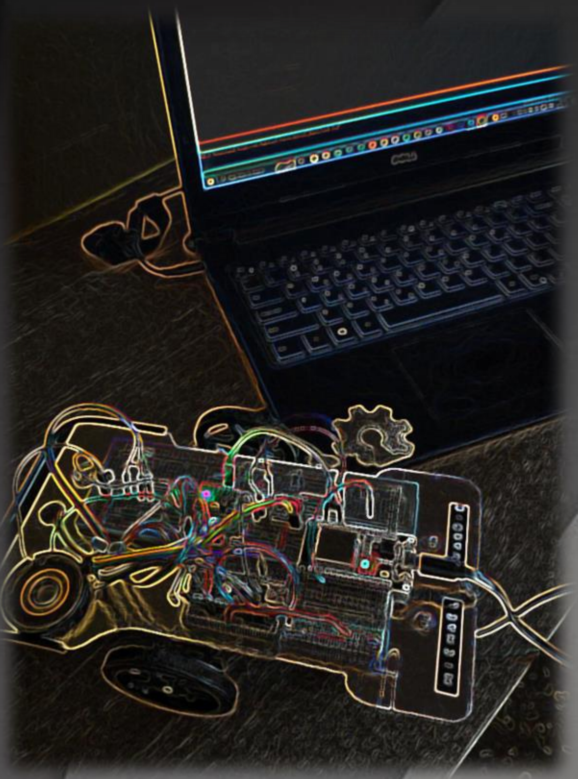


Internet of Things 2023

SEMESTER PROJECT

Instructor: Dr. Spyros Panagiotakis

Vardis Daskalakis - Michael Xirakis

Graduate Students of “Informatics Engineering”
Hellenic Mediterranean University
Department of Electrical & Computer Engineering
Heraklion, Greece 71410
Email: mth295@edu.hmu.gr - mth290@edu.hmu.gr

Table of Contents

Project’s Report

The purpose of this project is educational and has to do with applied IoT (Internet of Things) technologies. Specifically based on microcontrollers like Arduino and Esp32 we try to create our "smart" gadgets, using different types of sensors, make some measurements, send the data to the internet and control them remotely. As we will see both microcontrollers are required as they have different capabilities. Our topic of choice is the creation of a smart car, which has some sensors that used to control it make some measurements and see the data into the internet.

DIY Capacitance Sensor

First we have the DIY capacitance sensor connected to Arduino Nano. The idea behind it is to use it as an alarm system for our car and also to study and experiment with the theory of electronics required to build it. Using the official Arduino capacitance library [1] we can create the basic body of the sensor and also see some details about it.

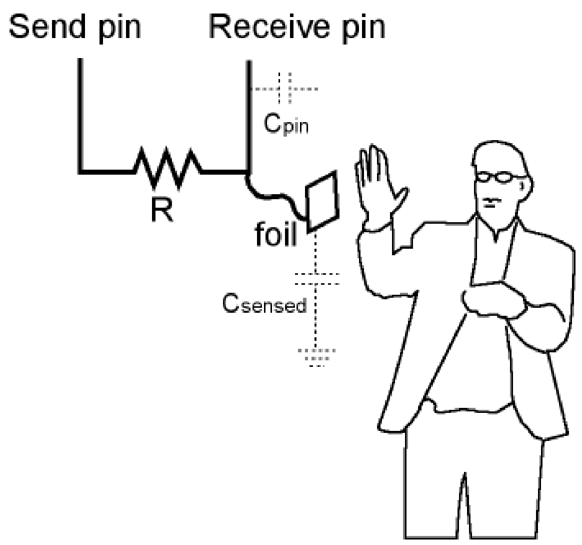


Figure 1. Basic form of capacitance sensor [1].

Starting with the basic form and analog reading of the sensor values, we face our first and biggest problem, which is noise. According to the diagram there are two pins. The send pin sends 5V DC through a 1.5 megOhm resistor. The resistor is necessary as it makes the whole circuit sensitive to electromagnetic interferences and thus works. Then, to make the sensor even more sensitive, there is an aluminum foil attached to it. After this the distorted signal is sent to the receive pin. The issue is that the more resistance we use (according to the Arduino documentation), to make the sensor work over longer distances, the more vulnerable it becomes to any ambient interference, providing noisy data. To overcome it to some extent (but not completely) we first try to shield it physically. The first thing we do is connect a 10 pFarad ceramic capacitor between the foil and the receiving pin. But because our capacitor is ceramic (meaning no polarity) we also use a 0.5W 12V Zener diode connected in parallel with the capacitor, we make the capacitor this way only discharge at the sensor pin. We also created a twisted wire by taking three thinner wires and twisting them around each other, making the sensor wire more shielded against interference since the electromagnetic fields around them tend to cancel out. An even better solution could be a coaxial cable, but it is too wide for our project. The result is less noise in our measurements than with a conventional cable. All these helps us partially solve the problem as it makes our sensor much more stable, but still not perfect. We also use a software approximation solution, using a moving average filter [2] to recover more stable values. Luckily there is the Moving Average library we can use, without need to implement the whole process from scratch. At this point after all this process, our sensor is working perfectly and providing a very smooth signal. The idea behind this is to read the distance between our hand and the car and also play a sine wave at some corresponding frequency, in our case we define a range from 800Hz to 1.5kHz, with the speaker we have connected

