Object Detection using SONAR based system

Report submitted for

Mini Project
(5th Semester)

in partial fulfillment of the degree of

Bachelor of Technology
in
Electronics and Communication Engineering

By

Resu Nikhil Reddy, 14UEC080
Sumit Sapra, 14UEC108
Surya Prakash Venkat, 14UEC109

Under the guidance of

Dr. M. V. Deepak Nair, Assistant Professor
Department of Electronics and Communication Engineering
The LNM Institute of Information Technology
Jaipur, India

The LNM Institute of Information Technology
Jaipur, India
http://www.lnmiit.ac.in

Title:
Object Detection using SONAR based system

Theme:
Embedded Systems

Project Period:
Odd Semester 2016

Project Group:
Resu Nikhil Reddy
Sumit Sapra
Surya Prakash Venkat

Supervisor:
Dr. M. V. Deepak Nair

Page Numbers: 42

Date of Completion:
December 10, 2016

Abstract:

The goal is to develop an object detection system which could determine the presence of objects in it's vicinity.

The system leverages the *HC-SR04 Ultrasonic Sensor* mounted on top of a *Micro-Servo* motor, both being controlled by an *Arduino UNO*. The HC-SR04 detects its separation from an obstacle placed in front of it and sends the data to the Arduino, while the servo rotates the sensor about its axis in order to have a field of view that sweeps over a sector with a radius specified within 4 metres.

The *Arduino* recieves sensor readings (distance measurements) from the sensor and sends it via *USART* to a computer for graphical display using the *Processing* IDE.

Contents

Introduction

The basic approach of our project, just like any other based on embedded systems (*microcontrollers*) largely depends on three main questions:

- Which microcontroller board would be appropriate?
- What *sensor*(s) should be used to fulfill the goals of the project?
- What actuators might be needed for introducing mechanical action, if needed?

We had a literature survey through the internet about deciding the components needed for our project. This included studying in detail, several project ideas and instructions posted on blogs and YouTube.
We selected the following major components for having a *minimal*, *cost effective* and *efficient* apparatus that could do the job for us:

- *Microcontroller Board - Arduino UNO*
- *Sensor - HC-SR04 Ultrasonic Sensor*
- *Actuator - Micro-Servo* Motor

The next few sections will explain the basis of selection of hardware for this project.

The *Arduino UNO*

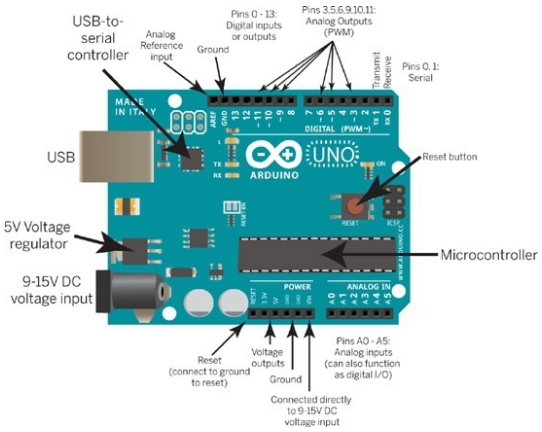

Figure 1.1: The Arduino UNO

Arduino is an open source electronics platform for fast prototyping of projects for users with minimal knowledge or experience in electronics and programming. We have used the *Arduino UNO*, the most widely used variant of the Arduino.
Technically, Arduino Uno is a microcontroller board based on the 8-bit ATMega328P microcontroller. With 14 digital input/output pins (including 6 PWM outputs), 6 analog inputs, a 16 MHz ceramic resonator, a USB connection, a power jack, an ICSP header, and a reset button, it has everything needed to support the microcontroller; we simply have to connect it to a computer with a USB cable or power it with a AC-to-DC adapter or battery to get started.
Arduino also comes with the Arduino IDE, in which we can write code and upload to the Arduino. The programming language used for an Arduino, called the Arduino Programming language, is very similar to C/C++ except that we can use inbuilt functions of the Arduino libraries which keep the code very simple and short.

Why Arduino?

We had the following reasons strongly backing our decision to choose the *Arduino UNO*:

Support - Thanks to its simple and accessible user experience, Arduino has been used in thousands of different projects and applications. The Arduino software is easy-to-use for beginners, yet flexible enough for advanced users. It runs on Mac, Windows, and Linux.
This has enabled a very large support base for Arduino, more than any other microcontroller board ever released in the market. Every problem/bug/question that we ever had throughout the course of this project was promptly answered by questions in online forums, as there was a very high probability that scores of people had already posted the same question and the issue was resolved by other experienced users.

Microcontroller	ATmega328P(8-bit)
Operating Voltage	(DC)5V
Input Voltage	(recommended) 7-12V; (limits) 6-20V
Digital I/O Pins	14 (of which 6 provide PWM output)
Analog Input Pins	6
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328) of which 0.5 KB used by bootloader
SRAM	2 KB (ATmega328)
EEPROM	1 KB (ATmega328)
Clock Speed	16 MHz
Need of External Programmer	NO
Built-in LED	pin13
Length	68.6 mm
Width	53.4 mm
Weight	25 g

Table 1.1: Features of Arduino UNO

Cost Effective - The Arduino UNO, its basic variant that every beginner to embedded systems usually starts with, is available for less than $25. Being an open source hardware and due to its simplicity, it is easily and widely replicable, which is also responsible for the growth of such a large community as mentioned in the earlier reason.

Features - It is generally observed that the computing power and inbuilt memory of the Arduino easily supports a wide variety of circuits with multiple components along with continuous serial communication with another device/computer. Compared to the system configuration of the board, our project would require fairly low amount of resources.

The no. of GPIO pins on the UNO is more than sufficient considering the need for the circuitry in our project, as we shall see when we look at the pinouts of the servo motor and the sensor - the only two other devices being used in the project.

One important advantage of using the UNO is that unlike most previous programmable circuit boards, the *Arduino UNO does not need a separate piece of hardware (called a programmer) in order to load new code onto the board* – we can simply use a USB cable. The ATmega328 on the Arduino/Genuino Uno comes preprogrammed with a *bootloader* that allows us to upload new code to it without the use of an external hardware programmer.

Moreover, the components we have used work under the same operating voltage and current conditions as an arduino, especially considering the fact that such components are nowadays sold with their compatibility with arduino in mind.

Considering the above factors, we found Arduino UNO to be the most appropriate microcontroller board to work with for this project.

Operation of Arduino

Once a program written for an *Arduino*, called a "*Sketch*" is uploaded to the board via the Arduino IDE, the Arduino environment performs some small transformations to make sure that the code is correct C/C++. It then gets passed to a compiler (*avr-gcc*), which turns the human readable code into machine readable instructions (*object* files). Then, the code gets combined with (*linked* against) the standard Arduino libraries that provide basic functions like *digitalWrite()* or *Serial.print()*. The result is a single Intel *hex* file, which contains the specific bytes that need to be written to the program(*flash*) memory of the chip on the Arduino board. This file is then uploaded to the board: transmitted over the USB or serial connection via the bootloader already on the chip or with external programming hardware.

The *HC-SR04 Ultrasonic Sensor*

Figure 1.2: HC-SR04 Ultrasonic sensor

The HC-SR04 ultrasonic sensor uses sonar to determine distance to an object like bats or dolphins do. It offers non-contact range detection from 2cm to 400 cm or 1″ to 13 feet. It's operation is not affected by sunlight or black material like sharp rangefinders are (although acoustically soft materials like cloth can be difficult to detect). It comes complete with ultrasonic transmitter and receiver module.
The human ear can only detect sounds betwen 20Hz-20kHz. The sound waves beyond 20kHz are called ultrasonic waves or ultrasound.
The principle of ultrasonic distance measurement uses the velocity of ultrasonic waves spreading in air, measuring the time from launch to reflection when it encounters an obstacle. We then calculate the distance between the transmitter and the obstacle according to the time and the velocity. Thus, the principle of ultrasonic distance measurement is the same as with a radar.

Features

The reason to choose this sensor was due to its reasonable range of taking measurements of the presence of an object for very less cost and electrical/computational resources. It's

Electrical Parameters	HC-SR04 Ultrasonic Module
Operating Voltage	(DC)5V(4.5-5.5V)
Quiescent Current(inactivity)	2mA(1.5-2.5mA)
Working Current	15mA(10-20mA)
Operating Frequency	40KHZ(ultrasonic)
Farthest Range	4m
Nearest Range	2cm
Resolution	0.3cm
Measuring Angle	15 Degrees
Input Trigger Signal	10us TTL pulse
Output Echo Signal	Output TTL level signal, proportional with range
Dimensions	45*20*15mm

Table 1.2: Features of HC-SR04

features are given in Table 1.2.

Note - Use of only stationary objects

Since the sensor considers the distance to an obstacle that reflects the ultrasonic wave, we cannot use it to make trustworthy measurements of the position of *moving* objects. This is majorly due to the fact that this sensor works on the principle of transmitting a set of pulses and recieving their reflection, before transmitting again, as we shall see later. Hence, there is no way that we can continuously track the position of an object with such a sensor. As far as we know, that can only be done by some type of a *camera*, which would've introduced the concepts of computer vision and taken us beyond the scope and costs of the current project.

