KANNAN TEAM

ADVANCED CONTROL AND PROGRAMMING TECHNIQUES FOR CEASER MOVEMENT

Table of Contents:

1. Introduction:

The Robot, named Caesar, employs two microcontrollers: Raspberry Pi and ESP32 DEVKIT V1. The ESP32 DEVKIT V1 is programmed within a Linux environment hosted on the Raspberry Pi. It establishes communication with the Raspberry Pi through a serial port. The designated platform for coding and uploading onto the ESP32 DEVKIT V1 is the Arduino IDE ARM version.

In this configuration, the ESP32 DEVKIT V1 microcontroller is programmed using the C++ language. Its functions encompass controlling the motor, steering wheel, and sensors (specifically, Ultrasonic). On the other hand, Python is employed for image processing on the Raspberry Pi. The program leverages appropriate libraries, including OpenCV, downloaded and configured to align with Python. This facilitates tasks such as image capture from the camera, clustering of red and green pillars, and the subsequent issuance of commands based on detected colors.

2. The Robot's Journey in Stage One:

In the initial stage of our project, the robot's primary task is to move autonomously, but without any pillars.

2.1 Deciphering Direction:

One of the challenges in this stage lay in developing an algorithm that allowed the robot to determine its direction, specifically whether it should move in a clockwise or counterclockwise direction.

Prior to the first turn, the robot operated in a state of directional uncertainty. To address this challenge, we incorporated ultrasonic sensors into our design. These sensors played a pivotal role in measuring the distance between the robot and its surroundings.

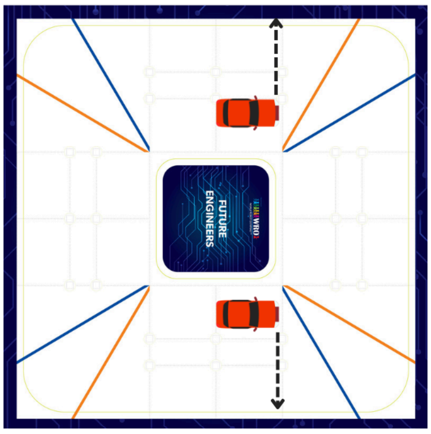
The algorithm we employed was ingeniously simple yet effective:

Ultrasonic Measurements: The robot utilized its ultrasonic sensors to measure the distance to objects in its vicinity.

Directional Indicators: Here's where the magic happened. Until the first turn was encountered, the robot remained clueless about its direction. However, it was equipped to make an informed decision when the time came.

If the left ultrasonic sensor registered a distance greater than 160 cm or zero (indicating more than 280cm meters of open space to the left), the robot inferred that it should proceed in a clockwise direction.

Conversely, if the right ultrasonic sensor measured a distance greater than 160 cm or zero (suggesting more than 280cm of open space to the right), the robot deduced that it should move counterclockwise.



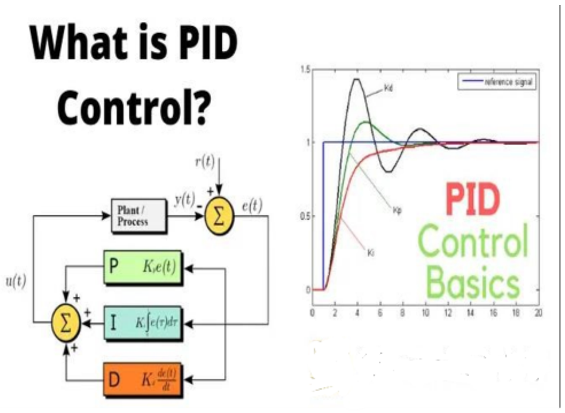
This ingenious algorithm ensured that the robot remained adaptive and selfaware. Until the moment of truth – the first turn – the robot's direction remained a mystery, but with ultrasonic measurement, it allowed him to decipher its path, reflecting the essence of autonomous navigation in this intriguing stage of our project.