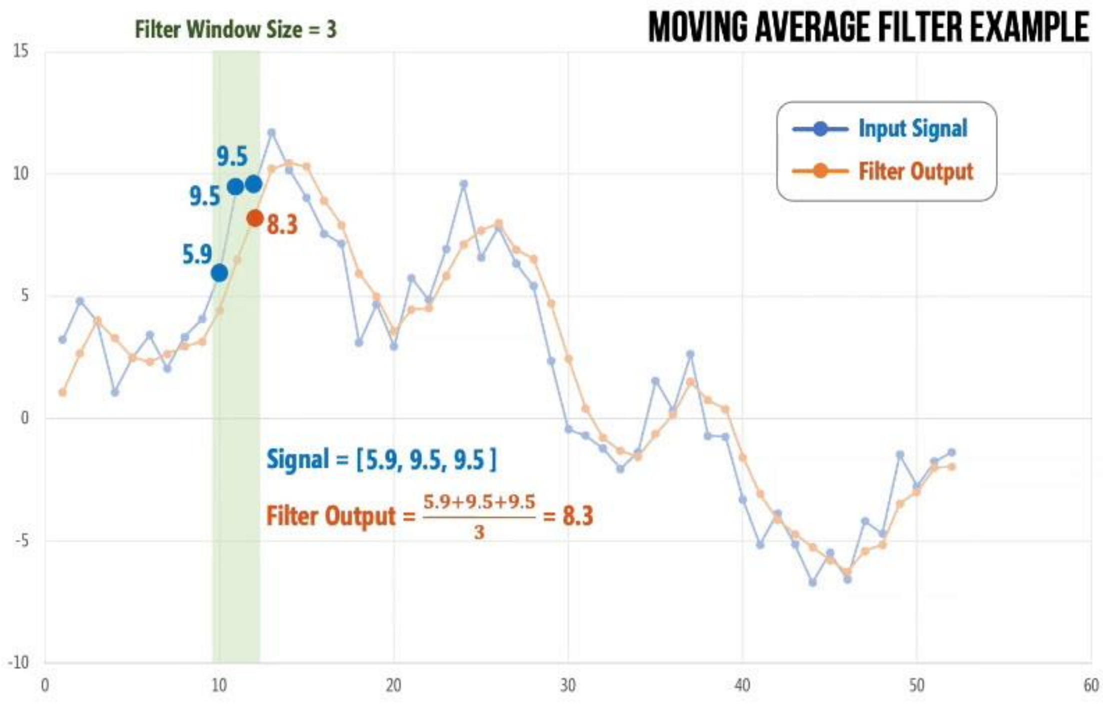
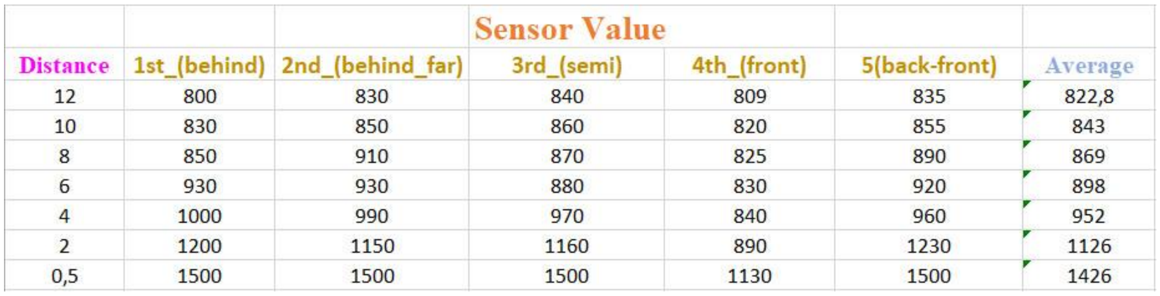


Figure 2. The moving average filter smooths our signal by averaging over a number of samples (window), removing any peaks caused by noise. [2]

to the car, simulating in this way a car alarm system (unfortunately after adding the rest of the electronic components like the speaker in the car we have interferences again and the sensor is not completely stable). To map the sensor values to the corresponding distance, what we do is take some measurements and import them into Microsoft Excel also providing the corresponding distance and try to find the best mapping function. After that we implement this function in the Arduino code and in this way we get an approximation of the distance between the car and our hand. For the acoustic part of the alarm system, things are simpler. We only apply a linear transformation (mapping) between the sensor values and the frequency range. Putting all this together as we can see in figures **3** and **4**.



			Sensor Value			
Distance	1st_(behind)	2nd_(behind_far)	3rd_(semi)	4th_(front)	5(back-front)	Average
12	800	830	840	809	835	822,8
10	830	850	860	820	855	843
8	850	910	870	825	890	869
6	930	930	880	830	920	898
4	1000	990	970	840	960	952
2	1200	1150	1160	890	1230	1126
0,5	1500	1500	1500	1130	1500	1426

Figure 3. The measurements for different states and the values we get.

As we can see we make measurements for different parts of our car (as it is whole covered with aluminum tape) and for different stages we get different values. The purpose is to find the relation between the average and the estimated distance.