HC-SR04 : Pin Definitions

- Vcc - 5V Power Supply
- Trig - Trigger Pin (Input)
- Echo - Receive Pin (Output)
- GND - Power Ground

The *Trig* and *Echo* pins are used for tranmission and reception of ultrasonic waves, respectively.

Operation of the HC-SR04

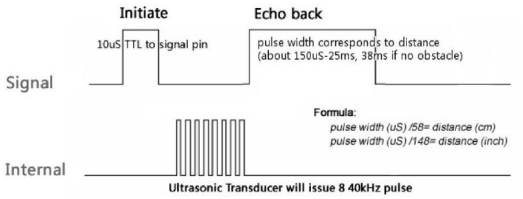

Figure 1.3: Timing Diagram of HC-SR04 Ultrasonic sensor

The timing diagram of HC-SR04 is shown in Figure 1.3. To start measurement, Trig of SR04 must receive a HIGH pulse (5V) for at least 10us, which will initiate the sensor to transmit 8 cycles of ultrasonic burst at 40kHz. It then waits for the reflected ultrasonic wave. When the sensor detects the ultrasonic wave through the receiver, it will set the Echo pin to HIGH (5V), for as long as it detects an incoming ultrasonic pulse at the receiver.

Calculation of distance

The delay between tranmission and reception of the final pulse (i.e., between when the Trig pin is set back to LOW and when the Echo pin turns HIGH) is a time period proportional to the distance travelled by it. To obtain the distance, we start a counter the moment the Trig pin is set to LOW which keeps counting till the microcontroller detects a HIGH pulse at the Echo pin. Time = Width of pulse, in uS (micro second).

- Distance in centimeters = Time / 58
- Distance in inches = Time / 148

as mentioned in its datasheet. We can also utilize the speed of sound, which is 340m/s in a generic case.
This entire process can be translated into the Arduino code as shown in example 1.1.

Example 1.1 (Arduino Code for calcuating distance from HC-SR04 sensor)

```
/* Function for calculating the distance measured by the
     Ultrasonic sensor*/
float calculateDistance(){
unsigned long T1 = micros();
digitalWrite(trigPin, LOW); // trigPin needs a fresh LOW
     pulse before sending a HIGH pulse that can be
    detected from echoPin
delayMicroseconds(2);//DELAY #2:time for which low trig
    pulse is maintained before making it high
digitalWrite(trigPin, HIGH);
delayMicroseconds(10);//DELAY #3:Sets the trigPin on
    HIGH state for 10 micro seconds
digitalWrite(trigPin, LOW);
duration = pulseIn(echoPin, HIGH); // Reads the echoPin,
     returns the sound wave travel time in microseconds
distance = (duration/2)/29.1;      //in cm. Calibrated
    form. Datasheet shows "duration/58" as the formula

return distance;
}
```

Need of calibration of the sensor

As we can notice in the given code, we had to change the formula for calculation of distance, as the values were not close to real values of distance of objects kept in front of it, as tested by us. We would discuss it in a later stage.

Micro-Servo Motor

Figure 1.4: Micro-Servo Motor

We had to use a servo motor in order to rotate the *HC-SR04* and give it a wider field of view over the area around it. This *Micro-Servo* can rotate approximately 180 degrees(90 in each side) with an operating speed of 0.1s/60degrees. Moreover, it has an operating voltage of 4.8V(5V) which is in the same range as of the *Arduino UNO*.
Other details:

- Weight: 9 g
- Dimension: 22.2 x 11.8 x 31 mm approx.
- Stall torque: 1.8 kgf·cm
- Operating speed: 0.1 s/60 degrees
- Operating voltage: 4.8 V (5V)
- Deadband width: 10 us (The servo will not move so long as subsequent commands to it are within the deadband timing pulse width)
- Temperature range: 0-55C

The *Micro-Servo* has a *signal* pin to which the arduino sends the value of angle that it should turn to. Figure 1.5 shows the pins of the *Micro-Servo*.

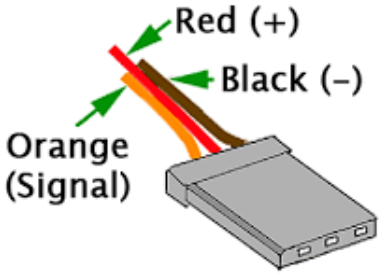

Figure 1.5: Pins of Micro-Servo

The *Ultrasonic Sensor* was mounted on top of it and the apparatus connected to the arduino was kept in a modified transparent plastic case, as shown in Figure 1.6.

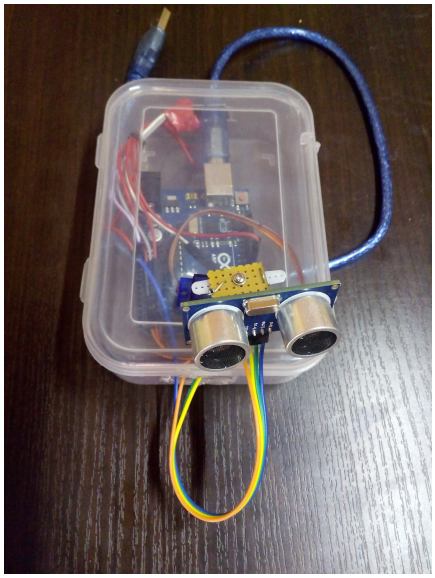

Figure 1.6: The Apparatus

We then proceeded to deciding the circuit connections, as shown in the next section.

Circuit Schematics

Arduino Pin Connections

Pin of Arduino	Connected to
Digital I/O (2)	Trig of HC-SR04
Digital I/O (4)	Echo of HC-SR04
Digital I/O (9)	Signal pin of Servo(orange)
Vcc	Vcc of HC-SR04 and Servo(Red)
GND Pin	GND of HC-SR04 and Servo(Black)

Breadboard View

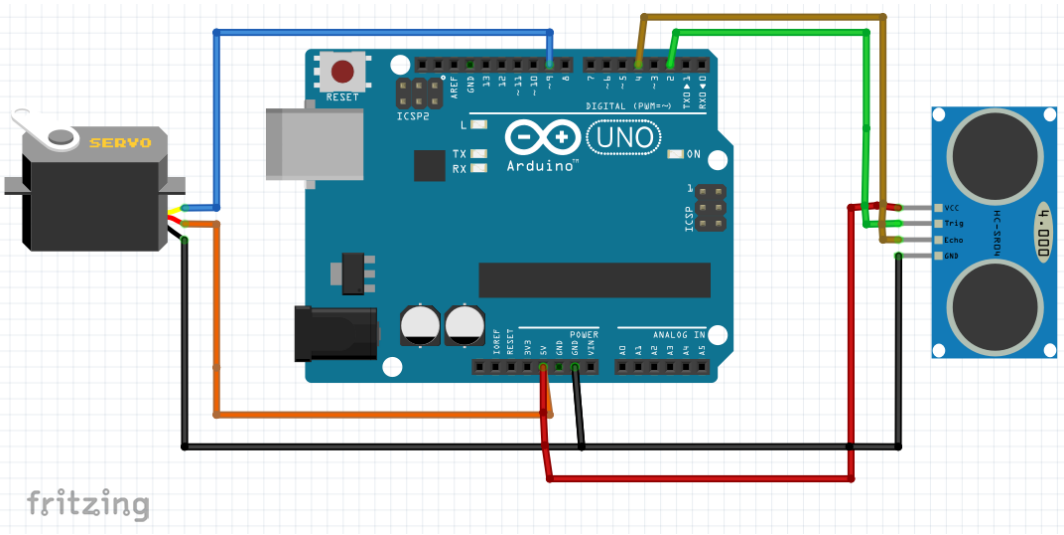

Figure 2.1: Circuit : Breadboard View

Schematic view

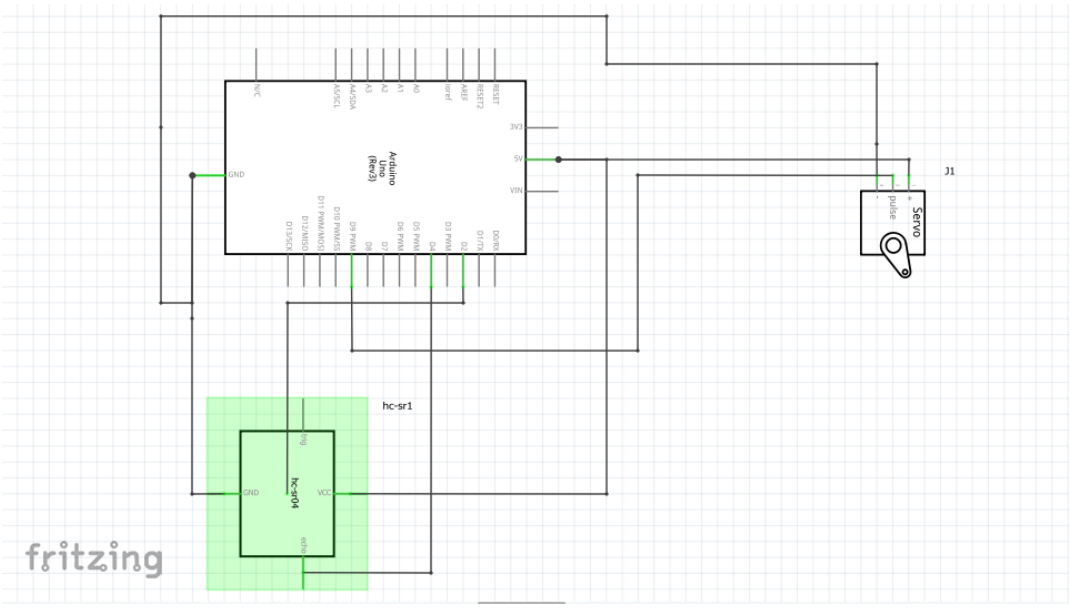

Figure 2.2: Circuit : Schematic view

Above circuit schematics were obtained using the Fritzing software.
Once we decided the connections, we had to two more steps left before connecting the actual circuit: setting up the **Development Environment** and **Circuit Simulation**.

