Application Note

Ultrasonic Sensor (HC-SR04) Documentation for Reverse Parallel Parking Application

Pratham Sharma

ABSTRACT

This project aims to design and implement a robust 360-degree ultrasonic sensor fusion system for automated reverse parallel parking applications. Leveraging multiple ultrasonic sensors arranged around a vehicle platform, the system captures comprehensive distance measurements to reliably detect obstacles in all directions. Improving accuracy and reliability in parking scenarios enhances safety and driver assistance capabilities.

Table of Contents

List of Tables

Executive Summary

Purpose

This project aims to design and implement a robust 360-degree ultrasonic sensor fusion system for automated reverse parallel parking applications. Leveraging multiple ultrasonic sensors arranged around a vehicle platform, the system captures comprehensive distance measurements to reliably detect obstacles in all directions. Improving accuracy and reliability in parking scenarios enhances safety and driver assistance capabilities.

Methodology

Step	**Task Description**	**Key Considerations**
Hardware Setup	Mount 14 sensors, each with a 15°f field of view (FOV), covering 210° if arranged with no overlap.	Ensure stable, non-overlapping coverage
Microcontroller	Use ESP32 to control sensors and collect readings sequentially	Avoid simultaneous triggers (crosstalk)
Timing Control	Implement 50 ms delay between each sensor trigger	Reliable echo reception
Data Structuring	Format readings as angle-distance pairs, tag with timestamp	Enables synchronization and fusion
Data Fusion	Combine all readings into a 360° map (array or JSON object)	Ready for visualization/processing
Transmission	Send stitched data to Raspberry Pi via UART/SPI/WiFi	Use structured packets (e.g., JSON)
Visualization	On Raspberry Pi, plot the stitched data as a polar plot or spider web diagram	Real-time feedback and monitoring

Table 1: Stepwise Tasks and Key Considerations for 360° Ultrasonic Sensor Stitching

Key Findings

- Environmental Effects Can Affect Ultrasonic Performance Significantly
- Sensor Choice Matters for Automotive Use
- Data Fusion with Cameras Provides Superior Results
- Temperature and Humidity Compensation is Achievable in Real Time

Introduction

This document underlines the important stages required to develop 360° view using 14 ultrasonic sensors. It covers sensor arrangement, data collection, synchronization, and communication to a central processor (Raspberry Pi). This documentation guides us through the steps of synchronizing multiple ultrasonic sensors (with ESP32), managing timing to avoid cross-talk, and visualizing stitched data in Python.

What is reverse parallel parking ?

Reverse parallel parking refers to the process of parking a vehicle in a parallel parking spot primarily by backing into the space.This means the vehicle is positioned adjacent to and slightly ahead of the chosen parking space, then reversed into the slot along the curb.

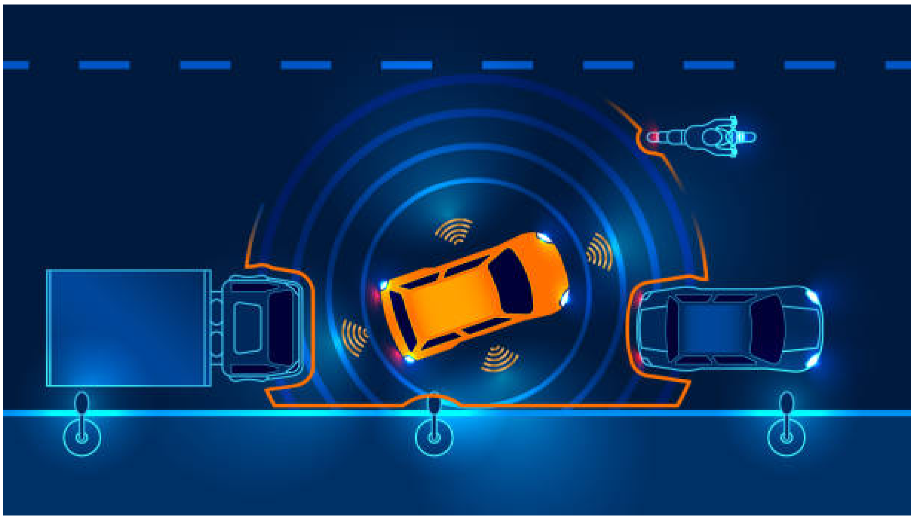


Figure 1: Parallel Park Car

What is ultrasonic sensing? and why sensor fusion is used?

Ultrasonic sensing is a technology that uses high-frequency sound waves to perceive the environment by detecting nearby obstacles. These sensors emit ultrasonic waves that bounce off objects and return as echoes, allowing the system to accurately measure the distance to those objects. Ultrasonic sensors are widely used in advanced driver assistance systems (ADAS) to support functions like parking, collision avoidance, and obstacle detection. They reliably provide precise distance measurements regardless of lighting conditions or object colors.

However, while ultrasonic sensors excel at measuring distances, they do not provide detailed information about the appearance, shape, or texture of objects. This is where cameras complement them by capturing rich visual data, including colors, textures, shapes, and edges, though cameras alone sometimes struggle with depth estimation, lighting variations, and transparent or reflective surfaces.

To overcome these individual limitations, ultrasonic sensing is fused with camera data. This sensor fusion combines the strengths of both technologies:

- Ultrasonic sensors provide precise, reliable distance measurements.
- Cameras provide detailed visual context to identify and classify objects.

By integrating these data sources, the system gains a more accurate and comprehensive understanding of the surroundings. This fusion enables better obstacle detection and improved driver alerts or autonomous control.

The fusion process often requires calibration to establish the spatial relationship between the camera and ultrasonic sensors. This calibration ensures that distance information from the ultrasonic sensors correctly maps onto corresponding parts of the camera's image, aligning distance data with visual context accurately for effective perception and decision-making.

while using the ultrasonic sensor for this application do note the following:

- Synchronization: Each scan cycle should be as fast as possible while ensuring no sensor crosstalk.
- Data Fusion Enhancements: For more advanced fusion, consider filtering (e.g., moving average) or interpolating between sensors.

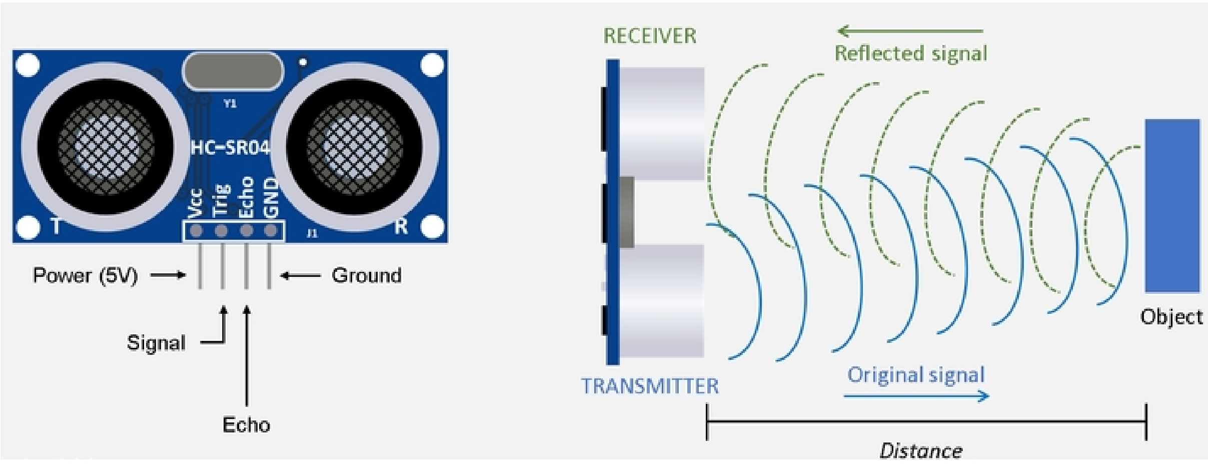


Figure 2: Ultrasonic sensing

Objectives for Autonomous Reverse Parallel Parking System

Ultrasonic Sensor Arrangement

- Use of 14 ultrasonic sensors that are placed strategically around the vehicle to achieve comprehensive 360-degree coverage. The front sensors (F1–F6) are tightly arranged end-to-end, to provide a full 90° coverage, each with a 15° field of view, the sensors should be arranged with their beams placed side-by-side without overlap or gaps. For example, the sensors can be positioned at angles: 315°, 330°, 345°, 0°, 15°, and 30°.

  This arrangement covers the front arc fully from 315° through 30° (wrapping across the 0° boundary) for a total of 90° coverage, ensuring no blind spots. The rear sensors (B1–B6) are similarly arranged from 150° through 210°, ensuring continuous rear coverage without overlap or gaps. Two side sensors, positioned at 90° (Right Side) and 270° (Left Side), complete the system. This configuration ensures true end-to-end coverage for both front and rear zones, with possible blind spots only along the vehicle flanks beyond the dedicated side sensors. The user interface and control logic are designed for serial integration of all 14 sensors, providing unified real-time data for autonomous maneuvers.

- Each sensor has a specific field of view (15° cone), with sequential triggering and timing control to avoid cross-talk interference.

Sensor Data Collection and Timing

- Trigger sensors sequentially with a fixed delay between each trigger (50 ms) to prevent signal interference.

- Collect distance readings tagged with angle and timestamp to sync multi-sensor data

Sensor Fusion

- Fuse ultrasonic sensor data with camera data (preferably fisheye or wide-angle) to cover weaknesses of each sensor modality.

Data Processing and Algorithm

- Use algorithms to stitch sensor readings into a 360° obstacle map.
- Interpolate between sensor readings for a smooth obstacle boundary.
- Use machine learning models or probabilistic filters (Kalman filters, Bayesian inference) to reduce noise, improve obstacle detection, and handle ambiguities.
- Semantic segmentation and occupancy grids can be employed for higher-level scene understanding.

Visualization and Control

- Visualize the fused data in real-time as polar or radar plots ("spider web" diagrams).
- Integrate with control logic to assist or automate reverse parallel parking maneuvers.
- Provide timely audio/visual alerts or automate steering and throttle control.

Implementation Platform

- Microcontroller like ESP32 for sensor control and data acquisition.
- A more powerful processor like Raspberry Pi for data fusion, visualization, and decision-making.
- Communication between devices via UART, SPI, or WiFi.

Environmental and Practical Considerations

- Compensate for temperature and humidity effects on ultrasonic sensor accuracy.
- Use waterproof and industrial-grade sensors (e.g., JSN-SR04T) for reliability in outdoor conditions.
- Consider error margins and sensor placement according to vehicle geometry and parking space dimensions (as per Indian IRC standards or other relevant regulations).

Requirements

- HCSR04 - Ultrasonic sensor
- ESP32
- Raspberry Pi
- Neccessary cables and interfacing equipment.
- Knowledge about sensors, microcontrollers and microprocessors.

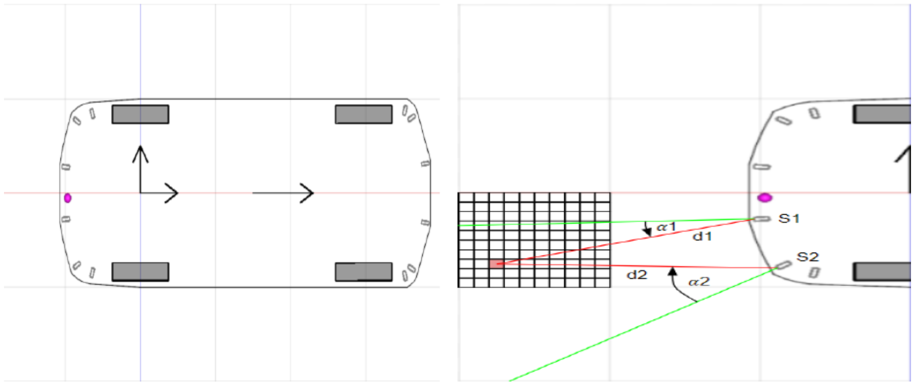


Figure 3: Mounting of ultrasonic sensors & schematic of an ultrasonic sensor gird map filling step for one exemplary grid cell and one exemplary signal way

Scope

- *Rear and Blind Spot Coverage:* The system is designed to detect objects behind the vehicle during reverse motion, covering the entire rear side including blind spots for enhanced safety.
- *Sensor Configuration:* Implemented with a four-sensor rear ultrasonic setup, extendable to six sensors for increased coverage and reliability.
- *Detection Range:* Capable of detecting obstacles at distances up to 4 meters (Theoretically), meeting typical park assist system requirements, and also able to sense objects at very close proximity (min. of 2 cm as per the datasheet) to the vehicle.
- *Sensor Characteristics:* Ultrasonic sensors utilized have specific properties critical to automotive applications including directivity (influenced by vibrating surface size, shape, and frequency), ringing time, sensitivity, and sound pressure, ensuring effective object detection.
- *Sensor Fusion:* Integration of ultrasonic sensors with cameras (and optionally other sensors like radar), enabling robust obstacle detection and scene understanding.
- *Real-Time Data Processing:* Acquisition, processing, and visualization of sensor data for immediate feedback and control.
- *Environmental Adaptation:* Operation in various lighting, weather, and environmental conditions, utilizing waterproof and temperature-compensated sensors for reliability.
- *Integration with Existing Platforms:* Can be adapted for microcontroller-based systems (ESP32, Arduino) and edge computing devices (Raspberry Pi), with options for UART/SPI/WiFi communication.
- *Sensor Quantity and Placement:* Limited to the arrangement and FOV of the installed 14 sensors.
- *Supported Maneuvers:* Primarily designed for reverse and parallel parking; not intended for high-speed obstacle avoidance or full autonomous navigation.
- *Low-Cost Implementation:* Emphasizes cost-effective design suited for widespread automotive application without compromising on essential functionality and diagnostics.

Algorithm for 360° Ultrasonic Sensor Fusion and Bird's-Eye View Visualization

1. *Sensor Placement & Angle Assignment*

   Fourteen ultrasonic sensors are placed around the vehicle for comprehensive 360-degree coverage:

   - **Front (F1–F6):** Sensors at 315°, 330°, 345°, 0°, 15°, and 30°, providing contiguous 90° front coverage.
   - **Rear (B1–B6):** Sensors at 150°, 160°, 170°, 180°, 190°, and 210°, forming a tight rear arc.
   - **Sides:** Sensors at 90° (right) and 270° (left) minimize lateral blind spots.

   Each sensor has a 15° field of view, placed with beams side-by-side for seamless coverage.

2. *Sequential Triggering & Timing Control*

   Sensors are triggered sequentially in a predefined order with a fixed inter-trigger delay (e.g., 50–150 ms) to prevent crosstalk. The microcontroller (e.g., ESP32) waits for each echo (or timeout) before proceeding to the next sensor.

3. *Data Acquisition*

   For each sensor:

   - Measure distance to obstacle (in cm)
   - Record the corresponding sensor angle
   - Log the timestamp of the reading

   After each scan cycle (all 14 sensors), a set of {angle, distance} pairs with timestamp is produced.

4. *Data Structuring & Transmission*

   Structure the acquired data as a list (or JSON object) of {angle, distance} pairs plus timestamp. **Transmit** the structured data to the Raspberry Pi via UART/SPI/WiFi.

5. *Fusion and Bird's-Eye View Generation on Raspberry Pi*

   The Raspberry Pi receives ultrasonic distance data and synchronizes it with camera input from fisheye or wide-angle cameras.

   Fusion Process:

   - Calibrate sensor positions relative to the camera's view for spatial alignment.
   - Overlay obstacle distances from ultrasonic readings as points or arcs on the bird's-eye (top-down) projection.
   - Merge visual context (objects, lane markings) from the camera feed with precise ultrasonic range data.

   Visualization is implemented in **Pygame**:

   - Pygame draws a polar or spider plot showing ultrasonic detections at correct angles/distances.
   - The bird's-eye processed camera view forms the background, with sensor-detected obstacles rendered on top.
   - Visualization updates happen in real-time, reflecting vehicle movement and changing surroundings.

6. *Result*

   The driver or autonomous system observes a real-time bird's-eye view: a fused graphical representation of nearby obstacles and free space around the vehicle—ideal for safe reverse parallel parking maneuvers.

