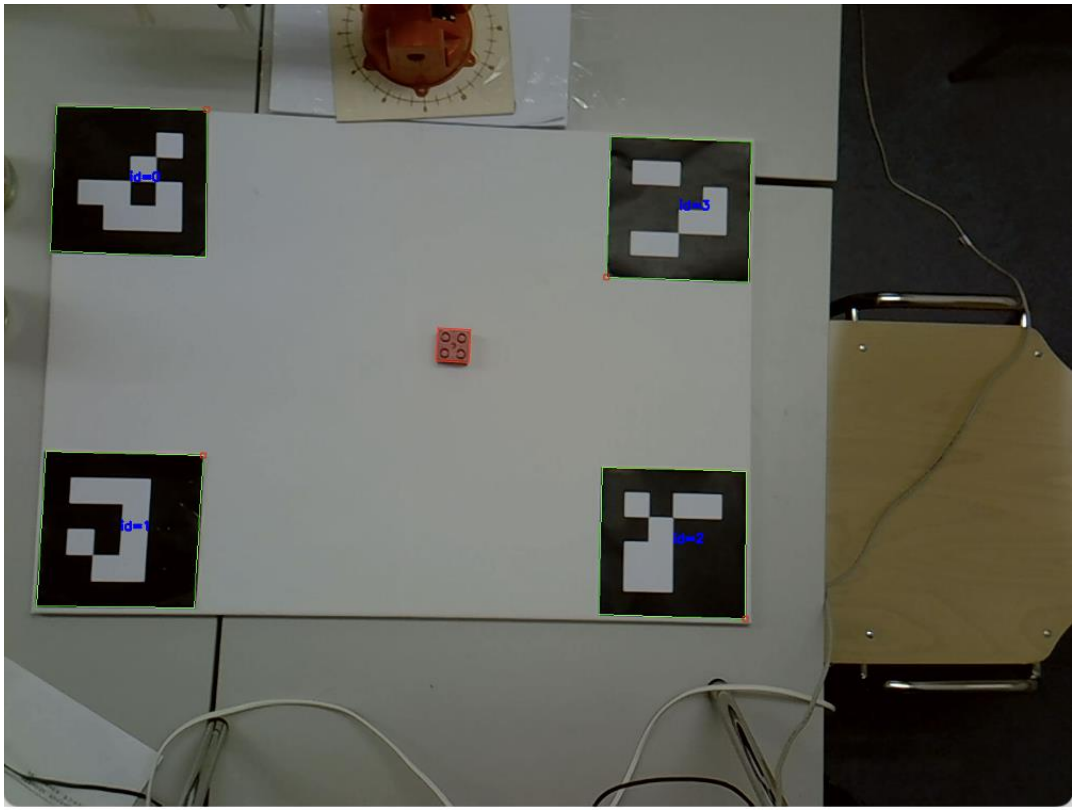
We use several method to exclude ArUco markers when detecting the LEGO bricks.

Firstly, we use the ArUco markers to exclude the area where the LEGO bricks detection performs, i.e., checking if the centroid of the bounding box lies inside any of the detected ArUco marker regions. However, the ArUco marker detection is not always successful across every frame of the camera. Once the ArUco marker detection is failed, the detection of LEGO bricks may turns on the ArUco. Thus we add another color filtering to ensure the ArUco markers are filtered since the ArUco marker is either black or white. Following is the code realization:

```
1. def is_not_grey(frame, contour):
2.         mask = np.zeros(frame.shape[:2], dtype=np.uint8)
3.         cv2.drawContours(mask, [contour], -1, 255, thickness=-1)
4.         mean_color = cv2.mean(frame, mask=mask)[:3]
5.         mean_color_hsv = cv2.cvtColor(
6.             np.uint8([[mean_color]]), cv2.COLOR_BGR2HSV)[0][0]
7.         min_saturation = 40
8.         min_value = 40
9.         if mean_color_hsv[1] > min_saturation and min_value < mean_color_hsv[2] < 220:
10.            return True  # Object is not black, white, or grey
11.        return False
12.
13. is_aruco = any(cv2.pointPolygonTest(corner.reshape(
14.                         4, 2), (x + w // 2, y + h // 2), False) >= 0 for corner in corners)
15.
```

```
16.
17. if is_not_grey(frame, contour) and not is_aruco and 0.5 <= aspect_ratio <= 2.0 and area >
max_area and area > MIN_AREA_THRESHOLD:
18.
```

Following is our result of object detection:



5.3.6. Coordinate System Transition

To get the coordinate system transition from camera pixel coordinate to real coordinate, we use cv2.findHomography function provided by OpenCV. Specifically, we set pixel coordinates and real coordinates of four ArUco markers obtained before as parameters, and the return value is called projection matrix, which can then be used to calculate the real coordinate of each pixel in the frames captured by camera.

```
 1.         # Only calculate projectio matrix if all required markers are detected
 2.         if len(aruco_centers) == 4:
 3.             pixel_coords = np.array(aruco_centers, dtype=np.float32)
 4.
 5.             # Define real-world coordinates of the ArUco markers
 6.             # This is an example
 7.             real_coords_big = np.array(
 8.                 [[275, 165, 0], [275, 515, 0], [-275, 515, 0], [-275, 165, 0]],
 9.                 dtype=np.float32
10.             )
11.
12.             # Calculate the 3x3 projection matrix
```

```
13.             projection_matrix, _ = cv2.findHomography(pixel_coords, real_coords_big)
```

Then we get the real coordinate of the LEGO brick with the calculated projection matrix.

```
1.    def calculate_real_coordinates(lego_center):
2.         lego_point = np.array([lego_center[0], lego_center[1], 1], dtype=np.float32)
3.         lego_real_coord = np.dot(projection_matrix, lego_point)
4.         lego_real_coord /= lego_real_coord[2]
5.         return lego_real_coord
```

5.3.7. Rotation Calculation

To improve the success rate of robotic gripper in grasping LEGO brick, it is essential to calculate its rotation degree. Here we calculate its angle with the x-axis in reality. So firstly we convert its camera pixel coordinate to real coordinate, and then we calculate the vectors from the geometric center of the LEGO brick to its four endpoints. Once all the angles are computed, they are summed and normalized to ensure that result is within 0 to 90 degrees since the LEGO brick is close to a square.

```
 1.      center_3d = calculate_real_coordinates((cx, cy))
 2.                 # Accumulate the 3D center coordinates and angles
 3.                 center_3d_accumulated += center_3d[:2]
 4.                 approx_3d = np.array(approx_3d, dtype=np.float32)
 5.                 # Calculate vectors and project onto the x-y plane
 6.                 vectors = approx_3d - center_3d
 7.                 vectors[:, 2] = 0
 8.                 # Calculate angles of each vector relative to the x-axis
 9.                 angles = np.arctan2(vectors[:, 1], vectors[:, 0]) * 180 / np.pi
10.                 sum_angle = np.sum(angles)
11.                 # Normalize angles to [0, 360]
12.                 while sum_angle < 0:
13.                     sum_angle += 360
14.                 while sum_angle > 360:
15.                     sum_angle -= 360
16.                 angles_accumulated += sum_angle / 4
17.
18.                 # Increment frame count
19.                 frame_count += 1
20.
21.                 # When 10 frames are accumulated, calculate and display the average
22.                 if frame_count == max_frames:
23.                     avg_angle = angles_accumulated / max_frames
```