Development and Simulation

Arduino IDE

The code to be burnt to the program memory of an *Arduino* is written and compiled in the *Arduino IDE*. The IDE also has a serial monitor, which displays values being recieved in real-time from the Arduino via serial communication through the serial port (COM ports). The IDE verifies for correct C/C++ syntax and compiles it, before linking it to Arduino Library files. This creates the *hex* file that contains the code in binary form. It can either be uploaded to the board directly, which the IDE does with the help of the Bootloader program pre-installed on the Arduino, or it can be fed to a *simulator*, which would simulate and show how the circuit would run and its associated parameters.

Figure 3.1: The Arduino IDE

Simulation with Proteus

It is a good practice to simulate a circuit before connecting it, in order to avoid hardware damage/debug issues in a circuit beforehand. We used *Proteus Professional* 8 for this purpose. Proteus has libraries for most commonly used electronics components, and it also allows us to make our own components, thanks to which we could download unavailable components from the internet, where many users have shared their own custom components.

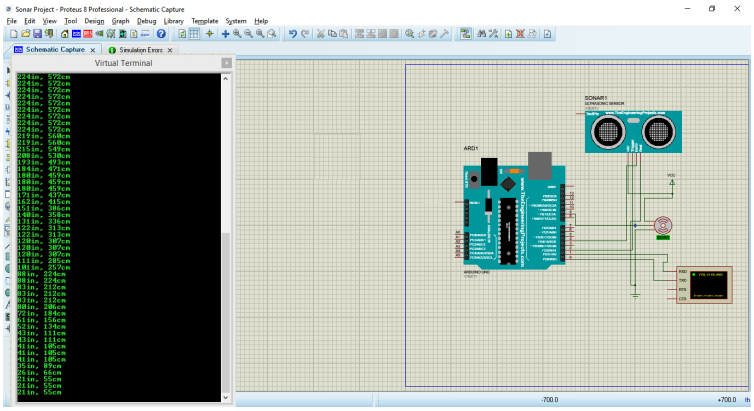

Figure 3.2: The Proteus Simulator

Processing

Processing is an interactive software to write programs with visual output. We have used Processing 3.2 for generating a 2D radar-map representation of the serial data being obtained from the Arduino through the computer's serial port.

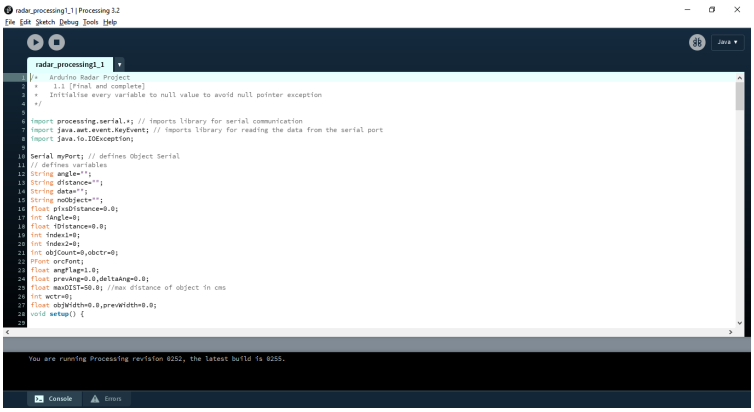

Figure 3.3: Processing

The code in Processing is written in *Java*, and primarily consists of drawing the lines

and borders of the radar map, and plotting the tracker-bar on the map. We added functionalities such as counting the number of objects, and also tried to calculate the width of the object, apart from just showing the distance of the object.
We made a 360 degrees map so that we could view the area according to our convenience if the angle of the set-up is changed. This angle is the "offset" that we introduced in the Arduino code. The entire Arduino and Processing codes are shown in Appendix C and Appendix D, respectively.

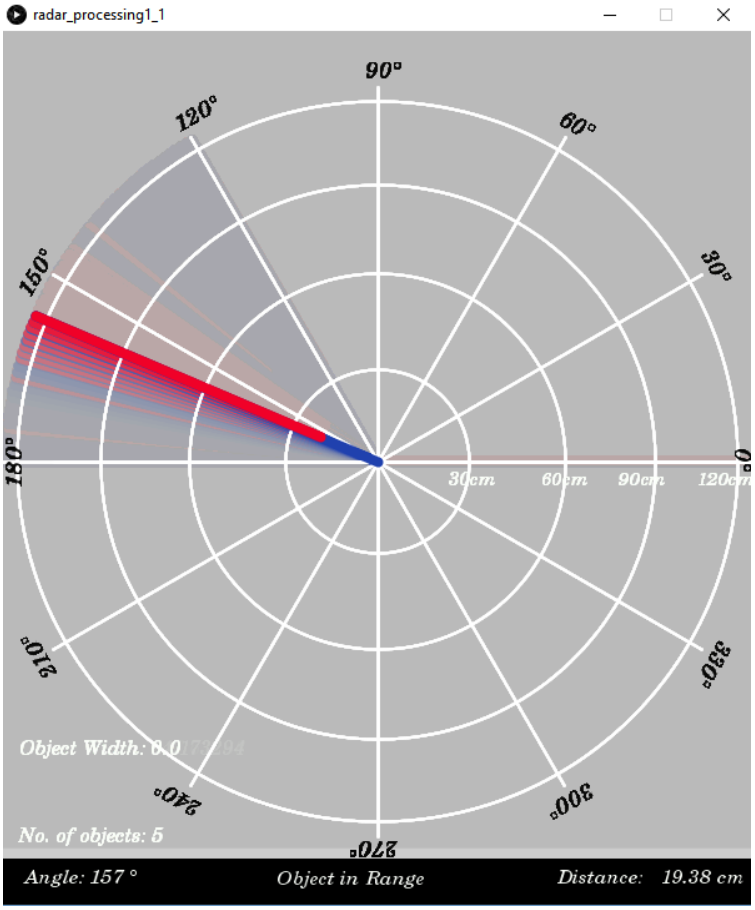

Figure 3.4: Running our code in Processing

Explaining the Codes

Here we'd be briefly going through the flow of the codes written for *Arduino* and in *Processing*. The entire Arduino and Processing codes are given (explained with comments) in Appendix C and Appendix D, respectively.

The Arduino Code

Every *Arduino* code consists of two standard functions - *void setup()* and *void loop()*. The former is used to define and initialise variables, pins and the baud rate of serial communication.
We have programmed the Arduino to control the circuit in the following flow:

1. Rotate the motor to an angle,
2. Use the ultrasonic sensor and calculate the distance,
3. Send both the above values via serial communication,
4. Rotate the motor to the next degree,
5. Repeat steps 2-5.

The following code snippets implement the above given steps:

Listing 4.1: Step 1

```
// rotates the servo motor from init_ang to fin_ang
    degrees
for(int i=init_ang;i<=fin_ang;i++)
{
myServo.write(i-offset);  //angle value to be passed to
    the servo library object for writing into the motor
delay(17);//DELAY #1:for time taken in motor rotation for
    one degree before calculating distance
```

Listing 4.2: Step 2

```
distance = calculateDistance();// Calls a function for
```

```
    calculating the distance measured by the Ultrasonic
    sensor for each degree
```

Listing 4.3: Function called above

```
// Function for calculating the distance measured by the
    Ultrasonic sensor
float calculateDistance(){
unsigned long T1 = micros();
digitalWrite(trigPin, LOW); // trigPin needs a fresh LOW
    pulse before sending a HIGH pulse that can be detected
     from echoPin
delayMicroseconds(2);//DELAY #2:time for which low trig
    pulse is maintained before making it high
digitalWrite(trigPin, HIGH);
delayMicroseconds(10);//DELAY #3:Sets the trigPin on HIGH
    state for 10 micro seconds
digitalWrite(trigPin, LOW);
duration = pulseIn(echoPin, HIGH); // Reads the echoPin,
    returns the sound wave travel time in microseconds
//distance= duration*0.034/2;
distance = (duration/2)/29.1;     //in cm,  datasheet
    gives "duration/58" as the formula
```

Listing 4.4: Step 3

```
Serial.print(i); // Sends the current degree into the
    Serial Port for graphical representation
Serial.print(","); // Sends addition character right next
    to the previous value needed later in the Processing
    IDE for indexing
Serial.print(distance); // Sends the distance value into
    the Serial Port for the graph
Serial.print(";"); // Sends addition character right next
    to the previous value needed later in the Processing
    IDE for indexing
```

Keeping the above code in a `void loop()` ensures that the steps get repeated in a loop, as desired.