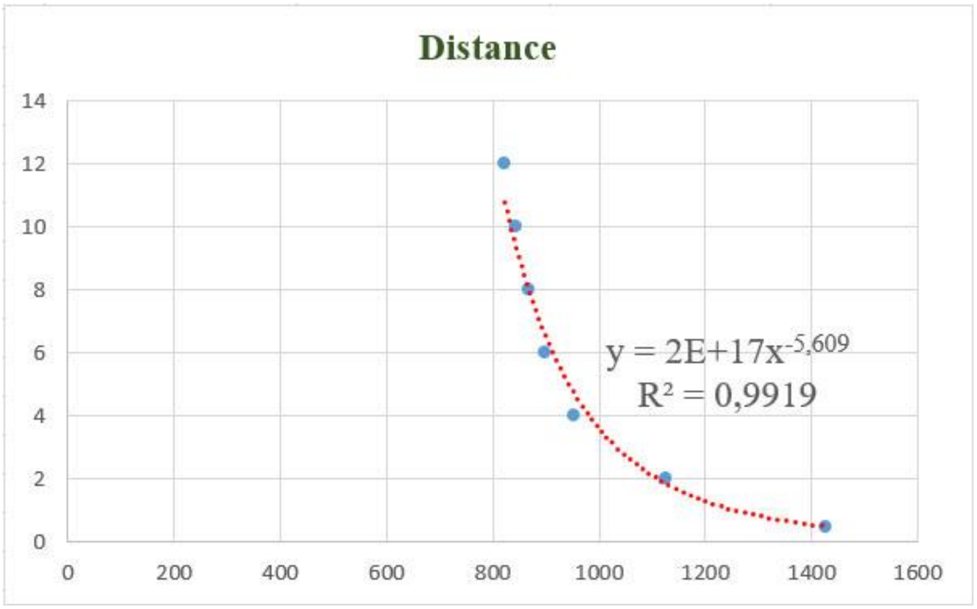
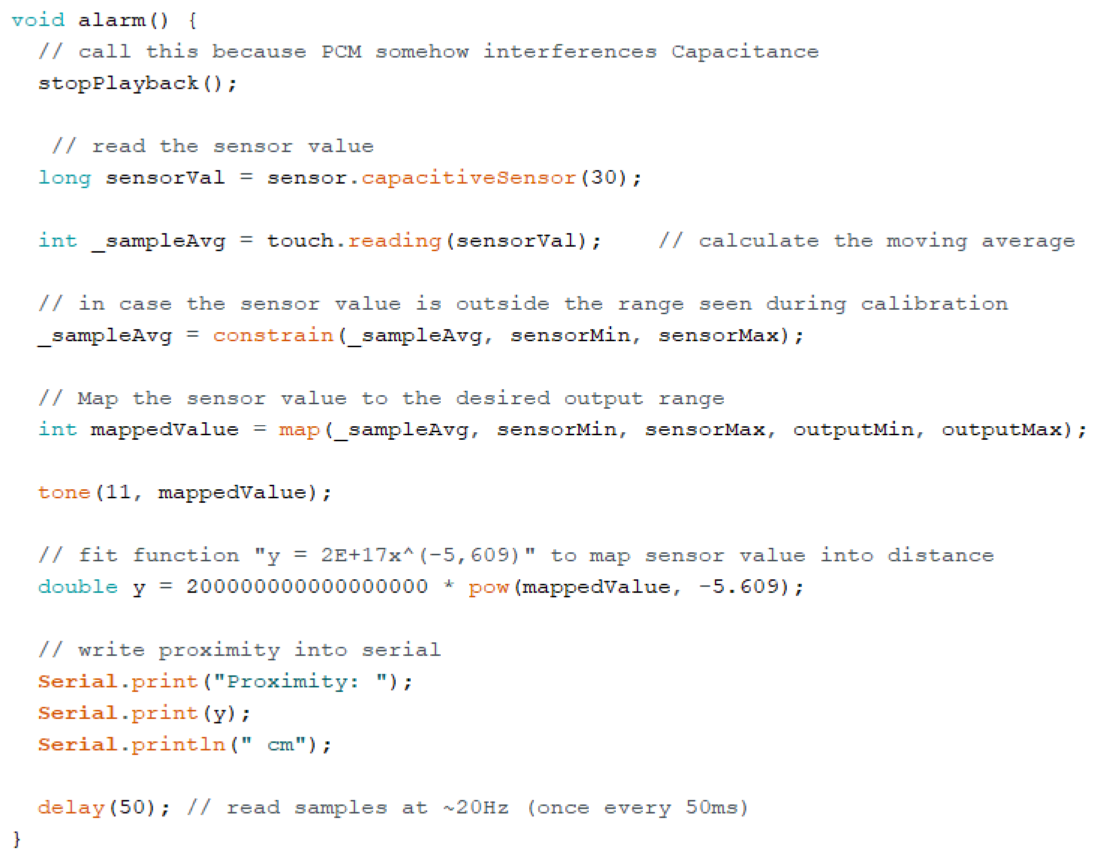


Figure 4. The relationship y=f(x) as a power function between the y-axis distance and the average of the x-axis measurements and also the R^2 score of the function. For scores closer to 1 we have a better fit to the data. As we can see the relationship is not linear.