6. Testing and Evaluation

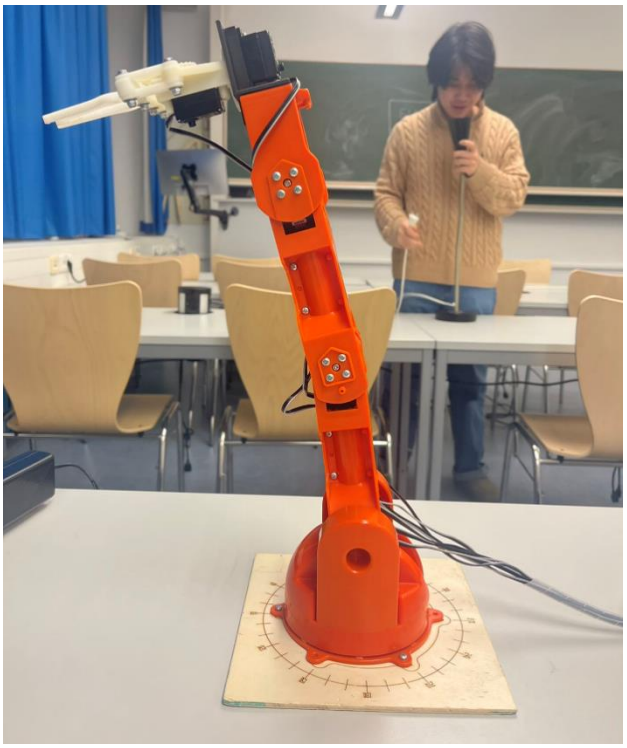
6.1. Robotic arm correction test

At the beginning of this project, the primary task was to test the accuracy of the robotic arm itself and to record and analyze its movement precision. This test aimed to evaluate the robotic arm's performance under specific conditions. To achieve this, we selected 10 random target points representing the positions the robotic arm might need to reach in real-world applications. We carefully measured the errors between the actual and expected positions of the robotic gripper's center point at these target points and conducted an in-depth analysis of the root causes of these errors.

Additionally, we developed corresponding code to implement error compensation to ensure that the robotic arm can reach its target positions with greater accuracy. This series of tests and analyses provided us with crucial data to assess whether the robotic arm is suitable for our application and identified areas for improvement to enhance its performance. The primary sources of error were found to be the misalignment of the robotic arm's center point and deviations in its angular positioning.

6.1.1. Robotic arm vertical test

To evaluate the movement accuracy of a robotic arm, conducting a vertical test is an essential first step. Positioning the manipulator in a vertical state allows for the identification and measurement of the baseline error in each joint during its initial state. Since these errors can become more pronounced in complex motion paths, it is critical to detect and document them early in the testing process.



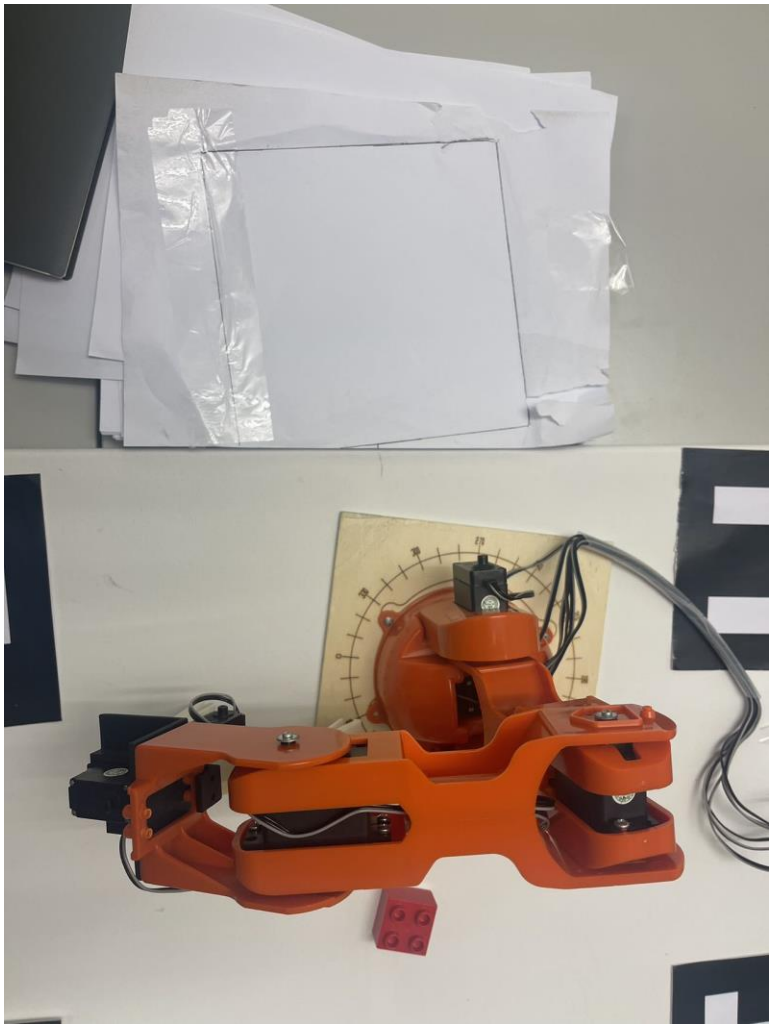
Joint No.	Given Angle	Actual Angle	Initial Error
Joint 2-	90	89	1
Joint 3	90	87	3
Joint 4	90	88	2

The maximum angular error is approximately indicating that in certain instances, the deviation of the robotic arm may surpass the acceptable threshold. This highlights the low precision of the robotic arm, necessitating measures to enhance its performance. By integrating a compensation mechanism into the software, the joint angles can be adjusted to improve the arm's accuracy and ensure it meets the required precision in real-world applications.

6.1.2. Robotic Arm Center Point Testing

Before starting the experiment, we observed that the center of the robotic arm was not aligned with the center of the square wooden board. Therefore, it was necessary to determine the true center of the robotic arm. By performing a center alignment test, we aimed to identify the offset of the robotic arm's center point relative to the board. These

offsets could significantly affect accuracy during operation, especially in more complex movement trajectories. Hence, determining and recording the true center at the beginning of the experiment was essential for ensuring reliable testing results.



Due to the alignment deviation between the robot and its base, we employed a specialized point-marking method to determine the robot's true coordinates. By performing multiple calculations, we were able to measure the offset between the center of the robot and the base. This data was then used to plot the deviation on a plane, allowing us to visualize the alignment errors. Finally, we adjusted the coordinate system to place the robot at the center and accurately plotted the X and Y axes.

To ensure precision, the point-marking process involved selecting reference points on the robot and base, then comparing their relative positions through repeated measurements. These measurements were analysed to determine the extent of misalignment, which was critical for improving the robot's performance during operations. By accurately mapping

the offsets, we were able to establish a reliable coordinate system, which serves as a foundation for future calibration and motion accuracy tests.