2.2 How Our Robot Moves: The Algorithm in Action:

In our project, we implemented a PID (Proportional, Integral, Derivative) control system to ensure precise steering and movement of our robot. This system is a sophisticated control mechanism widely used in robotics and automation. Here's how we put it to work:

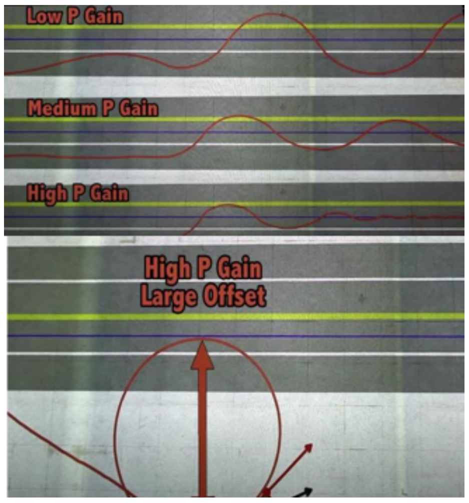
What is PID System:

The PID system is a feedback control loop that calculates and applies an error-correction value based on proportional, integral, and derivative terms. It helps to accurately maintain a desired set point or target.



Steering Control:

In our robot, we employed the PID system to govern the steering mechanism. Attaining the precise PID parameters, however, posed a substantial challenge. To simplify this endeavor, we devised a mobile application allowing us to directly input the PID values. This not only saved us considerable time but also spared us the repetitive task of manual adjustments on the laptop. Given the frequent necessity for PID tuning, this app emerged as an indispensable tool in guaranteeing optimal performance. For instance, setting the P gain too high could lead to overshooting, while setting it too low might result in sluggish responses from the robot.



Setting the Set Point:

Until the first turn occurs, the robot operates with an undetermined direction. Once the first turn takes place, the set point for the PID system is determined based on the outer edge. For clockwise movement, the outer edge is on the left side, and for counterclockwise, it's on the right.

Determining Turns:

In each subsequent turn, a new set point is established, based on the outer edge. The robot assesses which side to turn based on the ultrasonic sensor readings. If one of the side ultrasonic sensors depending on his direction measures more than 120cm or zero (indicating a distance greater than 280 cm), and the forward ultrasonic sensor measures less than 95cm, the robot recognizes that a turn is necessary.

Utilizing the MPU6050 Sensor:

When executing turns, we leverage the MPU6050 sensor, which aids the robot in achieving precise 90-degree rotations.

Turn Counter:

To keep track of progress, we integrated a turn counter. This enables the robot to discern when it's approaching the final turn, ensuring a timely stop.

Completion of Stage One:

This process is reiterated with every turn until the completion of 12 turns, distributed evenly across the four rounds. At the conclusion of the 12th turn, the robot executes a controlled stop, marking the successful completion of stage one. This meticulous approach to movement and navigation lays a solid foundation for the subsequent stages of our project.

3. CAESAR In Stage Two (with obstacles):

If pillars are present, we've devised an algorithm that optimally positions the robot for traffic light interaction in Stage 2.

3.1 The Far Near Algorithem In Stage 2:

Overview:

The Far Near Algorithm, applied in Stage 2, enhances the robot's navigation by synchronizing the camera’s movement with the robot's steering for real-time obstacle detection. The ESP32 microcontroller receives an array of three letters ('F' for Far, 'N' for Near) from the camera on the Raspberry Pi, where "Far" indicates areas close to the exterior wall or far from the internal square, and "Near" refers to areas near to the internal square or far from the exterior wall. This information guides the robot’s path, allowing for efficient and precise movement around obstacles. By dynamically adjusting based on real-time data, the robot can navigate complex environments effectively.

