Advanced Control and Programming Techniques for CEASER Movement

KANAAN TEAM

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.

2.1 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.

2.2 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:

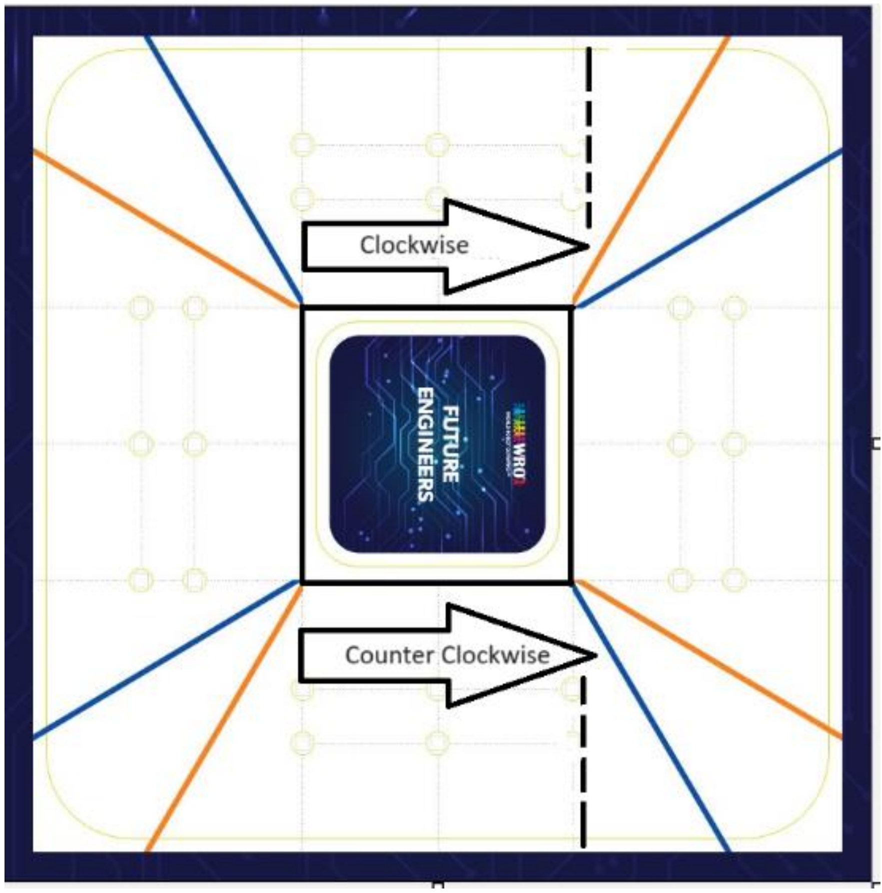
Ultrasonic Measurements: The robot utilized its ultrasonic sensors to measure the distance to objects in its vicinity.

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.

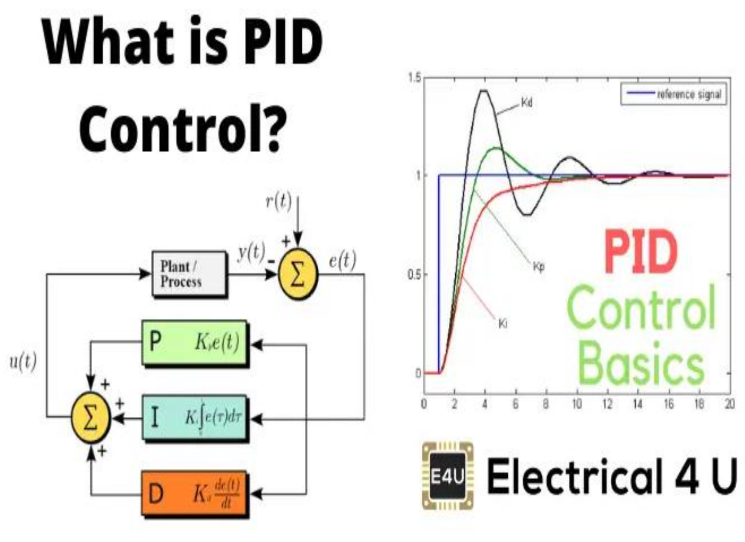
Conversely, if the right ultrasonic sensor measured a distance greater than 160 cm or zero (suggesting more than 280cm of open space to the right), the robot deduced that it should move counterclockwise.

This ingenious algorithm ensured that the robot remained adaptive and self-aware. 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.