Implementation and various considerations

Programming 8 ultrasonic sensors

Listing 1: Pin Definitions and Main Loop for 8 Ultrasonic Sensors

```
const int TRIG_PINS[8] = {5, 22, 12, 14, 32, 26, 2, 18}; // Example GPIOs
const int ECHO_PINS[1] = {4, 23, 13, 34, 33, 25, 15, 19}; // Must be INPUT capable

HardwareSerial SerialToPi(1); // use UART1

float readDistanceCM(int trigPin, int echoPin) {
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  long duration = pulseIn(echoPin, HIGH, 30000); // Timeout: 30ms
  float distance = duration * 0.034 / 2;
  return distance / 10.0; // Convert mm to cm
}

void setup() {
  SerialToPi.begin(115200, SERIAL_8N1, 18, 19); // RX=18, TX=19
  Serial.begin(115200);

  for (int i = 0; i < 8; i++) {
    pinMode(TRIG_PINS[i], OUTPUT);
    pinMode(ECHO_PINS[i], INPUT);
    digitalWrite(TRIG_PINS[i], LOW);
  }
}

void loop() {
  unsigned long timestamp = millis();
  float distances[8];

  for (int i = 0; i < 8; i++) {
    distances[i] = readDistanceCM(TRIG_PINS[i], ECHO_PINS[i]);
  }

  // --- Format: timestamp,0,0,0,0,0,0,0.000,0.000,... (6 dummy, 8 real), trailing dot ---
  SerialToPi.print(timestamp);
  for (int i = 0; i < 6; i++) {
    SerialToPi.print(",0");
  }
  for (int i = 0; i < 8; i++) {
    SerialToPi.print(",");
    SerialToPi.print(distances[i], 3);
  }
  SerialToPi.println(".");

  // Debug USB Output
  Serial.print(timestamp);
  for (int i = 0; i < 6; i++) {
    Serial.print(",0");
  }
  for (int i = 0; i < 8; i++) {
    Serial.print(",");
    Serial.print(distances[i], 3);
  }
  Serial.println(".");
```

```

  // Optional delay to avoid crosstalk
  // delay(80);
}
```

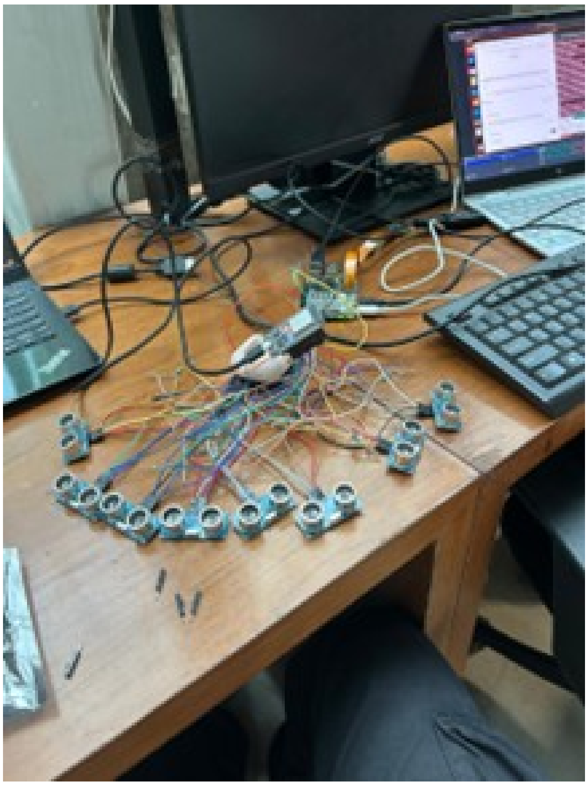


Figure 4: 8-ultrasonic sensor setup

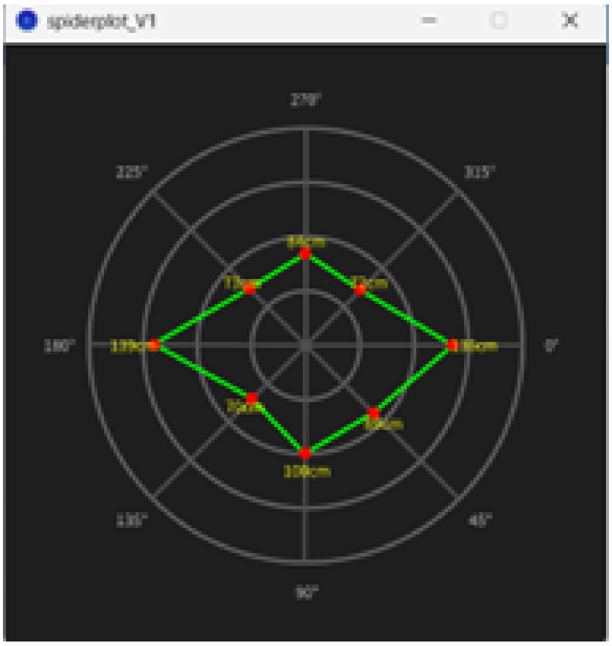


Figure 5: Spider web created using Processing

Key considerations to mount ultrasonic sensors

When you mount ultrasonic sensors on an autonomous vehicle, you need to **think of each sensor as a little "range-finder" whose exact 3D position and pointing direction must be known in the vehicle's coordinate frame. In practice that means:**

- **Fix each sensor's (x, y, z) in the ISO 8855 vehicle frame:**
  - Define your vehicle reference point (e.g., sprung-mass CG) and the X_V (forward), Y_V (left), Z_V (up) axes.
  - Measure (and hard-code) each sensor's longitudinal (X_V), lateral (Y_V), and height (Z_V) offset from that point.
  - This enables you to precisely project every echo onto your 2D grid or bird's-eye-view (BEV) map.
- **Align the beam axis with the local road plane:**
  - In ISO 8855 terms, your sensor's beam should lie in the tyre-axis X_T–Y_T plane (parallel to the road), with Z_T up.
  - Mount the sensor flush to the bumper so that its principal axis neither points into the hood nor toward the ground

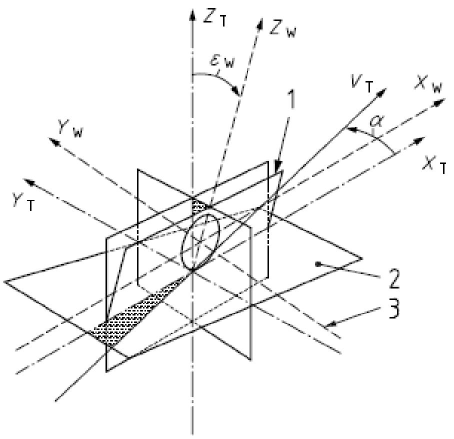
2.14
tyre axis system
(X_T, Y_T, Z_T)
axis system (2.3) whose X_T and Y_T axes are parallel to the local **road plane** (2.7), with the Z_T axis normal to the local road plane, where the orientation of the X_T axis is defined by the intersection of the **wheel plane** (4.1) and the road plane, and the positive Z_T axis points upward

NOTE A local tyre axis system may be defined at each wheel (see Figure 3).

2.15
tyre coordinate system
(x_T, y_T, z_T)
coordinate system (2.4) based on the **tyre axis system** (2.14) with the origin fixed at the **contact centre** (4.1.4)

2.16
wheel axis system
(X_W, Y_W, Z_W)
axis system (2.3) whose X_W and Z_W axes are parallel to the **wheel plane** (4.1), whose Y_W axis is parallel to the **wheel-spin axis** (4.1.1), and whose X_W axis is parallel to the local **road plane** (2.7), and where the positive Z_W axis points upward

NOTE A local wheel axis system may be defined for each wheel (see Figure 3).



Key
1 wheel plane
2 road plane
3 wheel-spin axis

Figure 6: Tyre and wheel axis system

4 Vehicle geometry and masses

4.1
wheel plane
plane normal to the **wheel-spin axis** (4.1.1), which is located halfway between the rim flanges

4.1.1
wheel-spin axis
axis of wheel rotation

NOTE This axis is coincident with the Y_W axis.

4.1.2
wheel centre
point at which the **wheel-spin axis** (4.1.1) intersects the **wheel plane** (4.1)

NOTE This point is the origin of the **wheel coordinate system** (2.17).

4.1.3
contact line
intersection of the **wheel plane** (4.1) and the **road plane** (2.7)

Figure 7: Vehicle geometry

- **Ensure full, overlapping coverage:**
  - The figure shows 12 sensors spaced around the front and rear bumpers. You want their fields-of-view ($\alpha_1, \ldots, \alpha_2$ in the right-hand schematic) to overlap, so there are no blind-spots—especially at the corners.
  - Use the known beam aperture (semi-discrete angle bins) to size your BEV grid cells (d_1, d_2 in the diagram) so every cell in front of and behind the car is "seen."
- **Choose a sensible mounting height:**
  - Too low: you pick up many ground echoes.
  - Too high: small or low-lying obstacles may vanish under your beam.
  - Typically, aim for a height where the beam just grazes the road at its maximum useful range.
- **Avoid obstructions and beam distortions:**
  - Keep sensors away from metallic trim, tow-hooks, or deep recesses that could shadow or reflect their acoustic pulse.
  - Ensure the plastic bumper cover over each sensor is as acoustically transparent as possible.

By treating every sonar as a tiny node in your BEV grid—each with a known XYZ mount point and a fixed beam direction—you can fuse all returns into a coherent obstacle map with no surprises.

Sensor Placement Guides

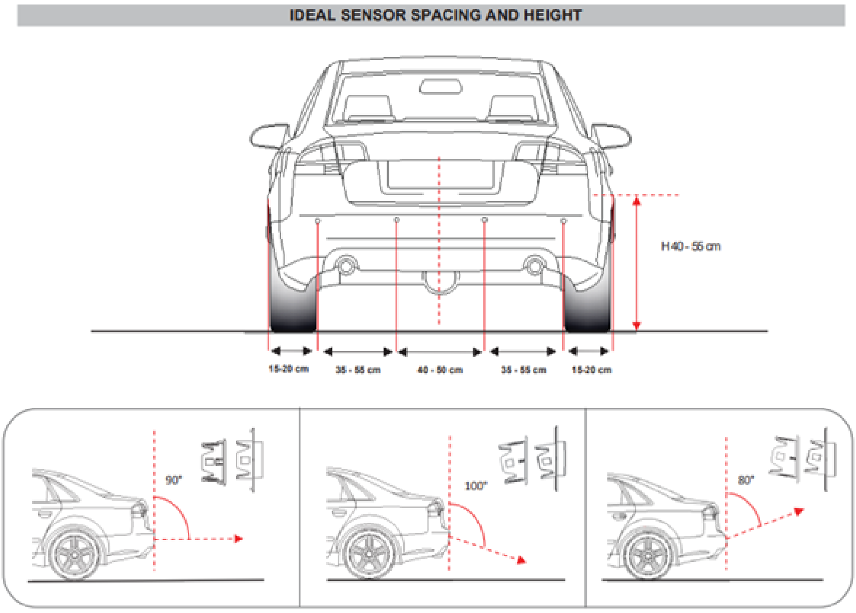
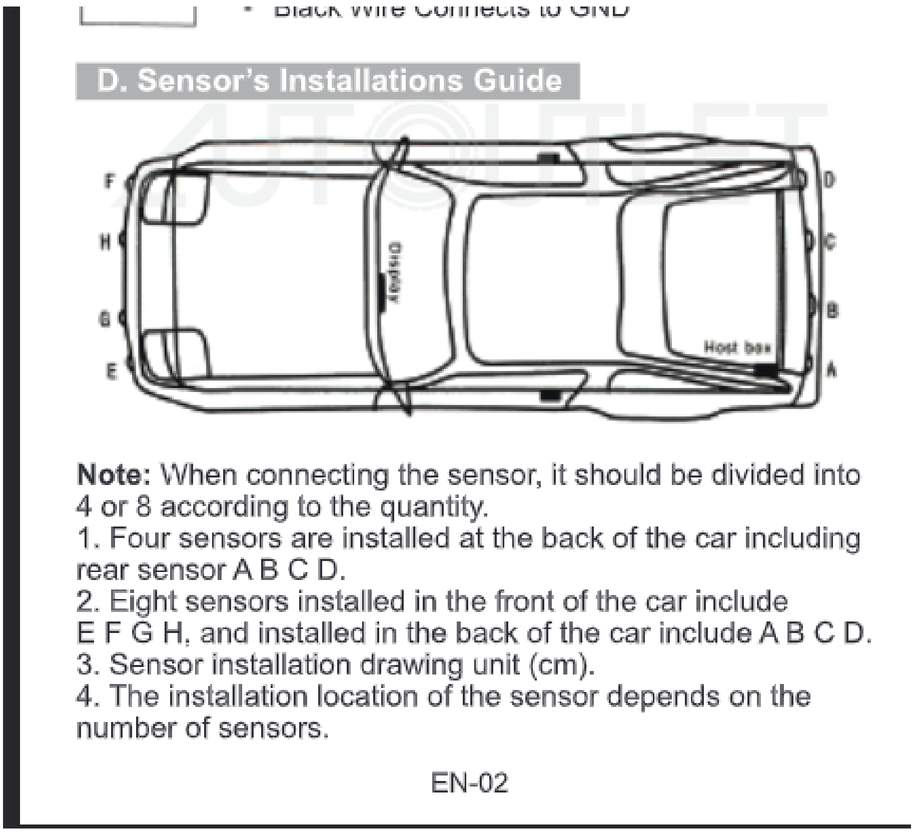


Figure 8: Caption



• Black Wire Connects to GND

D. Sensor's Installations Guide

Note: When connecting the sensor, it should be divided into 4 or 8 according to the quantity.

1. Four sensors are installed at the back of the car including rear sensor A B C D.
2. Eight sensors installed in the front of the car include E F G H, and installed in the back of the car include A B C D.
3. Sensor installation drawing unit (cm).
4. The installation location of the sensor depends on the number of sensors.

EN-02

Figure 9: Caption

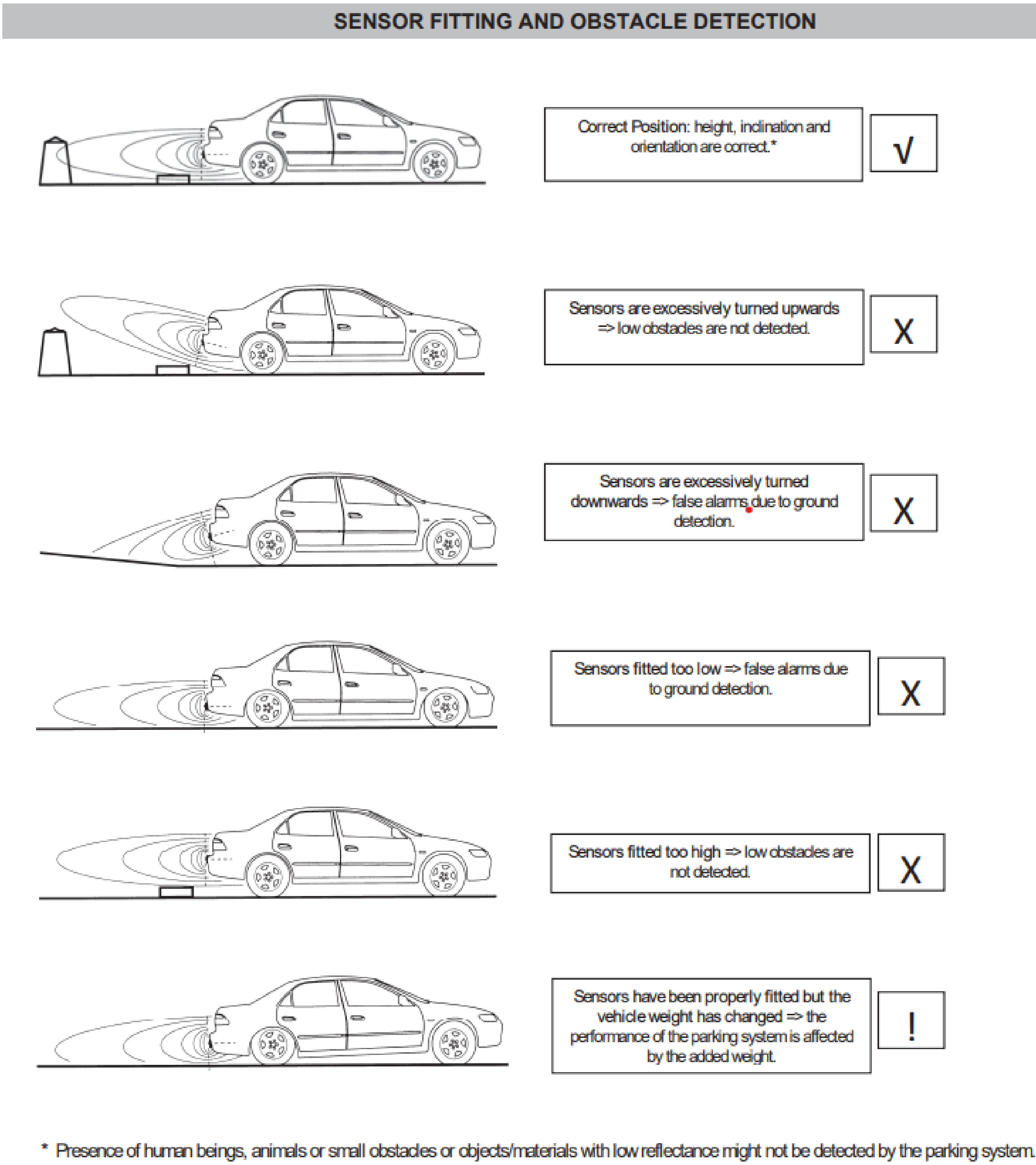


Figure 10: Caption

Dealing with Environmental Factors

Legend:

- RTTC = Reference Target Temperature Compensation
- RH = Relative Humidity

Temperature is a big player when it comes to ultrasonic sensor accuracy. As the temperature changes,

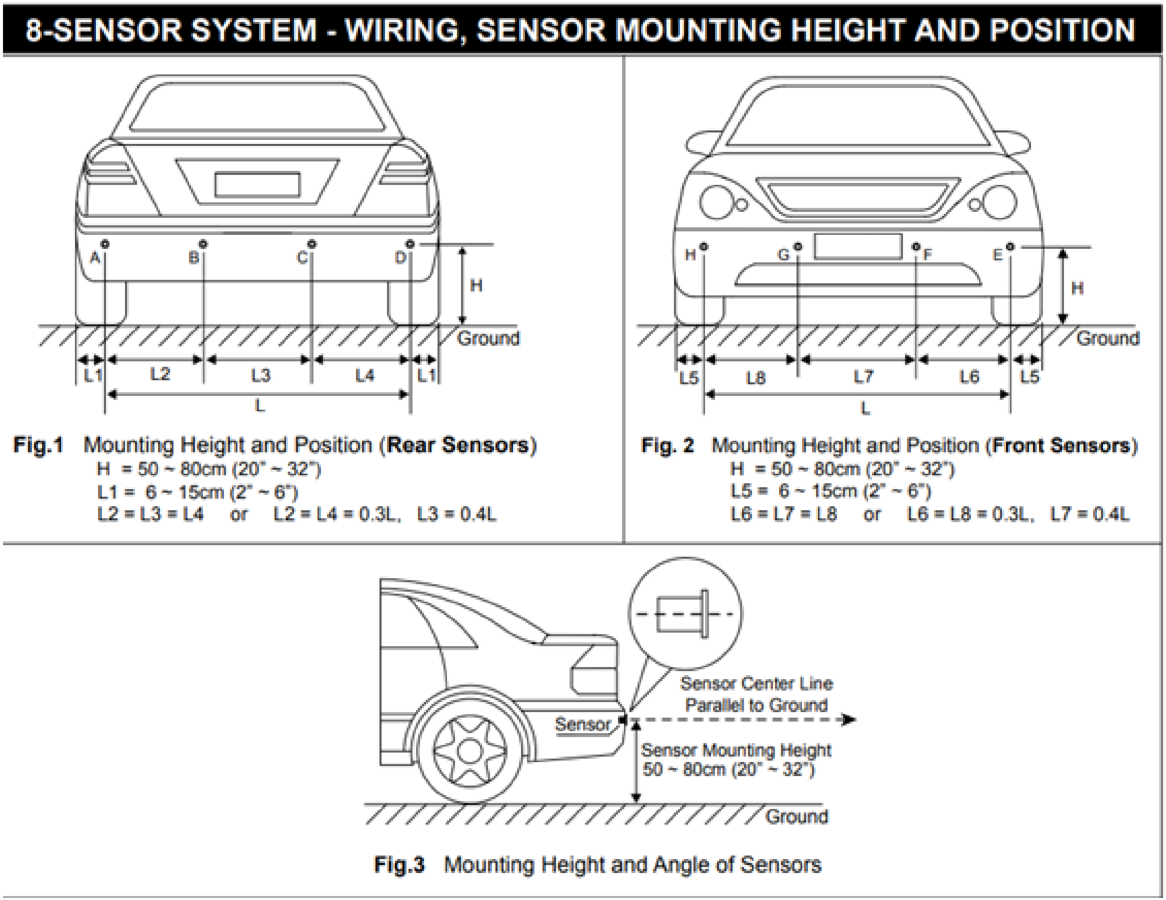


Figure 11: Caption

Chart 2. **Beeper Only System (F2R4B , F4R4B)**
Obstacle Distance, Alert Zones, and Audible Alert

