CS4278/CS5478 Final Report
Intelligent Robot Arm

Nguyen Quoc Anh **Pham Quang Minh** **Hoang Le Tri Cuong**
A0274968E A0170723L A0196640Y
National University of Singapore
{e1124714,e0196678,e0389126}@u.nus.edu

Abstract

Our project aims to use a robotic arm to detect and grasp a bottle of water and move it to a designated spot. We run our program on a Raspberry Pi 4 with one camera module. The degree of freedom of our robotic arm is 3 as there are 3 rotational motors to move the arms and 1 motor for grasping with the end effector. We utilized the yolov8 [8] for bottle detection. After coordinates mapping, calibration, and planning, we were able to get the robotic arm to grab and deliver the bottle with a 65% success rate and minimal error. For the code repository, see https://github.com/BatmanofZuhandArrgh/GimmeAHand, and all demo videos, see https://shorturl.at/hlCJR

1 Proposal

Recent advancements in intelligent robots hold the potential to significantly enhance assistive technology for seniors and individuals with disabilities. For instance, individuals with conditions like Cerebral Palsy, characterized by a spectrum of motor impairments, could greatly benefit from real-time object detection and grasping capabilities offered by intelligent robot arms, assisting in detecting, grabbing, and delivering objects to the user.

As such, our project aims to develop a robotic arm to detect and grasp targeted objects and move the objects to the user. We used a camera and object detection to identify the desired object to grab. Then, we converted the object coordinates in the image to the configuration space. At last, we wrote a control modules to command the robot to reach that configuration and grasp. However, due to certain resource and time constraints, we simplified our goals by making certain design choices. They are as follows:

- As our robotic arm uses two-finger end-effector, there are not a lot of objects that it can grasp. As such, something cylindrical is preferred, in our case, a small water bottle.
- We also utilize only one camera as our sole sensor of the environment. Therefore, we are not able to ascertain the depth of the targeted object. To simplify this, we opt to provide the target depth information to the system
- We have also fixed the position of the user, where the robotic arm will deliver the object.

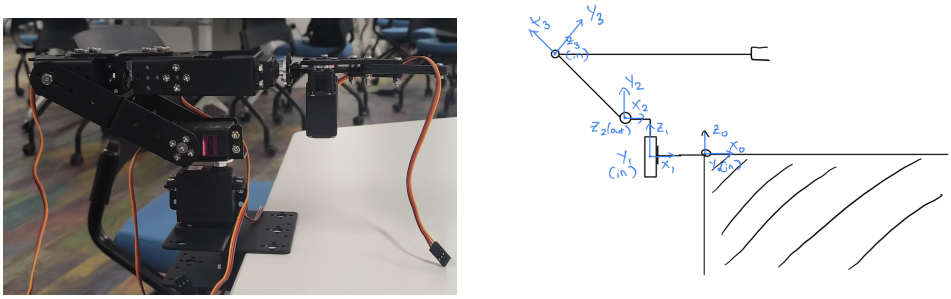
2 Implementations and Challenges

2.1 Actuators and Kinematics

Although usually when developing a robot, researchers design their own parts and have them manufactured or 3D printed, for this project, we purchased a pre-made robot arm from Shopee ([4]), and assembled them like in Figure1a. Although such robot kits are commonly sold on many

e-commerce sites, it was impossible to enquire the robot's specifications (specs) from any sellers or even traced the parts to their manufacturer. Eventually, after measuring the parts ourselves and verify them, we found the rare and most accurate specs of separate parts on [5].

For the kinematics of the robot, we assigned the coordinate systems to rotational joints of the robot like in Figure1b. For our purpose, we ignored the motor that rotate the end effector (not the motor that generate grasp). Therefore, we have 3 motors/rotational joints, assigned index 1,2,3 and EE (End-Effector). Planning and controlling the robot also requires the DH-parameters [19], so we both calculated them using the found specs, and measured them in real life, then confirmed that the two had minimal error. The kinematics for this particular design is recorded in Figure8



(a) Design (b) Coordinate System

Figure 1: Robot Arm

2.2 Computer

For our computing module, we chose to use the Raspberry Pi 4B 8G - an affordable and powerful computing device that are widely used in robotics. Its small size, modular design, and general-purpose input/output pin layout makes it one of the most suitable choice for our application

In order to run the object detection program in the Raspberry Pi 4(RPI), we have reinstalled the 64-bit version of Debian OS so that it can runs the PyTorch library.