In the plot diagram of Figure 4, we can see that the best matching function between the sensor values and the estimated distance is the power function, with a score of almost 1, which is very good. For more information, watch the tutorial below [3].



```
void alarm() {
  // call this because PCM somehow interferences Capacitance
  stopPlayback();

   // read the sensor value
  long sensorVal = sensor.capacitiveSensor(30);

  int _sampleAvg = touch.reading(sensorVal);     // calculate the moving average

  // in case the sensor value is outside the range seen during calibration
  _sampleAvg = constrain(_sampleAvg, sensorMin, sensorMax);

  // Map the sensor value to the desired output range
  int mappedValue = map(_sampleAvg, sensorMin, sensorMax, outputMin, outputMax);

  tone(11, mappedValue);

  // fit function "y = 2E+17x^(-5,609)" to map sensor value into distance
  double y = 200000000000000000 * pow(mappedValue, -5.609);

  // write proximity into serial
  Serial.print("Proximity: ");
  Serial.print(y);
  Serial.println(" cm");

  delay(50); // read samples at ~20Hz (once every 50ms)
}
```

Figure 5. The code implementation of the alarm system.

DIY Light Sensor

Our next car component is a DIY light sensor, created with a photocell and a 1 megohm resistor. Its purpose is to activate a connected led when there is darkness in the environment. The way it works is to first calibrate it, set a sensor min and max value, read some samples for the first five seconds in setup mode. Then setting the threshold as the average value between the min and max value, and if the sensor value is lower than the threshold, turn on the led. To get the minimum value of the sensor during calibration, all we have to do is cover it with our finger.

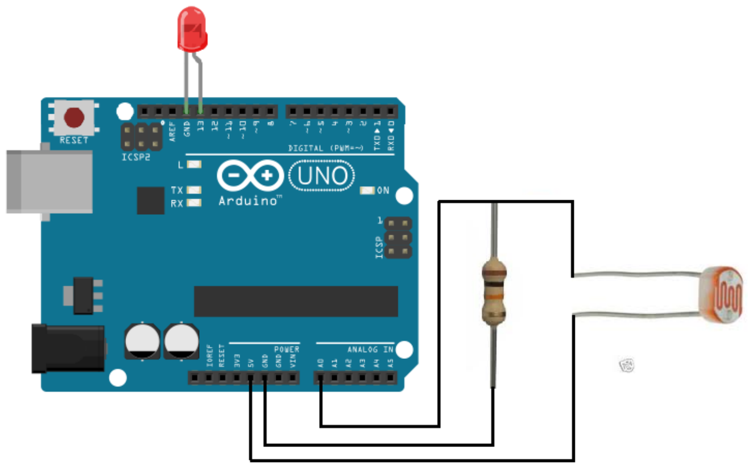


Figure 6. Example photo very similar to our light sensor circuit.

Servos, Gesture control & Speaker

Then of course we have our car wheels. Specifically, the car operates with two SG-90 servos attached to the Nano. First of all, as mentioned earlier, there is a speaker connected to the car to play audio messages and also a sine wave for the alarm system. The speaker works with a PCM (Pulse Code Modulation) library [4], which causes the following problem, both the PCM and Servo libraries use Arduino timer 1, making it impossible to place the two libraries in same project. For this reason we use the ServoTimer2 library [5], to avoid any conflicts with the PCM. A major difference between Servo and ServoTimer2, is that in the former, the motors operate with angles in degrees, while in the latter, they operate with PWM (Pulse Width Modulation) signal values. Here, Esp32 comes to make things interesting. Because our goal is to control the car with gestures, but not have to create a controller using extra components like microcontrollers, accelerometers, etc. we use the Dabble app [6] which reads values from our smartphone sensors and sends them to the Esp32 remotely via bluetooth which is really nice as it makes our lives easier. This application has its own library and depending on the sensors of our phone, it provides us with different features. For the needs of our project, simple accelerometer values are fine, as we want to read the pitch and roll of our smartphone and make the car move accordingly. After that, using the UART (Universal Asynchronous Receiver/Transmitter) protocol, we send the accelerometer values from the Esp32 to the Nano, which we map to a range of PWM signal values, to drive the servos. Communication between the two microcontrollers is via the TX and RX pins of each module. For safety reasons, we don't connect the two boards directly, but through a logic level converter that steps up or down the voltage, since the two boards operate at different voltages (Esp32 at 3.3V and Nano at 5V). The whole system works quite well, although it has some drawbacks. The biggest one is that the data from the master board (Esp32) to the slave (Nano) is not sent continuously. We must set a time period of at least more than 1 second (in our case 1050 milliseconds) delay on the master before sending the data, otherwise there is a communication conflict and the data is not received correctly by the slave. Also, because of this time delay, we don't have analog control of the servos, as expected, but rather digital, if the roll or tilt angle of the smartphone is more than 30 degrees, then the servos kick in at full speed.