6.1.3. Random-Points Testing of the Robotic Arm

After addressing the initial errors, we conducted random-points testing of the robotic arm. This involved transmitting a set of random coordinate points and monitoring its movement and accuracy in reaching these locations. These tests provided a deeper understanding of the robotic arm's performance, and the measured results enabled further program optimization. We chose 10 random positions within the working range of the robotic arm, measuring the **X, Y, Z coordinates and the Angle** at each position to evaluate its precision and reliability.

Note: The units for X Coordinate, Y Coordinate, Z Coordinate, and Angle are mm and °

1. 1st set of tests:

	Input Value	Actual Value	Error Value
X Coordinate	-51	-51	0
Y Coordinate	265	263	2
Z Coordinate	1	0	1
Angle	37	36	1

2. 2nd set of tests:

	Input Value	Actual Value	Error Value
X Coordinate	-129	-131	2
Y Coordinate	272	272	0
Z Coordinate	1	0	1
Angle	86	86	0

3. 3rd set of tests

	Input Value	Actual Value	Error Value
X Coordinate	77	77	0
Y Coordinate	245	245	0
Z Coordinate	1	0	1
Angle	24	25	1

4. 4th set of tests:

	Input Value	Actual Value	Error Value
X Coordinate	91	91	0
Y Coordinate	271	270	1
Z Coordinate	1	0	1
Angle	87	87	0

5. 5th set of tests:

	Input Value	Actual Value	Error Value
X Coordinate	-20	-19	1
Y Coordinate	303	302	1
Z Coordinate	1	0	1
Angle	8	8	0

6. 6th set of tests:

	Input Value	Actual Value	Error Value
X Coordinate	-14	-14	0
Y Coordinate	293	293	0
Z Coordinate	1	0	1

Angle	51	50	1

7. 7th set of tests:

	Input Value	Actual Value	Error Value
X Coordinate	-76	-75	1
Y Coordinate	239	239	0
Z Coordinate	1	0	1
Angle	36	36	0

8. 8th set of tests:

	Input Value	Actual Value	Error Value
X Coordinate	77	77	0
Y Coordinate	242	242	0
Z Coordinate	1	0	1
Angle	89	88	1

9. 9th set of tests:

	Input Value	Actual Value	Error Value
X Coordinate	79	80	1
Y Coordinate	284	285	1
Z Coordinate	1	0	1
Angle	87	87	0

10. 10th set of tests:

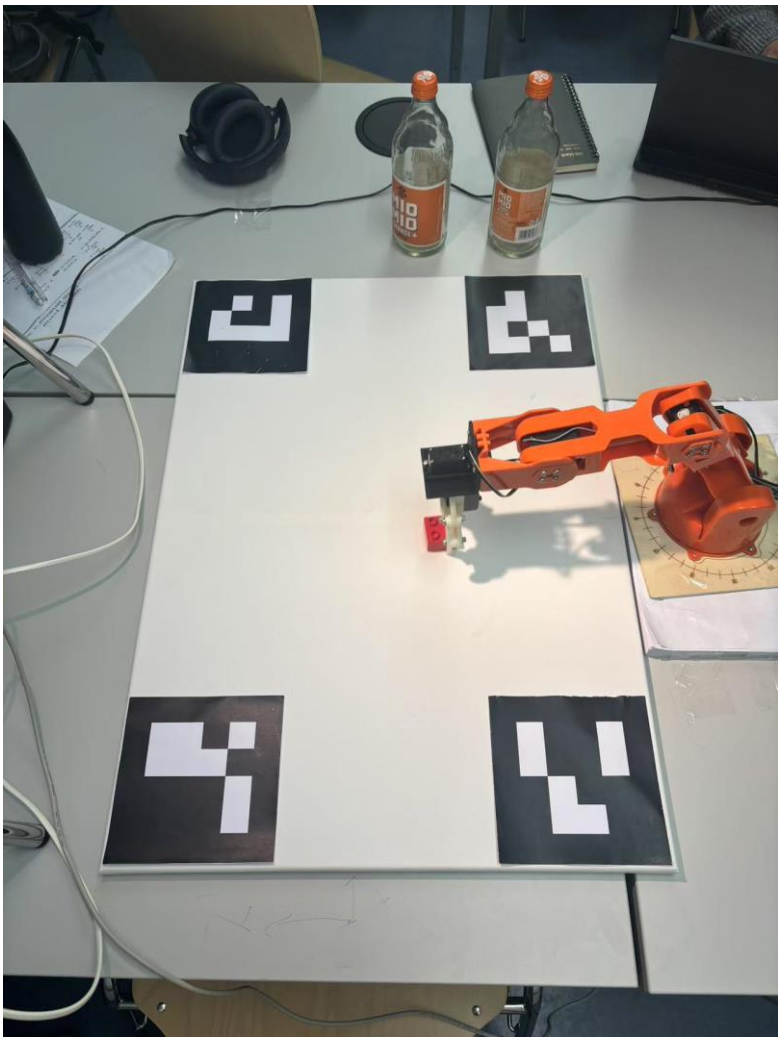
	Input Value	Actual Value	Error Value
X Coordinate	110	109	1
Y Coordinate	295	295	0
Z Coordinate	1	0	1
Angle	52	52	0

The results from the ten sets of random-point tests on the robotic arm reveal a degree of discrepancy between the actual values and the input values in each test. These discrepancies can be attributed to potential measurement errors as well as the inherent inaccuracies of the robotic arm. The findings suggest that the magnitude of error varies across different coordinate positions. To enhance the precision and reliability of the robotic arm, we plan to introduce dynamic error compensation tailored to its grabbing actions at various positions.