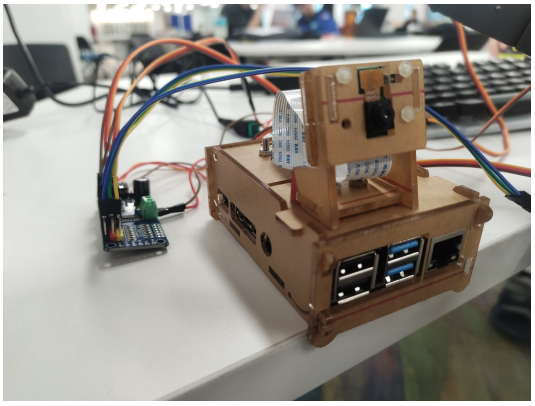


Figure 2: RPI 4B, RPI Camera v1 and PCA9685 Servo Driver

2.3 Motor and Motor Driver

We use the MG996R Servo Motor [6] for motors. For measurements, we used [3] as the most accurate model and used [7] for other specs. Since the stall torque at 6V of the motor is 11kmf.cm, it can reasonably be used to control a light robot with short arm to pick up a light plastic bottle.

The servo motor requires only 5V of electricity input, which can technically be connected directly to RPI power pins. However, since we need to use 4 motors in total and the current drawn directly

through the RPI board would heat up the computer, causing possible damage, we use the Adafruit PCA9685 Pulse Width Modulator [1]. This servo driver draws 3V of power and instructions directly from RPI, while controlling up to 16 motors. These motors' powers are connected directly through a 6V-1.0A AC-DC adapter to the 240V regular AC power source. Adafruit provided a python API to control the motors, including a function that maps pulse width values to the target angles [2]. Angular range of the motors is stated on the seller's website to be 180°, but we found the irl range to be about 220°, managing to calibrate the motor to be able to move in the range of 208°.

2.4 Camera and Calibration

2.4.1 Camera setup

The robot use a single camera that is located beside the base of the robot and tilted slightly upward to capture the target image and determine its location. As this is a system with only one vision source, it is very difficult to estimate the target depth information and thus we accepted this as one of the limitations of our system and have opted to provide the target depth information beforehand to the system so that it can infer the target coordinate in the real world from its location in the image

2.4.2 Camera intrinsic matrix calibration

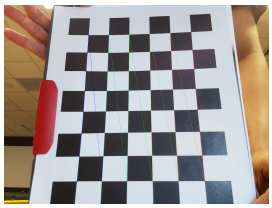


Figure 3: Checkerboard pattern used for camera calibration

We find the camera intrinsic matrix by using the capturing the images ($w_{calib} \times h_{calib} = 2592 \times 1944$ pixels) of a checkerboard at different angles and use the OpenCV library to calculate the intrinsic matrix. From the output intrinsic matrix, we found the focal lengths to be $f_x = f_y = 2500$

As the dimension of the camera footage of the object detection program is different ($w_{cam} \times h_{cam} = 640 \times 480$ pixels), we need to modify the measured intrinsic matrix so that it can be used for the coordination calculation during the operation of the robot. Thus, our final intrinsic matrix is calculated to be:

$$A = \begin{bmatrix} f_x \times w_{cam}/w_{calib} & 0 & (w_{cam}-1)/2 \\ 0 & f_y \times h_{cam}/h_{calib} & (h_{cam}-1)/2 \\ 0 & 0 & 1 \end{bmatrix}$$

The matrix above scales the focal length of the first two columns to the new camera resolution, while the values of the third column make sure that the center of the camera lens is the origin point of the camera's world coordinate.