The Processing Code

The JAVA code written for processing divides its work into a few main functions, which are customary for every code written in Processing. The functions are

- `void setup()`
  - to initialise variables and baud rate of serial communication,
- `void draw()`
  - to make a 2D/3D drawing on the screen on which to show the output,
- `void serialEvent()`
  - the function dealing with incoming/outgoing serial data, and operations on it.

Customised functions were used to draw several parts of the displayed graph, which were called from `void draw()`. These are :

- `void drawRadar()`
  - to draw the basic circular layer of graphics
- `void drawLine()`
  - to draw a line with trails which would be used to sweep over the area implying scanning
- `void drawObject()`
  - to draw a colored line over the scanning figure to indicate presence of an object, whose length would be dependant on the distance of the object from the sensor
- `void drawText()`
  - to place text at different places on the map to show the serial and calculated data values

Observations

We observed data from both the serial monitor of the Arduino IDE, as well as from the graph in processing. Since we wanted our device to give the best results as quickly as possible, we decided to calibrate it.

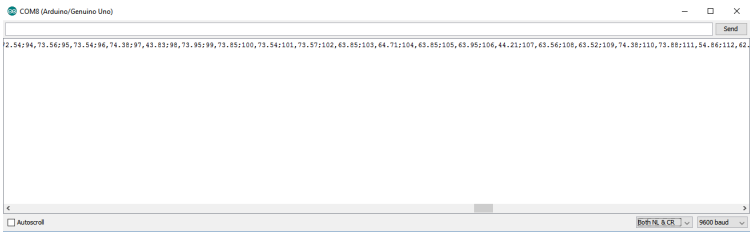

Figure 5.1: Data Observed in the Serial Monitor of the Arduino IDE

Calibration

Calibration of Measurements

We realised that actual measurements of distance/count/width of an object placed in the vicinity of the apparatus were different from the ones provided by our device. Hence, we proceeded to calibrate each part of our device that was capable of taking a measurement.

The quantities that were measured in our project before going for any further calculations were: Angle and Distance. The rest were only obtained by playing around with these two main quantities. Hence, we needed them to be as accurate as possible.

We did not notice any error in the value of angle that the motor was rotating to, hence we did not have to do any sort of calibration for it.

To calibrate the distance, we placed several objects at fixed distances from the sensor on a white chart paper, with values of distances marked on it with the help of a ruler. We then made measurements keeping the sensor still, and made an adjustment to the formula in the code by observing the multiplication factor that needed to be introduced into the previous formula in order to get the correct values. Hence, the formula was changed from "duration/58" to "duration/58.2".

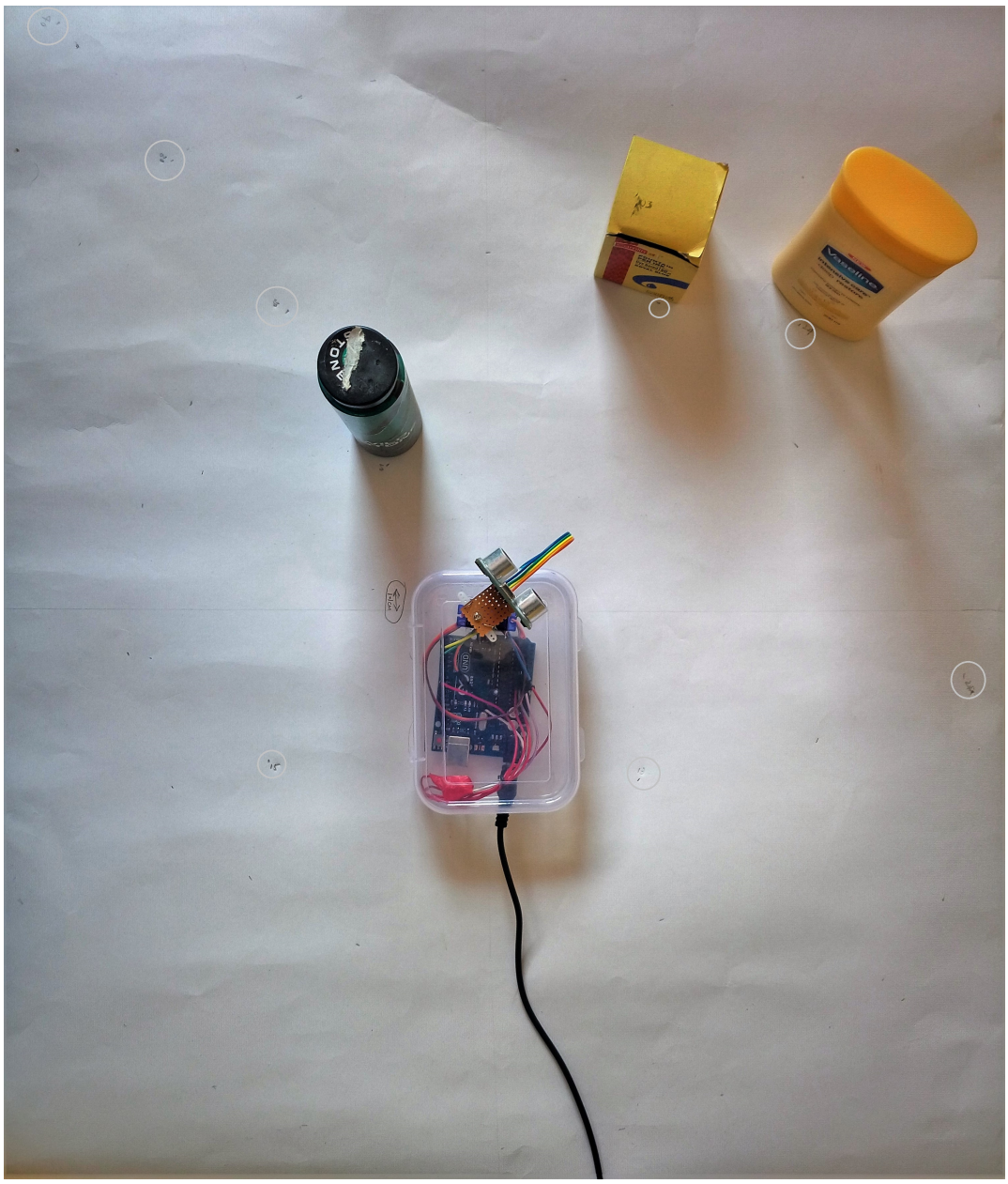

Figure 5.2: Calibration on chart paper with marked distances

Optimising Speed

We wanted to optimise our device to work as quickly as possible, without affecting normal execution of the program. So we optimised our code, considering the delays in physical movement, time periods of recieved/transmitted waves and time taken to process the data.

We took into account the several delays introduced in the code, along with the delay in rotation of the motor(specified by its speed given in its datasheet, tested to be accurate) and delay in calculation of distance. Accordingly, we minimised each of the values given as arguments to the

`delay()`

function used at different parts of the Arduino code, to the safest least value possible. Each of the delays with descriptions can be observed in the Arduino code given with comments in Appendix C

Graphs

We obtained the following [error vs. position] graphs for three different kinds of objects kept in front of the sensor. In each case, we noted down values for both the sensor being stationary, and rotating. The three different kinds of objects were

- Object1 = Hollow plastic bottle (width=6cm)
- Object2 = Solid DC battery (width=3cm)
- Object3 = Metallic pen (width=0.8cm)