6.2. Success rate test for grasping the LEGO cube

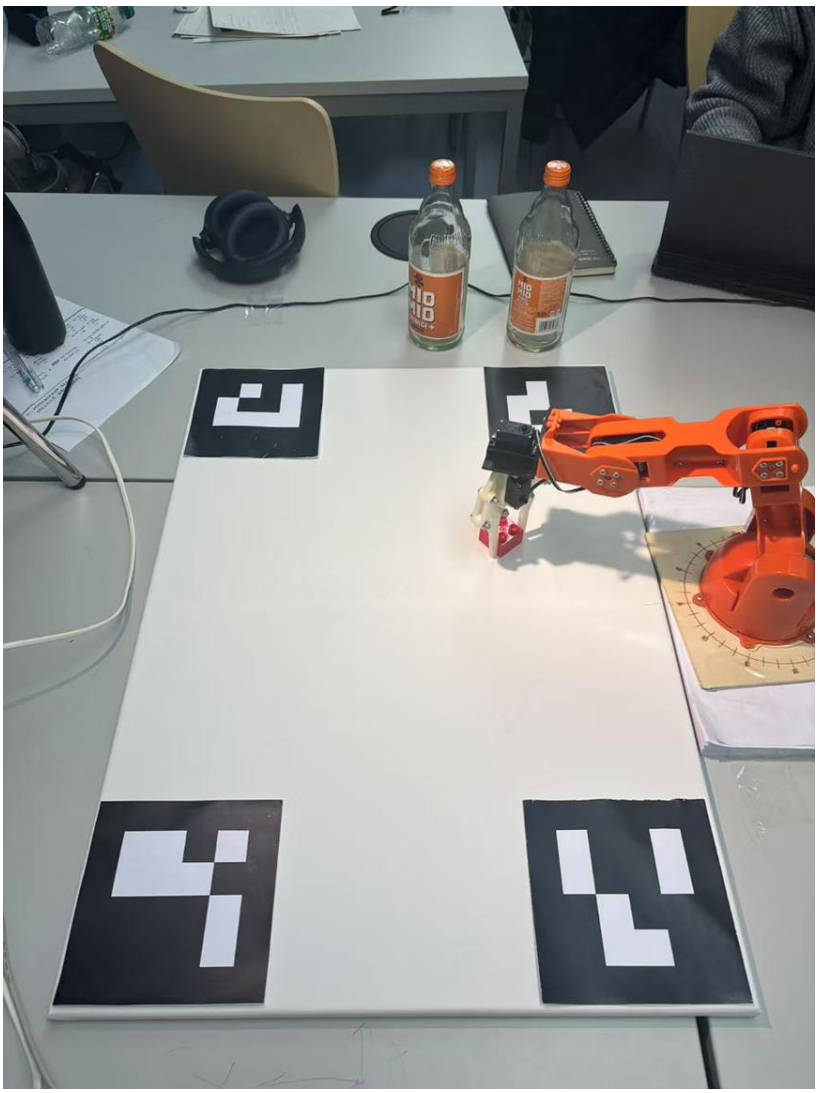
1. 1st set of tests: Coordinates of the cube: [-51, 262, 1], Angle: 37°

Metric	Success(Y/N)
Whether the gripper successfully moves precisely above the cube	Y
Whether the angle of the gripper turned to the same degree as the cub	Y
Whether the grasp is successfully completed	Y
Whether the brick is successfully placed	Y



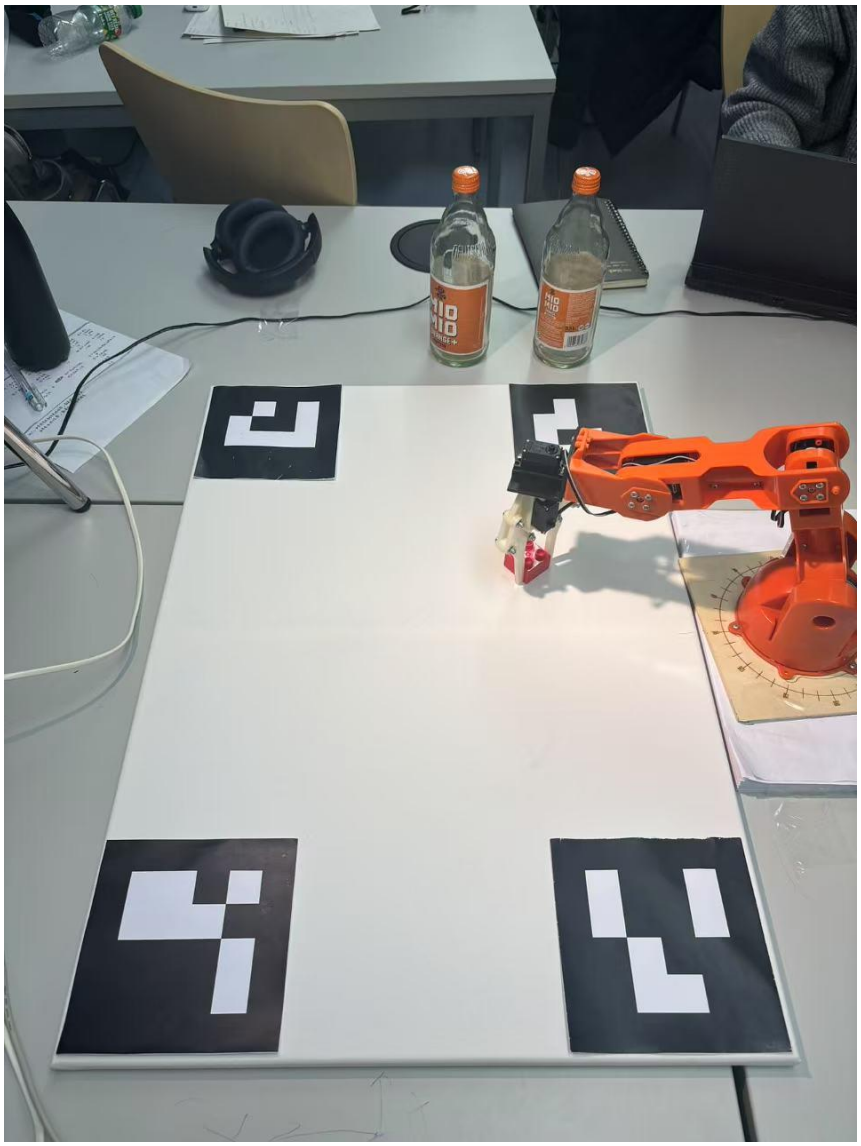
2. 2nd set of tests: Coordinates of the cube: [-129, 272, 1], Angle: 86°

Metric	Success(Y/N)
Whether the gripper successfully moves precisely above the cube	Y
Whether the angle of the gripper turned to the same degree as the cub	Y
Whether the grasp is successfully completed	Y
Whether the brick is successfully placed	Y



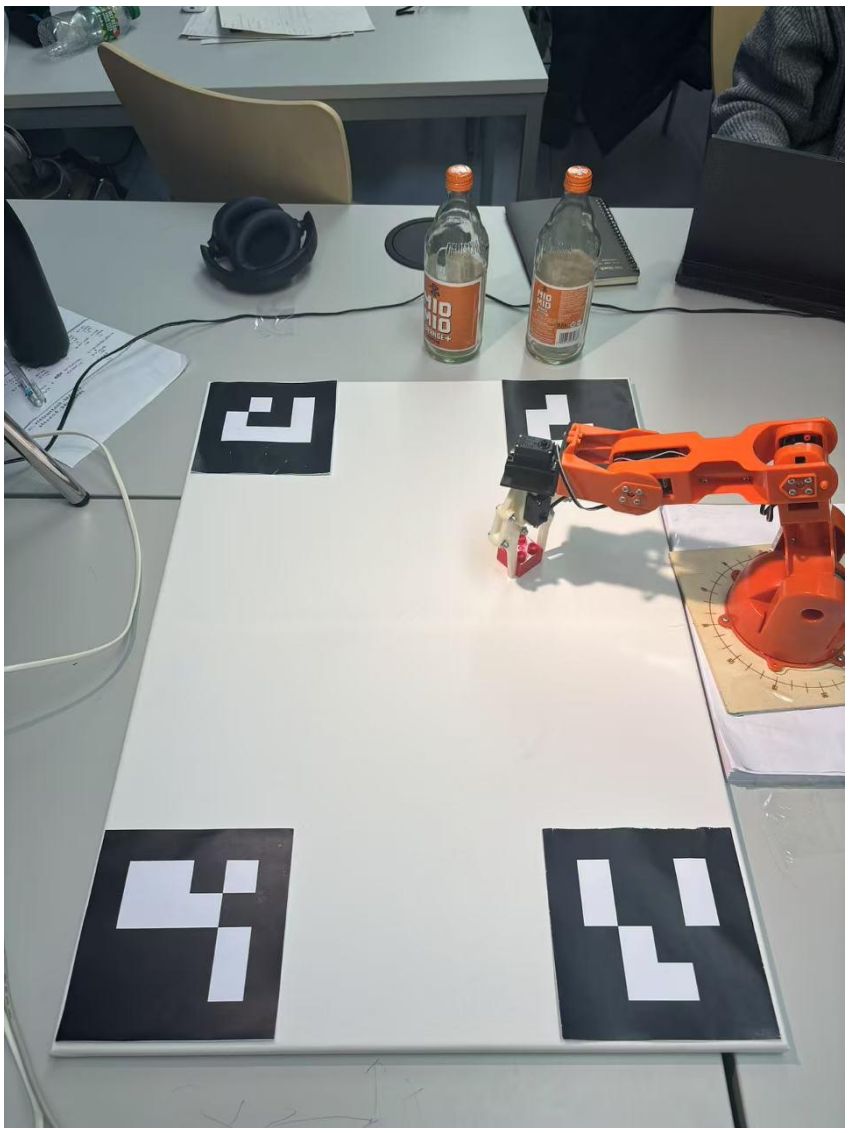
3. 3rd set of tests: Coordinates of the cube: [77, 245, 1], Angle: 24°