3.2 the First Part of the Algorithm (Learning Phase):

1. Initial Position Detection:

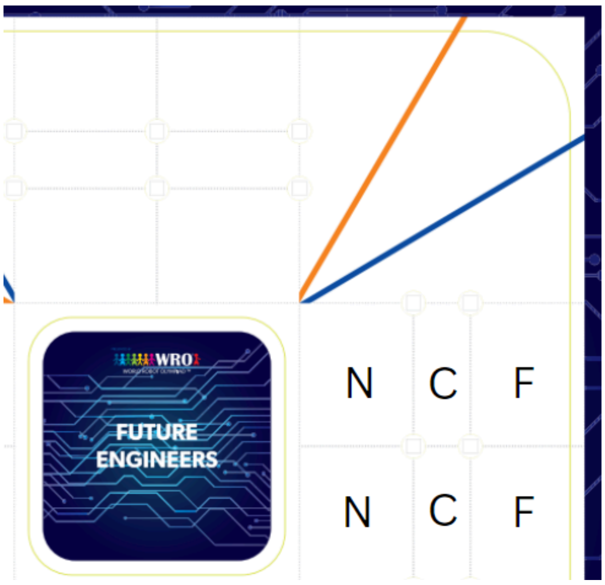
After pressing the start button, the robot determines its initial position using ultrasonic sensor measurements, stored in a variable called current_position.

Possible positions include:

'C' (Center): The robot is in the center of the game mat.

'F' (Far): The robot is near the exterior wall.

'N' (Near): The robot is far from the exterior wall and close to the inner square.



2.Movement to Determine Direction:

The robot moves straight until its front reaches the corner section, where it uses ultrasonic sensors to determine its direction (**clockwise or counterclockwise**). This direction detection is detailed in the Open Challenge algorithm.

3.Camera and Steering Synchronization:

The camera's movement is synchronized with the robot's steering, a key feature of this algorithm. Once the robot detects its direction at the corner section, it adjusts its steering to position the camera:

Counterclockwise Direction: The steering moves left, enabling the camera to capture the pillars on the straight forward section. The robot remains stationary for 3-4 seconds, allowing the camera, controlled by a Raspberry Pi, to process the surroundings.

Clockwise Direction: The steering moves right, enabling the camera to capture the pillars on the straight forward section. Similarly, the robot stays stationary while the camera captures the environment.

During this stationary period: The camera captures images to identify the positions of red and green pillars. The results are stored in an array of three letters ('N' for near, 'F' for far), guiding the robot's movement to avoid obstacles:

Counterclockwise Movement:

Seeing a **green pillar**: The robot should move near (**'N'**), staying close to the inner square to avoid the pillar on the left.

Seeing a **red pillar**: The robot should move far (**'F'**), maintaining a distance from the inner square to avoid the pillar on the right.

Clockwise Movement:

Seeing a **green pillar**: The robot should move far (**'F'**), staying away from the inner square to avoid the pillar on the left.

Seeing a **red pillar**: The robot should move near (**'N'**), staying close to the inner square to avoid the pillar on the right.

Example: If the array is ['F', 'F', 'N'], it means:

The robot starts far from the inner square to avoid the first obstacle.

It continues far to avoid the second obstacle.

Finally, it moves near to the inner square to avoid the last obstacle.

This array (next_position) helps the robot adjust its path dynamically, ensuring it avoids obstacles and navigates the track efficiently based on real-time environmental data.