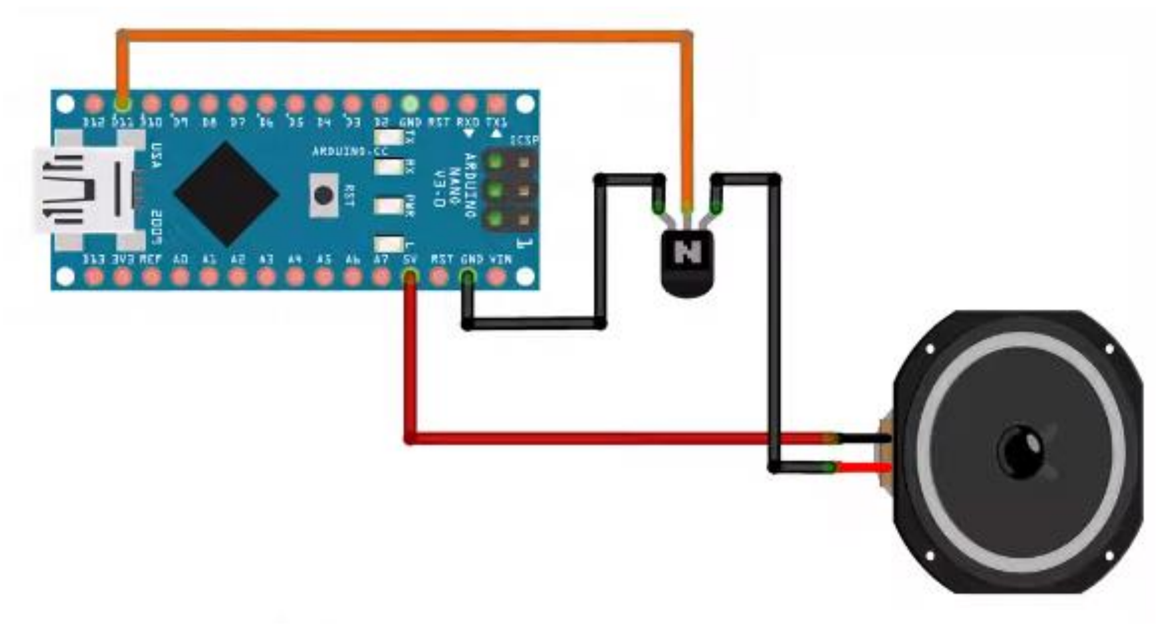
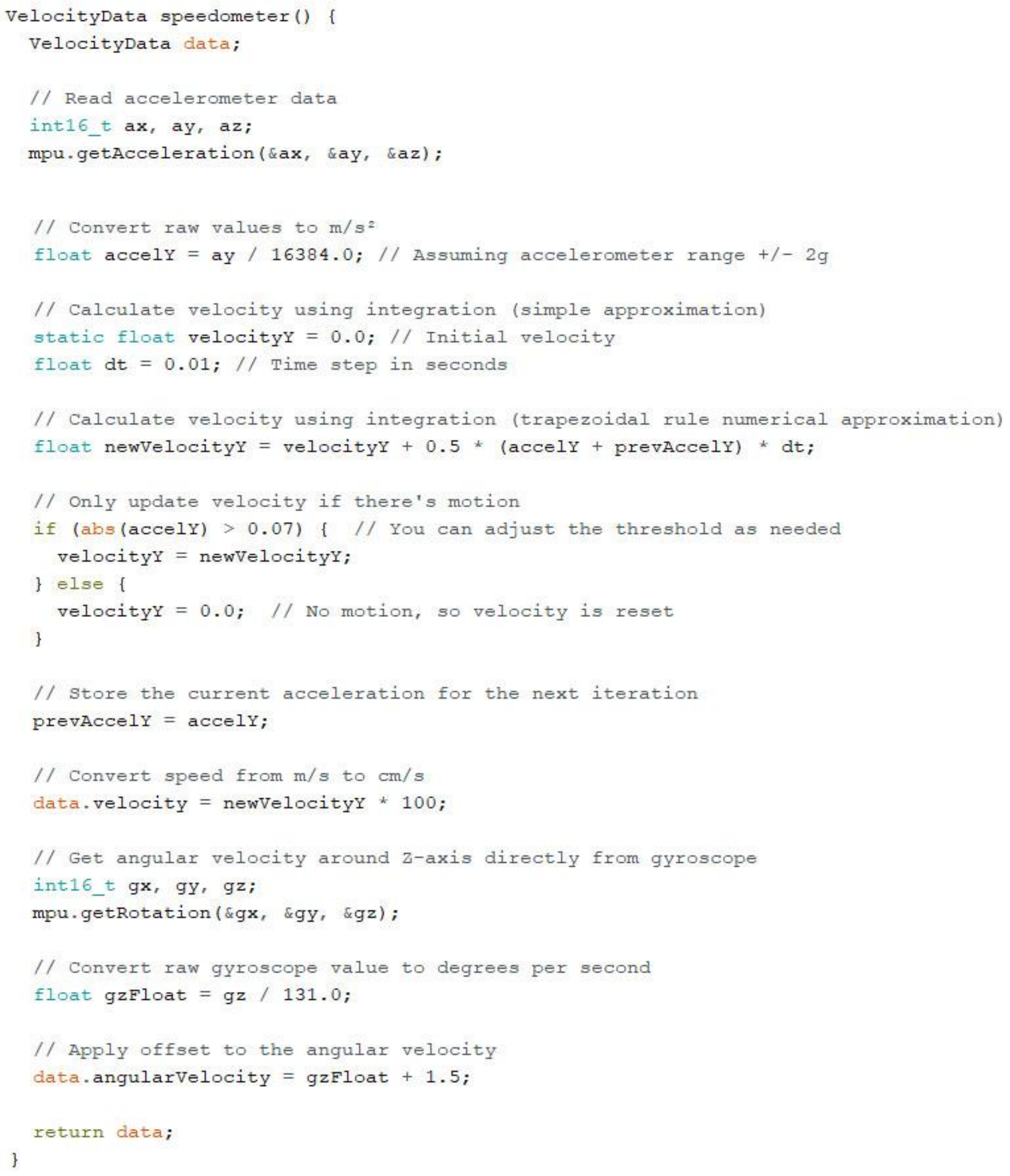

D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 GND RST RX0 TX1
D13 3V3 REF A0 A1 A2 A3 A4 A5 A6 A7 5V RST GND VIN
ARDUINO NANO V3.0
ICSP
N

Figure 7. Our car speaker circuit. As an amplifier we used a bipolar NPN transistor.