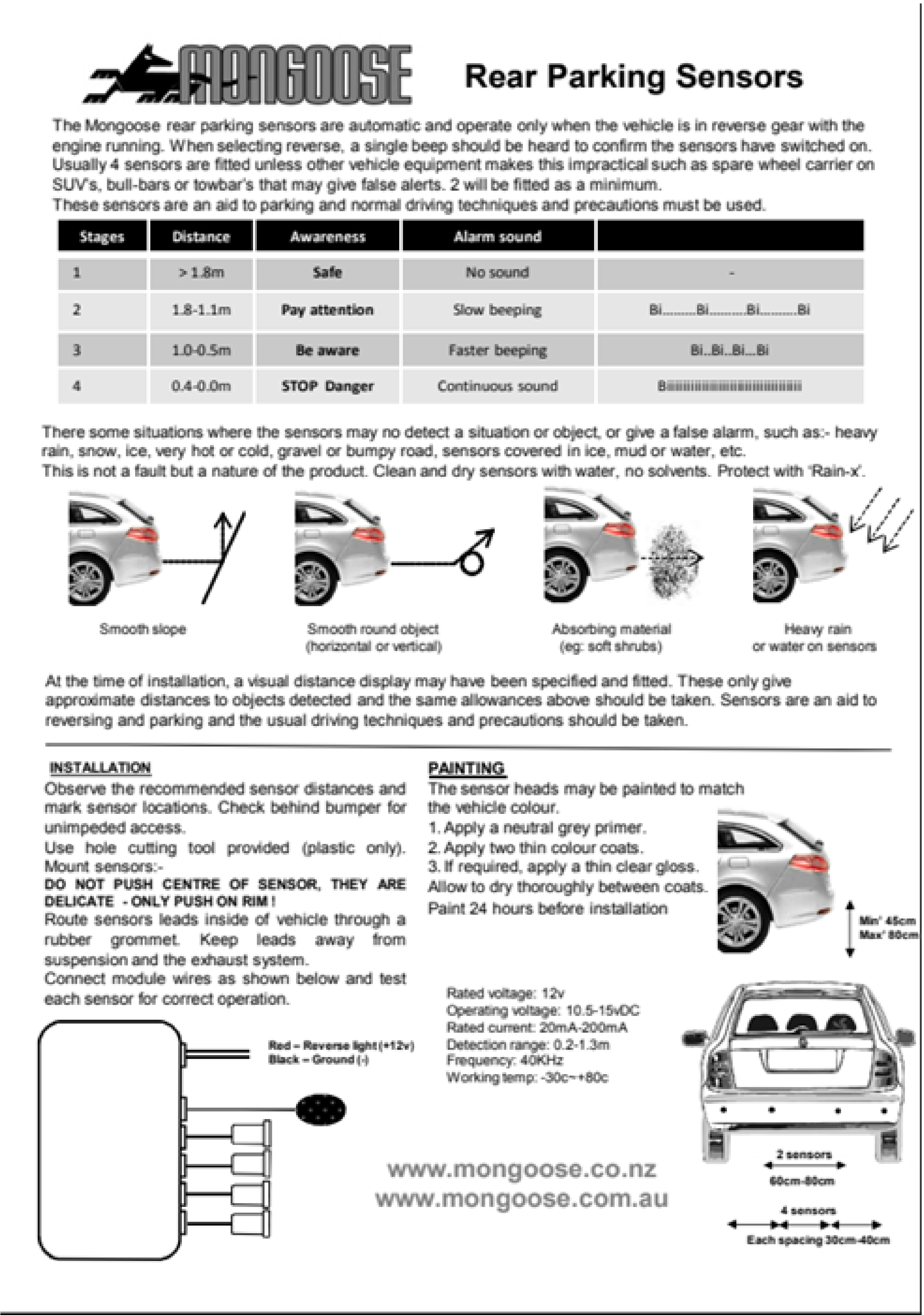
Alert Zones	Obstacle Distance	Beeping (Audible Alert)
Safe Zone	6.6~4.9ft	Starting to beep at 4.9ft
Safe Zone	4.9~3.3ft	Beeping Fast to Faster
Safe Zone	3.3~2.3ft	Beeping Faster to Fastest
Warning Zone	2.3~1.0ft	Beeping Fastest
Stop Zone	1.0~0.0ft	Beeping Continuously

- *(Note: 1ft = 0.3m)*

Figure 12: Caption

so does the speed of sound, which can throw off your distance calculations. For every degree Celsius change, the speed of sound varies by about 0.17%. This means that a 20°C temperature swing could lead to errors of several centimeters in your measurements.

To combat this, many high-quality ultrasonic sensors come with built-in temperature compensation. Some even use rapid temperature compensation systems to adjust for quick temperature changes. If you're working on a project that needs to be super accurate, consider using a sensor with this feature or



MONGOOSE **Rear Parking Sensors**

The Mongoose rear parking sensors are automatic and operate only when the vehicle is in reverse gear with the engine running. When selecting reverse, a single beep should be heard to confirm the sensors have switched on. Usually 4 sensors are fitted unless other vehicle equipment makes this impractical such as spare wheel carrier on SUV's, bull-bars or towbar's that may give false alerts. 2 will be fitted as a minimum.
These sensors are an aid to parking and normal driving techniques and precautions must be used.

Stages	Distance	Awareness	Alarm sound	
1	> 1.8m	Safe	No sound	-
2	1.8-1.1m	Pay attention	Slow beeping	Bi.........Bi.........Bi.........Bi
3	1.0-0.5m	Be aware	Faster beeping	Bi..Bi..Bi...Bi
4	0.4-0.0m	STOP Danger	Continuous sound	Biiiiiiiiiiiiiiiiiiiiiiiiiiii

There some situations where the sensors may no detect a situation or object, or give a false alarm, such as:- heavy rain, snow, ice, very hot or cold, gravel or bumpy road, sensors covered in ice, mud or water, etc.
This is not a fault but a nature of the product. Clean and dry sensors with water, no solvents. Protect with 'Rain-x'.

At the time of installation, a visual distance display may have been specified and fitted. These only give approximate distances to objects detected and the same allowances above should be taken. Sensors are an aid to reversing and parking and the usual driving techniques and precautions should be taken.

INSTALLATION

Observe the recommended sensor distances and mark sensor locations. Check behind bumper for unimpeded access.
Use hole cutting tool provided (plastic only). Mount sensors:-
DO NOT PUSH CENTRE OF SENSOR, THEY ARE DELICATE - ONLY PUSH ON RIM !
Route sensors leads inside of vehicle through a rubber grommet. Keep leads away from suspension and the exhaust system.
Connect module wires as shown below and test each sensor for correct operation.

PAINTING

The sensor heads may be painted to match the vehicle colour.
1. Apply a neutral grey primer.
2. Apply two thin colour coats.
3. If required, apply a thin clear gloss.
Allow to dry thoroughly between coats.
Paint 24 hours before installation

Rated voltage: 12v
Operating voltage: 10.5-15vDC
Rated current: 20mA-200mA
Detection range: 0.2-1.3m
Frequency: 40KHz
Working temp: -30c~+80c

www.mongoose.co.nz
www.mongoose.com.au

Figure 13: Caption

adding an external temperature sensor to your setup.

Environmental Factor	Effect on Ultrasonic Sensor	Details/Notes
Air Temperature	Greatest impact on accuracy	Speed of sound increases with temperature; readings appear closer as temperature rises. A 20°C swing can cause up to ±8.5 cm error at 1 m if uncompensated. Internal compensation helps, but rapid changes or external heating/-cooling can cause lag.
Humidity	Minimal at room temperature	Up to 0.036% change per 10% RH at room temperature; more effect at high temperatures due to lighter water vapor molecules.
Air Pressure & Altitude	Negligible in most cases	Less than 1% speed change up to 3,000 m altitude. Atmospheric fluctuations at a fixed location are negligible.
Air Currents & Wind	Can cause unstable readings or signal loss	Strong winds, turbulence, or hot objects (e.g., red-hot metal) can scatter/deflect echoes, causing loss of signal.
Paint Mist	No effect unless settling	Mist does not affect operation, but buildup on transducer surface can reduce sensitivity.
External Noise	Possible interference	Usually distinguished from target echoes. If noise is at the same frequency and higher amplitude, it may cause errors (e.g., compressed air jets).
Types of Gas	Major errors in non-air gases	Designed for atmospheric air; gases like CO_2 can cause large errors or loss of function due to different sound speeds/attenuation.
Tank Configurations & Dimensions	Affects echo path and accuracy	Flat-bottom, straight-sided tanks are easiest for accurate measurement; irregular shapes complicate calculations.
External Reference Targets	Improves compensation	RTTC accessory uses a reference target in the measurement path for fast, accurate compensation during rapid temperature swings or diurnal changes.
Rapid Temperature Changes	Compensation lag possible	Internal sensors may not track fast ambient changes; external reference targets or accessories can help maintain accuracy.

Table 2: Environmental Factors Affecting Ultrasonic Sensor Accuracy

Humidity and air pressure also affect ultrasonic sensors, but to a lesser extent. At room temperature, humidity doesn't make much difference, but it becomes more significant at higher temperatures. Air pressure changes with altitude can also slightly impact readings, but it's usually not a big concern unless you're working at extreme heights.

Temperature and Humidity Effects and Mitigation in HC-SR04 Ultrasonic Sensor Measurements

The HC-SR04 ultrasonic sensor is widely used for distance measurement in robotics and embedded systems. Its operation is based on measuring the time taken for an ultrasonic pulse to travel to an object and back. The sensor assumes a fixed speed of sound, but in reality, temperature and humidity affect this speed, leading to errors in distance measurement.

1. Effect of Temperature and Humidity on Sound Velocity

1.1 Temperature

- *Physical Principle:* The speed of sound in air increases as temperature rises because molecules move faster and transmit sound waves more quickly.
- *Formula:*

$$V = 331.3 + 0.606 \times T$$

  where V is the speed of sound in m/s and T is the temperature in °C.

- *Impact on Measurement:*
  - If the sensor assumes a constant speed (e.g., 343 m/s at 20°C), but the actual temperature is different, the calculated distance will be off.
  - Higher temperature: Sound travels faster $\rightarrow$ measured distance appears shorter than actual.
  - Lower temperature: Sound travels slower $\rightarrow$ measured distance appears longer than actual.

Real-time temperature is required through:

Sensor	*Measures*	*Temp Range (°C)*	*Temp Accuracy*	*Hum Range (%)*	*Hum Accuracy*
DHT11	Temp & Humidity	0 to 50	±2.0	20 to 80	±5
DHT22	Temp & Humidity	-40 to 80	±0.5	0 to 100	±2–5
LM35	Temperature only	-55 to 150	±0.5	–	–
DS18B20	Temperature only	-55 to 125	±0.5	–	–
DHT12	Temp & Humidity	-20 to 60	±0.5	20 to 95	±5

Table 3: Comparison of Temperature and Humidity Sensors

1.2 Humidity

- *Physical Principle:* More water vapor (higher humidity) lowers the density of air, which slightly increases the speed of sound.
- *Formula (with humidity):*

$$V = 331.4 + 0.6 \times T + 0.0124 \times RH$$

  where RH is relative humidity (%).

- *Impact on Measurement:* Humidity has a minor effect at room temperature but can become significant at high temperatures or in very humid environments.

2. Mitigation Strategies

2.1 Temperature Compensation

Use a temperature sensor (e.g., DHT11 or DHT22) to measure ambient temperature in real time and adjust the speed of sound in the distance calculation. This reduces distance errors from several centimeters (for a 20°C swing) to just a few millimeters.

2.2 Temperature and Humidity Compensation

Use a sensor that measures both temperature and humidity and apply the full compensation formula to further improve accuracy in varying humidity environments.

Temperature (°C)	*Speed of Sound (m/s)*	*Error at 100 cm (approx.)*
-10	325.2	+5.5 cm
20	343.4	0 cm (reference)
40	355.6	-3.6 cm

Table 4: Effect of Temperature on Speed of Sound and Corresponding Measurement Error at 100 cm

3. Quantitative Example

Distance error at 100 cm (no compensation):

With compensation, error is typically reduced to less than 1 cm, even for large temperature swings.

5. Code Example: HC-SR04 with Temperature and Humidity Compensation

Listing 2: Arduino sketch for HC-SR04 with DHT11 temperature and humidity compensation

```
#include // For DHT sensor
#include          // DHT sensor library
#include       // HC-SR04 library

// Pin definitions
#define DHTPin 6      // DHT11 signal pin
#define DHTType DHT11 // Sensor type
#define Trigger_pin 13 // HC-SR04 trigger pin
#define Echo_pin 10   // HC-SR04 echo pin
#define Max_distance 400 // Maximum distance to measure (in cm)

// Initialize objects
DHT dht(DHTPin, DHTType);
NewPing sonar(Trigger_pin, Echo_pin, Max_distance);

void setup() {
  Serial.begin(9600);
  dht.begin();
}

void loop() {
  delay(1000); // Allow DHT11 to stabilize

  // Read temperature and humidity
  float temp = dht.readTemperature(); // in Celsius
  float hum = dht.readHumidity(); // in %

  // Calculate speed of sound with temp and humidity compensation
  // Speed of sound (m/s) = 331.4 + 0.606 * temp + 0.0124 * hum
  float sound_speed = 331.4 + (0.606 * temp) + (0.0124 * hum);

  // Convert speed of sound to cm/us (1 m = 100 cm, 1 s = 1,000,000 us)
  float sound_cm_per_us = sound_speed * 100 / 1000000.0;

  // Get median duration from sensor (better than a single reading)
  int iterations = 5;
  float duration = sonar.ping_median(iterations);

  // Calculate distance (cm)
  float distance = (duration / 2.0) * sound_cm_per_us;

  // Output results
  Serial.print("Temp: ");
  Serial.print(temp);
  Serial.print(" C, Humidity: ");
  Serial.print(hum);
```

```
  Serial.print(" %, Speed of Sound: ");
  Serial.print(sound_speed);
  Serial.print(" m/s, Distance: ");
  if (distance >= 400 || distance <= 2) {
    Serial.println("Out of range");
  } else {
    Serial.print(distance);
    Serial.println(" cm");
  }
}
```

5. Conclusion

Without compensation, the HC-SR04 can have errors of several centimeters due to temperature and humidity changes. With real-time compensation using a DHT11 sensor, errors are reduced to less than 1 cm, making the sensor suitable for more demanding applications. Mitigation is straightforward: always use a temperature (and optionally humidity) sensor, and adjust the speed of sound in your calculations.

Minimizing Interference and Cross-Talk

When you're using multiple ultrasonic sensors or working in an environment with other ultrasonic devices, interference can be a real headache. This is known as cross-talk, and it happens when one sensor picks up signals from another.

To minimize cross-talk, you can use a technique called time division multiple access (TDMA). This is a fancy way of saying you make your sensors take turns. By carefully timing when each sensor sends out its pulse, you can avoid overlap and reduce interference.

Another approach is to use different frequencies for each sensor, known as frequency division multiple access (FDMA). However, this can be tricky and expensive to implement with standard ultrasonic sensors.

Enhancing Accuracy and Reliability

To boost the accuracy and reliability of your ultrasonic sensors, consider these tips:

- Choose the right sensor for your environment. Some sensors are better suited for outdoor use or harsh conditions.
- Pay attention to sensor placement. Make sure the sensor face is perpendicular to the target surface for the best results.
- Use signal processing techniques. Advanced algorithms can help filter out noise and improve accuracy.
- Consider using multiple sensors. This can provide redundancy and help verify readings.
- Regularly calibrate your sensors. This helps maintain accuracy over time.

Timing Problem Solution

Approaches to Prevent Ultrasonic Sensor Interference

1. Time Division Multiple Access (TDMA)

- **Each sensor is assigned a unique time slot to operate, ensuring only one sensor is active at any given time.**
- **This method evenly divides the operation time among all sensors.**
- **Limitation:** In environments with multiple vehicles (each with its own sensors), it is impossible to coordinate all sensors and prevent overlapping time slots, leading to potential interference.

2. Frequency Division Multiple Access (FDMA)

- **Each sensor operates at a different frequency, allowing simultaneous operation without mutual interference.**
- **Limitation:** Manufacturing ultrasonic transducers with different resonance frequencies and wide bandwidths is costly and impractical for most sensor manufacturers.

3. Code Division Multiple Access (CDMA)

- **Each transmitter (sensor) is assigned a unique orthogonal code as its ID.**
- **Sensors transmit signals encoded with their unique code.**
- **Receivers distinguish between signals by correlating received signals with these orthogonal codes.**
- **Advantage:** CDMA does not require time or frequency resource allocation, making it effective even in environments with many independent sensors (like multiple vehicles).
- **Effectiveness:** Orthogonal codes only correlate strongly with themselves, allowing the system to identify and separate each sensor's signal.
- **Application:** Simulations and implementations have shown that CDMA can overcome interference and reliably detect obstacles in vehicular ultrasonic sensing systems.