4.Navigating to the Next Position :

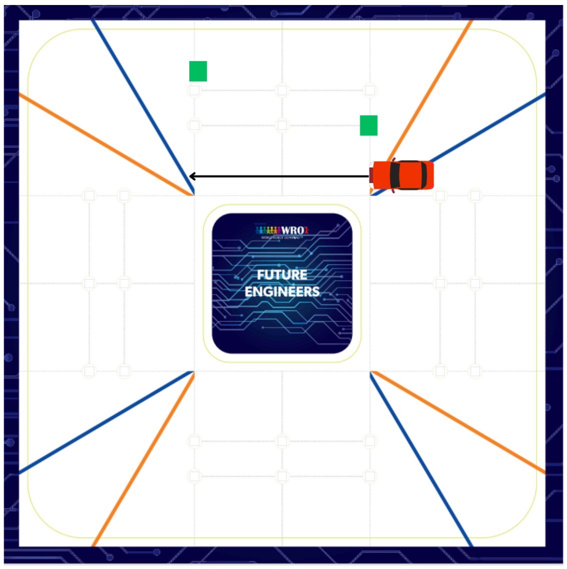
The robot uses the information stored in current_position and next_position to determine its subsequent movements. For instance, if current_position is 'C' and next_position is 'F', the robot executes a maneuver aligned with its programmed instructions to move to the 'F' position.

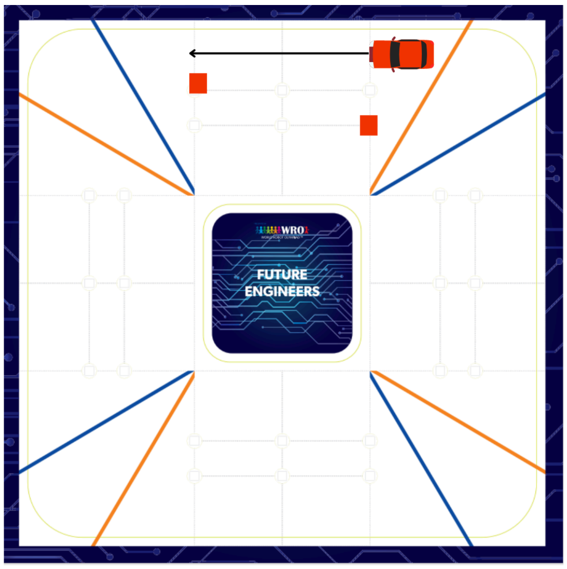
3.2 Between Obstacles Algorithm:

The Between Obstacles Algorithm manages the robot's movement between identified obstacles using the data from the learning phase:

1.Continuing in the Same Position (F to F or N to N):

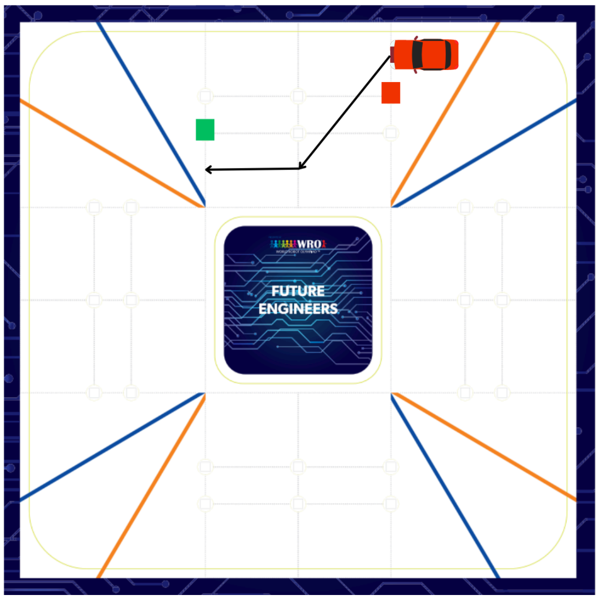
The robot moves straight ahead using PID control without altering its orientation. This ensures it maintains a steady path when consecutive positions in the array are the same.





