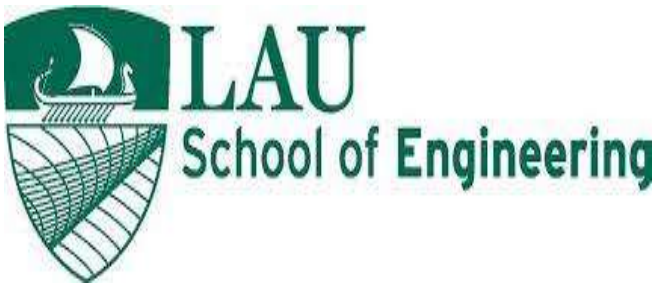
Lebanese American University



Mechatronics System Design II

Final Project

Khalid Kaddoura 202205440
Vartan Fakhreddine 202204238
Rayan Abou Matar 202102932

Contents

Table of figures

Overview / Abstract

This project develops an autonomous trash-collecting rover that integrates real-time computer vision and fuzzy logic control to detect, navigate to, and retrieve lightweight waste items. A YOLOv8 model running on a Raspberry Pi 5 processes Pi Camera v2 images to identify bottles and cans in real time. Detected object coordinates feed into a Mamdani fuzzy logic controller that computes smooth steering and velocity commands, which are transmitted wirelessly to motor drivers and an ESP32-controlled 6-DOF robotic arm the user controls through hand gestures using computer vision.

Project Objectives

- Detect & localize and track bottles/cans in real time
- Navigate using fuzzy logic
- Manipulate 6-DOF arm using hand gestures.

Hardware Components:

- Raspberry Pi 5
- Pi Camera v2
- 6-DOF robotic arm
- ESP32 (for wireless communication)
- DC motors and motor driver (L298N H-Bridge)

Process

To begin, we needed something with good computing power to run the object detection model on, so we went with the raspberry pi 5 as our microcontroller. After that, we need something to capture the environment around the rover with, so we chose the pi camera V2. For our robotic arm we went with a 6 DOF 3D-printed arm that can be mounted on the rover and used to pick up the bottles and cans. Our initial plan was to use flex sensors for controlling the joints of the arm, however that proved to be inaccurate, difficult, and inconvenient so we resorted to computer vision to capture hand movements. We used esp32 microcontroller to actuate the arm through the data received from the laptop via Bluetooth. For our rover motors, we wanted to control them in such a way that would steer the rover towards a detected target, center the object, then make its way to it, fuzzy control was perfect for this task.