Speedometer

Of course driving a car without speedometer is not safe ☺ and this introduces us to the next component. Using the MPU6050 module connected to Esp32, we try to calculate the linear and angular velocity through the accelerometer and gyroscope values we get. It is known from Physics that acceleration is the derivative of velocity. So by taking acceleration and applying the reverse process we can get velocity. Since we are working with discrete data and also do not know the acceleration function, we do integration using a numerical approximation, the trapezoidal rule. Also, to calculate the angular velocity, as the car rotates around its z-axis, we just take the gyro yaw value, which gives us the angular velocity in degrees per second. The result is satisfactory, although the values may not be 100% accurate, due to the delay of the code (10 milliseconds). After that, we send and store the data into InfluxDB and monitoring them with Grafana in order to read and plot them remotely. The sad thing is that we can't combine the code and libraries of both the remote with the Dabble app and the internet connection in one simple project, as the Esp32 gets an error about insufficient memory. So we separately upload each part of the code to the

Esp32 and just wave the car with our hand to read its speed (besides, the project has more of an educational, rather than a functional purpose, as previously mentioned). A possible solution to this could be to use an extra Esp32 to only connect to the internet and send data, as it has more than one serial channel.

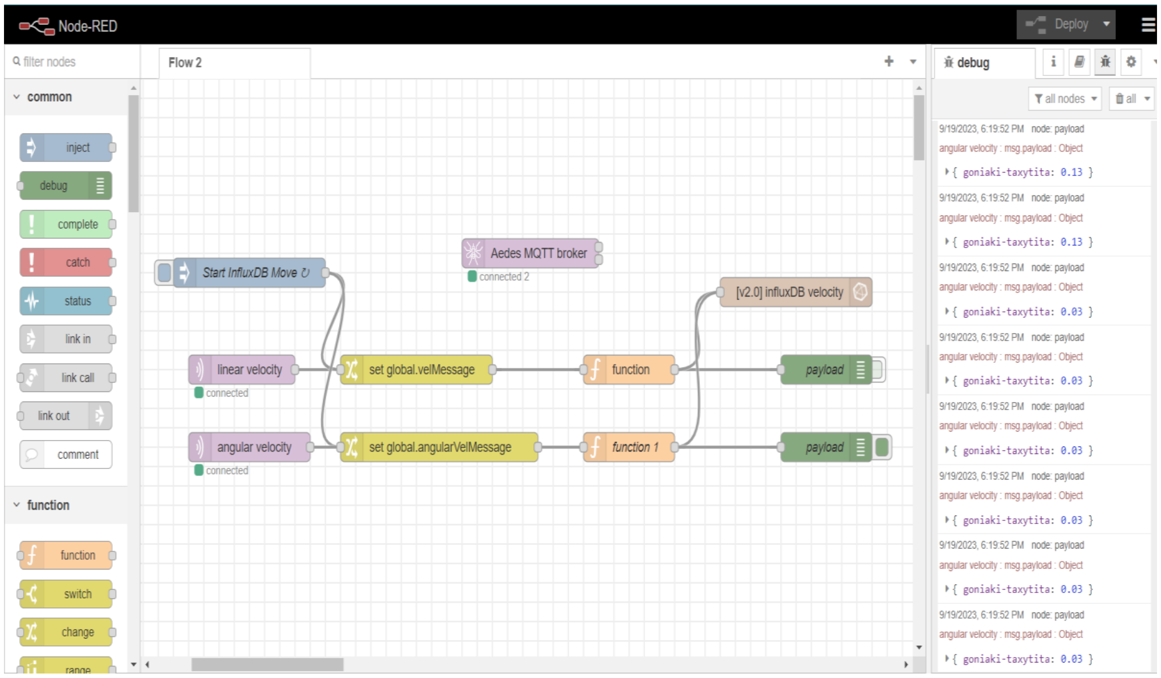


```
VelocityData speedometer() {
  VelocityData data;

  // Read accelerometer data
  int16_t ax, ay, az;
  mpu.getAcceleration(&ax, &ay, &az);


  // Convert raw values to m/s²
  float accelY = ay / 16384.0; // Assuming accelerometer range +/- 2g

  // Calculate velocity using integration (simple approximation)
  static float velocityY = 0.0; // Initial velocity
  float dt = 0.01; // Time step in seconds

  // Calculate velocity using integration (trapezoidal rule numerical approximation)
  float newVelocityY = velocityY + 0.5 * (accelY + prevAccelY) * dt;

  // Only update velocity if there's motion
  if (abs(accelY) > 0.07) {  // You can adjust the threshold as needed
    velocityY = newVelocityY;
  } else {
    velocityY = 0.0;  // No motion, so velocity is reset
  }

  // Store the current acceleration for the next iteration
  prevAccelY = accelY;

  // Convert speed from m/s to cm/s
  data.velocity = newVelocityY * 100;

  // Get angular velocity around Z-axis directly from gyroscope
  int16_t gx, gy, gz;
  mpu.getRotation(&gx, &gy, &gz);

  // Convert raw gyroscope value to degrees per second
  float gzFloat = gz / 131.0;

  // Apply offset to the angular velocity
  data.angularVelocity = gzFloat + 1.5;

  return data;
}
```

Figure 8. Part of the code where linear and angular velocities are calculated.