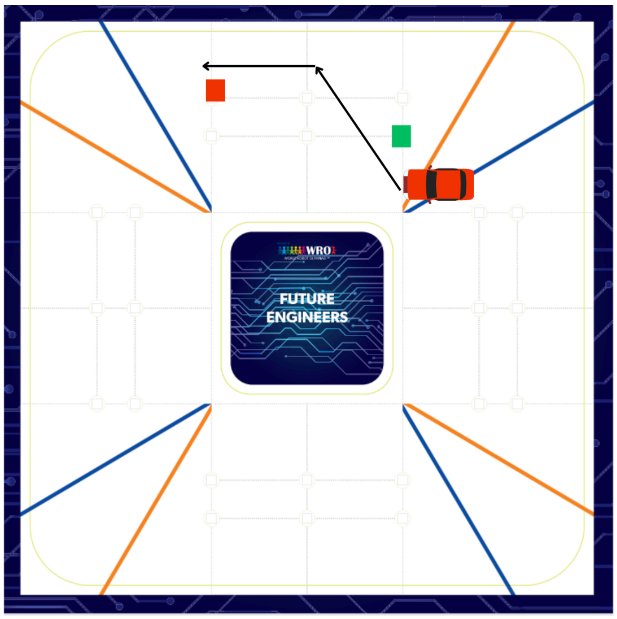
2.Transitioning from Far to Near (F to N):

When the robot needs to shift from a far to a near position, it performs a series of calculated movements and turns. This involves adjusting the steering angles and speed to navigate closer to the inner square, avoiding obstacles effectively.



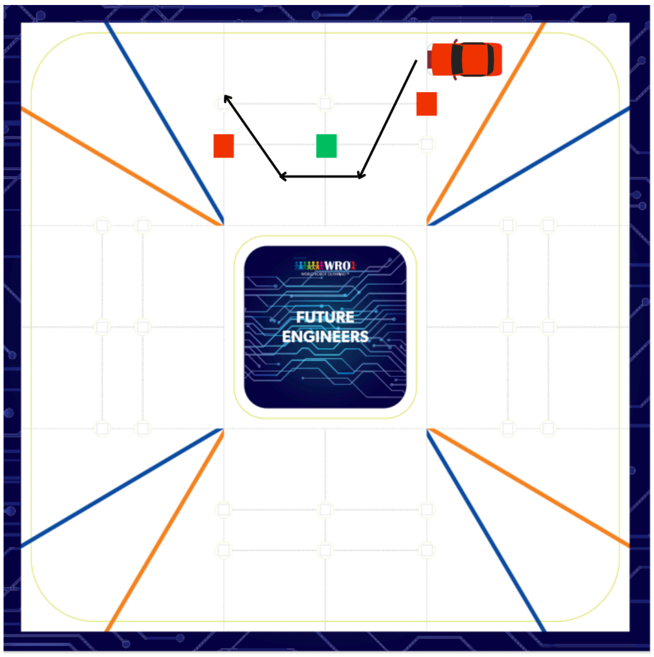
3.Transitioning from Near to Far (N to F):

In the opposite scenario, the robot adjusts its path to move away from the inner square. It uses a precise sequence of movements, steering adjustments, and speed control to ensure a smooth transition to a far position.



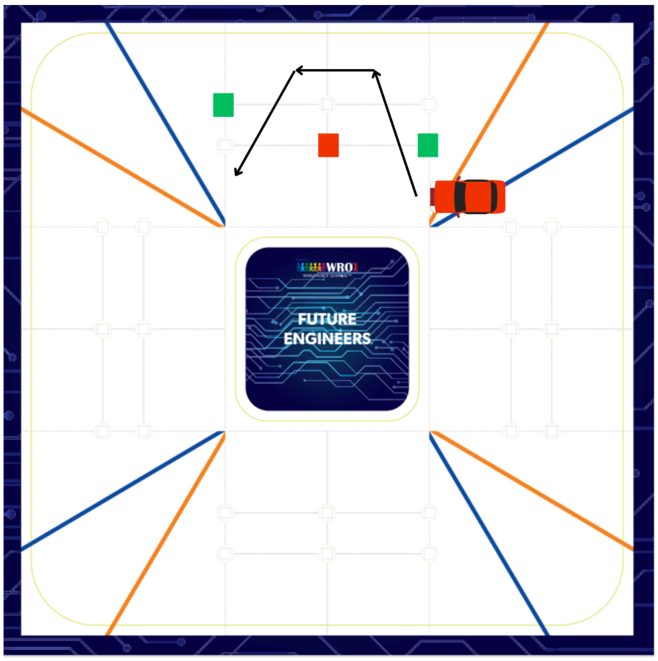
4.Transitioning from Far to Near to Far (F to N to F):