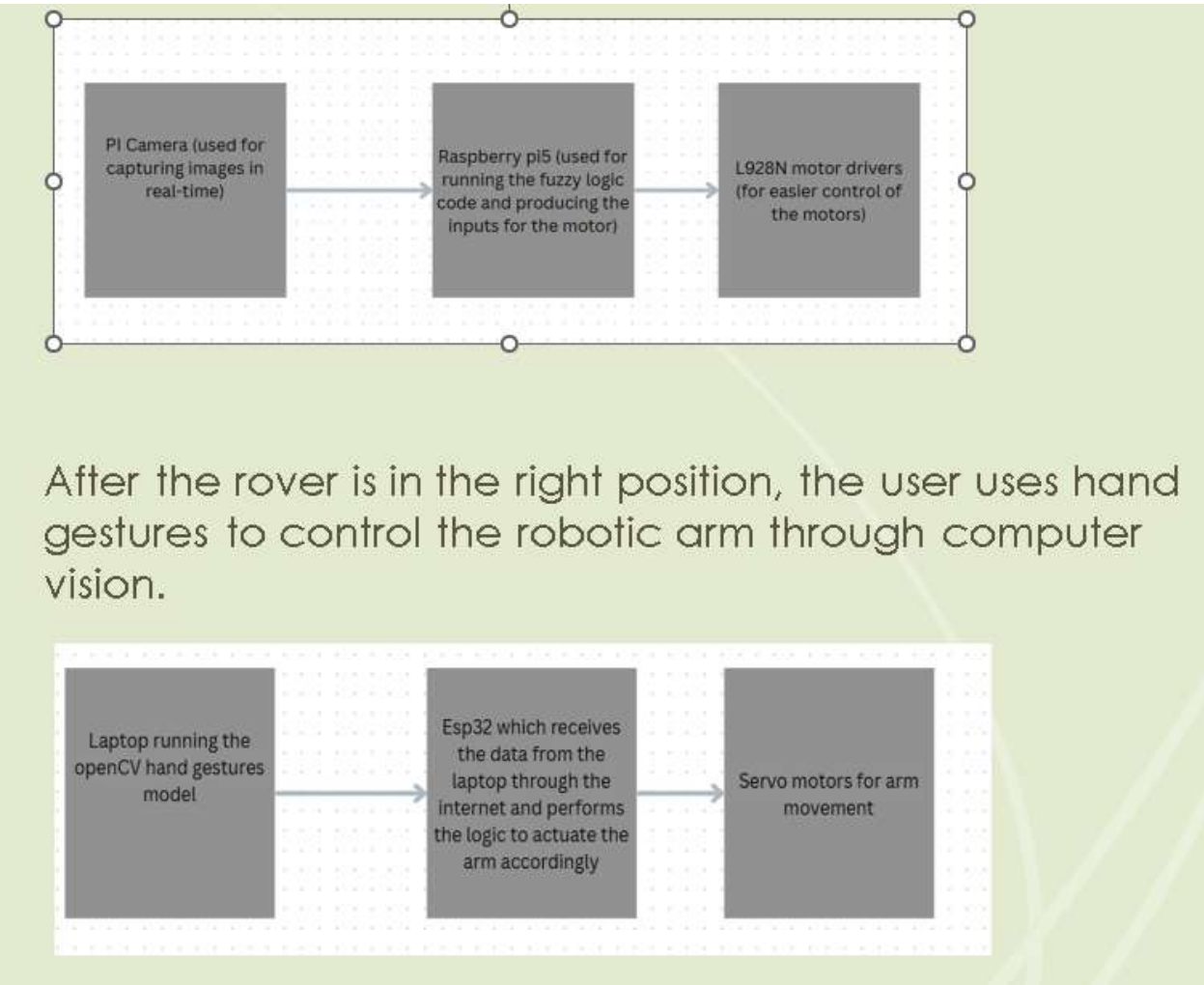


Figure 1:Project flowchart

YOLOv8 Model for Capturing Bottles and Cans

For the object detection module, we selected YOLOv8 nano to guarantee real-time performance on a Raspberry Pi without sacrificing accuracy. We assembled a custom dataset of 3,742 images drawn from multiple publicly available repositories as well as our own captures, ensuring roughly equal representation of two target classes—1,590 cans and 2,590 bottles. Prior to training, every image was uniformly resized to 640×640 pixels. We trained the model in google colab with a batch size of 16 for 25 epochs. Over the course of training, we observed a smooth and steady decline in both localization and classification losses: the training box loss fell from 1.25 → 0.52, cls loss from 1.70 → 0.45, and dfl loss from 1.24 → 1.02, while the corresponding validation losses mirrored these trends, dropping from box loss 1.07 → 0.70, cls loss 1.82 → 0.60, and dfl loss 1.50 → 1.10. Precision for bottles climbed from 0.70 to 0.92 and recall from 0.65 to 0.93; for cans, precision rose to 0.91 and recall to 0.90 by epoch 25. The model's mean Average Precision at IoU 0.5 (mAP@0.5) increased from an initial 0.75 to 0.95, and its stricter mAP@0.5–0.95 metric improved from 0.52 to 0.82, reflecting reliable detection across object scales and IoUs. A normalized confusion matrix reveals that true bottles were correctly identified 91% of the time (with just 7% missed and 2% mislabeled as cans) and true cans 98% (2% mislabeled as bottles), while background false positives remained under 16%. In real-world inference tests on our test set, the model successfully detected a white plastic water bottle on a stairwell twice—once with 0.79 confidence and again with 0.85—despite uneven stone textures and variable lighting; it also correctly recognized two different metal cans in close proximity with confidences of 0.93 and 0.92. The only notable false positive occurred when our pencil case–cushioned calculator was mistaken for a can at 0.73 confidence, underscoring the model's overall resilience but highlighting opportunities for future fine-tuning of hard negatives. On the Raspberry Pi, the final 25-epoch YOLOv8 nano model sustains over 15 FPS at 640×640 resolution, validating its suitability for embedded, low-power applications and confirming that this balanced, meticulously trained model is an effective, high-accuracy solution for on-device can and bottle detection.

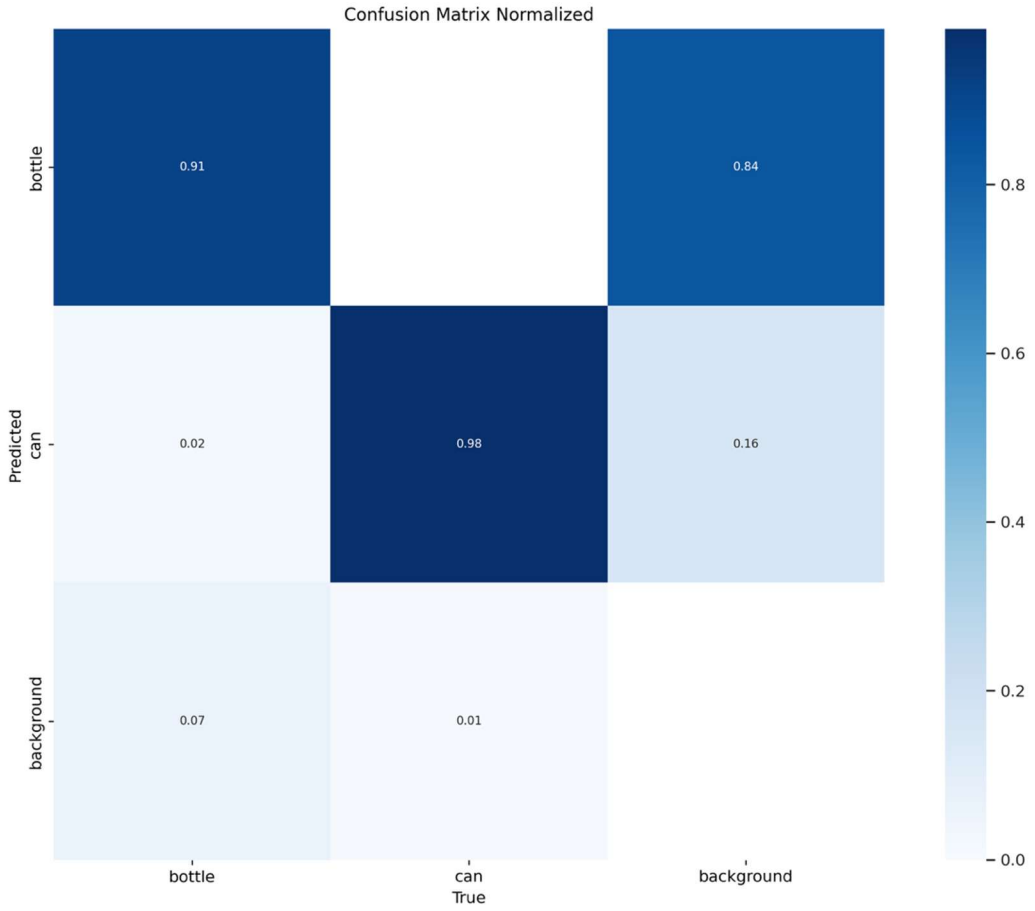


Figure 2:Confusion matrix.