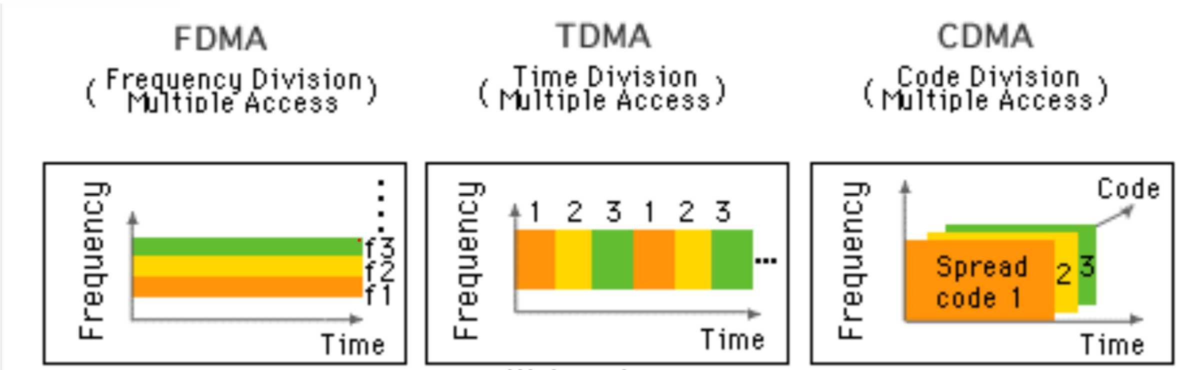


Figure 14: FDMA, TDMA and CDMA

Interference from the Other Transducer

CDMA is an efficient multiple access technique in terms of resource usage. However, the near–far problem is known as a weak point. A strong signal of a mobile station located near the base station interferes with detection of a weak signal of the mobile station far from the base station. Likewise, if two or more transducers transmit an ID sequence, the reflected wave has a smaller amplitude than the interference, as shown in Figure 3. The correlation between the interference and the ideal ID sequence is stronger than the correlation between the demodulated ID and the ideal ID sequence. In this situation, one solution is to control the signal strength of the transmitter so that the base station receives signals of the same strength from all transmitters. However, a single controller cannot change the sound pressure level of ultrasonic sensors on all the automotive. Therefore, the receiver needs a solution to overcome the interference by additional signal processing and adds a zero–crossing detector before the demodulation step.

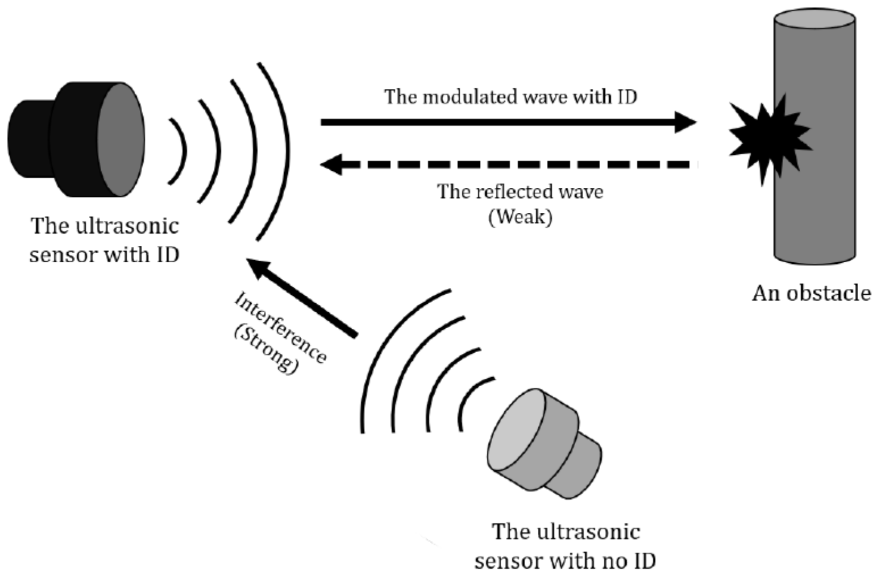


Figure 15: Near-Far Problem

CDMA Code Implementation

Listing 3: CDMA Implementation Code

```
#define NUM_SENSORS 8

// Define trigger and echo pins for each sensor
const uint8_t trigPins[NUM_SENSORS] = {2, 4, 6, 8, 10, 12, 14, 16};
const uint8_t echoPins[NUM_SENSORS] = {3, 5, 7, 9, 11, 13, 15, 17};

// Each sensor has a unique 8-bit code (example orthogonal-like codes)
const uint8_t codes[NUM_SENSORS][8] = {
  {1,0,1,0,1,0,1,0},
  {0,1,0,1,0,1,0,1},
  {1,1,0,0,1,1,0,0},
  {0,0,1,1,0,0,1,1},
  {1,0,0,1,1,0,0,1},
  {0,1,1,0,0,1,1,0},
  {1,1,1,0,0,1,0,1},
  {0,0,0,1,1,0,1,0}
};

// Duration of each bit pulse in microseconds
#define BIT_PULSE_DURATION 10
```

```

void setup() {
  Serial.begin(115200);

  // Initialize pins
  for (uint8_t i = 0; i < NUM_SENSORS; i++) {
    pinMode(trigPins[i], OUTPUT);
    digitalWrite(trigPins[i], LOW);
    pinMode(echoPins[i], INPUT);
  }
}

void loop() {
  for (uint8_t sensor = 0; sensor < NUM_SENSORS; sensor++) {
    long duration = sendCodedPulseAndReadEcho(sensor);
    float distance_cm = duration * 0.034 / 2.0; // Speed of sound approx. 0.034 cm/us

    Serial.print("Sensor ");
    Serial.print(sensor);
    Serial.print(" distance: ");
    if (duration == 0) {
      Serial.println("Out of range");
    } else {
      Serial.print(distance_cm);
      Serial.println(" cm");
    }

    delay(50); // Short delay before next sensor
  }
}

// Function to send coded pulse (simulated CDMA) and read echo
long sendCodedPulseAndReadEcho(uint8_t sensor) {
  // Send coded pulse
  for (uint8_t bit = 0; bit < 8; bit++) {
    digitalWrite(trigPins[sensor], codes[sensor][bit] ? HIGH : LOW);
    delayMicroseconds(BIT_PULSE_DURATION);
  }
  digitalWrite(trigPins[sensor], LOW);

  // Wait a short time before reading echo to allow pulse to propagate
  delayMicroseconds(100);

  // Read echo pulse duration (timeout 30ms)
  long duration = pulseIn(echoPins[sensor], HIGH, 30000);

  return duration;
}
```

What is implemented (in the code above):

- **Coded Trigger Pulses:** Each ultrasonic sensor is triggered using a unique 8-bit binary code. This means, instead of a single HIGH pulse to the trigger pin, the code sends a pattern (e.g., 10101010) of HIGH and LOW pulses.
- **Sequential Triggering:** The sensors are triggered one after another (not simultaneously), each with its own code. This helps reduce the chance of interference, especially if echoes from one sensor could overlap with another's trigger pulse.
- **Standard Echo Reading:** The code reads the echo pin after sending the trigger, measuring the time until the echo returns, and calculates the distance in the usual way.

What is not implemented (i.e., what true CDMA requires):

- **Simultaneous Sensor Operation:** True CDMA allows all sensors to transmit at the same time, each using a unique code. The current code does not do this; it triggers sensors one by one.
- **Coded Echo Transmission:** In true CDMA, the transmitted ultrasonic pulse itself is modulated (encoded) with the unique code, and the echo received is a coded signal.
- **Echo Signal Decoding (Correlation):** True CDMA requires the receiver to sample the incoming analog echo waveform and perform correlation/matched filtering with each code to determine which sensor's echo is present. This is not done in the provided code; it simply reads the echo as a digital pulse.
- **Analog Signal Processing:** True CDMA needs high-speed analog-to-digital conversion and digital signal processing to decode overlapping coded echoes—this is not feasible with standard Arduino/ESP32 hardware.
- **Noise/Interference Rejection via Code Orthogonality:** In real CDMA, the orthogonality of codes allows the receiver to distinguish between multiple simultaneous signals, even with noise or interference. The current code does not exploit this property.

CDMA summary:

- **Implemented:** The code uses unique codes for each sensor's trigger and triggers them sequentially.
- **Not implemented:** Simultaneous operation, coded echo transmission, and the critical CDMA decoding (correlation) process.

If you want to implement true CDMA, you would need:

- Hardware capable of high-speed analog signal sampling (ADC),
- Substantial memory and processing power for real-time correlation,
- Advanced DSP algorithms (not available on Arduino/ESP32 without significant external hardware).

Climate Extremes in Indian Regions/Cities

Region/City	**Summer Max (°C)**	**Winter Min (°C)**	**Notes**
Rajasthan (Thar Desert)	50	Below 0	Exceptionally hot summers; sub-zero winters in desert areas
Gujarat (Ahmedabad)	41	12	Hot, dry summers; mild winters
Punjab/Haryana (Hisar)	47	-4	Extreme summer heat; some of the lowest plains temperatures in winter
Delhi	46	4	Hottest month: June (46°C); coldest: January (4°C)
Eastern Maharashtra (Nagpur)	45	10	Very hot summers; cool winters
South India (Chennai)	38	20	Warm year-round; smaller temperature range
Darjeeling (Himalayas)	20	2	Cool summers, cold winters at moderate altitude
Himalayas (Leh, Ladakh)	25	-15	Extreme cold at high altitudes; alpine regions often well below freezing in winter
Western Ghats/Nilgiris (Ooty)	25	5	High-elevation areas can drop below freezing; mild summers

Table 5: Regional Temperature Extremes in Various Parts of India

Key Extremes

- Hottest recorded: Up to 50 °C (Phalodi, Rajasthan, source)
- Coldest recorded: Down to -15 °C (Leh, Ladakh, source)
- Coastal/southern: Generally 20–38 °C all year, minimal seasonal swing (Chennai)

Figure 16: Accuracy ref. for ultrasonic sensor

Temperature and Humidity Mitigation with CDMA-like Ultrasonic Sensor Array

Listing 4: Arduino code for Temperature and Humidity Compensation with CDMA Ultrasonic Sensor Array

```
#include <Adafruit_Sensor.h> // For DHT sensor
#include <DHT.h>       // DHT sensor library
#include <NewPing.h>  // HC-SR04 library

// DHT11 Temperature & Humidity Sensor Definitions
#define DHTPin 6    // DHT11 signal pin
#define DHTType DHT11 // DHT11 sensor type
DHT dht(DHTPin, DHTType); // Create DHT sensor object

// CDMA-like Ultrasonic Sensor Array Definitions
#define NUM_SENSORS 8 // Number of ultrasonic sensors

// Define trigger and echo pins for each sensor (change as per your wiring)
const uint8_t trigPins[NUM_SENSORS] = {2, 4, 6, 8, 10, 12, 14, 16};
const uint8_t echoPins[NUM_SENSORS] = {3, 5, 7, 9, 11, 13, 15, 17};

// Each sensor has a unique 8-bit code (example orthogonal-like codes)
const uint8_t codes[NUM_SENSORS][8] = {
  {1,0,1,0,1,0,1,0},
  {0,1,0,1,0,1,0,1},
  {1,1,0,0,1,1,0,0},
  {0,0,1,1,0,0,1,1},
  {1,0,0,1,1,0,0,1},
  {0,1,1,0,0,1,1,0},
  {1,1,1,0,0,1,0,1},
  {0,0,0,1,1,0,1,0}
};

#define BIT_PULSE_DURATION 10 // Duration of each code bit in microseconds

// Global variables for temp/humidity and speed of sound
float temp = 25.0;   // Default temperature (Celsius)
float hum = 50.0;    // Default humidity (%)
float sound_speed = 343.0; // Speed of sound (m/s), will be updated
float sound_cm_per_us = 0.0343; // Speed of sound in cm/us, will be updated

void setup() {
  Serial.begin(115200); // Initialize serial communication

  // Initialize DHT11 sensor
  dht.begin();

  // Initialize trigger and echo pins for all ultrasonic sensors
  for (uint8_t i = 0; i < NUM_SENSORS; i++) {
    pinMode(trigPins[i], OUTPUT);
    digitalWrite(trigPins[i], LOW); // Ensure trigger is LOW
    pinMode(echoPins[i], INPUT);
  }
}

void loop() {
  // Read temperature and humidity from DHT11
  temp = dht.readTemperature(); // Read temperature in Celsius
  hum = dht.readHumidity(); // Read relative humidity (%)

  // If failed to read, use last valid value or default
  if (isnan(temp)) temp = 25.0;
  if (isnan(hum)) hum = 50.0;

  // Calculate speed of sound using temp and humidity compensation
  // Speed of sound (m/s) = 331.4 + 0.606 * temp + 0.0124 * hum
```

```
  sound_speed = 331.4 + (0.606 * temp) + (0.0124 * hum);

  // Convert speed of sound to cm/us (1 m = 100 cm, 1 s = 1,000,000 us)
  sound_cm_per_us = sound_speed * 100.0 / 1000000.0;

  // Print environmental info
  Serial.print("Temp: "); Serial.print(temp); Serial.print(" C, ");
  Serial.print("Humidity: "); Serial.print(hum); Serial.print(" %, ");
  Serial.print("Speed of Sound: "); Serial.print(sound_speed); Serial.println(" m/s"
      );

  // CDMA Ultrasonic Sensor Loop
  for (uint8_t sensor = 0; sensor < NUM_SENSORS; sensor++) {
    // Send unique code and read echo duration
    long duration = sendCodedPulseAndReadEcho(sensor);

    // Calculate distance using compensated speed of sound
    float distance_cm = (duration / 2.0) * sound_cm_per_us;

    // Print distance result for each sensor
    Serial.print("Sensor ");
    Serial.print(sensor);
    Serial.print(" distance: ");
    if (duration == 0 || distance_cm < 2 || distance_cm > 400) {
      Serial.println("Out of range");
    } else {
      Serial.print(distance_cm);
      Serial.println(" cm");
    }
    delay(50); // Short delay before next sensor
  }

  delay(1000); // Wait before next full cycle
}

// Function: Send coded pulse (CDMA) and read echo duration
long sendCodedPulseAndReadEcho(uint8_t sensor) {
  // Send the 8-bit code as a pulse train
  for (uint8_t bit = 0; bit < 8; bit++) {
    digitalWrite(trigPins[sensor], codes[sensor][bit] ? HIGH : LOW); // Output code
        bit
    delayMicroseconds(BIT_PULSE_DURATION); // Wait for bit duration
  }
  digitalWrite(trigPins[sensor], LOW); // Ensure trigger is LOW after code

  // Wait briefly for pulse propagation
  delayMicroseconds(100);

  // Read echo pulse duration (timeout 30ms)
  long duration = pulseIn(echoPins[sensor], HIGH, 30000);

  return duration; // Return echo duration in microseconds
}
```

Classification of CARS

India [edit]

The Society of Indian Automobile Manufacturers (SIAM) divides Indian passenger vehicles into the segments A1, A2, A3, A4, A5, A6, B1, B2 and SUV. The classification is done solely based on the length of the vehicle. The details of the segments are below:

Car segment	Length of the car	Classification	Car model belonging to the segment
A1	Up to 3,400 mm	Ultracompact cars (A)	Suzuki Alto, Tata Nano, Mahindra e2o
A2	3,401 to 4,000 mm	Sub-four metre (B)	Maruti Suzuki Wagon R, Hyundai i10, Suzuki Swift, Suzuki Baleno (subcompact), Hyundai Xcent, Honda Amaze, Maruti Suzuki Dzire, Ford Aspire, Mahindra Verito, Hyundai i20, Tata Zest
A3	4,001 to 4,500 mm	Entry-level mid-size sedans (C)	Hyundai Verna, Honda City, Suzuki Ciaz
A4	4,501 to 4,700 mm	Small family cars (C)	Toyota Corolla, Škoda Octavia, Chevrolet Cruze
A5	4,701 to 5,000 mm	Mid-size (D) Executive cars (E)	D-segment: Toyota Camry, Škoda Superb E-segment: Mercedes-Benz E-Class, BMW 5 series
A6	More than 5,000 mm	Grand saloons (F)	Mercedes-Benz S-Class, Audi A8, BMW 7 series, Jaguar XJ
B1	<4,001 mm	Small vans	Maruti Omni, Tata Venture
B2	>4,000 mm	Mid-size MPVs/minivans	Toyota Innova, Suzuki Ertiga, Mahindra Marazzo, Kia Carnival
SUV	Any	SUVs	Renault Duster, Honda CR-V, Ford Endeavour, Hyundai Creta, Audi Q7, Toyota Land Cruiser

Figure 17: Classification of cars by SIAM

How Long Is a Car?: Average Car Length

The **average car length is about 14.7 feet**. Car length plays a significant role in your daily driving experience, especially when navigating tight spaces and choosing a storage facility. Different body types can greatly impact the length of a car, ranging from compact to full-size vehicles, SUVs, and more.

Average Car Width

Car width also differs between various types of vehicles. Compact vehicles tend to have a narrower design, while full-size vehicles, SUVs, and trucks are generally wider. The **average car width** is approximately 5.8 feet, although specific widths will vary depending on the make and model of the vehicle.

Car Types and Their Sizes

Compact Cars

A compact car or mini car is a vehicle that is **less than 14 feet long**, providing increased maneuverability in urban settings. The average length of a compact car is approximately 10–14 feet. Typical examples include:

- Nissan Versa
- Honda Civic
- Hyundai Elantra

Generally, minicars have a width of approximately **5.8 feet to 6 feet** and a height of 4.5 to 5 feet. These smaller vehicles are designed to be more fuel-efficient and easier to navigate in tight parking lots, making them ideal if you prioritize maneuverability and efficiency.

Midsize Cars

A midsize car has an **average length of approximately 14-16 feet**. Midsize cars are popular because they provide a balance between the compactness of smaller vehicles and the comfort of larger ones. They also offer a balance between space and fuel efficiency.