Node-Red

The Node-RED flow below allows you to establish an MQTT communication link between an ESP32 publishing linear and angular velocity data and a Node-RED instance. The data is then visualized in the debug console for monitoring and debugging purposes.



```
9/19/2023, 6:19:52 PM   node: payload
angular velocity : msg.payload : Object
▸{ goniaki-taxytita: 0.13 }
9/19/2023, 6:19:52 PM   node: payload
angular velocity : msg.payload : Object
▸{ goniaki-taxytita: 0.13 }
9/19/2023, 6:19:52 PM   node: payload
angular velocity : msg.payload : Object
▸{ goniaki-taxytita: 0.03 }
9/19/2023, 6:19:52 PM   node: payload
angular velocity : msg.payload : Object
▸{ goniaki-taxytita: 0.03 }
9/19/2023, 6:19:52 PM   node: payload
angular velocity : msg.payload : Object
▸{ goniaki-taxytita: 0.03 }
9/19/2023, 6:19:52 PM   node: payload
angular velocity : msg.payload : Object
▸{ goniaki-taxytita: 0.03 }
9/19/2023, 6:19:52 PM   node: payload
angular velocity : msg.payload : Object
▸{ goniaki-taxytita: 0.03 }
9/19/2023, 6:19:52 PM   node: payload
angular velocity : msg.payload : Object
▸{ goniaki-taxytita: 0.03 }
```

Figure 9. Node-Red code, which receives linear and angular velocity values from Esp32.

1. **Aedes MQTT Broker**: The flow begins with an Aedes MQTT broker node. Aedes is a popular MQTT broker for Node.js. This node serves as the MQTT broker, and it's configured to listen for incoming MQTT messages.
2. **MQTT Subscribe Nodes (Linear Velocity and Angular Velocity)**: Connected to the MQTT broker, there are two MQTT subscribe nodes. One is subscribed to the MQTT topic where the linear velocity values are published, and the other is subscribed to the MQTT topic for angular velocity values. These nodes are responsible for receiving incoming messages from the ESP32.
3. **Payload Messages:** When the ESP32 publishes linear velocity and angular velocity values to their respective MQTT topics, these values are captured as payload messages by the MQTT subscribe nodes. These messages contain the velocity data in a format that can be processed further.

4. **Debug Console**: To visualize and monitor the incoming velocity data, the flow includes a "Debug" node. The payload messages received from the MQTT topics are routed to this node. The "Debug" node displays the payload messages in the Node-RED debug console, making it easy for you to view and analyze the velocity values in real-time.

InfluxDB

Once the data is stored in InfluxDB in our database, we can use the InfluxDB Data Explorer to visualize the data as tables and graphs. Also we can use queries to build our custom graphs and manage our values.

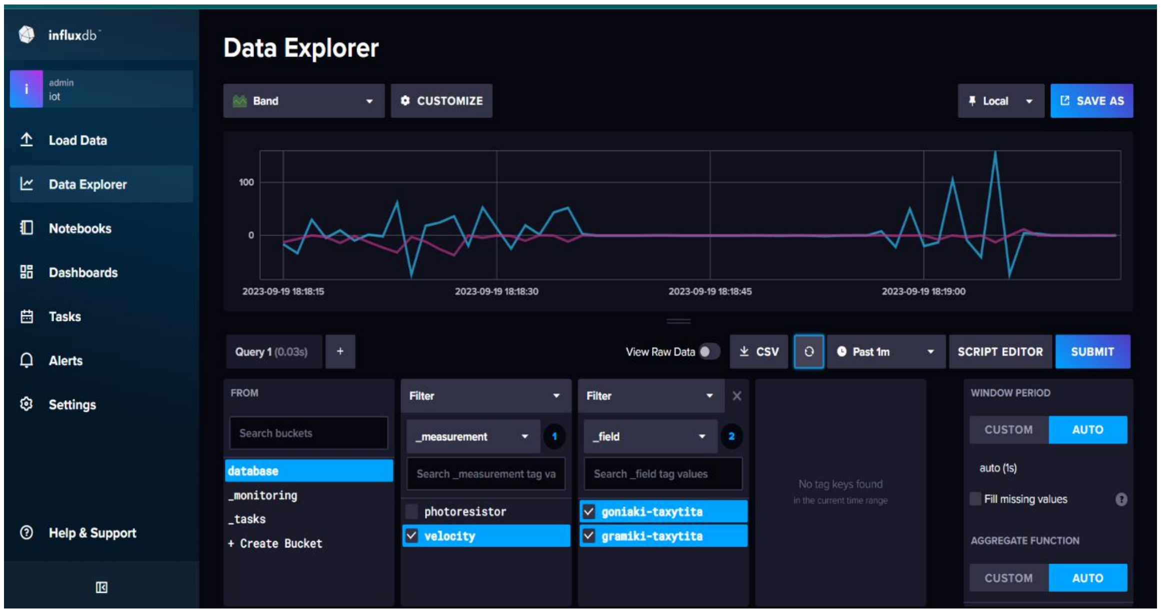


Figure 10. Screenshot of the received data waveform as displayed in InfluxDB.