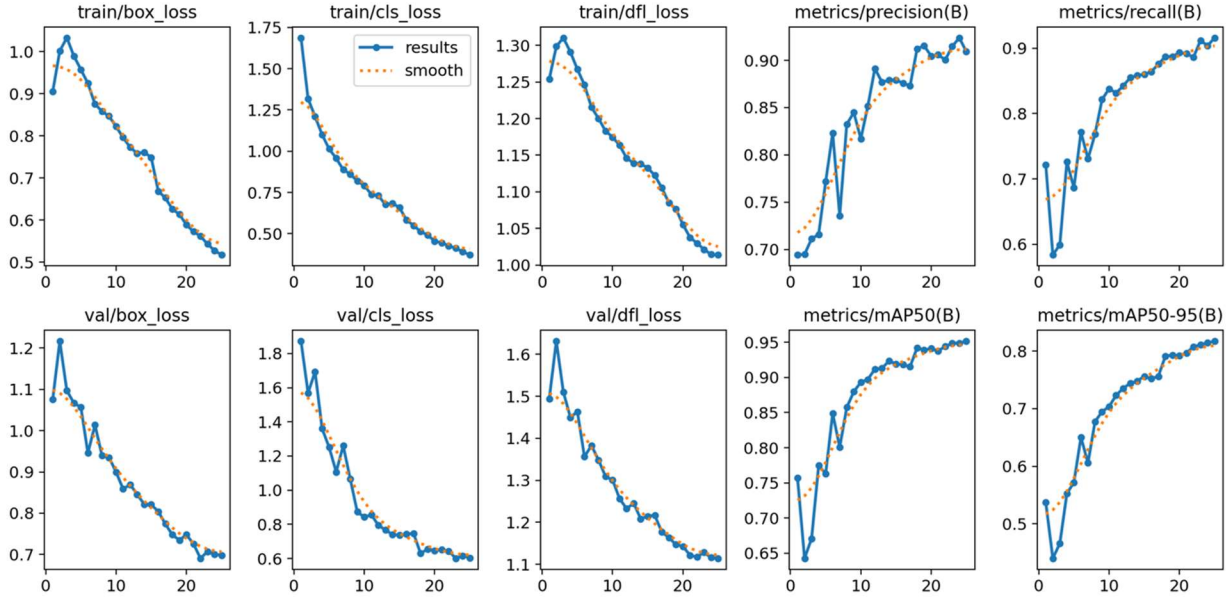


Figure 3:Graphs for different metrics

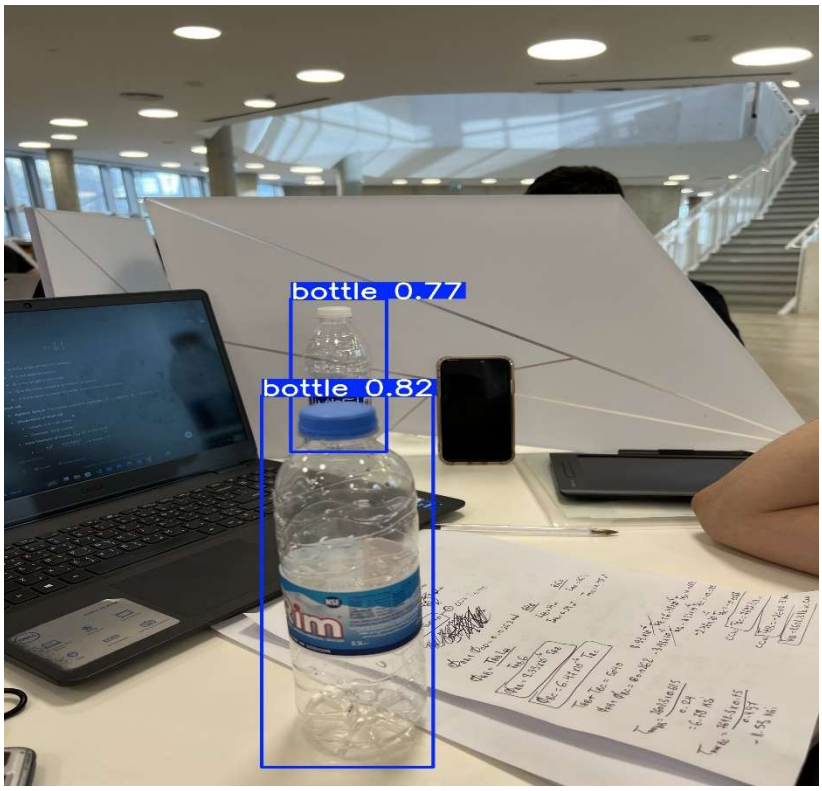


Figure 4:Sample picture 1

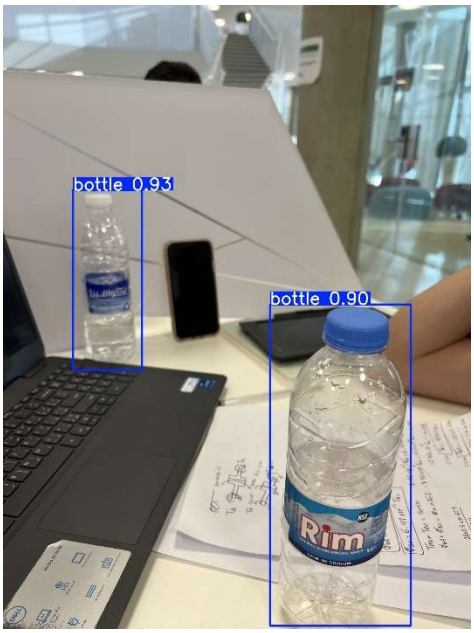


Figure 5:sample picture 2

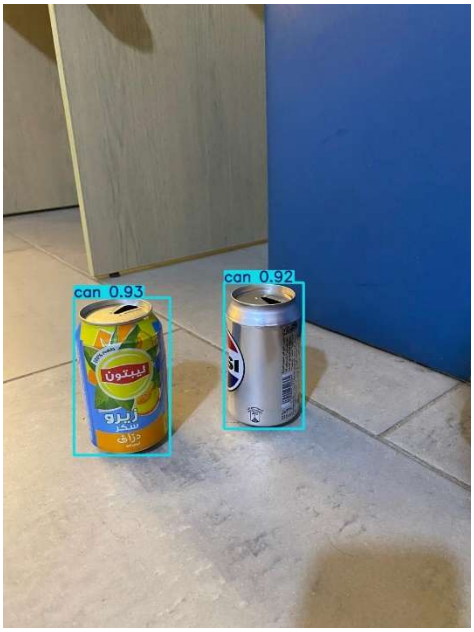


Figure 6:sample picture 3

Fuzzy Controller:

After training the model successfully and running it on our Pi using the pi camera, the next step was to