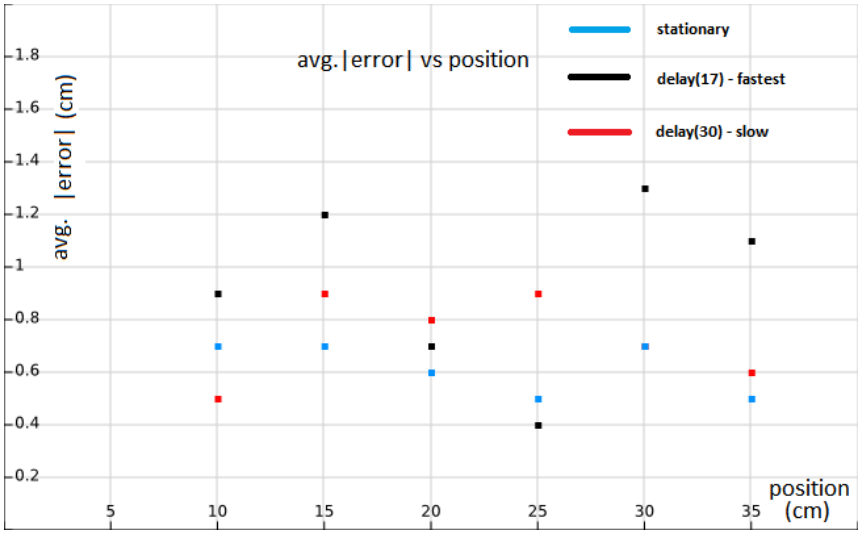

Figure 5.3: Object1-avg.|error| vs position

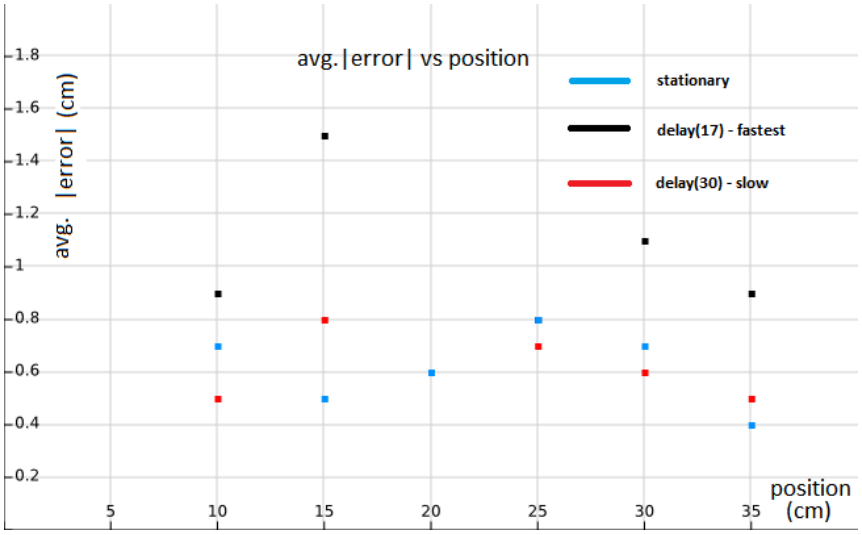

Figure 5.4: Object2-avg.|error| vs position

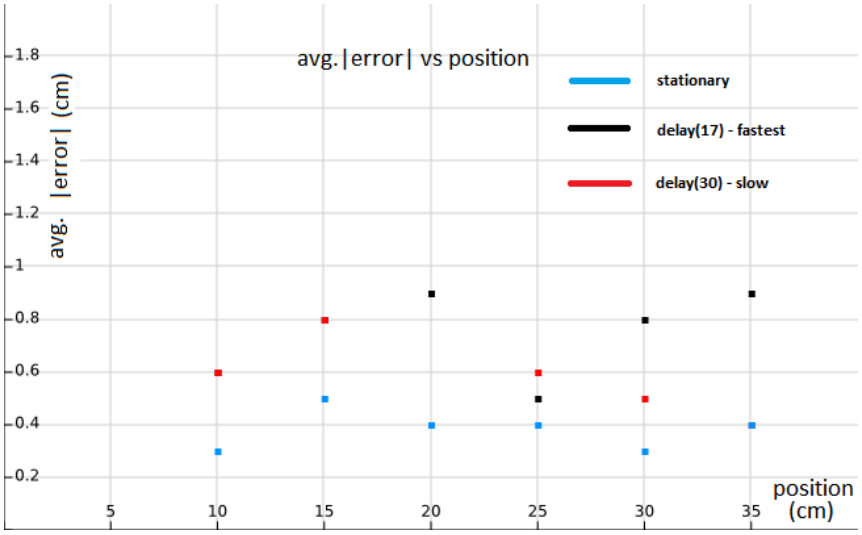

Figure 5.5: Object3-avg.|error| vs position

Finally, we also plotted the error in data of width of object vs. position, which was something like this:

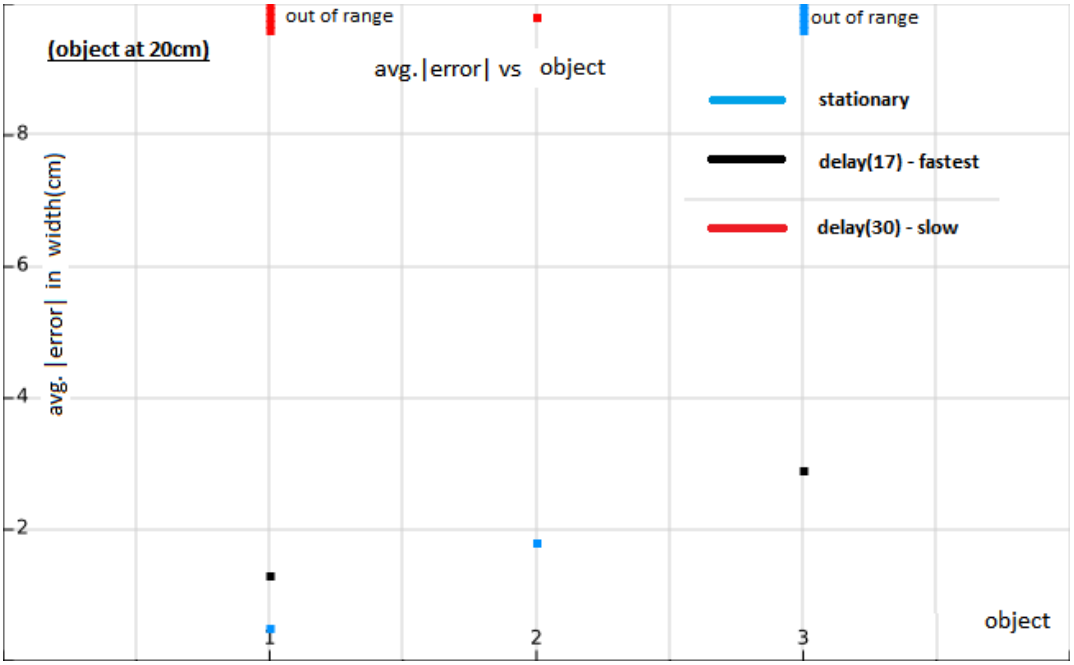

Figure 5.6: Object-wise avg.|error| in width at x=20cm

Challenges

As observed from the graphs plotted in the previous section, we are not able to determine any particular increasing/decreasing order being followed by the errors in measurement of the "distance" quantity. Moreover, by increasing the speed of measurement of the device (by reducing the manual delays introduced inside the Arduino code within safe limits), the error does not necessarily seem to increase with speed. The "width of the object" quantity seems to be correct at one instant and completely wrong at any other instant with very random error values.
However, what we did observe were sudden jumps in errors. Among several trials, their occurrence increased with the speed. That is, the device was detecting *stray* values, and the amount of stray values per experiment increased with the speed of the device.

Identified Problem

With the help of ideas from several online electronics forums, we carefully studied the flow of the code, which exposed the flaw in our logic. The problem lied in the fact that the code took variable amount of time in calculating the distance of an object, proportional to the distance itself. This fact could be derived from the line in the code

```
duration = pulseIn(echoPin, HIGH);
```

where the function pulseIn starts a counter which counts till the value at echoPin is HIGH. Here, our program only waited to recieve the first returning pulse, before quickly sending the output and turning the motor to the next angle. At some instances, this could mean that by the time the sensor was ready to take a new measurement, the older echo pulses were still arriving at the receiver, hence a wrong measurement. The older echo pulses could still be around due to
(a) The program having quickly moved to the next set of commands and having made the sensor ready to receive ultrasonic waves - this, within the time in which the remaining of the 8 pulses were still being recieved at the reciever, or
(b) Due to the waves being spherical in nature and the field of view of the sensor being 15 degrees, the waves could've been reflected by some undesired obstacles in the field of view. It could also be the result of multiple reflections.

Proposed Solution

Therefore, we edited the algorithm in such a way, that between each degree of rotation of the motor, the program takes the exact amount of time to run. That is, the program makes up for the time left out compared to the extreme-time-taking case by waiting.
We did this by using the fact to our advantage that the maximum time taken by the calculateDistance() function is when there is no obstacle. The *HC-SR04* sensor waits for 38ms in case no object is detected, before giving a HIGH echo signal. Hence, we ensure that the program always takes a total of 38ms for each iteration of the loop. For this, we used the

micros() function inside the *calculateDistance()* function of the arduino code, which calculates the number of microseconds elapsed till the code execution started.
Here's the added portion of code which did our solution:

Listing 5.1: Corrected calculateDistance() function

```
// Function for calculating the distance measured by the
    Ultrasonic sensor
float calculateDistance(){
unsigned long T1 = micros();
digitalWrite(trigPin, LOW); // trigPin needs a fresh LOW
    pulse before sending a HIGH pulse that can be detected
     from echoPin
delayMicroseconds(2);//DELAY #2:time for which low trig
    pulse is maintained before making it high
digitalWrite(trigPin, HIGH);
delayMicroseconds(10);//DELAY #3:Sets the trigPin on HIGH
    state for 10 micro seconds
digitalWrite(trigPin, LOW);
duration = pulseIn(echoPin, HIGH); // Reads the echoPin,
    returns the sound wave travel time in microseconds
//distance= duration*0.034/2;
distance = (duration/2)/29.1;     //in cm,  datasheet
    gives "duration/58" as the formula

//To avoid sending data at variable time intervals due to
    varying time duration taken between execution of above
     code inside this function depending on distance of
    obstacle
//if no object, echo pulse is HIGH after 38ms
while(micros()-T1<38000)
{
;
}

return distance;
}
```