2.4.3 Camera extrinsic rotation matrix calculation

As the camera is slightly tilted upward (with angle α) in our setup, we are interested in figuring out the rotation portion inside the extrinsic matrix for our calculation. We were able to calculate the theoretical rotation matrix to be:

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -\cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & -\cos\alpha \end{bmatrix}$$

From the camera calibration step above, we were able to find the output extrinsic matrix when the calibration checkerboard is standing upright and verified the angle of the camera to be $\alpha = 20°$

2.5 Object Detection

For real-time object detection on an edge device, the robot arm needs a model that is light, fast and accurate. We uses yolov8 [8], the current state-of-the-art benchmarked on COCO [20], specifically yolov8-nano to be run on RPI's processor at about 1 fps. After experimentation, we find that for many bottle objects, the model mistaken them for the class "vase", so we include both "bottle" and "vase" to be detected, and put the confidence threshold at 0.25, to accommodate for lower certainty predictions. We've also found that putting a bottle cap onto the capless bottles makes it more likely to be detected as a bottle. See Figure4.

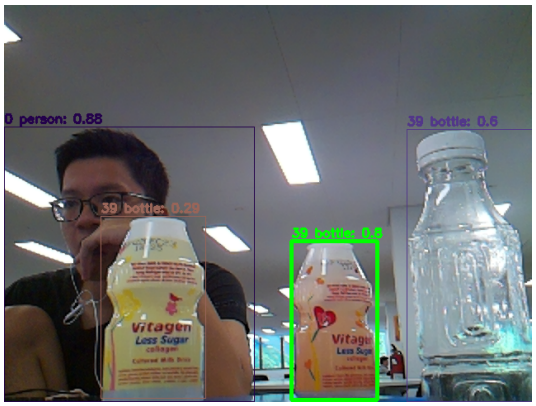


Figure 4: Target objects detection

2.6 Coordinate Mapping

2.6.1 World coordinate calculation

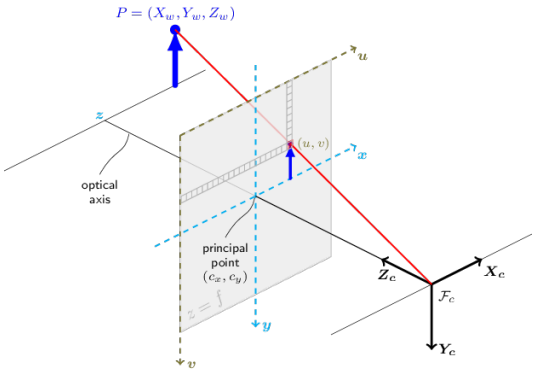


Figure 5: Pinhole camera model (Source: OpenCV)

From the object detection, the bounding box of the target in the camera are given as the pixel coordinates of the top left and bottom right corners u_1, v_1, u_2, v_2. The robot calculates the middle point of the bounding box to be the pixel coordinate of the target in the image:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} (u_1 + u_2)/2 \\ (v_1 + v_2)/2 \end{bmatrix}$$

The robot then uses the intrinsic and extrinsic matrices that we found from the camera calibration steps to calculate the scaled world coordinate of the target:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = R^{-1} A^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

Given that we manually provide the depth z_{target} of the target compared to the camera position and the position offset $(x_{offset}, y_{offset}, z_{offset})$ of the camera compared to the base of the robot, the robot is then able to calculate the world coordinate of the target compared to world coordinate origin (at the base of the robot Figure1b) to be:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \times \left| \frac{z_{target}}{z'} \right| + \begin{bmatrix} x_{offset} \\ y_{offset} \\ z_{offset} \end{bmatrix}$$

2.6.2 Operational coordinate calculation

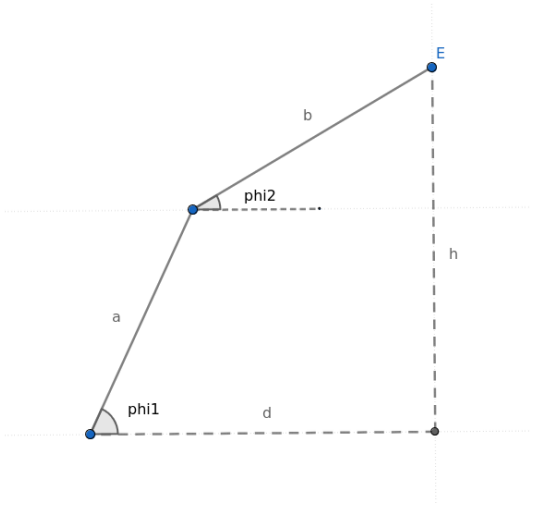


Figure 6: Model used for calculating the angles of the two arms

For ease of communication, we call the link connecting joint 2 and 3 arm 1, and the link connecting arm 3 and the EE arm 2.