Metric	Success(Y/N)
Whether the gripper successfully moves precisely above the cube	Y
Whether the angle of the gripper turned to the same degree as the cub	Y
Whether the grasp is successfully completed	Y
Whether the brick is successfully placed	Y



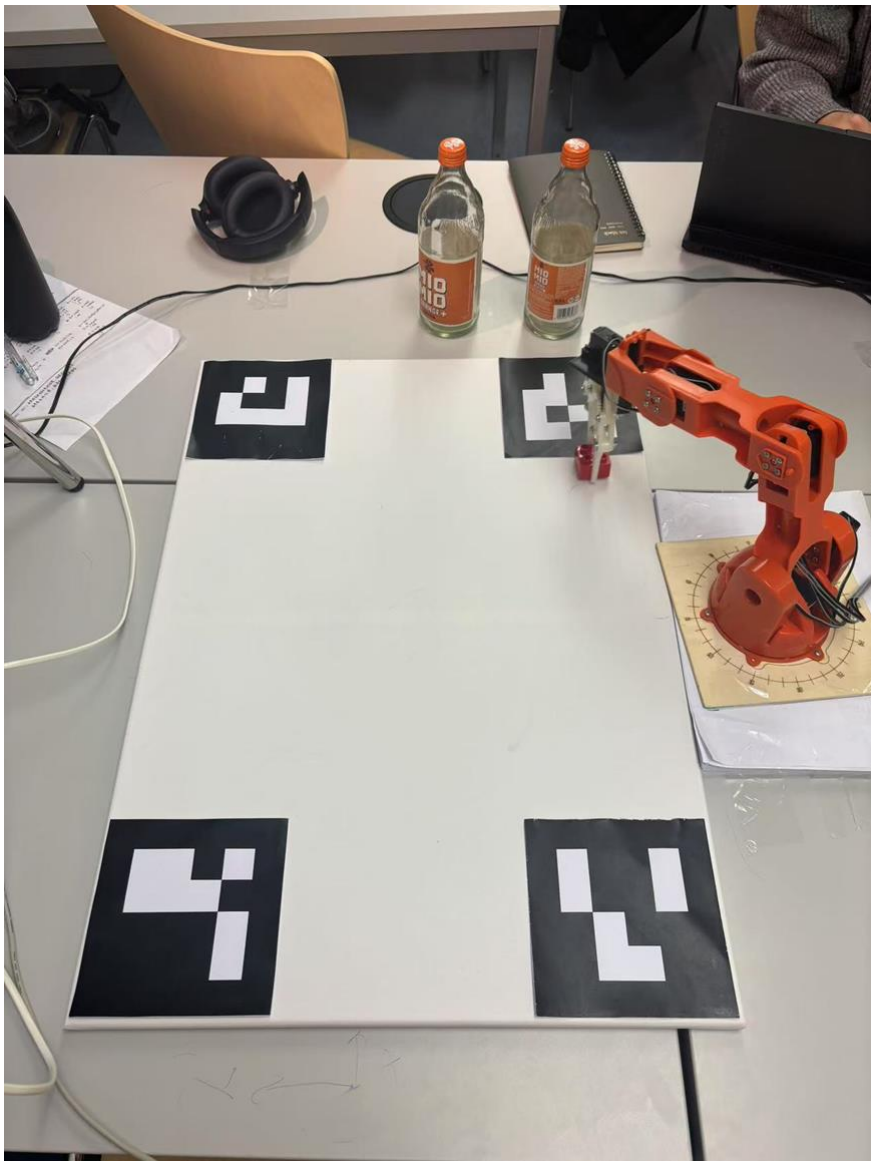
4. 4th set of tests: Coordinates of the cube: [91, 271, 1], Angle: 87°

Metric	Success(Y/N)
Whether the gripper successfully moves precisely above the cube	Y
Whether the angle of the gripper turned to the same degree as the cub	Y
Whether the grasp is successfully completed	Y
Whether the brick is successfully placed	Y



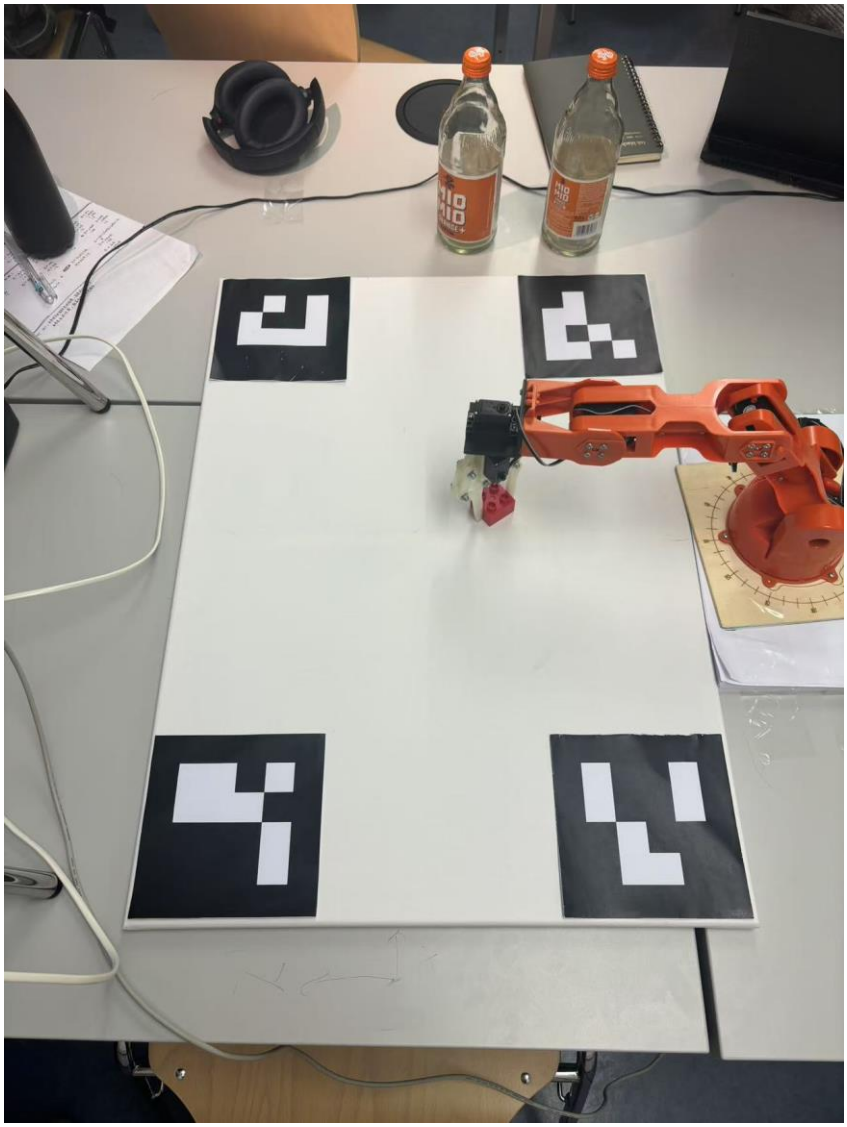
5. 5th set of tests: Coordinates of the cube: [-20, 303, 1], Angle: 8°