The robot initially moves from a far to a near position, navigating closer to the inner square to avoid obstacles on the far side. After this, it transitions back to a far position, moving away from the inner square. This sequence ensures that the robot efficiently avoids obstacles on both sides while maintaining a smooth path.



5.Transitioning from Near to Far to Near (N to F to N):

The robot moves away from the inner square (near to far), then transitions back to a near position to remain close to the inner square. This maneuver helps the robot effectively navigate areas with alternating obstacles, ensuring safe and efficient movement.



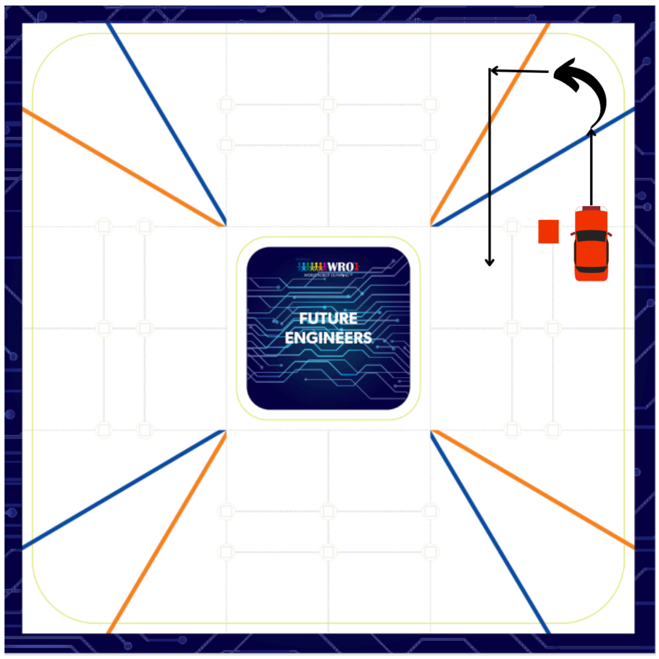
3.3 U-Turn Algorithm:

The U-turn algorithm is employed when the robot needs to reverse direction. To navigate obstacles after making a U-turn, the robot manipulates the array of detected positions:

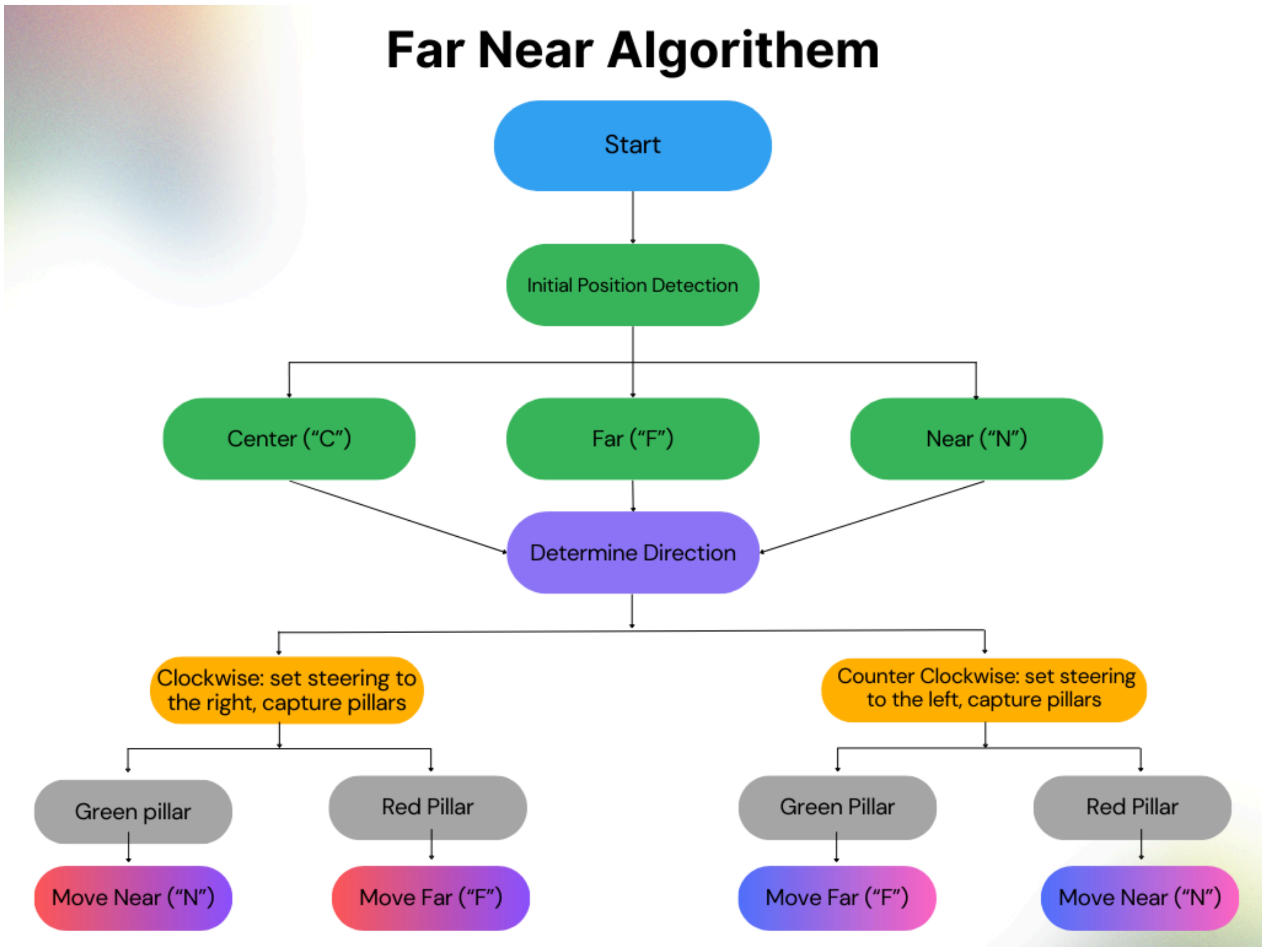
1.Flip the Array: The sequence of detected positions is flipped. For example, an array of ['N', 'N', 'F'] becomes ['F', 'N', 'N'].

2.Reverse the Letters: After flipping, each letter in the array is reversed to reflect the new orientation. Thus, the flipped array ['F', 'N', 'N'] changes to ['N', 'F', 'F'].

This transformation ensures that the robot correctly interprets the environment and continues to avoid obstacles effectively even after reversing direction. By systematically flipping and reversing the detection array, the robot adapts its path dynamically to maintain efficient navigation throughout the course.



3.4 The Flowchart For The Far Near Algorithem:



3.5 Summary:

The Far Near Algorithm, including **the initial learning phase**, **the Between Obstacles Algorithm**, and **the U-turn Algorithm**, enables the robot to navigate complex environments by synchronizing camera movements with steering, processing real-time data to adjust its path, and executing efficient maneuvers to avoid obstacles. This comprehensive approach ensures the robot can handle varying scenarios effectively, making it an optimal solution for precise and safe navigation in competitive settings.

4. DC Encoder:

In our car, we utilized a 130 RPM encoder for smooth and high-performance operation. Using the encoder, we found that each cm equals to 20 pulses in the encoder.