A typical medium sedan is about 14 feet long. Examples of midsize cars include:

- Toyota Camry
- Nissan Altima
- Honda Accord
- Hyundai Sonata

The **average width of a midsize car is approximately 6 feet**, with a height of around 5.6 feet. These dimensions allow you to sit comfortably while maintaining a manageable size for navigating urban environments.

Full-Size Cars

Full-size cars are typically **16 feet to 18 feet long** and are the best if you prioritize passenger room and a smooth, comfortable ride. The **average width of a full-size car is in excess of 6 feet**, while the height is about 4.7 feet depending on the manufacturer. With their spacious interiors, full-size cars are ideal if you want optimal cargo space.

Examples include:

- 2021 Dodge Charger
- Nissan Maxima
- Chrysler 300

Sports Cars

Sports cars have an **average length measuring between 13 feet and 16.4 feet**. These cars prioritize speed and aesthetics over practicality.

Known for their performance and style, **sports cars have an average width ranging from 5.7 feet to 6.5 feet** and a **height of between 3.9 feet and 4.2 feet**. The wider design of sports vehicles often contributes to their aggressive appearance and improved handling capabilities.

Examples include:

- Chevrolet Camaro
- Ford Mustang
- Dodge Viper
- Porsche 911

SUVs and Crossovers

Sport utility vehicles (SUVs) and crossovers span from **compact models measuring 15 feet long to large models measuring 16.5–17 feet long**. Compact SUVs, such as Honda CR-V and Toyota RAV4, offer a smaller footprint than their larger counterparts while providing ample cargo space and versatility.

Midsize SUVs typically measure 15 to 16 feet in length. Examples include:

- Ford Explorer
- Honda Pilot

Full-size SUVs like Chevrolet Tahoe and Ford Expedition are over **16 feet long**. SUVs and crossovers have **widths that range from 6 feet for compact models to 6.5 to 7 feet** for larger models. The height range for compact and big SUVs is between 4.9 feet and 5.5 feet.

The SUV dimensions offer more space and versatility than traditional sedans, making them popular choices when you need additional cargo space.

Pickup Trucks

The **average length** of a pickup truck can vary from **16 feet to over 20 feet**, depending on the cab and bed configurations. This range is longer than most vehicles, including sedans and compact vehicles.

The **average width** of a pickup truck can vary from **6.3 to 6.8 feet**, while the height is between 5.5 to 6.2 feet. Pickup trucks are known for their versatility and utility, with an open-bed rear cargo area and a cab in the front.

Examples include:

- Ford Ranger
- Ram 1500
- Chevrolet Silverado 1500

Minivans

Minivans have an **average length of 16–18 feet**, providing additional space for passengers and cargo compared to most cars. These vehicles are designed with families in mind, offering a higher roofline and more interior space than a sedan.

The **average width of a minivan is 6 feet 10 inches**, with a height of 5.8 to 6 feet.

Examples of minivans include:

- Toyota Sienna
- Honda Odyssey
- Kia Carnival

Luxury Cars

Luxury cars can span a wide range of dimensions, with certain models reaching a length of more than 18 feet. The **average length of a luxury car is typically around 14.7 feet** (or 4.5 meters), although the length may vary according to the manufacturer and model.

The **average width of a luxury car is approximately 6 feet**, with an average height of 4.9 feet. Luxury cars can also vary greatly, with some models being wider than usual.

These cars prioritize comfort, performance, and features over practicality.

Examples include:

- Mercedes Benz S

- Mercedes Benz GLS
- Audi A8
- Jaguar XJ

Station Wagons

Station wagons have average dimensions of **15 to 16 feet long, width of 5.8 to 6.2 feet**, and height of 4.5 to 5.4 feet. Also, they offer versatile cargo space with fuel efficiency and passenger comfort.

Examples of station wagons include:

- 2020 Buick Regal TourX
- Audi A4 Quattro
- Volvo V-60

How to Determine Your Car's Dimensions

You can determine your car size through the following methods: *using a VIN decoder, contacting a dealership, or measuring manually*. Each method has advantages and can provide accurate information about your car's length and width.

VIN Decoder

This tool can provide car dimensions and specifications based on your vehicle's VIN (Vehicle Identification Number). Input your car's VIN into the decoder to get details such as car dimensions, transmission, engine, color, standard features, and safety equipment.

If you're into car-sharing, the VIN method is the easiest approach to determining the length of the car you're hiring.

You can find the VIN in various places on the vehicle, especially the driver's door. Double-check that the VIN is complete (17 characters) and accurate before using the decoder.

Dealership Inquiry

Contact your local dealer with your license number or VIN to obtain information about your car's length and other specifications, including engine and mechanical equipment. Dealers can also provide information about car dimensions through new car specification sheets or the manual for a pre-owned car. Additionally, a car valuation website might offer such details.

Manual Measurement

You can know your car size using a tape measure or yardstick. To measure your car manually, run the tape measure or yardstick from the front to the rear to determine the car's length.

When measuring the width of your car, consider the distance the side mirrors extend outwards. To measure the width, utilize a tape measure and stand at the driver's side door, running the tape under the vehicle and positioning it at the opposite end.

In Summary

Aside from personal preference, the width and length of a car should, ultimately, inform where you park your new ride. It's important to measure your car's dimensions accurately and consider any aftermarket components that may alter its size.

If your measurements show that a second vehicle won't fit into your existing garage or driveway, consider a peer-to-peer storage marketplace like Neighbor for your car storage needs. For more insight into alternative parking options and the most popular storage unit sizes for cars, check out this resource.

Different Types of Parking

Parking is one of the major problems created by increasing road traffic. Availability of space in urban areas has increased the demand for parking space, especially in central business districts. There are two main types of parking: on-street parking and off-street parking.

On Street Parking

On-street parking means vehicles are parked on the sides of the street itself, usually controlled by government agencies. Classification is based on the angle vehicles are parked relative to the road alignment. According to IRC, standard car dimensions are taken as 5×2.5 meters, and for a truck, 3.75×7.5 meters.

Common types of on-street parking include:

- **Parallel Parking:** Vehicles are parked along the length of the road with no backward movement involved. It is the safest parking from an accident perspective but consumes maximum curb length, allowing only a minimum number of vehicles to park. This method causes least obstruction to ongoing traffic since minimum road width is used.
- **30° Parking:** Vehicles are parked at 30° to the road alignment. More vehicles can be parked compared to parallel parking, with better maneuverability and minimal delay to traffic. The length available for parking N vehicles is $L = 0.58 + 5N$ meters.
- **45° Parking:** As the angle increases, more vehicles can be accommodated. The length available for parking N vehicles is $L = 3.54N + 1.77$ meters.
- **60° Parking:** Vehicles are parked at 60° to the road, allowing even more vehicles. Length available is $L = 2.89N + 2.16$ meters.
- **Right Angle (90°) Parking:** Vehicles are parked perpendicular to the road. It requires the least curb length but more road width and complex maneuvering, which may cause accidents. Length required for N vehicles is $L = 2.5N$ meters.

Off Street Parking

Off-street parking is provided in designated areas away from the main traffic stream. These areas may be operated by public agencies or private firms. A typical layout involves parking lots or garages, which help reduce congestion on roads.

Parking Space Dimensions & Rules in India

Parking spaces are an essential component of urban infrastructure, yet many property owners and developers are unclear about the parking spaces dimensions and regulations that govern them. Whether planning a residential complex or commercial establishment or simply curious about the rules, understanding these standards is crucial for compliance and efficient space utilisation.

Parking Space Regulations by the Government of India

The Development Code by the Government of India provides comprehensive guidelines for parking lot dimensions. These regulations are typically incorporated into local building bylaws and development plans to ensure consistency in implementation.

The car parking space dimensions are specified in Equivalent Car Spaces (ECS) per 100 sq m of floor area. Different zones and activities have various sizes of parking spaces. These are:

Residential Areas

Group Housing, Plotted Housing (plots above 250 sq.m.), and mixed-use: 0.50 - 1.50 ECS per 100 sq m.

Commercial Areas

- Wholesale Trade and Freight Complex (including parking for loading and unloading): 1.50 - 2.50 ECS per 100 sq m.
- City centre, district centre, hotel, cinema and others: 1.00 - 2.00 ECS per 100 sq m.
- Community centre, local shopping centre, convenience shopping centre: 0.50 - 1.50 ECS per 100 sq m.

Public and Semi-Public Facilities

- Nursing homes, hospitals (other than government), social, cultural and other institutions, government and semi-government offices: 0.50 - 1.50 ECS per 100 sq m.
- Schools, colleges, universities and government hospitals: 0.25 - 0.75 ECS per 100 sq m.

Industrial Areas

Light and service industries, flatted group industries, and extensive industries: 0.50 - 1.00 ECS per 100 sq m.

Standard Dimensions for Parallel Parking

Parallel parking spots are generally built to handle cars parked alongside edges or walls.

The normal measurements for parallel parking usually include the following:

- **Length:** 22 feet (6.7 meters) to fit most sedan-sized cars
- **Width:** 5 feet (2.6 meters) from the curb
- **Maneuvering space:** Additional room at the ends of a row of parallel parking spots is often suggested for easy entry and exit

For places with high shift rates, such as business zones, slightly bigger measurements may be suggested to allow faster parking moves and reduce traffic jams.

Standard Parking Space Size

According to the Development Code, the normal car parking size is clearly defined. The measurements vary based on the type of parking:

- **Open parking:** 0 sq m. per similar car space
- **Ground floor covered parking:** 0 sq m. per similar car space
- **Basement parking:** 0 sq m. per similar car space

These measurements ensure adequate room not just for the car itself but also for safe movement when entering and leaving the parking place. The larger area required for basement parking accounts for the structural elements like pillars and the need for proper ventilation systems.

JSN-SR04T Waterproof Ultrasonic Sensor for Smart Parking

The **JSN-SR04T** is specifically designed to be **waterproof** — this is its main advantage over the **HC-SR04**. It uses a waterproof ultrasonic transducer that can be mounted outside vehicles for applications like **reverse parking sensors** in all weather conditions.

Key Advantages of JSN-SR04T's Modes for Parking

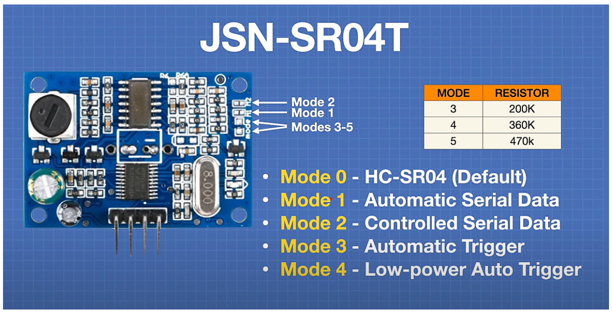


Figure 18: Caption

Here's why the **JSN-SR04T** modes are better **than just a simple HC-SR04** for reverse parking:

Mode	Functionality	Benefit
Mode 0	HC-SR04 compatible (default)	Can replace HC-SR04 directly if needed
Mode 1	Automatic Serial Data	Sends continuous distance — great for real-time parking assist without extra triggers
Mode 2	Controlled Serial Data	Microcontroller can trigger sensor, saving power and avoiding false triggers
Mode 3	Automatic Trigger	Continuously sends distance without manual trigger — suitable for simple reverse alarms
Mode 4	Low Power Trigger	Reduces power consumption — ideal for parked vehicles
Mode 5	1.5 Meter Switch	Acts as a switch detecting obstacles within 1.5 meters — useful for automatic braking

Table 6: Operational Modes of the JSN-SR04T Ultrasonic Sensor and Their Benefits

Comparison Table: HC-SR04 vs. JSN-SR04T

Why JSN-SR04T Modes are Better for Smart Parking

HC-SR04: Modes

- Only 1 Mode: Trigger → Wait for Echo → Calculate distance.
- MCU must send a trigger pulse every time you want a reading.
- **Problems:**
  - No automatic or serial streaming.
  - Needs constant microcontroller attention.
  - Can miss readings if timing isn't precise.
  - Not practical for real-time or multi-sensor setups in outdoor parking.

Feature	HC-SR04	JSN-SR04T	Why HC-SR04 is Not Suitable
Waterproof	$\times$	yes	Can fail in rain, dirt, mud
Detection Range	2 cm – 400 cm	20 cm – 600 cm	Shorter range for cars
Accuracy	$\pm$0.3 cm (ideal)	$\pm$1 cm (real outdoor)	Better spec but unreliable outdoors
Voltage Range	5 VDC	3.0 – 5.5 VDC	JSN-SR04T is more flexible
Serial Data Mode	$\times$	yes	Cleaner data, less noise
Cost	$1 USD	$10 USD	Cheap but impractical outdoors
Modes	Single manual	5 modes: Serial, Trigger, Auto, Switch	Flexible for smart parking
Practical Coverage Angle	$\sim 15^\circ$	Up to 75° (practical multi-sensor setup)	Narrow beam means more blind spots

Table 7: Comparison between HC-SR04 and JSN-SR04T Ultrasonic Sensors

JSN-SR04T-2.0: Multiple Modes

The **JSN-SR04T-2.0** supports **3 practical operating modes**, selectable by soldering a resistor (R27):

Mode	How it works	Why it's useful for smart parking/outdoor
Mode 1	Manual Trigger Mode — Same basic mode as HC-SR04, but improved waterproof housing	Good if you want full control. Industrial-grade housing for reliability in bad weather.
Mode 2	Auto Serial Mode — Sensor measures distance every 100 ms and streams TTL serial data in millimeters	Perfect for parking: no precise timing needed, easy to integrate, cleaner data, multiple sensors on same bus.
Mode 3	Command Serial Mode — Sensor waits for command (0x55), then measures and replies with serial distance frame	Good for smart multi-sensor systems: MCU polls sensors one by one, reducing cross-talk, ideal for smart parking bays and gates.

Table 8: Operating Modes of JSN-SR04T and Advantages for Smart Parking

Key Advantages of JSN-SR04T Modes

1. Auto Mode means less MCU load: no tight timing loops to handle triggers.
2. Serial Data = Noise Resistant: less prone to electrical noise than echo pulse width signals.
3. Easier Multi-Sensor System: Multiple sensors on one MCU manageable with unique serial responses.
4. Flexible Integration:
    - Manual mode for simple testing.
    - Auto mode for plug & play distance feed.
    - Command mode for custom smart parking logic.

Bottom Line

Why HC-SR04 fails	Why JSN-SR04T Modes help
Single mode only: hard to scale for smart parking	Multiple modes: Easy smart parking lanes, multiple bays
Needs constant MCU timing	Auto streaming: Reliable continuous monitoring
Echo pin can get noisy in cars	Serial output: Robust, cleaner signal
No flexible wiring options	Works over longer cables, serial is stable

So for smart parking: The **JSN-SR04T's modes** provide a real plug & play solution for rugged outdoor distance sensing, automatic data streaming, and easy multi-sensor setups — things the **HC-SR04** cannot do reliably outdoors.

Extra Components for Decoding CDMA Logic

- HC-SR04 sensors don't expose *raw echo waveform*. They output a single digital echo pulse only.
- **You cannot do CDMA decoding in software** with HC-SR04 because no raw signal is provided.
- To decode signals in software, you need:
  - Separate ultrasonic receiver transducer.
  - Amplifier circuit.
  - ADC pin on microcontroller for digitizing waveform fast enough.
  - Software correlation algorithms.

Techniques Used in the Paper for Ultrasonic Sensor Accuracy Enhancement

(See the detailed video and article here: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10870780&tag=1`)

1. Error Correction with an Artificial Neural Network (ANN)

- Developed a machine learning model (ANN) trained on real measurement data.
- The model learns to **predict the true distance** more accurately by compensating for systematic sensor errors.
- It corrects for deviations caused by environmental factors.

2. Incorporating Environmental Variables

- Used additional data from a **DHT11 sensor**:
  - Temperature
  - Humidity
- Also considered air pressure.
- These values were added as input features for the ANN.
- This compensates for how temperature and humidity affect the **speed of sound**, which changes the time-of-flight measurement.

3. Feature Selection and Reduction

- Used a **Random Forest Regression** model to check which features matter most.
- Ensures the ANN only uses significant inputs, avoiding overfitting.
- Result: *All four features (raw sensor reading, temperature, humidity, air pressure) were kept.*

4. Feature Scaling

- Normalized the data to a common scale (e.g., -1 to 1) to help the ANN train more effectively.
- Minimizes errors caused by differences in measurement units and scales.

5. Hyperparameter Tuning

- Experimented with different:
  - Activation functions (ReLU, tanh, sigmoid)
  - Number of epochs
  - Neural network layer sizes
- Found the **hyperbolic tangent activation function** with 250 epochs worked best for their dataset.

6. Prototype Testing in Real-World Scenarios

- Deployed multiple HC-SR04 sensors on a vehicle for parking assistance.
- The ANN model ran in real time to adjust raw distance readings.
- The system achieved **98.42% accuracy** — slightly better than using the raw sensor data alone.

Core Practical Insight

The **main idea** is that **raw ultrasonic sensors like the HC-SR04 are vulnerable to weather conditions**, and low-cost sensors lack built-in compensation. This paper uses:

- **Environmental sensing (DHT11)**
- **Machine learning (ANN)**

to automatically correct systematic measurement errors *without changing the hardware*.

Summary Approach

Raw ultrasonic reading + temperature + humidity + air pressure $\rightarrow$ ANN $\rightarrow$ corrected distance

Implementing Techniques Using the HC-SR04 Sensor: Feasibility and Alternatives

Below is a breakdown of how each advanced technique can or cannot be implemented using the basic HC-SR04 sensor.

Technique	Feasibility with HC-SR04	Notes
Constrained Stimulus Optimization	x	Requires custom waveform; not feasible with HC-SR04's fixed pulse output.
Dual Threshold Detection	x	Needs analog signal access; HC-SR04 only provides digital pulse.
Noise-Adaptive Waveform	**Partial**	Partially feasible via firmware by discarding noisy readings; no waveform change possible.
Automatic Gain Control	x	Requires analog AGC stage; not available in HC-SR04 hardware.
Temperature Compensation	**Yes**	Easily implemented by adding an external temperature sensor.
High-Resolution Timing	**Yes**	Requires a good microcontroller with hardware timers or interrupts.