Metric	Success(Y/N)
Whether the gripper successfully moves precisely above the cube	Y
Whether the angle of the gripper turned to the same degree as the cub	Y
Whether the grasp is successfully completed	Y
Whether the brick is successfully placed	Y



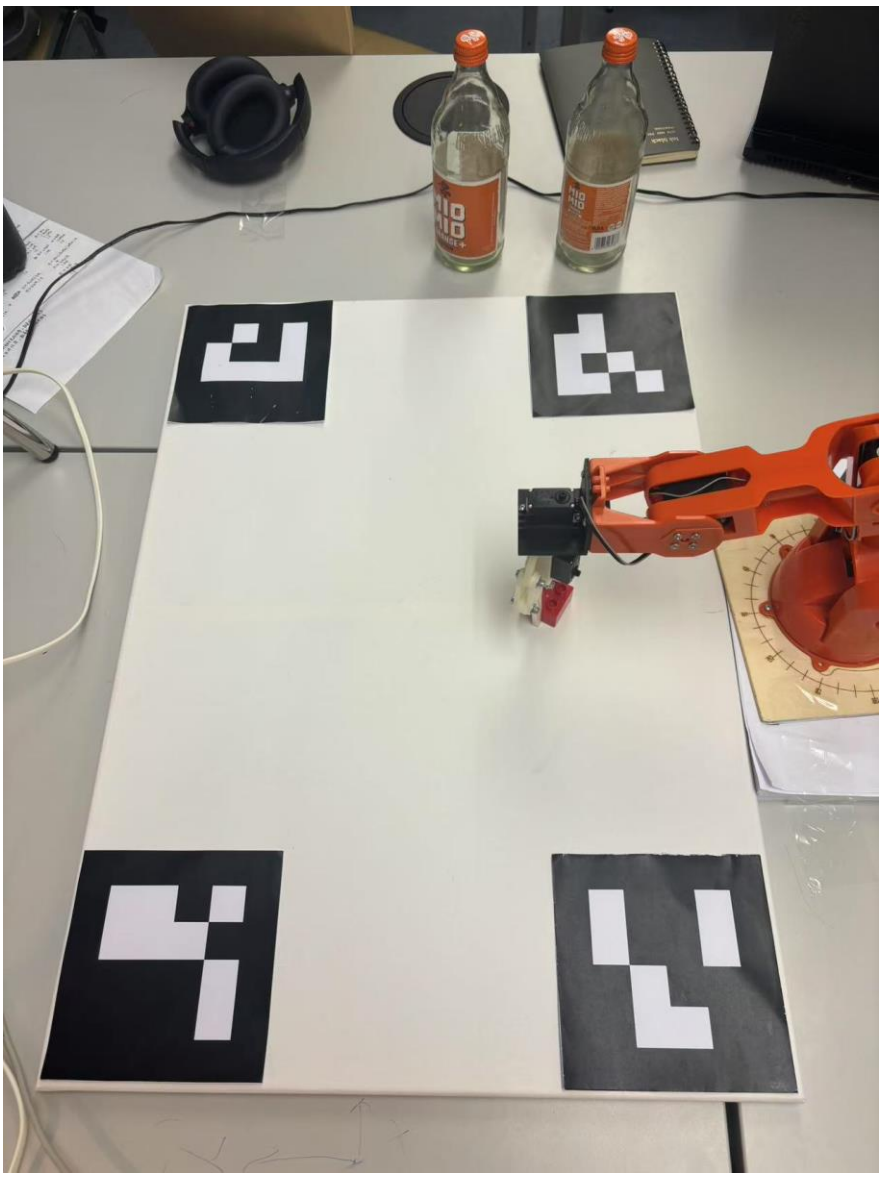
6. 6th set of tests: Coordinates of the cube: [-14, 293, 1], Angle: 51°

Metric	Success(Y/N)
Whether the gripper successfully moves precisely above the cube	Y
Whether the angle of the gripper turned to the same degree as the cub	Y
Whether the grasp is successfully completed	Y
Whether the brick is successfully placed	Y