2.3 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:

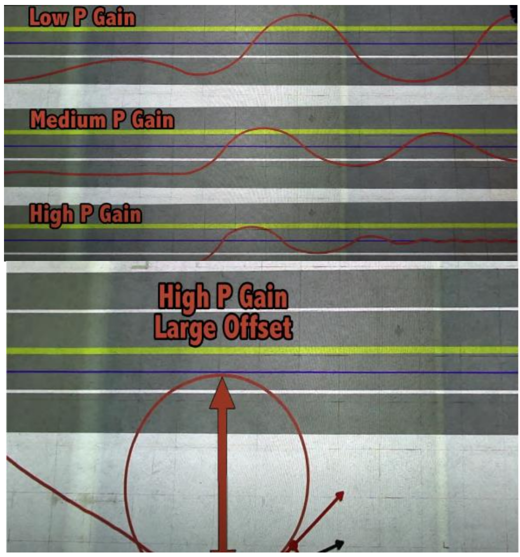


What is PID System:

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:

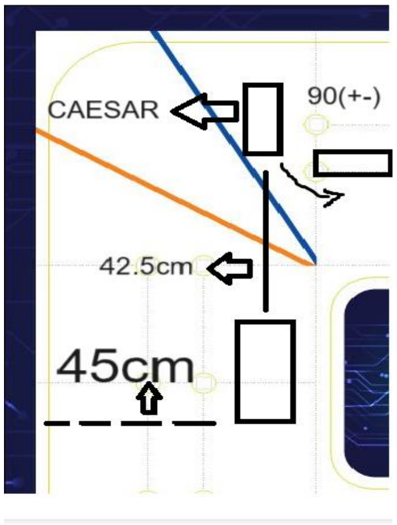
This process is reiterated with every turn until the completion of 12 turns, distributed evenly across the four rounds. At the conclusion of the 12th turn, the robot executes a controlled stop, marking the successful completion of stage one. This meticulous approach to movement and navigation lays a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

3.1 CAESAR In Stage Two (with pillars):

If pillars are present, we've devised an algorithm for rotation that accounts for all possible positions the vehicle may occupy.

3.2: Cause 1(when the vehicle maintains a distance greater than 45 cm from the outer frame while in motion):

One specific situation arises when the vehicle maintains a distance greater than 45 cm from the outer frame while in motion. In this case, we employ the "straightReverse()" function, which operates as follows:

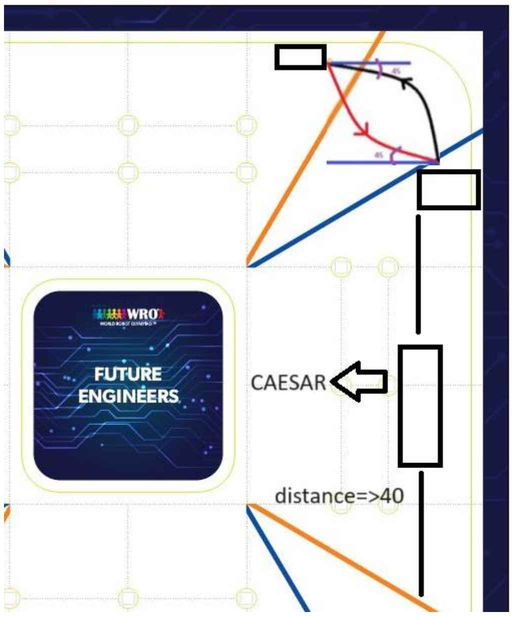


The vehicle advances halfway towards the front wall, effectively covering a distance of approximately 42.5 cm. It then executes a turnback maneuver, rotating by either 90 degrees plus or minus the angle of the slope (theta) to align itself, ensuring a thorough examination of the area in front of it before proceeding.

Upon experimentation, we determined that each degree in radians, when the steering wheel is positioned at 45 degrees, corresponds to an approximate distance of 0.77 cm. Consequently, a 90-degree rotation equates to approximately 66cm of movement.

3.3 Cause 2(When the gap between the vehicle and the outer frame is 40 centimeters or less):

For the second scenario, if the gap between the vehicle and the outer frame is 40 centimeters or less, we employ the turnleft() or turnright() functions. Here's how these functions are implemented:



We marked out a circle within the ring, with a radius of 42 centimeters, which is half the length of the distance in the corner region. Subsequently, we calculated the arc length in the corner area using the equation shown the Figure , denoted as:

Arc length (Arc) = radius (r) × $\pi/2$

During our computations, we determined that the length of a single arc measures 32 centimeters. The combined length of both arcs is 65 centimeters. However, based on practical testing, we established that the length of the first arc is approximately 37.25 centimeters, while the second arc measures about 32.25 centimeters. This variation is attributed to the initial tilt of the vehicle, resulting in an inclination of nearly 5 centimeters.