control the rover in a way such that it centers the object and makes its way to it.

- Inputs: Angle offset of the detected object, distance from object.
- Outputs: PWM signal for left and right motors.
- Fuzzification and Defuzzification: We used a classical Mamdani min-max approach, we also used the centroid method for defuzzification.

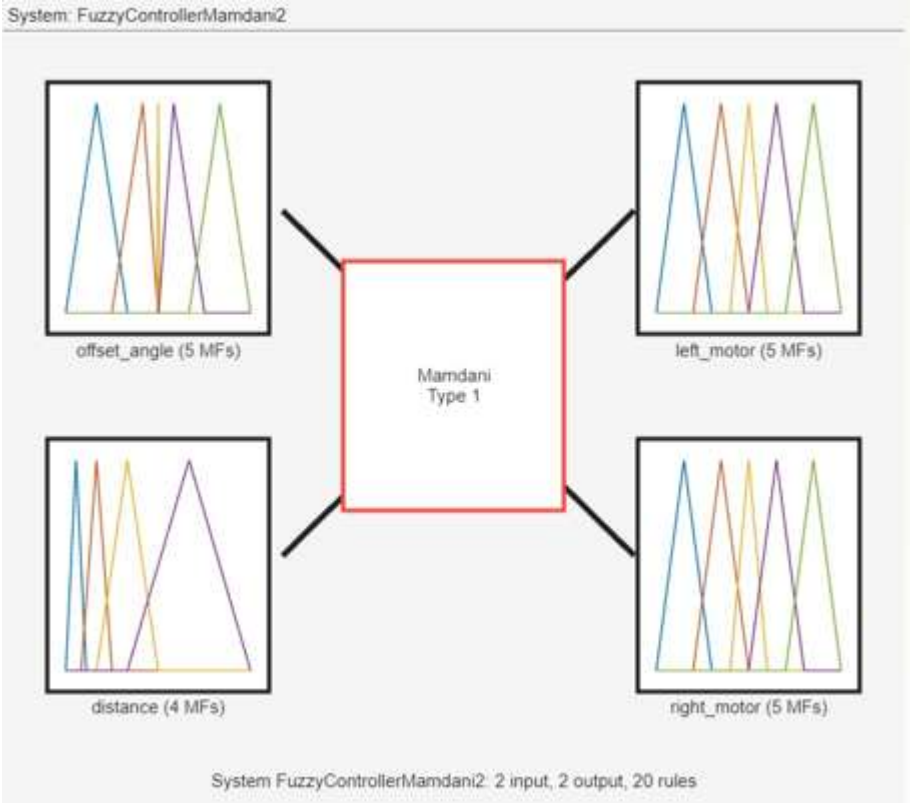


Figure 7: Fuzzy controller overview.