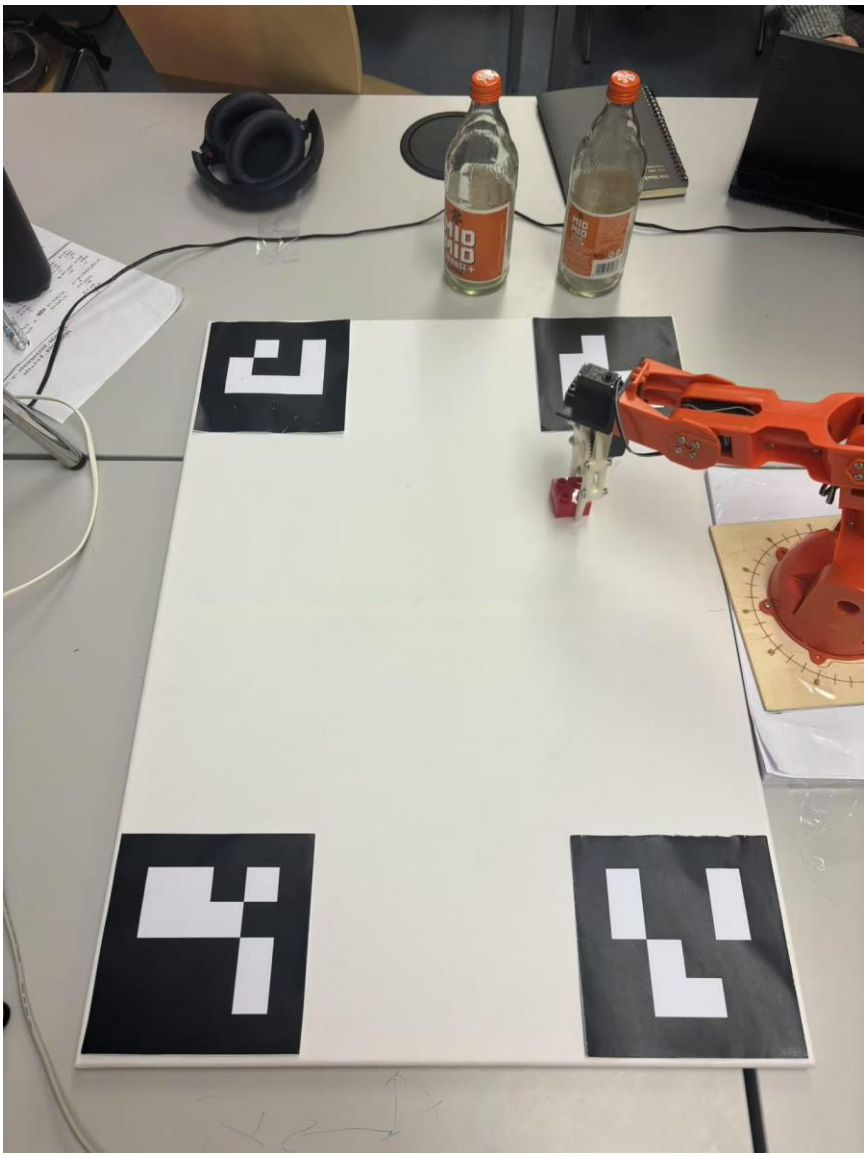
7. 7th set of tests: Coordinates of the cube: [-76, 293, 1], Angle: 36°

Metric	Success(Y/N)
Whether the gripper successfully moves precisely above the cube	Y
Whether the angle of the gripper turned to the same degree as the cub	Y
Whether the grasp is successfully completed	Y
Whether the brick is successfully placed	Y



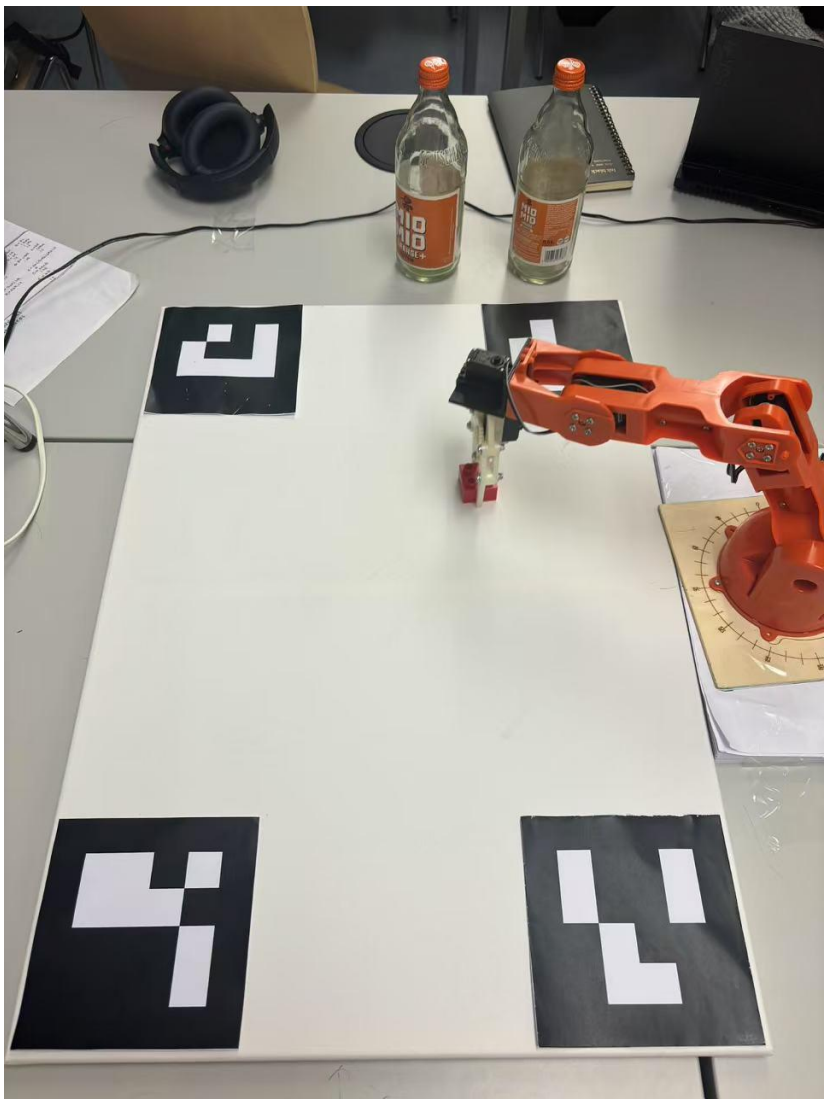
8. 8th set of tests: Coordinates of the cube: [77, 242, 1], Angle: 89°

Metric	Success(Y/N)
Whether the gripper successfully moves precisely above the cube	Y
Whether the angle of the gripper turned to the same degree as the cub	Y
Whether the grasp is successfully completed	Y
Whether the brick is successfully placed	Y



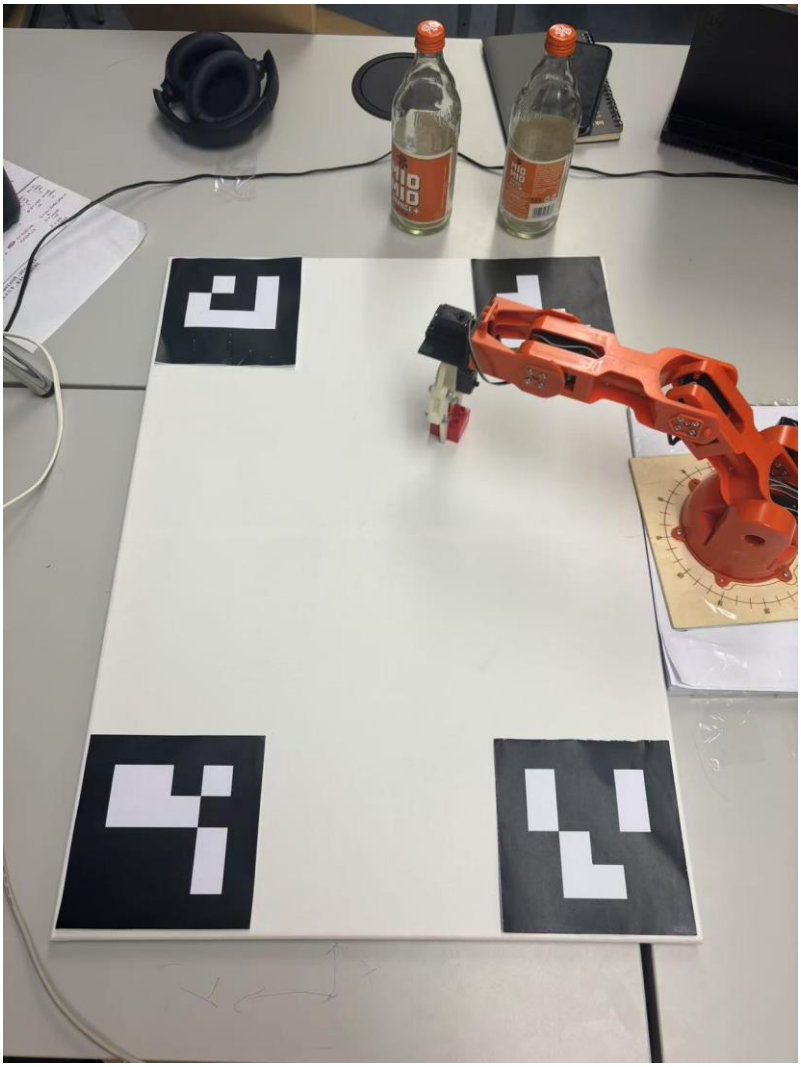
9. 9th set of tests: Coordinates of the cube: [79, 284, 1], Angle: 87°