- Motor 1 (rotate the arms around the vertical axis): $\theta_1 = 0$ when the robot arm is in the middle, the spinning axis is the positive y-axis of the calculated world coordinate
- Motor 2 (rotate the first and second arms): $\theta_2 = 0$ when the arm 1 is vertical, the spinning axis is the positive x-axis of the calculated world coordinate
- Motor 3 (rotate the second arm): $\theta_3 = 0$ when the arm 2 is aligned with the arm 1, the spinning axis is the negative x-axis of the calculated world coordinate

Given that the robot knows the (x, y, z) coordinate of the target, the robot needs to calculate the operational coordinates (the angles of the motors) so that the endpoint of the arm is at the location of the target.

We can easily calculate the first operational coordinate as:

$$\theta_1 = \arctan \frac{x}{z - a_0}$$

To calculate the other two operational coordinates, we simplify the calculation of the angles of the arms by converting the problem into the problems in Figure 6 where we can calculate the angles of the two arms ϕ_1 and ϕ_2 compared to the horizontal line with the following system of equation:

$$a = a_2,\ b = a_3$$
$$d = \sqrt{x^2 + (z - a_0)^2} + a_1$$
$$h = y - d_2$$

$$a\cos\phi_1 + b\cos\phi_2 = d$$
$$a\sin\phi_1 + b\sin\phi_2 = h$$

Solving the above will give us two possible solutions, and we opt to choose the one where the two arms are above ground which is:

$$\phi_1 = +90^\circ + \arctan\frac{h}{d} - \arcsin\frac{d^2+h^2+a^2-b^2}{2a\sqrt{d^2+h^2}}$$
$$\phi_2 = -90^\circ + \arctan\frac{h}{d} + \arcsin\frac{d^2+h^2+b^2-a^2}{2b\sqrt{d^2+h^2}}$$

We then can convert these two angles to the final two operational coordinates:

$$\theta_2 = \phi_1 - 90^\circ$$
$$\theta_3 = \phi_1 - \phi_2$$

2.7 Controls

2.7.1 Calibration

To rotate the motors through the Adafruit's Python API, the motors were input with Pulse-Width Modulation (PWM) signals. We calibrated the motors and found a $MIN\_PULSE = 400$ and $MAX\_PULSE = 2700$ which are the safe limits for our MG9965R servos that provide a 0° to 208° of turn.

Then, we proceeded to do some calibration from the operational coordinates in part 2.6.2 to input for the servos (0° to 208°). We first find the correct servo inputs offsets corresponding to $\theta_1 = 0^\circ$, $\theta_2 = 0^\circ$, $\theta_3 = 0^\circ$, which are servo 1 = 98°, servo 2 = 158°, servo 3 = 180°, respectively. We then have a formula to convert from operational coordinates to servo input, which is:

$$servo\_input = offsets + direction \times op\_coord$$

The direction is -1 or 1 depending on whether the servo's rotation axis direction is opposite or parallel to the rotation axis direction of the operational coordinate.

We also put some limit on how much each servo can rotate, because if the torque is too high when the servo gets obstructed by itself or the ground, the servo heats up a lot which can damage itself. Servo 2 cannot turn below 60° and servo 3's limits are dependent on the angle of servo 2.

2.7.2 Naive Controls

We first tested the most naive scheme for controlling the arm. The full flow is as follows in Figure9. The process starts with running object detection model on frames streamed from the camera. RPI will stopped as soon as it detects bottle (or vase) at a confidence score more than 0.25, saves the frame and run coordinate mapping to get and log the target angles for the 3 joints. Since the width of the grasp is only 55mm [5], we picked a small empty plastic bottle. Although since the camera is pitch about 20° upwards, the short bottles need to put onto a box to fit into the camera's Field-of-View (FOV). See Figure**??**b.

We set 3 fixed pre-defined configurations for the robot. DEFAULT_ONLINE is when the arm 1 and 2 hold themselves up. DEFAULT_OFFLINE is both arms are rested in a stable position. USER_LOCATION is the target location on the table for delivery, where joint 1 is at 150°.