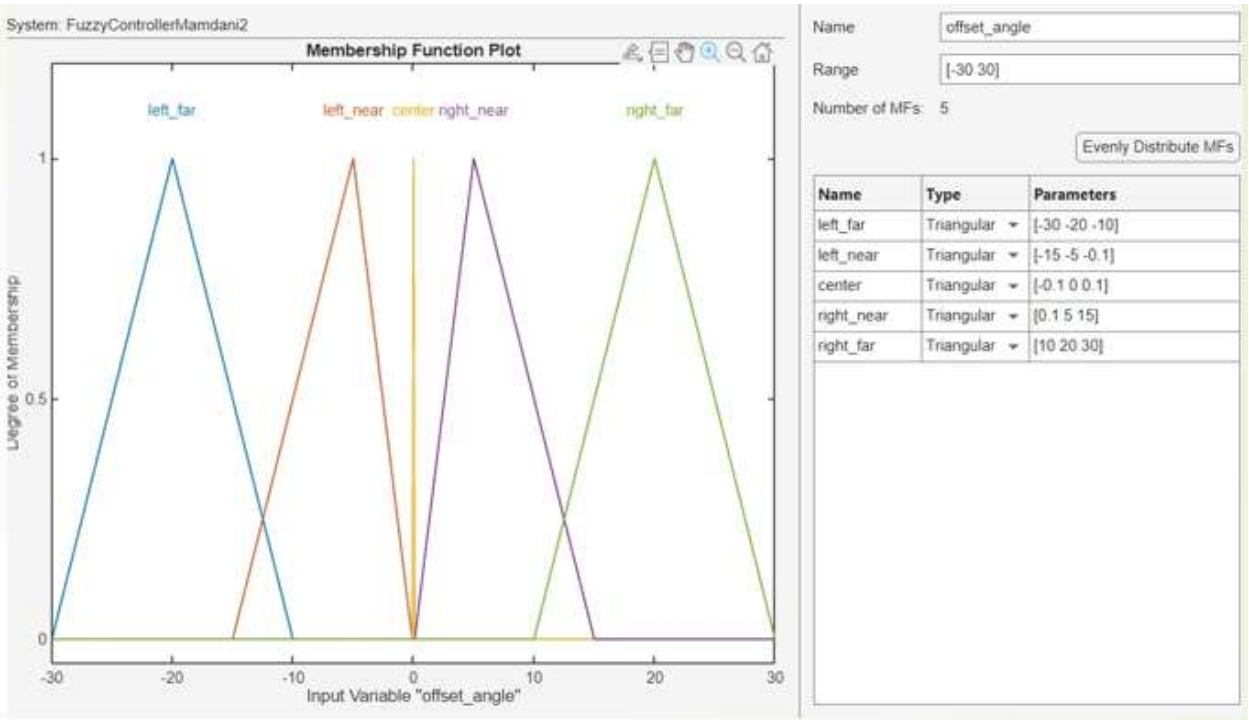


Figure 8: Membership function for offset angle

We wanted the rover to precisely center the object thus the small range of the “Center” membership function

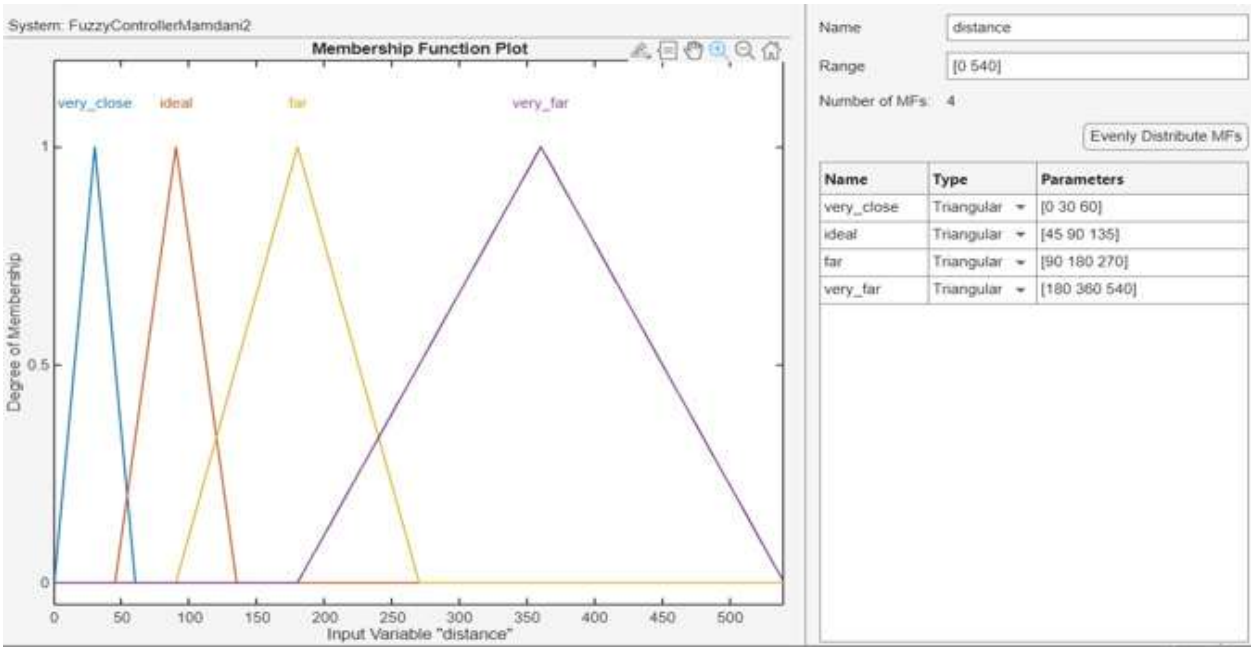


Figure 9:Membership function for distance.

Our rover, in most cases, started at far distances from the object which demanded the rover to move faster towards the object as long as it is “very far” thus the membership function for “very far” spans most of our universe of discourse for distance.