Metric	Success(Y/N)
Whether the gripper successfully moves precisely above the cube	Y
Whether the angle of the gripper turned to the same degree as the cub	Y
Whether the grasp is successfully completed	Y
Whether the brick is successfully placed	Y



10. 10th set of tests: Coordinates of the cube: [110, 295, 1], Angle: 52°

Metric	Success(Y/N)
Whether the gripper successfully moves precisely above the cube	Y
Whether the angle of the gripper turned to the same degree as the cub	Y
Whether the grasp is successfully completed	Y
Whether the brick is successfully placed	Y



From the results of these ten sets of random-point tests on the robotic arm, we observed a 100% success rate in the tests. However, the robotic gripper did not perfectly align with the LEGO bricks during the grabbing process. These minor misalignments can be attributed to two main factors.

First, our use of the Canny operator for edge detection, combined with non-vertical lighting, caused shadows to be cast on the LEGO bricks, leading to slight deviations in detecting their true center. This shadow-induced offset resulted in small inaccuracies in the measured positions.

Second, the robotic arm itself, due to its gear-based structure, has inherent precision errors that cannot be entirely eliminated. These structural limitations contribute to slight inaccuracies in its movements.

Together, these factors explain the minor misalignments observed during the grabbing process.

7. Future Directions

In this project, we have successfully developed a monocular vision-based robotic arm grasping program that meets our initial objectives. However, as our understanding deepens and the system evolves, we have identified significant areas for improvement. To further enhance the accuracy and performance of the vision system and robotic arm, the following directions are proposed:

1. Replacing the robotic arm:

The current Arduino Tinkerkit Braccio robotic arm, while adequate for basic operations, exhibits limitations in precision, with errors exceeding 1 cm. For applications requiring high accuracy, replacing it with a more advanced robotic arm is essential. Options such as the Universal Robots UR3e or the KUKA LBR iiwa are known for their superior precision and reliability, making them suitable candidates for improving the system's overall performance.

2. Introducing stereo or depth cameras:

Our current monocular camera can only capture two-dimensional information, limiting the system's depth perception. Replacing it with a stereo camera or a depth camera, such as the Basler blaze series, would enable the system to acquire high-precision 3D depth data. This enhancement would significantly improve object recognition accuracy and the success rate of grasping tasks.

3. Applying YOLO for object detection:

YOLO (You Only Look Once) is a state-of-the-art real-time object detection algorithm capable of identifying multiple objects in a single pass. The latest versions of YOLO offer remarkable improvements in speed and accuracy. Incorporating YOLO into the image processing pipeline would greatly enhance the system's object detection capabilities and efficiency.

4. Utilizing SLAM technology:

Simultaneous Localization and Mapping (SLAM) allows a system to simultaneously build a map of its environment and localize itself within that map. Recent advancements in SLAM algorithms have improved their efficiency and adaptability to dynamic environments. Integrating SLAM into our system would enable the robotic arm to navigate and operate

autonomously in unknown or changing environments, reducing the reliance on predefined setups.

5. Leveraging machine learning for trajectory optimization:

Incorporating machine learning models to optimize the robotic arm's motion planning could greatly improve efficiency and precision. Reinforcement learning algorithms, for instance, can be used to learn optimal trajectories for grasping tasks, reducing wear on the robotic arm and increasing operational speed.

6. Implementing cloud-based data processing:

Cloud-based platforms can provide enhanced computational resources for real-time processing of image data, complex kinematic calculations, and system simulations. By offloading computationally heavy tasks to the cloud, the system could handle larger datasets and improve overall responsiveness, making it more scalable for complex environments.

By exploring and implementing these directions, we aim to significantly improve the system's performance, expand its application potential, and meet the demands of more complex tasks.

8. Conclusion

As we reach the conclusion of our project, we reflect on a journey that has been both intellectually rewarding and demanding. Venturing into the realm of industrial robotics with a focus on a camera-based pick-and-place system has offered us a profound appreciation of the intricate challenges inherent in robotics engineering.

This project allowed us to explore the fascinating fields of kinematics and vision processing, uncovering the detailed mechanics behind robotic motion and the seamless interplay of components to achieve precision and fluidity. Our deep dive into vision processing further expanded our understanding of how robots perceive and interpret their surroundings, enabling them to make informed decisions and perform deliberate actions.

Through hands-on experience, we have gained a far-reaching understanding of robotics, bridging the gap between theoretical knowledge and practical application. Designing and implementing a robotic system that interacts effectively with its environment demanded not only technical proficiency but also adaptability and problem-solving skills. The integration of kinematic principles with vision processing has been central to our success, enabling the development of a system that is both mechanically efficient and intelligently responsive.

Looking back, we take pride in the progress we have made, from conceptualizing the design to overcoming technical challenges in programming and calibration. Every hurdle we faced contributed to a deeper learning experience, reinforcing the value of teamwork, determination, and the practical use of theoretical foundations.

Ultimately, this project has been an invaluable learning journey, not only in the field of robotics but also in collaboration, critical thinking, and perseverance. As aspiring engineers, we emerge from this experience more knowledgeable, confident in our abilities, and excited to tackle future challenges that await us.

9. Contributions Breakdown

The names are listed alphabetically, without implying any ranking of contributions.

Chaoyue Ni:

1. Conducted tests, analyzed the results to identify and address potential issues and evaluated the sources of errors.

2. Assisted team members with using ArUco Markers for the localization of LEGO bricks.

Chenyi He:

1. Utilized ArUco markers to achieve precise localization by calculating projection matrix.

2. Calculated the rotation angle of LEGO bricks.

Hongming Yang:

1. Implemented the testing and validation

2. Analyzed the state of the art solutions

3. Engaged in the vision part and image processing.

Xinqiang Zhang:

1. Designed LEGO bricks detection algorithm

2. Revised vision code to avoid error detection

3. Finished related test and report

Xuefei Shi:

1. Developed and implemented of inverse kinematics code

2. Integrated inverse kinematics with machine vision system

3. Calibrated the robotic arm's internal deviations

4. Compiled the final experiment report.