Here are the graphs replotted for the data obtained after correction

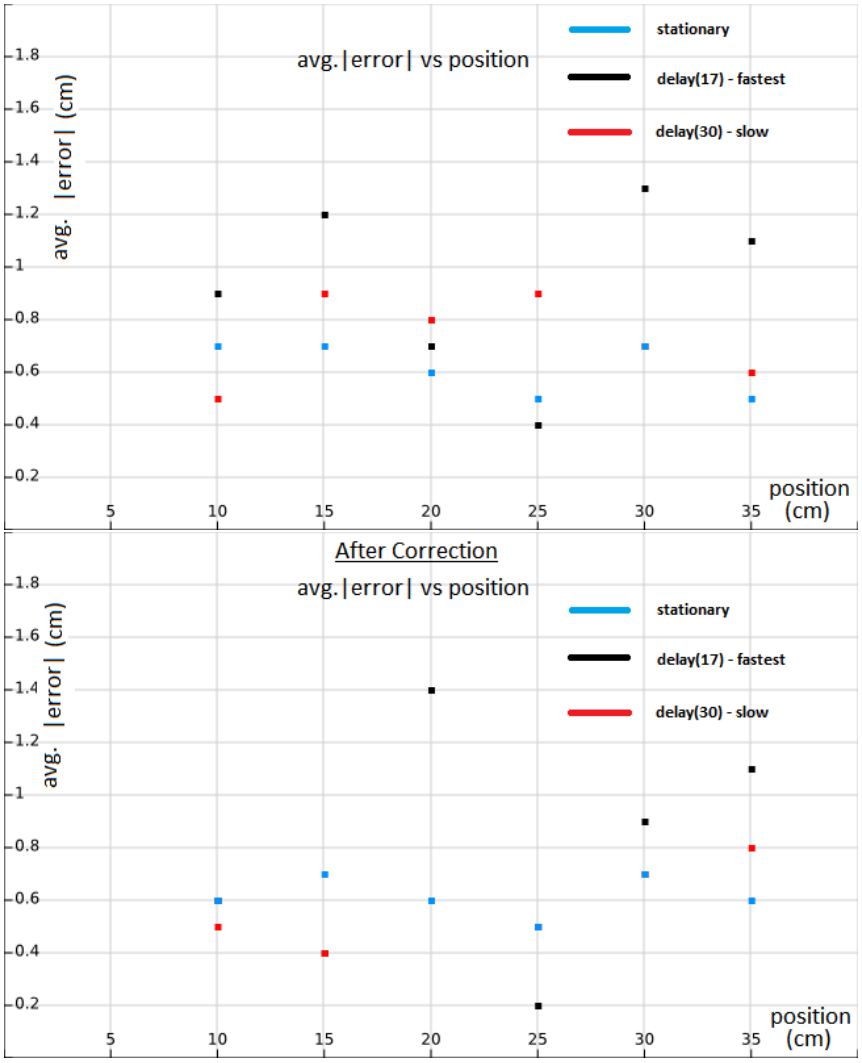

Figure 5.7: Object1-avg.|error| vs position, before vs. after correction

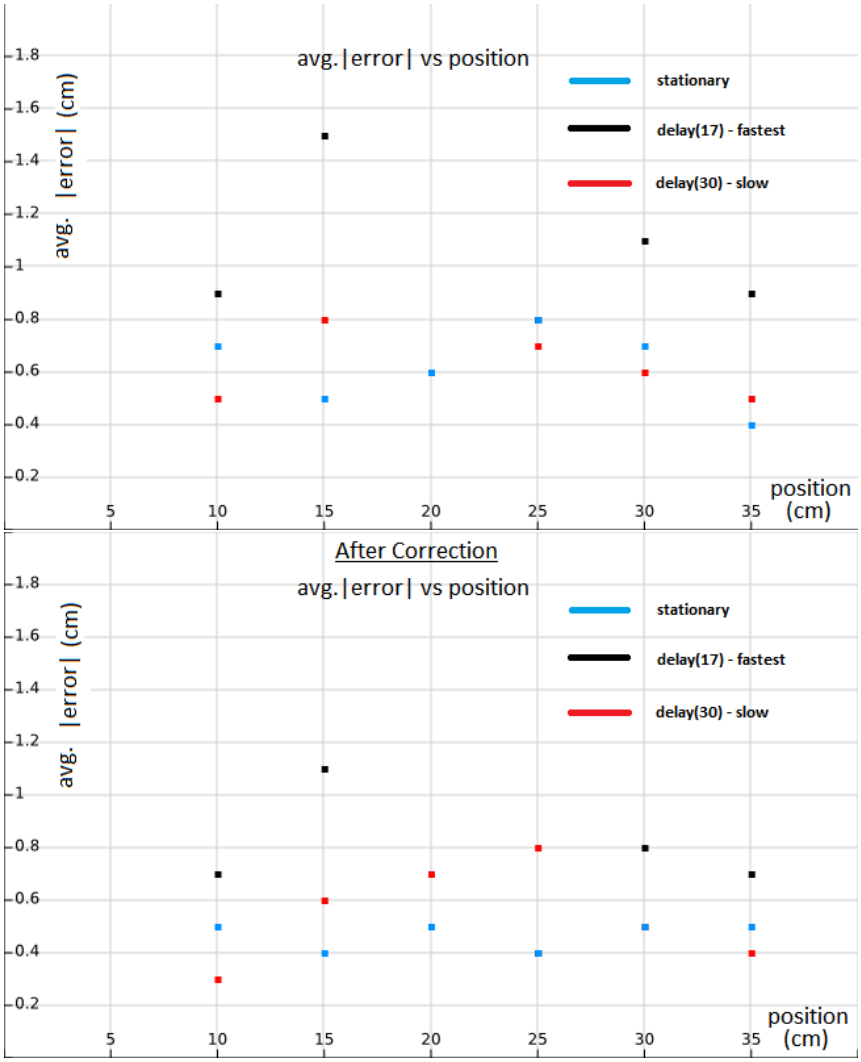

Figure 5.8: Object2-avg.|error| vs position, before vs. after correction

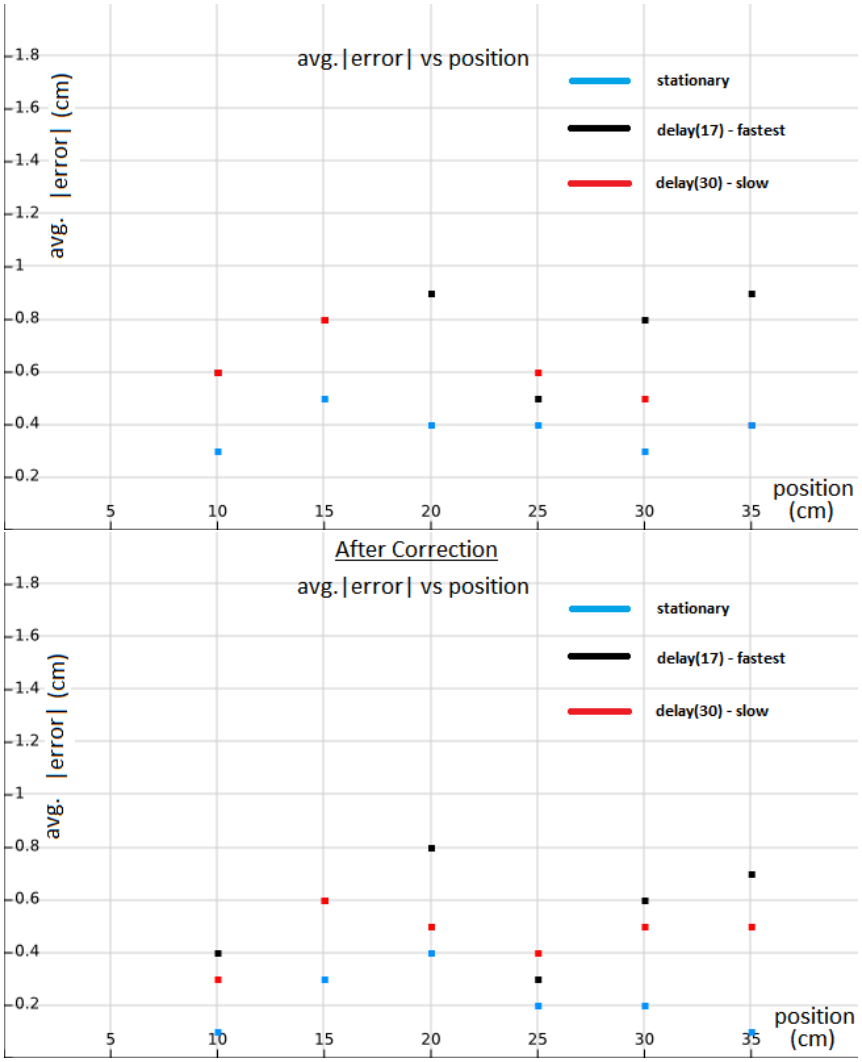

Figure 5.9: Object3-avg.|error| vs position, before vs. after correction

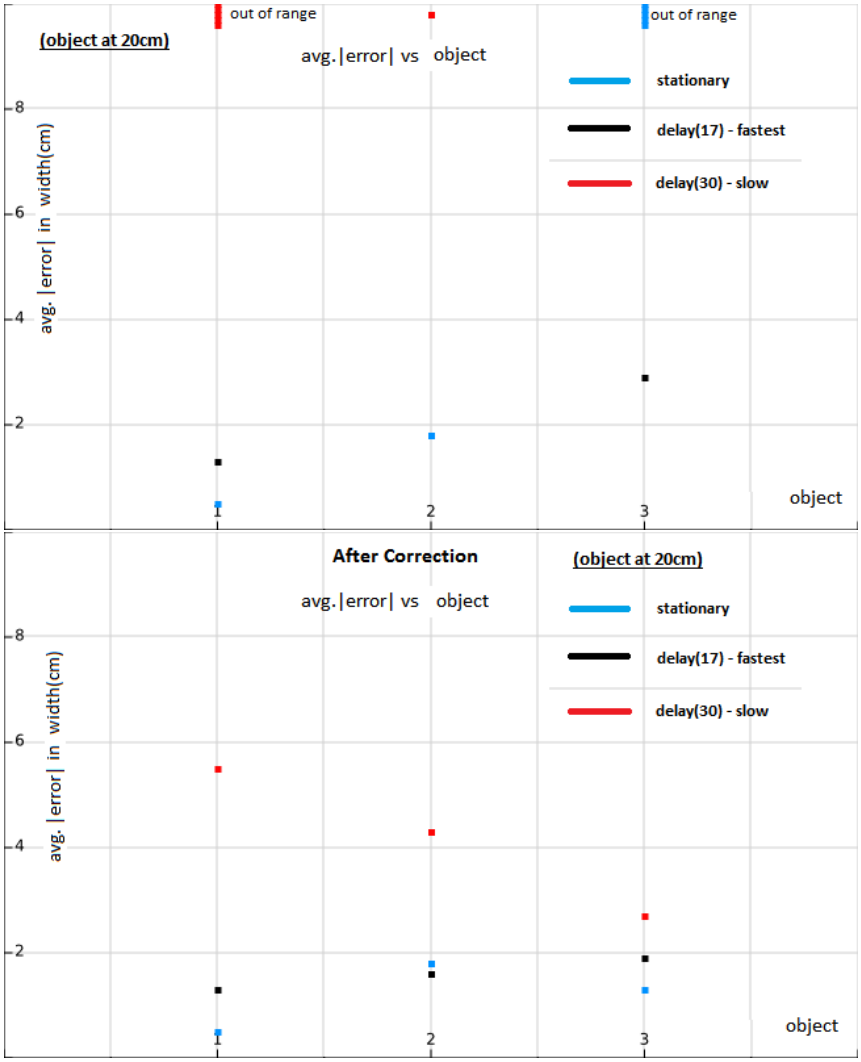

Figure 5.10: Object-wise:avg.|error| in width at x=20cm, before vs. after correction

Conclusion

We continue to observe that there are several challenges, some unexplainable, in obtaining a stable and accurate reading using the HC-SR04 ultrasonic sensor. After we curbed some of the error, we tried to understand what more could've possibly gone wrong with the algorithm/circuitry. This lead us to raise questions at the depth of its internal working, which was a dead end, as we did not have any understanding of the complicated circuitry in it.

We conclude by stating that the *HC-SR04 Ultrasonic Sensor* is a good device for detecting objects, but measurements taken from it can at best be suggestive, but not conclusive of the data about present objects/obstacle.

We believe that for best results, especially in a robust application, this device must be used for detecting "presence" of objects and so far as to get a fair idea, the data that it provides. But, to measure exact distances of objects lying in front of a sensor, especially if in motion, we must use a different type of sensor, possibly a camera, and even a depth sensor like devices running on Augmented-Reality incorporate.

Acknowledgements

We are deeply grateful to our teacher, mentor and team leader; Dr. Deepak Nair, who held us through the course of the semester and made us learn things in a very systematic way. This in itself was solely responsible for teaching us with deep practice, new skills and methods which could have been easily missed out, otherwise.

Thanks to sir's guidance, not a single portion of the project, however rudimentary it could've sounded, was devoid of new knowledge and appropriate professional methodology.

We'd like to thank our institute, the LNMIIT Jaipur, for having granted us the resources,freedom and opportunity to embark on a project of our choice, which has taught us so much.

The Team

Figure B.1: The Team : From Left to Right - Sumit, Nikhil and Surya

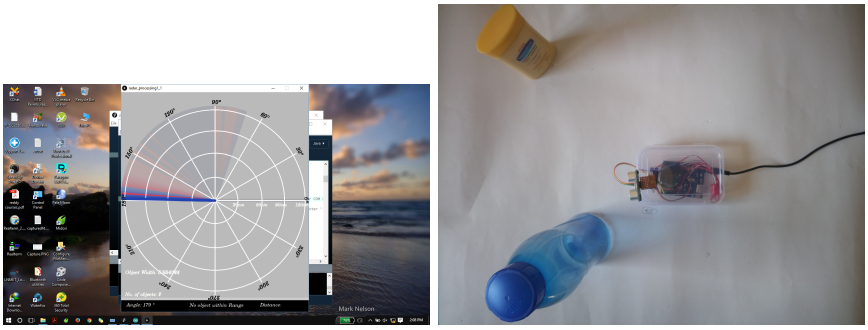

Figure B.2: The project

Appendix A : *Arduino* Code

Here is the code uploaded on to the *Arduino UNO*:

Listing C.1: The Arduino Code

```
/*ver 1.1 (Final)*/

// Includes the Servo library
#include <Servo.h>.

// Defines Trig and Echo pins of the Ultrasonic Sensor
const int trigPin = 2;
const int echoPin = 4;

// Variables for the duration & distance measured from the sensor
float duration;
float distance;

//Angle offset between graph and motor's XY axes
int offset = 0;
//Initial and final angles of motor's rotation
//Note that the motor can only take values 0-180.