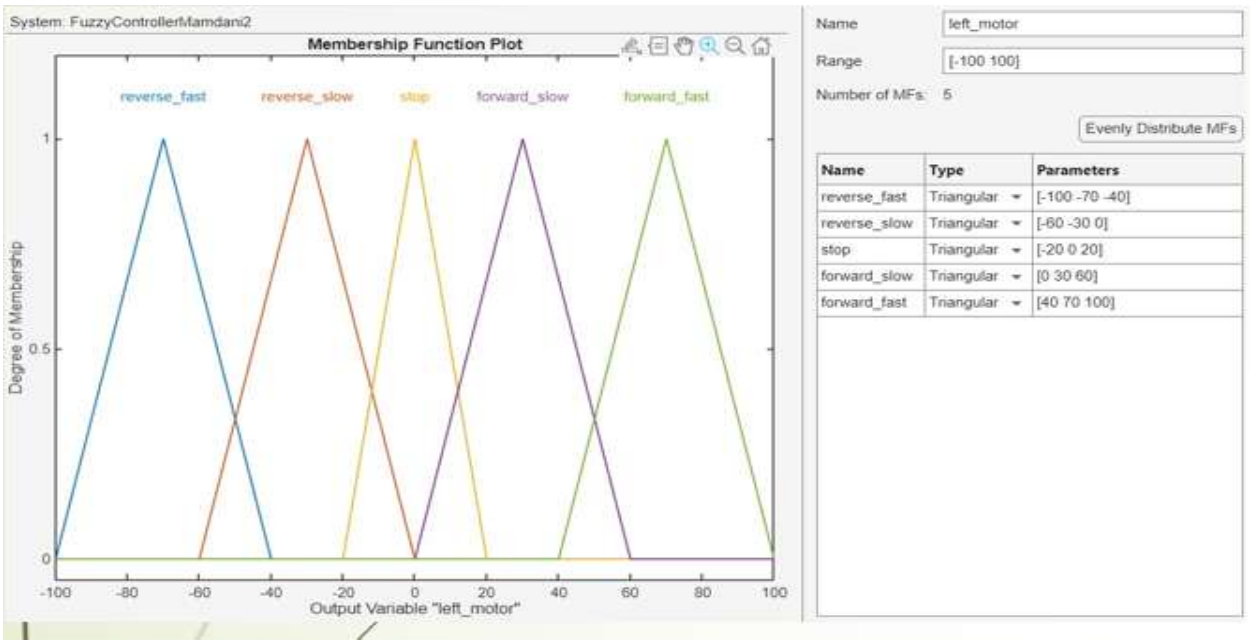


Figure 10:Membership function for left motors.

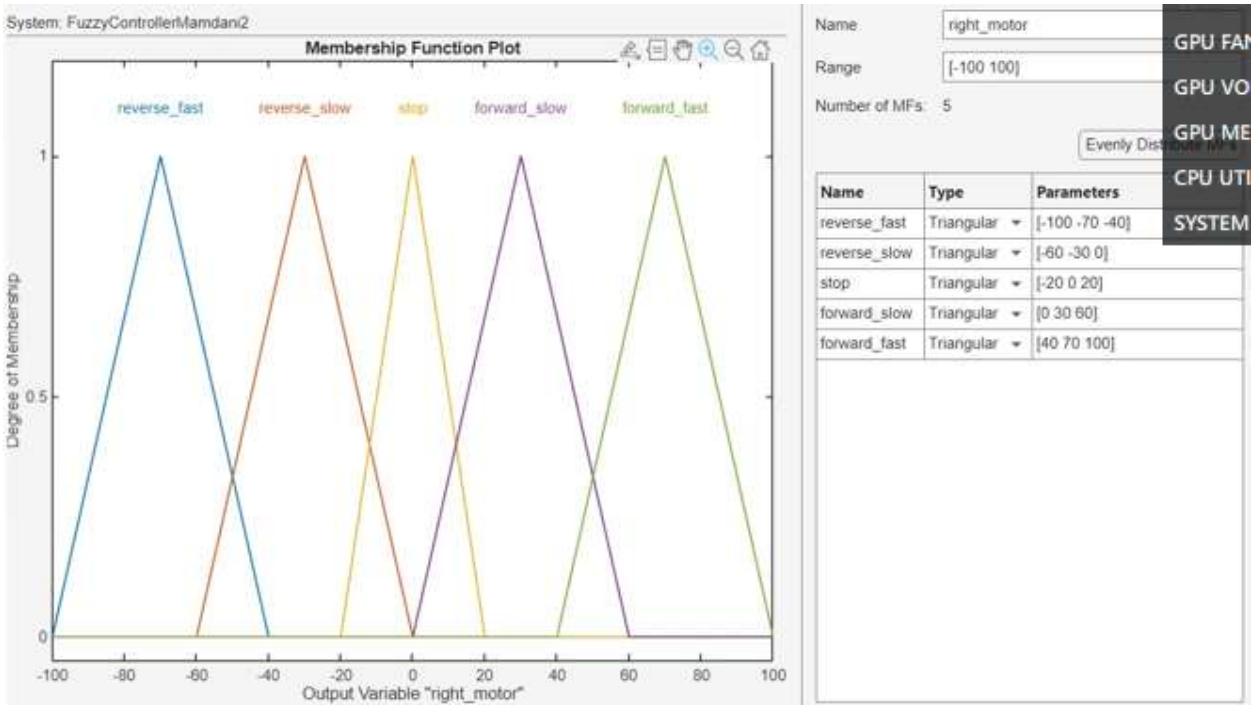


Figure 11:Membership function for right motors

After implementing the following membership function on MATLAB, we also needed to come up with a set of rules to cover all the possible cases;