Table 9: Feasibility of Implementing Advanced Ultrasonic Sensor Techniques Using the HC-SR04

Details on Key Limitations and Alternatives

Constrained Stimulus Optimization

- Not feasible with HC-SR04 due to fixed 8-cycle 40kHz transmission.
- Alternative: Build custom driver with waveform generator, but complex and against HC-SR04 purpose.

Dual Threshold Detection

- HC-SR04 only outputs echo pin pulse duration; no access to raw analog signals.
- Alternative: Custom analog ultrasonic receiver with comparator circuits required.

Noise-Adaptive Waveform

- Can't measure analog noise on HC-SR04.
- Partial alternative: Add microphone or ultrasonic noise sensor and discard readings in noisy conditions.

Automatic Gain Control (AGC)

- No AGC inside HC-SR04.
- Use amplifier circuits with AGC externally, or software smoothing by averaging multiple measurements.

Temperature Compensation

- Speed of sound varies about 0.6 m/s per °C, affecting distance readings.
- Use external temperature sensor (DS18B20, DHT22, TMP36).
- Adjust calculation for speed of sound: $v(T) \approx 331 + 0.6 \times T\ (m/s)$.

High-Resolution Timing

- Default Arduino pulseIn() too coarse (4 µs resolution).
- Use hardware timers, interrupts, or advanced MCUs (ESP32, STM32) for sub-microsecond timing.

Ultrasonic sensing visualization using Pycharm

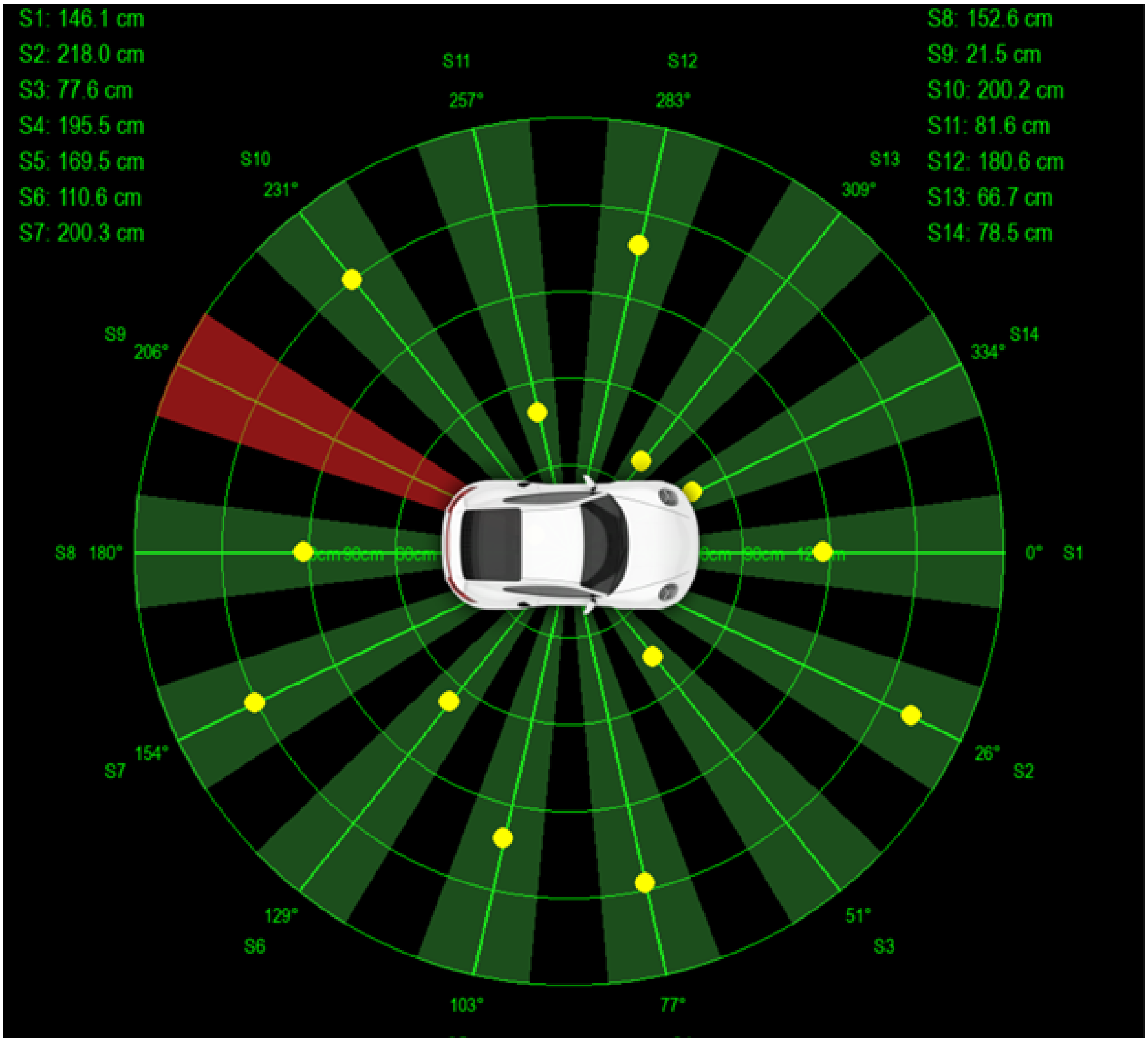


Figure 19: Caption

Pygame Code for 360° Radar with 14 Sensors(Configuration: 1)

Listing 5: Pygame 360° Radar with 14 Sensors (with Car Image Overlay)

```
import pygame
import math
import random

WIDTH, HEIGHT = 900, 900
CENTER = (WIDTH // 2, HEIGHT // 2)
RADAR_RADIUS = 340
MAX_DISTANCE = 250
SENSOR_CONE_ANGLE = 15
NUM_SENSORS = 14
ANGLE_STEP = 360 / NUM_SENSORS

```

```
SENSOR_ANGLES = [i * ANGLE_STEP for i in range(NUM_SENSORS)]
SENSOR_LABELS = [f"S{i+1}" for i in range(NUM_SENSORS)]
DISTANCE_MARKS = [30, 60, 90, 120]

pygame.init()
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("360 Radar - 14 Sensors with Car Image Overlay")
font_small = pygame.font.SysFont('Arial', 16)
font_large = pygame.font.SysFont('Arial', 20)

car_img = pygame.image.load('porscha.png').convert_alpha()
car_img = pygame.transform.smoothscale(car_img, (260, 250))
car_rect = car_img.get_rect(center=CENTER)

clock = pygame.time.Clock()
sensor_distances = [MAX_DISTANCE] * NUM_SENSORS

HORIZONTAL_ANGLES = [0, 180]

running = True
while running:
   for event in pygame.event.get():
      if event.type == pygame.QUIT:
         running = False

   sensor_distances = [random.uniform(1, MAX_DISTANCE) for _ in range(NUM_SENSORS)]

   # Set radar background to black
   screen.fill((0, 0, 0))

   # Draw bright green concentric circles
   for i in range(1, 6):
      r = RADAR_RADIUS * i / 5
      pygame.draw.circle(screen, (0, 255, 0), CENTER, int(r), 1)

   # Draw bright green bearing lines
   for angle in SENSOR_ANGLES:
      a_rad = math.radians(angle)
      endpt = (CENTER[0] + RADAR_RADIUS * math.cos(a_rad), CENTER[1] + RADAR_RADIUS * math.sin
          (a_rad))
      pygame.draw.line(screen, (0, 255, 0), CENTER, endpt, 2)

   # Draw sensor cones and labels
   for i, angle in enumerate(SENSOR_ANGLES):
      a_rad = math.radians(angle)
      cone = math.radians(SENSOR_CONE_ANGLE)
      points = [CENTER]
      steps = 10
      for s in range(steps + 1):
         t = a_rad - cone / 2 + (cone * s / steps)
         x = CENTER[0] + RADAR_RADIUS * math.cos(t)
         y = CENTER[1] + RADAR_RADIUS * math.sin(t)
         points.append((x, y))
      dist = sensor_distances[i]
      if dist < 30:
         color = (255, 40, 40, 140)
      elif dist < 60:
         color = (255, 165, 0, 100)
      else:
         color = (80, 220, 80, 90)
      poly_surf = pygame.Surface((WIDTH, HEIGHT), pygame.SRCALPHA)
      pygame.draw.polygon(poly_surf, color, points)
      screen.blit(poly_surf, (0, 0))

      # Distance text on 0 and 180 only
      if int(round(angle)) % 360 in HORIZONTAL_ANGLES:
         for mark in DISTANCE_MARKS:
            rad = mark / MAX_DISTANCE * RADAR_RADIUS
            mx = CENTER[0] + rad * math.cos(a_rad)
            my = CENTER[1] + rad * math.sin(a_rad)
            label = font_small.render(f"{mark}cm", True, (0, 250, 0))
            offset = 14 if angle == 0 else -54
            screen.blit(label, (mx + offset, my - 8))

      # Angle label at the end of each line (green)
      end_label = font_small.render(f"{int(round(angle))}", True, (0, 250, 0))
```

```
        ex = CENTER[0] + (RADAR_RADIUS + 23) * math.cos(a_rad)
        ey = CENTER[1] + (RADAR_RADIUS + 23) * math.sin(a_rad)
        screen.blit(end_label, end_label.get_rect(center=(ex, ey)))

        # Detection point for active sensors
        if dist < MAX_DISTANCE:
            r = (dist / MAX_DISTANCE) * RADAR_RADIUS
            px = CENTER[0] + r * math.cos(a_rad)
            py = CENTER[1] + r * math.sin(a_rad)
            pygame.draw.circle(screen, (255, 255, 0), (int(px), int(py)), 8)

    # Draw car image
    screen.blit(car_img, car_rect)

    # Draw sensor labels around radar (spread further for left/right)
    label_radius = RADAR_RADIUS * 1.16
    for i, angle in enumerate(SENSOR_ANGLES):
        a_rad = math.radians(angle)
        x = CENTER[0] + label_radius * math.cos(a_rad)
        y = CENTER[1] + label_radius * math.sin(a_rad)
        text_surf = font_small.render(SENSOR_LABELS[i], True, (0, 255, 0))
        rect = text_surf.get_rect(center=(x, y))
        screen.blit(text_surf, rect)

    # Sensor value panelsside columns spaced for clarity
    for i, dist in enumerate(sensor_distances):
        col_x = 20 if i < 7 else 730
        row_y = 20 + (i if i < 7 else i-7)*28
        dist_text = font_large.render(f"{SENSOR_LABELS[i]}: {dist:.1f} cm", True, (0, 255, 0))
        screen.blit(dist_text, (col_x, row_y))

    pygame.display.flip()
    clock.tick(30)
pygame.quit()
```

How Placement Works in Code

- **Sensor Count and FOV:** There are 14 sensors, each with a 15° field of view (FOV), covering 210° if arranged with no overlap.
- **Distribution for ”6F 6B 2S”:**
  - 6 Front sensors: Span the front arc of the car.
  - 6 Rear sensors: Span the rear arc of the car.
  - 2 Side sensors: Cover exactly left and right.
- **Angular Positioning:**
  - Front sensors typically cover from -45° to +45° (total 90°, divided among 6 sensors, with a small overlap or extra coverage if desired).
  - Back sensors cover from 135° to 225° (centered at 180°, spanning 90° total across 6 sensors).
  - Side sensors are positioned at 90° (left) and 270° (right).
- **Sensor Angle Calculation:**
  - Each sensor's central angle is given by its slot in `SENSOR_ANGLES`. The code evenly spaces all 14 sensors around the 360° circle, i.e. every 25.71° (360° / 14).
  - In your code, the placement starts at 0° (front), then goes to 25.71°, 51.43°, and so on, wrapping around the car.

Coverage and Blind Spots

- **Cone Overlap:** Each sensor cone is 15°, but centers are 25.71° apart, leading to a gap of about 10.7° between adjacent cone edges.

- **Blind Spot Calculation:**
  - Angular gap between cones $= 25.71^\circ \; - \; 15^\circ \approx 10.7^\circ$
  - Each space between neighboring sensors leaves up to 10.7° that is not covered at all. This is a worst-case gap if the cones are not overlapped or precisely angled for overlap.
- **Total Uncovered Angle:**
  - Total cone coverage: $14 \times 15^\circ = 210^\circ$
  - Total circle: 360°
  - Total uncovered: $360^\circ - 210^\circ = 150^\circ$, distributed as $\approx 10.7^\circ$ between each cone slot.
- **Practical Impact:**
  - Blind spots will appear between each sensor's field of view.
  - The ideal "6F 6B 2S" arrangement could be enhanced by increasing overlap on critical zones (front, side, rear), and manually shifting the angular assignment of side sensors to exactly $90^\circ/270^\circ$ if needed, depending on vehicle geometry and safety requirements.

Visual Summary

- **Best practice:** For true 6 Front, 6 Back, 2 Side arrangement, explicitly define angles for front/back clusters and place the 2 side sensors at 90° and 270°, rather than even spacing. The even-spaced 14 sensors approximates this, but may not align perfectly with car axes.
- **Blind spots:** Roughly 10.7° between each sensor cone, totaling 150° out of 360° as uncovered regions.

Pygame Code for 360° Radar with 14 Sensors(Configuration: 2)