//Make sure init_angle-offset >=60 to avoid jumper wire
    obstruction.
int init_ang = 70+offset;
int fin_ang = 180+offset;

// Declares a structure variable/object of the type Servo,
//(defined in the servo library) for controlling the servo motor.
Servo myServo;

void setup() {
pinMode(trigPin, OUTPUT); // Sets the trigPin as an Output
pinMode(echoPin, INPUT); // Sets the echoPin as an Input
Serial.begin(9600); //Sets baud rate of serial communication
myServo.attach(9); // Defines on which pin is the servo motor
    attached
```

```
}

void loop() {
// if (Serial.available() > 0)  /*piece of code to be included in
    case of serial communication with IoT board*/
// {
// rotates the servo motor from init_ang to fin_ang degrees
for(int i=init_ang;i<=fin_ang;i++)
{
myServo.write(i-offset);  //angle value to be passed to the servo
    library object for writing into the motor
delay(17);//DELAY #1:for time taken in motor rotation for one
    degree before calculating distance
distance = calculateDistance();// Calls a function for calculating
     the distance measured by the Ultrasonic sensor for each
    degree
Serial.print(i); // Sends the current degree into the Serial Port
    for graphical representation
Serial.print(","); // Sends addition character right next to the
    previous value needed later in the Processing IDE for indexing
Serial.print(distance); // Sends the distance value into the
    Serial Port for the graph
Serial.print(";"); // Sends addition character right next to the
    previous value needed later in the Processing IDE for indexing
}
// Repeats the previous lines from fin_ang to init_ang degrees
for(int i=fin_ang;i>init_ang;i--)
{
myServo.write(i);
delay(17);  //DELAY #1 //can be minimised to 17 or 1667 microsec
distance = calculateDistance();
Serial.print(i);
Serial.print(",");
Serial.print(distance);
Serial.print(";");
}
//}
}
// Function for calculating the distance measured by the
    Ultrasonic sensor
float calculateDistance(){
unsigned long T1 = micros();
digitalWrite(trigPin, LOW); // trigPin needs a fresh LOW pulse
    before sending a HIGH pulse that can be detected from echoPin
delayMicroseconds(2);//DELAY #2:time for which low trig pulse is
    maintained before making it high
digitalWrite(trigPin, HIGH);
delayMicroseconds(10);//DELAY #3:Sets the trigPin on HIGH state
    for 10 micro seconds
digitalWrite(trigPin, LOW);
```

```
duration = pulseIn(echoPin, HIGH); // Reads the echoPin, returns
    the sound wave travel time in microseconds
//distance= duration*0.034/2;
distance = (duration/2)/29.1;     //in cm,  datasheet gives "
    duration/58" as the formula

//To avoid sending data at variable time intervals due to varying
    time duration taken between execution of above code inside
    this function depending on distance of obstacle
//if no object, echo pulse is HIGH after 38ms
while(micros()-T1<38000)
{
;
}

return distance;
}
```

Appendix B : *Processing* Code

The *Java* code that we ran in *Processing* is given below :