	Rule	Weight	Name
1	If offset_angle is center and distance is very_close then left_motor is reverse_slow, right_mot...	1	rule1
2	If offset_angle is center and distance is ideal then left_motor is stop, right_motor is stop	1	rule2
3	If offset_angle is center and distance is far then left_motor is forward_slow, right_motor is forw...	1	rule3
4	If offset_angle is center and distance is very_far then left_motor is forward_fast, right_motor is...	1	rule4
5	If offset_angle is left_near and distance is very_close then left_motor is reverse_fast, right_motor is reverse_slow	1	rule5
6	If offset_angle is left_near and distance is ideal then left_motor is stop, right_motor is forward...	1	rule6
7	If offset_angle is left_near and distance is far then left_motor is forward_slow, right_motor is fo...	1	rule7
8	If offset_angle is left_near and distance is very_far then left_motor is forward_fast, right_motor...	1	rule8
9	If offset_angle is left_far and distance is very_close then left_motor is reverse_fast, right_moto...	1	rule9
10	If offset_angle is left_far and distance is ideal then left_motor is reverse_slow, right_motor is f...	1	rule10
11	If offset_angle is left_far and distance is far then left_motor is stop, right_motor is forward_fast	1	rule11
12	If offset_angle is left_far and distance is very_far then left_motor is forward_slow, right_motor i...	1	rule12
13	If offset_angle is right_near and distance is very_close then left_motor is reverse_slow, right_...	1	rule13
14	If offset_angle is right_near and distance is ideal then left_motor is forward_slow, right_motor i...	1	rule14
15	If offset_angle is right_near and distance is far then left_motor is forward_fast, right_motor is f...	1	rule15
16	If offset_angle is right_near and distance is very_far then left_motor is forward_fast, right_mot...	1	rule16
17	If offset_angle is right_far and distance is very_close then left_motor is stop, right_motor is rev...	1	rule17
18	If offset_angle is right_far and distance is ideal then left_motor is forward_slow, right_motor is ...	1	rule18
19	If offset_angle is right_far and distance is far then left_motor is forward_fast, right_motor is stop	1	rule19
20	If offset_angle is right_far and distance is very_far then left_motor is forward_fast, right_motor...	1	rule20

Figure 12:Rules for the fuzzy controller

After implementing the entire controller on MATLAB we used sckit fuzzy to convert it to python code to run on the Pi.

```
class FuzzyController:
    """Fuzzy logic controller for rover navigation"""

    def __init__(self, target_distance=TARGET_DISTANCE, offset_interval=OFFSET_INTERVAL):
        """Initialize fuzzy controller with membership functions"""
        self.target_distance = target_distance
        self.offset_interval = offset_interval

        # Define fuzzy sets for inputs
        # Offset angle (in degrees)
        self.offset_angle_ranges = {
            'left_far': [-30, -20, -10],    # Left far: -30 to -10 degrees
            'left_near': [-15, -5, -self.offset_interval],  # Left near: -15 to -offset_interval degrees
            'center': [-self.offset_interval, 0, self.offset_interval],  # Center: -offset_interval to +offset_interval degrees
            'right_near': [self.offset_interval, 5, 15],    # Right near: +offset_interval to 15 degrees
            'right_far': [10, 20, 30]       # Right far: 10 to 30 degrees
        }

        # Distance (in cm)
        self.distance_ranges = {
            'very_close': [0, target_distance/3, target_distance*2/3],        # Very close
            'ideal': [target_distance/2, target_distance, target_distance*1.5],  # Ideal distance
            'far': [target_distance, target_distance*2, target_distance*3],   # Far
            'very_far': [target_distance*2, target_distance*4, target_distance*6]  # Very far
        }

        # Define fuzzy sets for outputs
        # Motor speeds (-100 to 100)
        self.left_motor_ranges = {
            'reverse_fast': [-100, -70, -40],   # Reverse fast: -100 to -40
            'reverse_slow': [-60, -30, 0],      # Reverse slow: -60 to 0
            'stop': [-20, 0, 20],               # Stop: -20 to 20
            'forward_slow': [0, 30, 60],        # Forward slow: 0 to 60
            'forward_fast': [40, 70, 100]       # Forward fast: 40 to 100
        }

        self.right_motor_ranges = {
            'reverse_fast': [-100, -70, -40],   # Reverse fast: -100 to -40
            'reverse_slow': [-60, -30, 0],      # Reverse slow: -60 to 0
            'stop': [-20, 0, 20],               # Stop: -20 to 20
            'forward_slow': [0, 30, 60],        # Forward slow: 0 to 60
            'forward_fast': [40, 70, 100]       # Forward fast: 40 to 100
        }
```

Figure 13: Defining the ranges for inputs and outputs.

```
  # Define fuzzy rules for smooth navigation
  # Each rule is (offset_angle_set, distance_set, left_motor_set, right_motor_set)
  self.rules = [
      # When object is centered
      ('center', 'very_close', 'reverse_slow', 'reverse_slow'),    # Back up if too close
      ('center', 'ideal', 'stop', 'stop'),                         # Stop at ideal distance
      ('center', 'far', 'forward_slow', 'forward_slow'),           # Move forward slowly
      ('center', 'very_far', 'forward_fast', 'forward_fast'),      # Move forward quickly

      # When object is to the left
      ('left_near', 'very_close', 'reverse_fast', 'reverse_slow'),  # Turn right while backing up
      ('left_near', 'ideal', 'stop', 'forward_slow'),               # Turn right in place
      ('left_near', 'far', 'forward_slow', 'forward_fast'),         # Turn right while moving forward
      ('left_near', 'very_far', 'forward_fast', 'forward_fast'),    # Speed up right motor

      # When object is far to the left
      ('left_far', 'very_close', 'reverse_fast', 'stop'),          # Hard right while backing up
      ('left_far', 'ideal', 'reverse_slow', 'forward_slow'),       # Hard right turn
      ('left_far', 'far', 'stop', 'forward_fast'),                 # Sharp right
      ('left_far', 'very_far', 'forward_slow', 'forward_fast'),    # Wide right

      # When object is to the right
      ('right_near', 'very_close', 'reverse_slow', 'reverse_fast'),  # Turn left while backing up
      ('right_near', 'ideal', 'forward_slow', 'stop'),               # Turn left in place
      ('right_near', 'far', 'forward_fast', 'forward_slow'),         # Turn left while moving forward
      ('right_near', 'very_far', 'forward_fast', 'forward_fast'),    # Speed up left motor

      # When object is far to the right
      ('right_far', 'very_close', 'stop', 'reverse_fast'),          # Hard left while backing up
      ('right_far', 'ideal', 'forward_slow', 'reverse_slow'),       # Hard left turn
      ('right_far', 'far', 'forward_fast', 'stop'),                 # Sharp left
      ('right_far', 'very_far', 'forward_fast', 'forward_slow')     # Wide left
  ]
```