Naive controls is designed as follows. Firstly, the robot will go in DEFAULT_ONLINE, open EE, and rotate joint 1 to angle the arm towards the detected object, creating a singular 2D vertical surface with both arms and the object in it, like in Figure 6. Secondly, joint 2 and 3 rotate will into its target angle, in that order, and close EE. Presumably, the bottle would be picked up and robot retracted to DEFAULT_ONLINE position (by moving joint 3 first then 2), then rotate joint 1 to USER_LOCATION, lower its arms to drop the bottle, before moving to DEFAULT_OFFLINE position and relaxes all its motors. The whole process can be seen here [15]. The order in which joints are rotated was found through trial and error experiments, to avoid obstruction with itself, the ground and the target objects.

2.7.3 Segmented Controls

Due to the accumulated errors, the EE will usually missed by very little [14] (although sometimes, we still get lucky with minor deviations [13]). And as the EE goes down with the default angular speed of joint 3, the very light bottle gets hit by part of the EE and bounces off. Therefore, we remediate this by first slows down joint motion, by waiting a small interval for every degree rotated. Then to reach to the object for grasping, the robot employs what we call segmented reach. For joint 2 and 3, the difference between the current angles (DEFAULT_ONLINE) and target angles were divided into 10 sub-target angles, equally spaced. Then joint 2 and 3 would alternate and move to each sequential sub-targets, creating a very delicate motion that slowly slide the object into the EE and visibly improve the rate of success. See [16, 17]

2.8 Fine-tuning

As we tested the robot arm, we found that defining the length of the second arm (the distance between motor 3 and the endpoint) is crucial in the accuracy of the arm in grasping the target. We initially defined this length to be from the joint 3 to the innermost part of the claw, but we found that the arm consistently overestimates the target location and makes the target fell over, or deviate the location of the target to one side which makes the claw hit and get stuck at the top of the target while trying to grasp the target [9, 10, 11]. We then defined the length to be from the motor to the tip of the claw, but this time the claw either underestimate the location of the target or tends to grasp the center of the target with the tip of the claw, pushing it just a little bit away from the grasp. As such, we decided to fine tune the length so that the endpoint is defined to be a point in the middle of the claw, and this led to a significant increase in the success rate of the robot. See [18] for experiments after fine-tuning.

3 Testing and Evaluation

3.1 Coordinate Mapping Evaluation

To test the accuracy of the robot's coordinate mapping, we set up an experiment where we randomly set the target at certain depths that are provided to the robot for it to calculate the coordinate of the target for us to measure the error of the estimation. After 10 measurements, the mean and standard deviation of the coordinate estimation is as followed:

Axis	Mean of error (unit: mm)	Standard deviation of error (unit: mm)
x (horizontal)	1.47	13.99
y (vertical)	16.08	14.54

From the results above, we see that the robot was able to estimate the horizontal coordinate very well, with the average error within a couple of millimeters. While the average estimation error of the vertical axis is noticeably higher at 16 millimeters, it only affects the grasping height of the claw on the target and does not prevent the claw from reaching the target and putting the target within its grasp. The standard deviation of error of both axis is around 1 centimeter, which is within the acceptable margin for the claw to be able to grasp the target without missing it. The measured standard deviation of errors are also affected by the lack of highly accurate measurement tools and experimental setup, making it theoretically larger than its true values.

3.2 Robot Operation Evaluation

For our final evaluation of the robot as a whole, we tasked it with the task of picking up the target that are put at random location at a specific depth. For this experiment, we chose the target to be a small 125ml bottle and we set it at 200mm depth compared to the camera, which is 128mm depth away from the robot. We performed the experiment 20 times and observed the following result:

Number of test cases	20
Number of times the robot successfully picked up the target	13
Number of times the robot was unable to pick up the target	7
Success rate	65%