Listing D.1: The Processing Code

```
/*   Arduino Radar Project
*    1.1 [Final and complete]
*   Initialise every variable to null value to avoid null pointer
    exception
*/

import processing.serial.*; // imports library for serial
    communication
import java.awt.event.KeyEvent; // imports library for reading the
     data from the serial port
import java.io.IOException;

Serial myPort; // defines Object Serial
// defines variables
String angle="";
String distance="";
String data="";
String noObject="";
float pixsDistance=0.0;
int iAngle=0;
float iDistance=0.0;
int index1=0;
int index2=0;
int objCount=0,obctr=0;
PFont orcFont;
float angFlag=1.0;
float prevAng=0.0,deltaAng=0.0;
float maxDIST=50.0; //max distance of object in cms
int wctr=0;
float objWidth=0.0,prevWidth=0.0;
void setup() {
```

```
size (600, 700); // ***CHANGE THIS TO YOUR SCREEN RESOLUTION***
smooth();

myPort = new Serial(this,"/dev/ttyACM0", 9600); // Enter the COM
    Port address as COM4 or COM 22.starts the serial communication

myPort.bufferUntil('.'); // reads the data from the serial port up
     to the character '.'. So actually it reads this: angle,
    distance.
orcFont = loadFont("CenturySchL-Ital-20.vlw");

}

void draw() {

fill(237,13,245);
textFont(orcFont);
// simulating motion blur and slow fade of the moving line
noStroke();
fill(184,13);
rect(0, 0, width, height-height*0.065);

fill(211,27,228); //  color
// calls the functions for drawing the radar
drawRadar();
drawLine();
drawObject();
drawText();
}

void serialEvent (Serial myPort) { // starts reading data from the
     Serial Port
// reads the data from the Serial Port up to the character '.' and
     puts it into the String variable "data".
try{
data = myPort.readStringUntil(';');
data = data.substring(0,data.length()-1);

index1 = data.indexOf(","); // find the character ',' and puts it
    into the variable "index1"
angle= data.substring(0, index1); // read the data from position
    "0" to position of the variable index1 or thats the value of
    the angle the Arduino Board sent into the Serial Port
distance= data.substring(index1+1, data.length()); // read the
    data from position "index1" to the end of the data pr thats
    the value of the distance

// converts the String variables into Integer
iAngle = int(angle);
iDistance = float(distance);
```

```
deltaAng=iAngle-prevAng;

if(deltaAng*angFlag<0.0){
objCount=0;
objWidth=0;
}
//anticlockwise--deltaAng>0
//clockwise--deltaAng<0
if(deltaAng>0.0)
angFlag=1.0;
else
angFlag=-1.0;
}
catch(Exception e){
println("Error parsing:");
e.printStackTrace();
}
}

void drawRadar() {
pushMatrix();
translate(width/2,height-height*0.507); // moves the starting
    coordinates to new location
noFill();
strokeWeight(2);
stroke(407,409,305);
// draws the arc lines
arc(0,0,(width-width*0.0385),(width-width*0.0385),PI,TWO_PI);
arc(0,0,(width-width*0.26),(width-width*0.26),PI,TWO_PI);
arc(0,0,(width-width*0.498),(width-width*0.498),PI,TWO_PI);
arc(0,0,(width-width*0.754),(width-width*0.754),PI,TWO_PI);
arc(0,0,(width-width*0.0385),(width-width*0.0385),0,PI);
arc(0,0,(width-width*0.26),(width-width*0.26),0,PI);
arc(0,0,(width-width*0.498),(width-width*0.498),0,PI);
arc(0,0,(width-width*0.754),(width-width*0.754),0,PI);
// draws the angle lines
line(-width/2,0,width/2,0);
line(0,0,(-width/2)*cos(radians(30)),(-width/2)*sin(radians(30)));
line(0,0,(-width/2)*cos(radians(60)),(-width/2)*sin(radians(60)));
line(0,0,(-width/2)*cos(radians(90)),(-width/2)*sin(radians(90)));
line(0,0,(-width/2)*cos(radians(120)),(-width/2)*sin(radians(120))
    );
line(0,0,(-width/2)*cos(radians(150)),(-width/2)*sin(radians(150))
    );
line(0,0,(-width/2)*cos(radians(180)),(-width/2)*sin(radians(180))
    );
line(0,0,(-width/2)*cos(radians(210)),(-width/2)*sin(radians(210))
    );
line(0,0,(-width/2)*cos(radians(240)),(-width/2)*sin(radians(240))
    );
```

```
line(0,0,(-width/2)*cos(radians(270)),(-width/2)*sin(radians(270))
    );
line(0,0,(-width/2)*cos(radians(300)),(-width/2)*sin(radians(300))
    );
line(0,0,(-width/2)*cos(radians(330)),(-width/2)*sin(radians(330))
    );
line((-width/2)*cos(radians(30)),0,width/2,0);
popMatrix();
}

void drawLine() {
pushMatrix();
strokeWeight(8);
stroke(32,65,174); //color for blue line
translate(width/2,height-height*0.507); // moves the starting
    coordinates to new location
line(0,0,(height-height*0.575)*cos(radians(iAngle)),-(height-
    height*0.575)*sin(radians(iAngle))); // draws the line
    according to the angle
popMatrix();
}

void drawObject() {
pushMatrix();
translate(width/2,height-height*0.508); // moves the starting
    coordinats to new location
strokeWeight(8);
stroke(240,1,40); // red color
pixsDistance = iDistance*((height-height*0.1666)*0.013/3); //
    covers the distance from the sensor from cm to pixels
if(iDistance<=maxDIST){
if(wctr>2){    //Assuming that an object will be thick enough to
    be detected for 2 degrees of rotation.
objWidth=0.0;
// draws the object according to the angle and the distance
line(pixsDistance*cos(radians(iAngle)),-pixsDistance*sin(radians(
    iAngle)),(width-width*0.505)*cos(radians(iAngle)),-(width-
    width*0.505)*sin(radians(iAngle)));
obctr++;
}
}
popMatrix();
}

void drawText() { // draws the texts on the screen
pushMatrix();
if(iDistance>maxDIST) {
noObject = "No object within Range";
if(wctr==0)
objWidth=prevWidth;
```

```
else
objWidth=float(wctr)*iDistance*0.0174;  //width of object in cm
prevWidth=objWidth;
wctr=0;
if(obctr>0){
objCount++;
obctr=0;
}
}
else {
wctr++;
if(wctr>2)  //Assuming that an object will be thick enough to be
    detected for 2 degrees of rotation.
{
noObject = "Object in Range";

}
}

fill(0,0,0);  //black background of bottom text
noStroke();
rect(0, height-height*0.0521, width, height);
fill(251,255,249);
textSize(15);

text("30cm",width-width*0.4041,height-height*0.4793);
text("60cm",width-width*0.281,height-height*0.4792);
text("90cm",width-width*0.177,height-height*0.4792);
text("120cm",width-width*0.0729,height-height*0.4792);
textSize(16);
text(noObject, width-width*0.634, height-height*0.0218);
text("Angle: " + iAngle +" °", width-width*0.97, height-height
    *0.0232);
text("Distance: ", width-width*0.26, height-height*0.0235);
textSize(16);
text(" No. of objects: "+ objCount +"", width-width*0.986, height-
    height*0.0714);
text(" Object Width: "+ objWidth +"", width-width*0.986, height-
    height*0.1714);
if(iDistance<=maxDIST) {
if(wctr>2)
text("        " + iDistance + " cm", width-width*0.185, height-
    height*0.0237);
}
textSize(19);
fill(7,7,6); //color for degrees text
translate((width-width*0.5020)+width/2*cos(radians(30)),(height-
    height*0.5283)-width/2*sin(radians(30)));
rotate(-radians(-60));
text("30°",0,0);
```

```
resetMatrix();
translate((width-width*0.507)+width/2*cos(radians(60)),(height-
    height*0.5139)-width/2*sin(radians(60)));
rotate(-radians(-29));
text("60°",-4,-5);
resetMatrix();
translate((width-width*0.507)+width/2*cos(radians(90)),(height-
    height*0.5149)-width/2*sin(radians(90)));
rotate(radians(0));
text("90\degree",-5,-2);
resetMatrix();
translate(width-width*0.513+width/2*cos(radians(120)),(height-
    height*0.51286)-width/2*sin(radians(120)));
rotate(radians(-30));
text("120°",0,0);
resetMatrix();
translate((width-width*0.5298)+width/2*cos(radians(150)),(height-
    height*0.4803)-width/2*sin(radians(150)));
rotate(radians(-60));
text("150°",0,0);
resetMatrix();
translate(width-width*0.475+width/2*cos(radians(180)),(height-
    height*0.47791)-width/2*sin(radians(180)));
rotate(radians(-90));
text("180°",0,0);
resetMatrix();
translate(width-width*0.494+width/2*cos(radians(210)),(height-
    height*0.4797)-width/2*sin(radians(210)));
rotate(radians(-120));
text("210°",0,0);
resetMatrix();
translate(width-width*0.484+width/2*cos(radians(240)),(height-
    height*0.4867)-width/2*sin(radians(240)));
rotate(radians(-150));
text("240°",0,0);
resetMatrix();
translate(width-width*0.474+width/2*cos(radians(270)),(height-
    height*0.50151)-width/2*sin(radians(270)));
rotate(radians(-180));
text("270°",0,0);
resetMatrix();
translate(width-width*0.470+width/2*cos(radians(300)),(height-
    height*0.51167)-width/2*sin(radians(300)));
rotate(radians(-210));
text("300°",0,0);
resetMatrix();
translate(width-width*0.478+width/2*cos(radians(330)),(height-
    height*0.5174)-width/2*sin(radians(330)));
rotate(radians(-240));
text("330°",0,0);
```

```
resetMatrix();
translate(width-width*0.523+width/2*cos(radians(360)),(height-
    height*0.52132)-width/2*sin(radians(360)));
rotate(radians(-270));
text("0°",0,0);
popMatrix();
prevAng = iAngle;
}
```