Figure 14:Defining the rules.

We also used the following formula to estimate distance to our best ability while being limited to 1 camera:

```
def estimate_distance(self, bbox_width_pixels, image_width_pixels, class_id=None):
    """Estimate distance to object based on apparent size

    This uses the pinhole camera model: distance = (object_width_real * focal_length) / object_width_pixels

    Args:
        bbox_width_pixels: Width of object bounding box in pixels
        image_width_pixels: Width of image in pixels
        class_id: Object class ID for real-world size lookup

    Returns:
        Estimated distance in cm
    """
    # Get real-world object width based on class
    object_width_cm = self.get_object_width(class_id)

    # Calculate the focal length in pixels
    # focal_length_pixels = (image_width_pixels * focal_length_mm) / sensor_width_mm
    focal_length_pixels = (image_width_pixels * self.focal_length_mm) / self.sensor_width_mm

    # Estimate distance using the pinhole camera model
    if bbox_width_pixels > 0:
        distance_cm = (object_width_cm * focal_length_pixels) / bbox_width_pixels
        return distance_cm
    else:
        return float('inf')  # Return infinity for zero-width objects
```

Figure 15:Distance calculation.

Moreover we also added a manual control feature using the keyboard keys to help in cases where an object goes out of frame.

Arm Control

For our robotic arm, it had 6 DOF meaning that we needed 12 unique combinations using fingers. Each finger had 2 states (open or closed) representing a binary 1 or 0 in the code, the OpenCV model was run on the laptop and the data binary combination of the fingers was sent to the esp32(through internet) for performing the logic and actuating the motors.
This Python script implements real-time hand-gesture control of a six-joint robotic arm by combining OpenCV's webcam capture with MediaPipe's hand-landmark detection and a WebSocket link to an ESP32 microcontroller. At launch, the user is prompted for the ESP32's IP address.. Each video frame is processed to extract normalized hand landmarks, from which a

five-bit “finger code” is generated to represent which fingers are raised. Once a gesture has been held stably for a few frames, the code maps that finger code to one of 13 predefined joint commands—updating a joint_commands dictionary for base rotation, shoulder, elbow, wrist pitch, wrist roll, or gripper—and sends it as JSON over the websocket. After the video loop ends (or the user quits), a final “stop” message is sent and all resources are released. The recognized gestures and their associated joint actions are:

Gesture	Finger Code	Joint Command
Fist	00000	STOP ALL
Index only	01000	Shoulder ↑
Index + Middle	01100	Shoulder ↓
Thumb only	10000	Elbow ↑
Thumb + Index	11000	Elbow ↓
Pinky only	00001	Wrist Pitch ↑
Pinky + Index	01001	Wrist Pitch ↓
Thumb + Index + Pinky	11001	Wrist Roll →
Thumb + Pinky	10001	Wrist Roll ←
Thumb + Index + Middle	11100	Base Rotate →
Ring + Middle + Pinky	00111	Base Rotate ←
All fingers	11111	Gripper Open
All except Thumb	01111	Gripper Close

Challenges & Limitations:

We faced a lot of challenges and setbacks during our project, mainly:

Computational power constraints: Although we were using the pi5, we still faced a lot of issues with the lack of computational power, as running the YOLOv8 model in real-time is a very demanding task, moreover we couldn’t find a portable power supply with 5A current output, so we stuck with a 3A one which also bottlenecked the performance.

Inaccuracies: for the sake of our project, measuring the distance to an object in real time would make the process easier, however measuring the distance to an object (which doesn't have predefined dimensions) in real-time proved to be a very challenging task with only 1 camera. Installing another camera and running it on the same pi was also out of the question due to the lack of computational power (1st camera was already running at 7-10 fps) so we had to resort to other methods.

Delay: As mentioned above, the camera was running at a very low frame rate, however every single frame needed to be processed and used for producing the crisp fuzzy output at that instant which caused some delay in the actuation of the motors.

Model: Dataset collection, preventing overfitting and underfitting.

Objects leaving frame: As mentioned above, due to the limitation of having only one camera, if the targeted object went out of frame the rover would lose track of it and the user would need to manually adjust the rover until the object is in frame again.

Future work:

We produced a simple prototype for our project that can be expanded with more budget and less constraints, mainly:

- The integration of an additional camera can be extremely helpful in measuring distances and covering a wider scope of terrain.
- The possibility of expanding the prototype to a larger dataset of garbage which makes it capable of identifying more kinds garbage.

- The possibility of making the rover and the arm completely autonomous

Note that the video link and full codes along with additional info are uploaded to GitHub.