The rotation process involves moving the vehicle along the first arc while simultaneously turning the steering wheel 45 degrees for a distance of 37 centimeters. Afterwards, the vehicle is reversed to traverse the second arc, covering a distance of 32 centimeters.

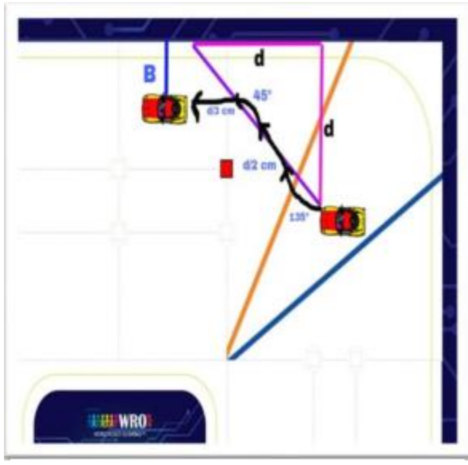
The functions turnleft() and turnright() are employed in this process.

3.4 Actions for red and green colors:

When the vehicle detects the presence of the color red, it should execute a right turn. To achieve this, we devised a specific algorithm, outlined the Figure:

By applying the Pythagorean theorem, we computed the value of 'd' and directed the vehicle's movement based on this calculation:

$$\sqrt{a^2 + b^2} = c$$

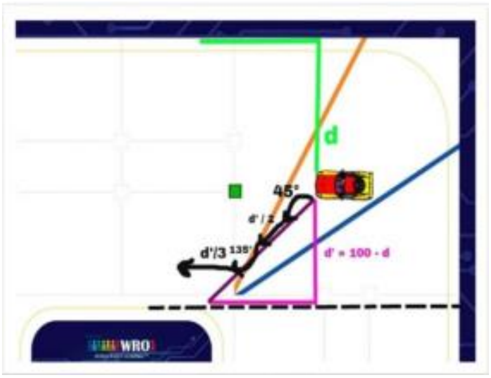


In our case, we made the assumption that 'a' is equivalent to 'b', simplifying the equation to:

$$\sqrt{a^2 + a^2} = c$$

$$a\sqrt{2} = c$$

If the vehicle detects the presence of the color green, it should initiate a left turn. To achieve this, we developed a specific algorithm, outlined in the Figure .



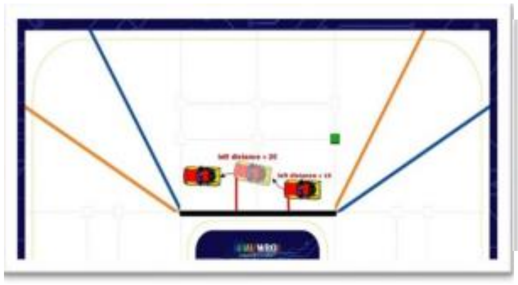
4. Tuning:

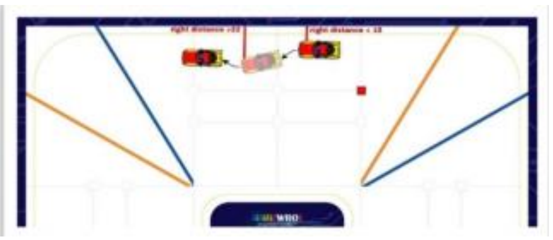
We devised a function to optimize the vehicle's trajectory, aiming for it to stay as centered as possible within the path. This was achieved by gauging the distance between the vehicle and either the outer or inner edge.

When the distance from the outer edge is under 15 units, the vehicle shifts towards the left. Conversely, if the distance is from the inner edge, the vehicle shifts right.

This behavior is illustrated in the following Figures.

Furthermore, if the distance exceeds 22 units from the outer edge, the vehicle will veer to the right. Conversely, if it's from the inner edge, it will steer to the left.





5. 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:

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

6. The Camera Coding:

Initially, we transformed the image's color representation from RGB to HSV.

Next, we established a color mask by defining the specific green and red color ranges in the HSV system.

Afterward, we employed functions provided by the OpenCV library to isolate the regions corresponding to these masks within the image.

Subsequently, we applied functions to extract the area, taking into consideration their rectangular shape.

Following this, we computed the area covered by these masks, excluding those with less than 300 pixels.

We then initialized global variables to keep track of the count of masks for both colors.

To minimize potential noise during the live video, we iterated the minimum number of these masks for both colors ten times.

Subsequently, we established a connection between the Raspberry Pi controller and the ESP32 via the serial port, utilizing the serial library.

Lastly, we transmitted commands through the ESP32's serial port, allowing it to execute the necessary actions.

The End

KANAAN TEAM