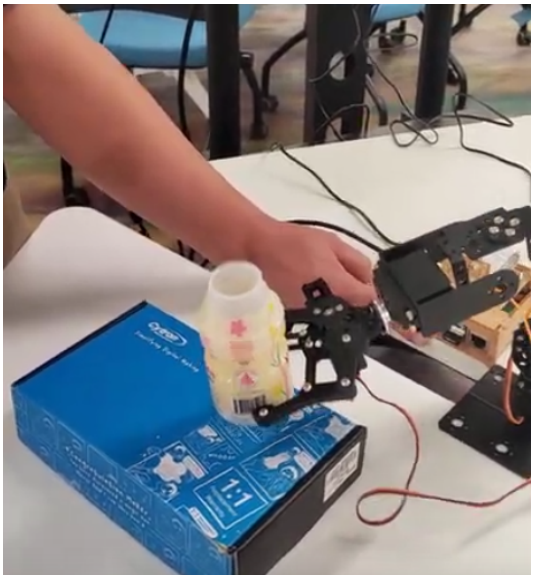


Figure 7: Robot successfully grasped and picked up the target [18]

The robot was able to successfully grasped and picked up a total of 13 times out of 20 tries, reaching the success rate of 65%. In the failure cases, we observed that the main reason why the robot was not able to pick up the target was not due the claw missing its target. Instead, the majority of the failure cases are when the robot successfully put the target within its claw, but due to the claw angle tilting downward, the tip of the claw is at the bottom of the target so when the claw closed to grasp the target, the bottle slip upward and get launched out of the grasp [12]. We believe that with more fine-tuning of the grasping location of the claw, it is possible to further improve the success rate of the robot.

4 Future works, improvements, and other notes

Our robot only uses a one camera system for vision, so possible future works involved replacing this system with a more robust one that can automatically infer the depth of the target without requiring manual target depth inputs

Another improvement of our robot that we can also make is to improve how the robot decides where to grasp the target. Currently our implementation only involves letting the robot grasp the middle point of the target, which will be a problem if the target is in spherical shape or if the best possible grasping point is not at the middle of the object

For our simple robot working in an environment with no obstacles, we could get away with a trivial control process. Even if we were to generate a C-Space for joint 2 and 3, and use simple planning algorithms to find a path, we believe the output would be similar to the aforementioned Segmented Controls (it would be a straight line from source to target configuration in a 2D C-Space of parameters θ_2 and θ_3). However, with obstruction, there should be a planning algorithms implemented to find the fitting control sequence

All of our work was done manually, and so using pre-established software like ROS [21] would possibly save time. Furthermore, with 1 irl robot, work needs to be done sequentially, while using the simulation software accompanied with ROS would enable us to work in parallel. Although, the installation and learning to develop with ROS itself would take a lot of effort.

References

[1] Adafruit 16-Channel 12-bit PWM/Servo Driver - I2C interface - PCA9685. https://learn.adafruit.com/16-channel-pwm-servo-driver?view=all, 2023.

[2] Adafruit CircuitPython driver for PCA9685 16-channel, 12-bit PWM LED and servo driver chip. https://github.com/adafruit/Adafruit_CircuitPython_PCA9685, 2023.

[3] Lonypan's GrabCAD MG996R Servo Motor CAD model. https://grabcad.com/library/servo-motor-mg996r-2/details?folder_id=11413121, 2023.

[4] Shopee DIY 6-DOF Robot Mechanical Arm Kits for Learning Robotics Assembly Kits. https://tinyurl.com/2xh27n52, 2023.

[5] Standard steering gear bracket double-axis servo bracket robot multi-function oblique U-shaped L-shaped one-legged beam bearing. https://tinyurl.com/mr2s77e9, 2023.

[6] Tower Pro MG996R Servo Motor. https://www.towerpro.com.tw/product/mg996r/, 2023.

[7] Tower Pro MG996R Servo Motor Datasheet. https://www.electronicoscaldas.com/datasheet/MG996R_Tower-Pro.pdf, 2023.

[8] Ultralytics yoloV8 Github repository. https://github.com/ultralytics/ultralytics, 2023.

[9] (Video) Failure due to minor deviations 1. https://drive.google.com/file/d/1dkJ-y7NnOXSleBfHszXgT73o16oUXHvy/view?usp=sharing, 2023.

[10] (Video) Failure due to minor deviations 2. https://drive.google.com/file/d/1deFdXZL24Tc4X92joMWnxeV24K1Bzp32/view?usp=sharing, 2023.