Hence, we have an equation that establishes a connection between the distance covered and the count of pulses:

$$X(pulses) = cm*20$$

5.Camera coding:

1.Convert the image's color scheme from RGB to HSV:

```
hsvFrame = cv2.cvtColor(imageFrame, cv2.COLOR_BGR2HSV)
```

2.Create color masks to isolate green and red colors in the HSV system:

```
red_lower = np.array([170, 100, 50], np.uint8)
red_upper = np.array([180, 255, 255], np.uint8)
red_mask = cv2.inRange(hsvFrame, red_lower, red_upper)

green_lower = np.array([50,80, 50], np.uint8)
green_upper = np.array([80, 255, 255], np.uint8)
green_mask = cv2.inRange(hsvFrame, green_lower, green_upper)
```

3.Use OpenCV functions to extract regions defined by these masks:

```
red_mask = cv2.dilate(red_mask, kernal)
res_red = cv2.bitwise_and(imageFrame, imageFrame, mask=red_mask)

green_mask = cv2.dilate(green_mask, kernal)
res_green = cv2.bitwise_and(imageFrame, imageFrame, mask=green_mask)
```

4.Find contours to track red and green colors:

```
contours, hierarchy = cv2.findContours(red_mask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

contours, hierarchy = cv2.findContours(green_mask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

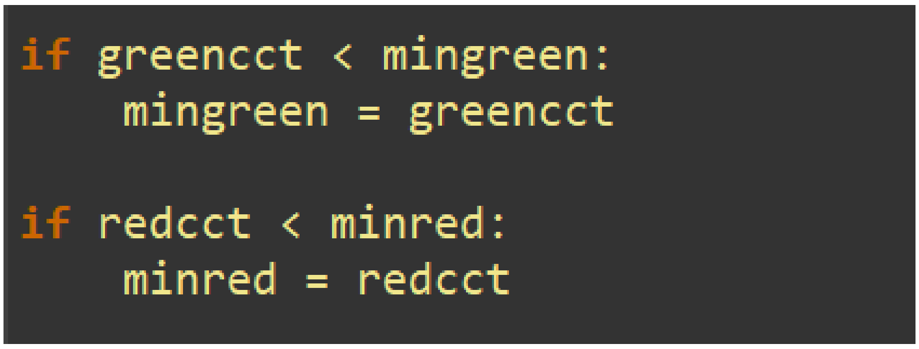
5.Calculate areas for these masks, ignoring those smaller than 300 pixels:

```
for pic, contour in enumerate(contours):
    # ...
    if (area > 300):
        # ...
```

6.Maintain global variables to keep track of the number of masks for each color:

```
counter = counter + 1
mingreen = 10
minred = 10
```

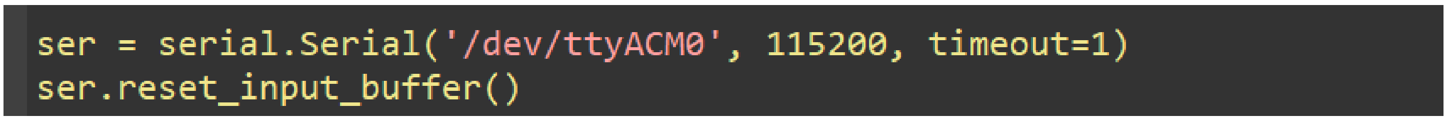
7.To minimize noise, repeat the process for the minimum number of masks (10 times) for both colors:



```python
if greencct < mingreen:
    mingreen = greencct

if redcct < minred:
    minred = redcct
```

8.Establish a connection between the Raspberry Pi and an Arduino via the serial port, utilizing the serial library:



```python
ser = serial.Serial('/dev/ttyACM0', 115200, timeout=1)
ser.reset_input_buffer()
```

9.Send commands through the serial port to the ESP32 for it to execute appropriate actions.

THE END

KANNAN TEAM