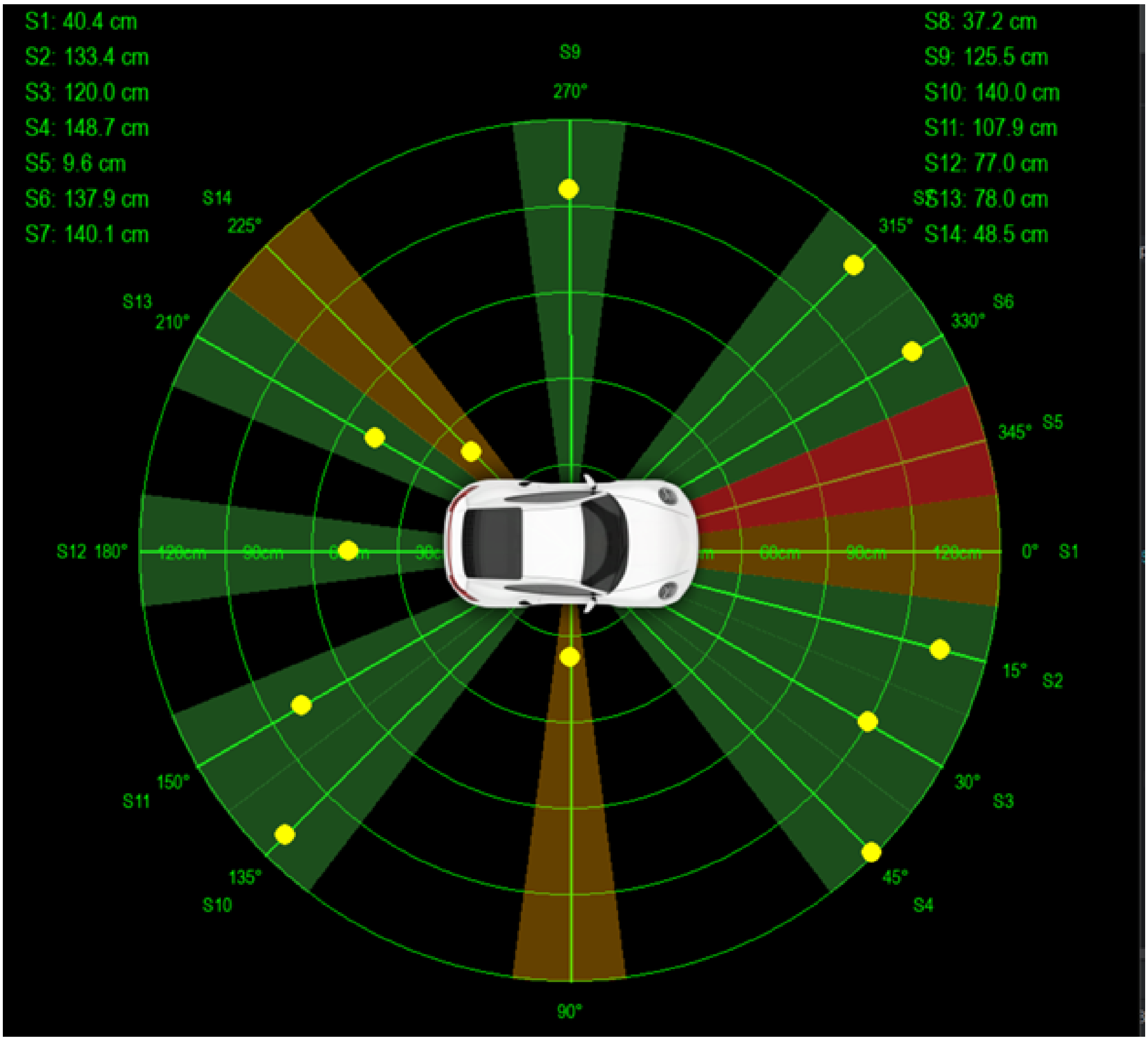


Figure 20: Caption

Listing 6: Pygame Radar Visualization with Serial Data Integration and Custom Sensor Placement

```
import pygame
import math
import serial
import serial.tools.list_ports
import threading
import queue

WIDTH, HEIGHT = 900, 900
CENTER = (WIDTH // 2, HEIGHT // 2)
RADAR_RADIUS = 340
MAX_DISTANCE = 150
SENSOR_CONE_ANGLE = 15 # Each sensor covers 15 degrees
NUM_SENSORS = 14

# Custom sensor placements for your layout
SENSOR_ANGLES = [
    0,    # S1: Front-Center
    15,   # S2: Front-Front-Right
    30,   # S3: Front-Right
    45,   # S4: Front-Far-Right
    345,  # S5: Front-Front-Left
    330,  # S6: Front-Left
    315,  # S7: Front-Far-Left
    90,   # S8: Right Side
```

```
    270,  # S9: Left Side
    135,  # S10: Rear-Right
    150,  # S11: Rear-Mid-Right
    180,  # S12: Rear-Center
    210,  # S13: Rear-Mid-Left
    225   # S14: Rear-Left
]
SENSOR_LABELS = [
    "S1", "S2", "S3", "S4", "S5", "S6", "S7",
    "S8", "S9", "S10", "S11", "S12", "S13", "S14"
]
DISTANCE_MARKS = [30, 60, 90, 120]
HORIZONTAL_ANGLES = [0, 180]

pygame.init()
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("360 Radar - 14 Sensors with Car Image Overlay")
font_small = pygame.font.SysFont('Arial', 16)
font_large = pygame.font.SysFont('Arial', 20)

car_img = pygame.image.load('porscha.png').convert_alpha()
car_img = pygame.transform.smoothscale(car_img, (260, 250))
car_rect = car_img.get_rect(center=CENTER)

clock = pygame.time.Clock()

# Serial setup and thread for asynchronous reading
serial_queue = queue.Queue()

def find_serial_port():
    ports = list(serial.tools.list_ports.comports())
    for port in ports:
        return port.device
    return None

ser = None
try:
    port_name = find_serial_port()
    if port_name:
        ser = serial.Serial(port_name, 9600, timeout=0.1)
        print(f"Connected to serial port: {port_name}")
    else:
        print("No serial port detected; running with blank data")
except Exception as e:
    print(f"Error opening serial port: {e}")

def read_serial(ser_obj):
    buffer = b""
    while True:
        if ser_obj is None:
            break
        try:
            data = ser_obj.read(ser_obj.in_waiting or 1)
            if data:
                buffer += data
                while b'.' in buffer:
                    pkt, buffer = buffer.split(b'.', 1)
                    line = pkt.decode(errors='ignore').strip()
                    serial_queue.put(line)
        except Exception as e:
            print(f"Serial read error: {e}")
            break

if ser is not None:
    thread = threading.Thread(target=read_serial, args=(ser,), daemon=True)
    thread.start()

def parse_serial_line(line):
    try:
        parts = line.split(",")
        if len(parts) >= NUM_SENSORS + 1:
            values = [float(x) for x in parts[1:NUM_SENSORS + 1]]
            values = [min(v, MAX_DISTANCE) for v in values]
            return values
    except Exception as e:
        print(f"Parse error: {e}")
```

```
    return None

sensor_distances = [MAX_DISTANCE] * NUM_SENSORS

running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    # Read from serial if available, else keep showing previous data
    while not serial_queue.empty():
        line = serial_queue.get()
        new_values = parse_serial_line(line)
        if new_values:
            sensor_distances = new_values

    screen.fill((0, 0, 0))

    # Draw radar graphics
    for i in range(1, 6):
        r = RADAR_RADIUS * i / 5
        pygame.draw.circle(screen, (0, 255, 0), CENTER, int(r), 1)
    for angle in SENSOR_ANGLES:
        a_rad = math.radians(angle)
        endpt = (CENTER[0] + RADAR_RADIUS * math.cos(a_rad), CENTER[1] + RADAR_RADIUS * math.sin(
            a_rad))
        pygame.draw.line(screen, (0, 255, 0), CENTER, endpt, 2)
    for i, angle in enumerate(SENSOR_ANGLES):
        a_rad = math.radians(angle)
        cone = math.radians(SENSOR_CONE_ANGLE)
        points = [CENTER]
        steps = 10
        for s in range(steps + 1):
            t = a_rad - cone / 2 + (cone * s / steps)
            x = CENTER[0] + RADAR_RADIUS * math.cos(t)
            y = CENTER[1] + RADAR_RADIUS * math.sin(t)
            points.append((x, y))
        dist = sensor_distances[i]
        if dist < 30:
            color = (255, 40, 40, 140)
        elif dist < 60:
            color = (255, 165, 0, 100)
        else:
            color = (80, 220, 80, 90)
        poly_surf = pygame.Surface((WIDTH, HEIGHT), pygame.SRCALPHA)
        pygame.draw.polygon(poly_surf, color, points)
        screen.blit(poly_surf, (0, 0))

        # Distance text on horizontal axes only
        if int(round(angle)) in HORIZONTAL_ANGLES:
            for mark in DISTANCE_MARKS:
                rad = mark / MAX_DISTANCE * RADAR_RADIUS
                mx = CENTER[0] + rad * math.cos(a_rad)
                my = CENTER[1] + rad * math.sin(a_rad)
                label = font_small.render(f"{mark}cm", True, (0, 250, 0))
                offset = 14 if angle == 0 else -54
                screen.blit(label, (mx + offset, my - 8))

        # Angle label at end of each line
        end_label = font_small.render(f"{int(round(angle))}", True, (0, 250, 0))
        ex = CENTER[0] + (RADAR_RADIUS + 23) * math.cos(a_rad)
        ey = CENTER[1] + (RADAR_RADIUS + 23) * math.sin(a_rad)
        screen.blit(end_label, end_label.get_rect(center=(ex, ey)))

        # Detection (object) marker
        if dist < MAX_DISTANCE:
            r = (dist / MAX_DISTANCE) * RADAR_RADIUS
            px = CENTER[0] + r * math.cos(a_rad)
            py = CENTER[1] + r * math.sin(a_rad)
            pygame.draw.circle(screen, (255, 255, 0), (int(px), int(py)), 8)

    screen.blit(car_img, car_rect)

    # Draw sensor labels at perimeter
    label_radius = RADAR_RADIUS * 1.16
```

```python
    for i, angle in enumerate(SENSOR_ANGLES):
        a_rad = math.radians(angle)
        x = CENTER[0] + label_radius * math.cos(a_rad)
        y = CENTER[1] + label_radius * math.sin(a_rad)
        text_surf = font_small.render(SENSOR_LABELS[i], True, (0, 255, 0))
        rect = text_surf.get_rect(center=(x, y))
        screen.blit(text_surf, rect)

    # Sensor value panels (left and right)
    for i, dist in enumerate(sensor_distances):
        col_x = 20 if i < 7 else 730
        row_y = 20 + (i if i < 7 else i-7)*28
        dist_text = font_large.render(f"{SENSOR_LABELS[i]}: {dist:.1f} cm", True, (0, 255, 0))
        screen.blit(dist_text, (col_x, row_y))

    pygame.display.flip()
    clock.tick(30)
pygame.quit()
```

Correct 360° Ultrasonic Sensor Arrangement for Your Vehicle

To match typical automotive sensor layouts, here is the recommended 14-sensor arrangement for full 360-degree coverage:

- **Sensor Placement Overview**
  - **Front:** 7 sensors (spread across the front bumper for detailed coverage)
  - **Back:** 5 sensors (spread across the rear bumper)
  - **Sides:** 2 sensors (one on each side, roughly at mid-doors)

Sensor Layout and Position Mapping

Sensor	Angle (Degrees)	Position Description
S1	0	Front-Center
S2	15	Front-Front-Right
S3	30	Front-Right
S4	45	Front-Far-Right
S5	345	Front-Front-Left
S6	330	Front-Left
S7	315	Front-Far-Left
S8	90	Right Side
S9	270	Left Side
S10	135	Rear-Right
S11	150	Rear-Mid-Right
S12	180	Rear-Center
S13	210	Rear-Mid-Left
S14	225	Rear-Left

- Angles increase clockwise from the top (front-center 0°), exactly matching a compass.
- **Front sensors:** S1–S7 (from center, sweeping through right to left across the front bumper).
- **Side sensors:** S8 (right), S9 (left).
- **Back sensors:** S10–S14 (sweeping right to left across the rear bumper).

Pygame Code for 360° Radar with 14 Sensors(Configuration: 3)

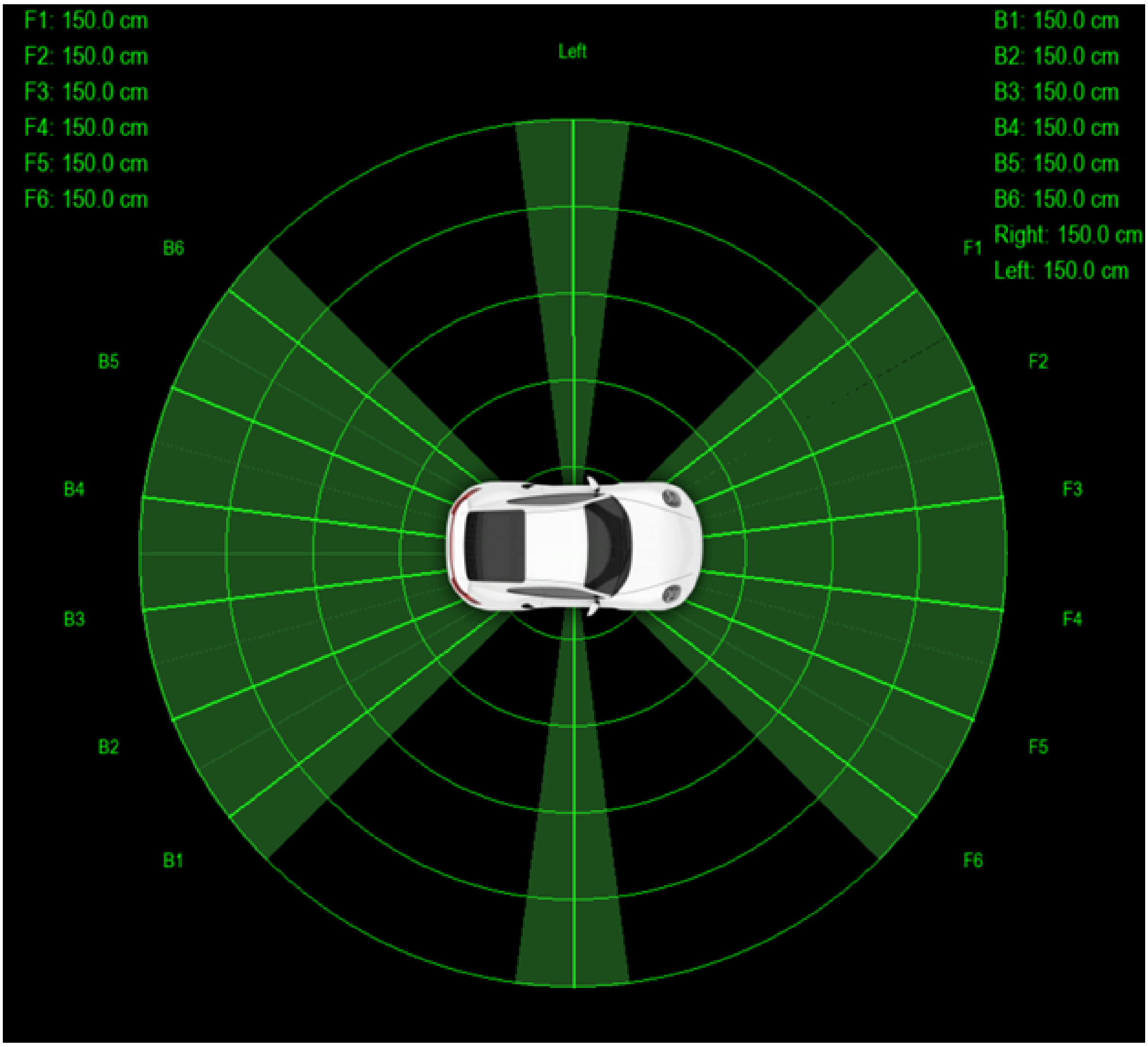


Figure 21: Caption

Sensor Placement Summary

Python Code: Radar Visualization

Listing 7: Pygame Visualization: 14 Sensors, Custom End-to-End Placement

```
import pygame
import math
import serial
import serial.tools.list_ports
import threading
import queue

WIDTH, HEIGHT = 900, 900
CENTER = (WIDTH // 2, HEIGHT // 2)
RADAR_RADIUS = 340
MAX_DISTANCE = 150
SENSOR_CONE_ANGLE = 15 # Each sensor covers 15 degrees
NUM_SENSORS = 14

# 6 front sensors, end-to-end, each 15 apart, centered near 0 (front)
front_start = -37.5
front_angles = [front_start + i * SENSOR_CONE_ANGLE for i in range(6)]
```

```
front_angles = [angle % 360 for angle in front_angles]

# 6 back sensors, end-to-end, each 15 apart, centered near 180 (back)
back_start = 142.5
back_angles = [back_start + i * SENSOR_CONE_ANGLE for i in range(6)]

# Sides
side_angles = [90, 270]

SENSOR_ANGLES = front_angles + back_angles + side_angles
SENSOR_LABELS = [f"F{i+1}" for i in range(6)] + [f"B{i+1}" for i in range(6)] + ["Right", "
    Left"]
DISTANCE_MARKS = [30, 60, 90, 120]
HORIZONTAL_ANGLES = [0, 180]

pygame.init()
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("360 Radar - 6 Front, 6 Back End-to-End, 2 Side Sensors")
font_small = pygame.font.SysFont('Arial', 16)
font_large = pygame.font.SysFont('Arial', 20)

car_img = pygame.image.load('porscha.png').convert_alpha()
car_img = pygame.transform.smoothscale(car_img, (260, 250))
car_rect = car_img.get_rect(center=CENTER)

clock = pygame.time.Clock()
serial_queue = queue.Queue()

def find_serial_port():
   ports = list(serial.tools.list_ports.comports())
   for port in ports:
      return port.device
   return None

ser = None
try:
   port_name = find_serial_port()
   if port_name:
      ser = serial.Serial(port_name, 9600, timeout=0.1)
      print(f"Connected to serial port: {port_name}")
   else:
      print("No serial port detected; running with blank data")
except Exception as e:
   print(f"Error opening serial port: {e}")

def read_serial(ser_obj):
   buffer = b""
   while True:
      if ser_obj is None:
         break
      try:
         data = ser_obj.read(ser_obj.in_waiting or 1)
         if data:
            buffer += data
            while b'.' in buffer:
               pkt, buffer = buffer.split(b'.', 1)
               line = pkt.decode(errors='ignore').strip()
               serial_queue.put(line)
      except Exception as e:
         print(f"Serial read error: {e}")
         break

if ser is not None:
   thread = threading.Thread(target=read_serial, args=(ser,), daemon=True)
   thread.start()

def parse_serial_line(line):
   try:
      parts = line.split(",")
      if len(parts) >= NUM_SENSORS + 1:
         values = [float(x) for x in parts[1:NUM_SENSORS + 1]]
         values = [min(v, MAX_DISTANCE) for v in values]
         return values
   except Exception as e:
      print(f"Parse error: {e}")
   return None
```

```
sensor_distances = [MAX_DISTANCE] * NUM_SENSORS

running = True
while running:
   for event in pygame.event.get():
      if event.type == pygame.QUIT:
         running = False

   while not serial_queue.empty():
      line = serial_queue.get()
      new_values = parse_serial_line(line)
      if new_values:
         sensor_distances = new_values

   screen.fill((0, 0, 0))

   for i in range(1, 6):
      r = RADAR_RADIUS * i / 5
      pygame.draw.circle(screen, (0, 255, 0), CENTER, int(r), 1)

   for angle in SENSOR_ANGLES:
      a_rad = math.radians(angle)
      end_pt = (CENTER[0] + RADAR_RADIUS * math.cos(a_rad), CENTER[1] + RADAR_RADIUS * math.
          sin(a_rad))
      pygame.draw.line(screen, (0, 255, 0), CENTER, end_pt, 2)

   for i, angle in enumerate(SENSOR_ANGLES):
      a_rad = math.radians(angle)
      cone_rad = math.radians(SENSOR_CONE_ANGLE)
      points = [CENTER]
      steps = 10
      for s in range(steps + 1):
         t = a_rad - cone_rad / 2 + (cone_rad * s / steps)
         x = CENTER[0] + RADAR_RADIUS * math.cos(t)
         y = CENTER[1] + RADAR_RADIUS * math.sin(t)
         points.append((x, y))
      dist = sensor_distances[i]
      # Color code by distance
      if dist < 30:
         color = (255, 40, 40, 140)
      elif dist < 60:
         color = (255, 165, 0, 100)
      else:
         color = (80, 220, 80, 90)
      poly_surf = pygame.Surface((WIDTH, HEIGHT), pygame.SRCALPHA)
      pygame.draw.polygon(poly_surf, color, points)
      screen.blit(poly_surf, (0, 0))

      # Detection blob
      if dist < MAX_DISTANCE:
         r = (dist / MAX_DISTANCE) * RADAR_RADIUS
         px = CENTER[0] + r * math.cos(a_rad)
         py = CENTER[1] + r * math.sin(a_rad)
         pygame.draw.circle(screen, (255, 255, 0), (int(px), int(py)), 8)

   screen.blit(car_img, car_rect)

   # Sensor labels
   label_radius = RADAR_RADIUS * 1.16
   for i, angle in enumerate(SENSOR_ANGLES):
      a_rad = math.radians(angle)
      x = CENTER[0] + label_radius * math.cos(a_rad)
      y = CENTER[1] + label_radius * math.sin(a_rad)
      text_surf = font_small.render(SENSOR_LABELS[i], True, (0, 255, 0))
      rect = text_surf.get_rect(center=(x, y))
      screen.blit(text_surf, rect)

   # Sensor value panels
   for i, dist in enumerate(sensor_distances):
      col_x = 20 if i < 6 else 780
      row_y = 20 + (i if i < 6 else i-6)*28
      dist_text = font_large.render(f"{SENSOR_LABELS[i]}: {dist:.1f} cm", True, (0, 255, 0))
      screen.blit(dist_text, (col_x, row_y))

   pygame.display.flip()
```

```
    clock.tick(30)
pygame.quit()
```

Label	Angle (deg)	Region
F1	322.5	Front
F2	337.5	Front
F3	352.5	Front
F4	7.5	Front
F5	22.5	Front
F6	37.5	Front
B1	142.5	Back
B2	157.5	Back
B3	172.5	Back
B4	187.5	Back
B5	202.5	Back
B6	217.5	Back
Right	90	Right Side
Left	270	Left Side

- Sensors in the front (F1–F6) are tightly arranged end-to-end from 322.5° through 37.5° (no overlap, no gaps).
- Sensors in the back (B1–B6) similarly cover 142.5° through 217.5° (no overlap, no gaps).
- Side sensors are exactly at 90° and 270°.
- The UI and logic are ready for serial sensor integration.
- Only 14 sensors are used for this arrangement, ensuring "end to end" coverage, no overlap, and no angular gaps in front or back. Blind spots exist only between front-back limits and at the sides beyond the two dedicated side sensors.