[11] (Video) Failure due to minor deviations 3. https://drive.google.com/file/d/1ddO79CwNf4T2JpZSoV6W7UmW-fLXfyOY/view?usp=sharing, 2023.

[12] (Video) Failure due to slippery bottle. https://drive.google.com/file/d/1eRSRMzNaUzIvb9OkN1whdyDyJIeRlNdm/view?usp=sharing, 2023.

[13] (Video) Naive Controls Demo with minor deviations, but still successfull. https://drive.google.com/file/d/1eBjn86UmOKu_DFpFsy8E-xQyGA9Bhvbe/view?usp=drive_link, 2023.

[14] (Video) Naive Controls Failed Demo. https://drive.google.com/file/d/1eBjn86UmOKu_DFpFsy8E-xQyGA9Bhvbe/view?usp=drive_link, 2023.

[15] (Video) Naive Controls Successful Demo. https://drive.google.com/file/d/1eAW2G0CA1eXhKOwbQ9RO1xXcrfkpaQ1J/view?usp=sharing, 2023.

[16] (Video) Segmented Controls Successful Demo 1. https://drive.google.com/file/d/1e98vmabiB1d5uL7HYW0GR3CBdxgCakxN/view?usp=drive_link, 2023.

[17] (Video) Segmented Controls Successful Demo 2. https://drive.google.com/file/d/1dD67-E5y2fpAM4hQMw8769RVJ9V5l_PL/view?usp=drive_link, 2023.

[18] (Video) Segmented Controls Successful Demo Final. https://drive.google.com/file/d/1eQ6K6PYiEYRY3DSOO71mzDSFzmhJ6ZH4/view?usp=sharing, 2023.

[19] J. Denavit and R. S. Hartenberg. A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices. *Journal of Applied Mechanics*, 22(2):215–221, 06 2021.

[20] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[21] Stanford Artificial Intelligence Laboratory et al. Robotic operating system.

Appendix

Joints(i)	a(i-1)	alpha(i-1)	d(i)	theta(i)
1	39.56	0 degs	50.4	var
2	21	90 degs	13	var
3	100	180 degs	0	var
Distance from joint3 to end of end effector: 262.9				
a > Distance (z(i-1), zi) along x(i-1) α > Angle (z(i-1), zi) along x(i-1) d > Distance (x(i-1), xi) along z(i) θ > Angle (x(i-1), xi) along z(i)				
Unit: mm				

Figure 8: DH Parameters

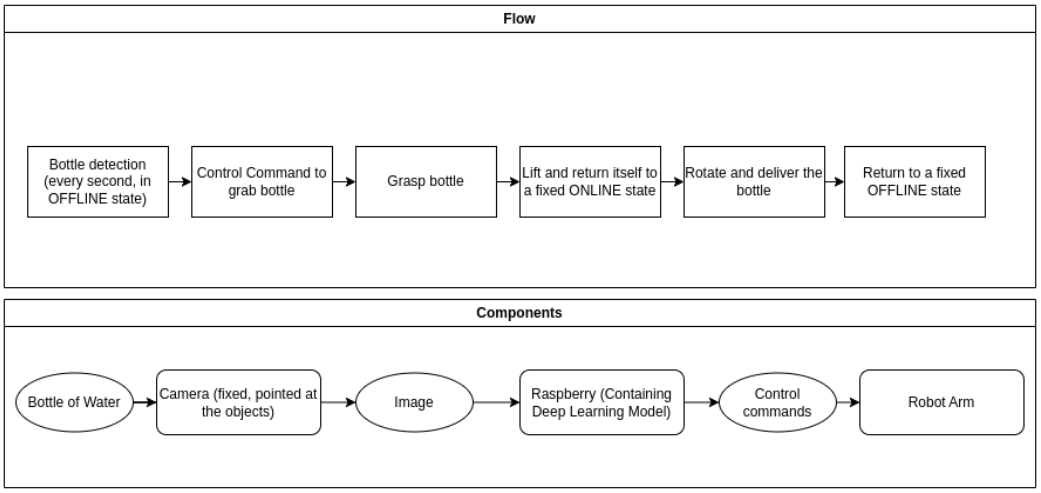


Figure 9: Full Control Flow