Radar Visualization

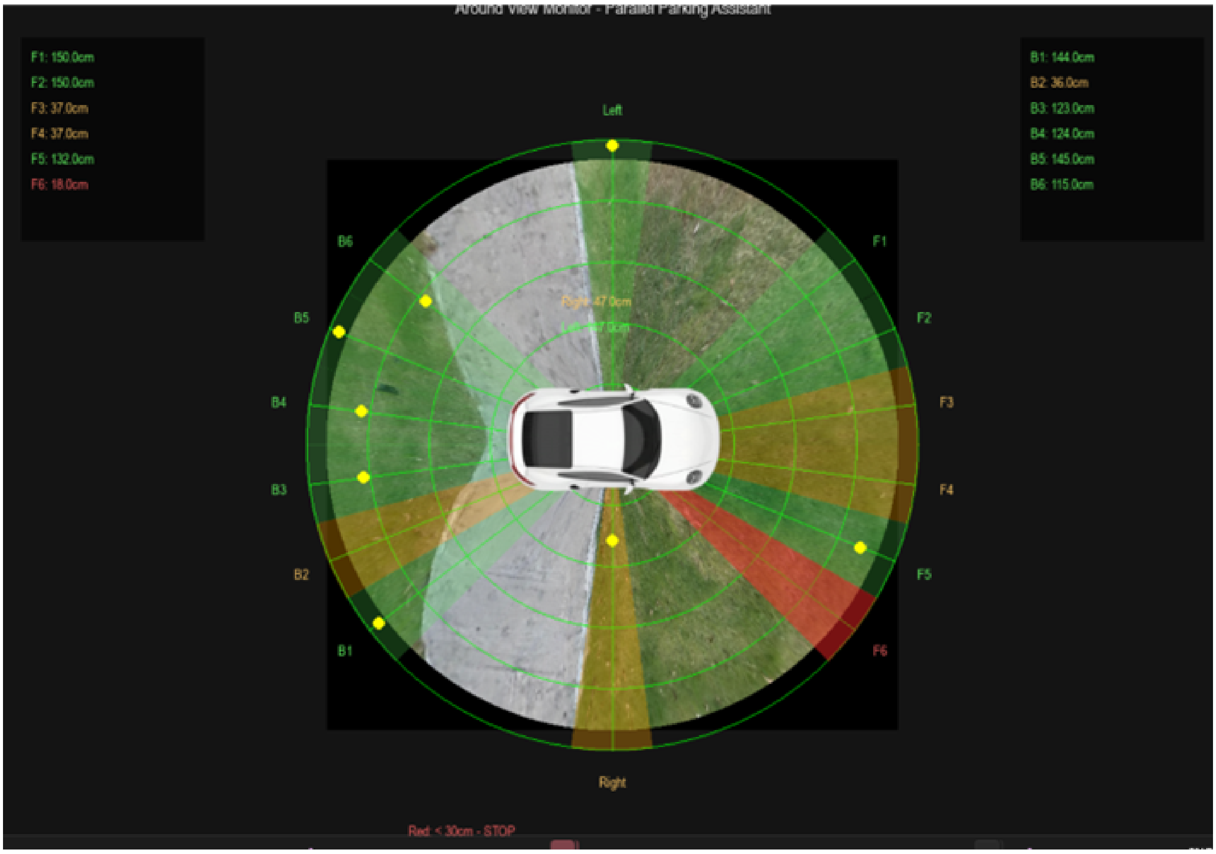


Figure 22: Caption

Python Code: Camera Stitching + Integrated AVM (Around View Monitor) Radar Visualization

Listing 8: Full Python Code: CV Camera Stitching + Pygame Radar AVM Visualization

```
import numpy as np
import cv2
import math
import imutils
import os
import random
import pygame

# ---------- CONSTANTS ----------
WIDTH, HEIGHT = 1200, 900
CENTER = (WIDTH // 2, HEIGHT // 2)
RADAR_RADIUS = 300
MAX_DISTANCE = 150
SENSOR_CONE_ANGLE = 15
NUM_SENSORS = 14
CAMERA_RING_OUTER = 280
CAMERA_RING_INNER = 40
CAR_SIZE = (270, 250) # PNG image will be scaled to this size

# Sensor angles and labels
front_start = -37.5
front_angles = [front_start + i * SENSOR_CONE_ANGLE for i in range(6)]
front_angles = [angle % 360 for angle in front_angles]
back_start = 142.5
back_angles = [back_start + i * SENSOR_CONE_ANGLE for i in range(6)]
side_angles = [90, 270]
SENSOR_ANGLES = front_angles + back_angles + side_angles
SENSOR_LABELS = [f"F{i + 1}" for i in range(6)] + [f"B{i + 1}" for i in range(6)] + ["Right",
    "Left"]
```

```python

class CameraStitcher:
    def __init__(self):
        self.stitched_image = None
        self.camera_overlay = None

    def load_images(self, start, end, ext_list=None, width=800):
        if ext_list is None:
            ext_list = ['.jpg', '.jpeg', '.png', '.JPG', '.JPEG', '.PNG']
        imgs = []
        for i in range(start, end + 1):
            for ext in ext_list:
                img_path = f"{i}{ext}"
                if os.path.exists(img_path):
                    img = cv2.imread(img_path)
                    if img is not None:
                        scale = width / img.shape[1]
                        imgs.append(cv2.resize(img, (width, int(img.shape[0] * scale))))
                    break
        return imgs

    def stitch_and_crop(self, start, end):
        imgs = self.load_images(start, end)
        if not imgs or len(imgs) 0:
            m = cv2.erode(m, None)
            sub = cv2.subtract(m, th)
        cnts2 = imutils.grab_contours(cv2.findContours(m, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE))
        c2 = max(cnts2, key=cv2.contourArea)
        x2, y2, w2, h2 = cv2.boundingRect(c2)
        return p[y2:y2 + h2, x2:x2 + w2]

    def generate_face(self, pano, size, face):
        H, W = pano.shape[:2]
        P = pano.astype(np.float32) / 255.0
        i = np.linspace(-1, 1, size)
        j = np.linspace(-1, 1, size)
        xx, yy = np.meshgrid(i, -j)
        if face == 'front':
            f, u, v = [0, 0, 1], [1, 0, 0], [0, 1, 0]
        elif face == 'right':
            f, u, v = [1, 0, 0], [0, 0, -1], [0, 1, 0]
        elif face == 'back':
            f, u, v = [0, 0, -1], [-1, 0, 0], [0, 1, 0]
        else:
            f, u, v = [-1, 0, 0], [0, 0, 1], [0, 1, 0]
        dirs = np.array(f) + xx[..., None] * np.array(u) + yy[..., None] * np.array(v)
        dirs /= np.linalg.norm(dirs, axis=2, keepdims=True)
        x_, y_, z_ = dirs[:, :, 0], dirs[:, :, 1], dirs[:, :, 2]
        theta = np.arctan2(x_, z_)
        phi = np.arcsin(y_)
        u_p = (theta + np.pi) / (2 * np.pi) * (W - 1)
        v_p = (np.pi / 2 - phi) / np.pi * (H - 1)
        xmap = u_p.astype(np.float32)
        ymap = v_p.astype(np.float32)
        F = cv2.remap(P, xmap, ymap, cv2.INTER_LINEAR, borderMode=cv2.BORDER_WRAP)
        return (F * 255).astype(np.uint8)

    def create_camera_ring(self):
        try:
            pano = self.stitch_and_crop(1, 12)
            if pano is None:
                return self.create_placeholder_ring()
            H, W = pano.shape[:2]
            bot = pano[H // 2:, :]
            face_sz = W // 4
            faces = {face: self.generate_face(bot, face_sz, face) for face in ('front', 'right', 'back', 'left')}
            R_out = CAMERA_RING_OUTER
            R_in = CAMERA_RING_INNER
            S = 2 * R_out
            ring = np.zeros((S, S, 3), dtype=np.uint8)
            arcs = [
                ('front', -math.pi / 4, math.pi / 4),
                ('right', math.pi / 4, 3 * math.pi / 4),
                ('back', 3 * math.pi / 4, 5 * math.pi / 4),
```

```
                ('left', 5 * math.pi / 4, 7 * math.pi / 4),
            ]
            for face_name, t0, t1 in arcs:
                self.paint_on_ring(ring, faces[face_name], t0, t1, R_in, R_out)
            return ring
        except Exception as e:
            print(f"[Camera] Error: {e}")
            return self.create_placeholder_ring()

    def create_placeholder_ring(self):
        R_out = CAMERA_RING_OUTER
        R_in = CAMERA_RING_INNER
        S = 2 * R_out
        ring = np.zeros((S, S, 3), dtype=np.uint8)
        center = S // 2
        y, x = np.ogrid[:S, :S]
        mask_outer = (x - center) ** 2 + (y - center) ** 2 = R_in ** 2
        ring[mask_outer & mask_inner] = [60, 60, 90]
        return ring

    def paint_on_ring(self, canvas, face, t0, t1, r_in, r_out):
        S = canvas.shape[0]
        cx = cy = S // 2
        h, w = face.shape[:2]
        ys = np.arange(h)[:, None]
        xs = np.arange(w)[None, :]
        radii = r_out - ys * (r_out - r_in) / (h - 1)
        thetas = t0 + xs * (t1 - t0) / (w - 1)
        xi = (cx + radii * np.cos(thetas)).astype(int)
        yi = (cy + radii * np.sin(thetas)).astype(int)
        xi = np.clip(xi, 0, S - 1)
        yi = np.clip(yi, 0, S - 1)
        canvas[yi, xi] = face[ys, xs]

class IntegratedAVM:
    def __init__(self):
        pygame.init()
        self.screen = pygame.display.set_mode((WIDTH, HEIGHT))
        pygame.display.set_caption("Integrated AVM - Camera + Ultrasonic Sensors")
        self.font_small = pygame.font.SysFont('Arial', 14)
        self.font_large = pygame.font.SysFont('Arial', 18)
        self.clock = pygame.time.Clock()
        self.camera_stitcher = CameraStitcher()
        self.camera_surface = None
        self.setup_camera_overlay()
        self.car_image = pygame.image.load("porscha.png").convert_alpha()
        self.car_image = pygame.transform.smoothscale(self.car_image, CAR_SIZE)
        self.sensor_distances = [random.randint(40, 120) for _ in range(NUM_SENSORS)]
        self.last_update = pygame.time.get_ticks()

    def setup_camera_overlay(self):
        camera_ring = self.camera_stitcher.create_camera_ring()
        if camera_ring is not None:
            camera_ring_rgb = cv2.cvtColor(camera_ring, cv2.COLOR_BGR2RGB)
            camera_ring_rotated = np.rot90(camera_ring_rgb)
            self.camera_surface = pygame.surfarray.make_surface(camera_ring_rotated)

    def draw_camera_background(self):
        if self.camera_surface:
            camera_rect = self.camera_surface.get_rect()
            x = CENTER[0] - camera_rect.width // 2
            y = CENTER[1] - camera_rect.height // 2
            self.screen.blit(self.camera_surface, (x, y))

    def draw_radar_grid(self):
        for i in range(1, 6):
            r = RADAR_RADIUS * i / 5
            pygame.draw.circle(self.screen, (0, 255, 0), CENTER, int(r), 1)
        for angle in SENSOR_ANGLES:
            a_rad = math.radians(angle)
            end_pt = (CENTER[0] + RADAR_RADIUS * math.cos(a_rad),
                      CENTER[1] + RADAR_RADIUS * math.sin(a_rad))
            pygame.draw.line(self.screen, (0, 255, 0), CENTER, end_pt, 1)

    def draw_sensor_cones(self):
        for i, angle in enumerate(SENSOR_ANGLES):
```

```
            a_rad = math.radians(angle)
            cone_rad = math.radians(SENSOR_CONE_ANGLE)
            points = [CENTER]
            steps = 10
            for s in range(steps + 1):
                t = a_rad - cone_rad / 2 + (cone_rad * s / steps)
                x = CENTER[0] + RADAR_RADIUS * math.cos(t)
                y = CENTER[1] + RADAR_RADIUS * math.sin(t)
                points.append((x, y))
            dist = self.sensor_distances[i]
            if dist 60cm - SAFE"
        ]
        for i, instruction in enumerate(instructions):
            color = (255, 100, 100) if i == 0 else (255, 200, 100) if i == 1 else (100, 255, 100)
            text = self.font_small.render(instruction, True, color)
            self.screen.blit(text, (40, HEIGHT - 80 + i * 24))

    def run(self):
        running = True
        while running:
            self.screen.fill((0, 0, 0))
            self.draw_camera_background()
            self.draw_radar_grid()
            self.draw_sensor_cones()
            self.draw_car()
            self.draw_sensor_labels()
            self.draw_sensor_panels()
            self.draw_instructions()
            pygame.display.flip()
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    running = False
            self.clock.tick(30)
        pygame.quit()

if __name__ == "__main__":
    avm = IntegratedAVM()
    avm.run()
```

Mounts for Ultrasonic sensor

Figure 23: Various mounts of ultrasonic sensor

Placement Configurations

Figure 24: Various placement configuration of ultrasonic sensor

System Verification and Validation

TO BE CONT. ....

Design Limitations and Future Improvements

This project demonstrates a practical implementation of a 360-degree ultrasonic sensor system combined with camera stitching and data fusion techniques to provide a comprehensive view of the vehicle surroundings. Despite the successful integration, several design limitations exist:

- **Sensor Blind Spots:** Due to fixed sensor cone angles and spacing, blind spots remain between sensor coverage areas. Increasing sensor overlap or repositioning sensors could mitigate this.
- **Environmental Sensitivity:** Ultrasonic sensors are affected by temperature, humidity, and atmospheric pressure variations, causing measurement inaccuracies without real-time compensation.
- **Interference and Crosstalk:** Multiple sensors firing simultaneously can cause echo interference. Sequential triggering and signal coding strategies are needed to reduce this.
- **Processing Latency:** Real-time stitching and fusion of multiple camera feeds and sensor data require significant processing power, possibly limiting frame rate or system responsiveness.

- **Hardware Constraints:** Limitations of microcontroller ports, power consumption, and cabling complexity can impact scalability for larger sensor arrays.

Future improvements may include adaptive sensor scheduling to reduce crosstalk, integration of temperature and humidity sensors for automatic compensation, and use of machine learning algorithms for enhanced data fusion and object recognition.

Glossary

AVM Around View Monitor – A system providing a 360-degree visual and sensor-based view around a vehicle.

BEV Bird's Eye View – A top-down perspective visualization often used in vehicle monitoring.

CDMA Code Division Multiple Access – Technique to separate signals in simultaneous sensor transmissions.

ESP32 A popular microcontroller with Wi-Fi and Bluetooth capabilities, commonly used for sensor integration.

FOV Field of View – Angular coverage of a sensor.

RTTC Reference Target Temperature Compensation – A method to improve ultrasonic sensor accuracy using a reference target.

RH Relative Humidity – The percentage of water vapor in the air relative to the maximum possible at that temperature.

Sensor Fusion The process of combining data from multiple sensors to create a cohesive understanding of the environment.

References

- *Understanding Ultrasonic Sensor Accuracy*, Senix Corporation. Available at: `https://senix.com/ultrasonic/accuracy/`
- *HC-SR04 Ultrasonic Sensor Datasheet and Application Guide*, NJIT. Available at: `https://ecelabs.njit.edu/fed101/resources/HC-SR04%20Ultrasonic%20Sensor.pdf`
- *Ultrasonic Sensor FAQ – Environmental Influences*, Pepperl+Fuchs. Available at: `https://blog.pepperl-fuchs.com/en/2018/ultrasonic-sensor-faq-external-influences/`
- *Improving Ultrasonic Sensor Readings with Environmental Compensation*, NerdyElectronics. Available at: `https://nerdyelectronics.com/how-to-improve-readings-of-ultrasonic-sen`
- *Ultrasonic Sensor and Temperature Compensation*, TeachMeMicro. Available at: `https://www.teachmemicro.com/ultrasonic-temperature-compensation/`
- Weather Data: *Hisar*, Weather Online. Available at: `https://weatheronline.in`
- Weather Data: *Delhi*, Weather Atlas. Available at: `https://weather-atlas.com`
- Weather Data: *Nagpur*, Weather Atlas. Available at: `https://weather-atlas.com`
- Weather Data: *Chennai*, Weather Atlas. Available at: `https://weather-atlas.com`
- Weather Data: *Darjeeling*, Weather Atlas. Available at: `https://weather-atlas.com`
- Weather Data: *Leh*, Weather Atlas. Available at: `https://weather-atlas.com`

- Weather Data: *Ooty*, Weather Atlas. Available at: `https://weather-atlas.com`
- *Climate of India*, Wikipedia. Available at: `https://en.wikipedia.org/wiki/Climate_of_India`
- *Weather Patterns in India*, Intrepid Travel. Available at: `https://www.intrepidtravel.com/india/weather`
- *India Climate Data*, World Bank Climate Knowledge Portal. Available at: `https://climateknowledgepor worldbank.org/country/india`
- *Parking Space Dimensions and Rules – India*, ICICI Lombard. Available at: `https://icicilombard.com/blogs/car-insurance/parking-space-dimensions`
- *Vehicle Dimension Reference*, Transport Info. Available at: `https://transportinfo.in`
- *Average Vehicle Sizes*, Neighbor Storage Blog. Available at: `https://neighbor.com/storage-blog/average-vehicle-sizes`
- *Parking Dimension Simulator*, AutomobileDimensions.com. Available at: `https://automobiledimension com`
- *Vehicle Size Classification*, Wikipedia. Available at: `https://en.wikipedia.org/wiki/Vehicle_size_class`
- *FCC Documentation*, Wireless Module. Available at: `https://fcc.report/FCC-ID/WE7H-113`
- *AVS Gemini User Guide*, AVSGemini. Available at: `https://avsgemini.com/documents/installation-guide.pdf`
- *Senix Ultrasonic Technical Video*, YouTube. Available at: `https://youtu.be/h632UB`
- *IEEE Paper on Ultrasonic Radar*, IEEE Xplore. Available at: `https://ieeexplore.ieee.org/document/10870780`