Grafana

Monitoring our data in Grafana is a valuable addition to IoT and data visualization toolkit. Grafana is a powerful open-source platform for creating, sharing, and exploring dashboards and graphs. To start monitoring our data in Grafana, we need to configure a data source. In our case we used InfluxDB with Grafana because it’s a time-series database that works well for IoT and sensor data.

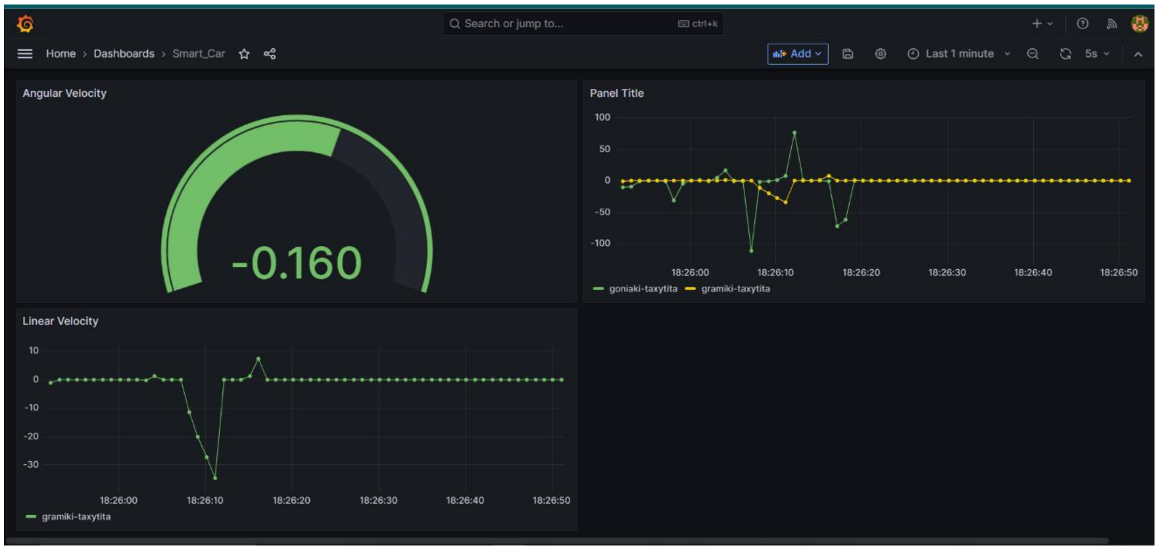
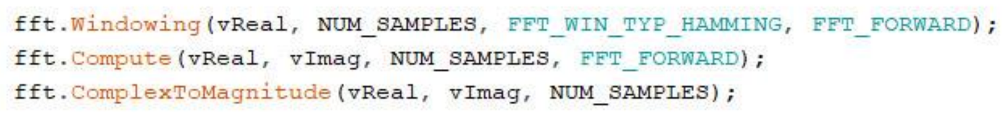


Figure 11. Velocity values as shown in the Grafana Dashboard.

Audio Classification

Finally, we save the best and most interesting part of the project for the end and that is the machine learning. Specifically, by "machine learning" we mean the sound classification we do, using a KY-038 sound sensor, connected to the Esp32. Here we will try to describe the general process by which audio classification can be achieved on the Esp32 or any microcontroller of similar specifications. For more information take a look at [7], [8], [9] and [10]. The idea behind this is for the car to classify the horn sounds of bicycles and nearby cars into two different categories. The way classification works can be summarized in the following steps. First we calibrate the audio sensor to detect the background sound and subtract it from the audio signal. If the audio signal is above a certain threshold, the microphone records 512 samples in a line vector format. After that we perform FFT (Fast Fourier Transform) on the time signal to calculate its magnitude spectrum and recover its frequency component, using the Arduino FFT library [11]. For practical reasons, due to the periodic nature of the FFT, we only keep the first half samples of the magnitude vector, as the remaining halves are the "reflection" (phenomenon known as “folding”) of the first half (256 samples) and also reject the first “fake” bin, as it happens due to the fundamental DC frequency offset, which is not useful. Another detail is that the number of samples must be a power of 2 number for FFT efficiency.



```
fft.Windowing(vReal, NUM_SAMPLES, FFT_WIN_TYP_HAMMING, FFT_FORWARD);
fft.Compute(vReal, vImag, NUM_SAMPLES, FFT_FORWARD);
fft.ComplexToMagnitude(vReal, vImag, NUM_SAMPLES);
```

Figure 12. Part of the Esp32 code, where the FFT is performed. Windowing applies a Hamming envelope to the time signal to reduce spectral leakage. Then the FFT is calculated and finally we retrieve its magnitude spectrum. It might have been more appropriate to run the STFT "Short Term Fourier Transform" for more accurate results, but this is difficult for real-time implementation on a microcontroller.

We then print the feature vector to the serial screen to copy and paste into a CSV (Comma Separated Value – Microsoft Excel) file. For each sound we record 15 samples, creating a dataset of each category (bike-car). Then, using Python machine learning packages, we build and train a classification model and also apply PCA (Principal Component Analysis) to test whether our data is linearly separable.

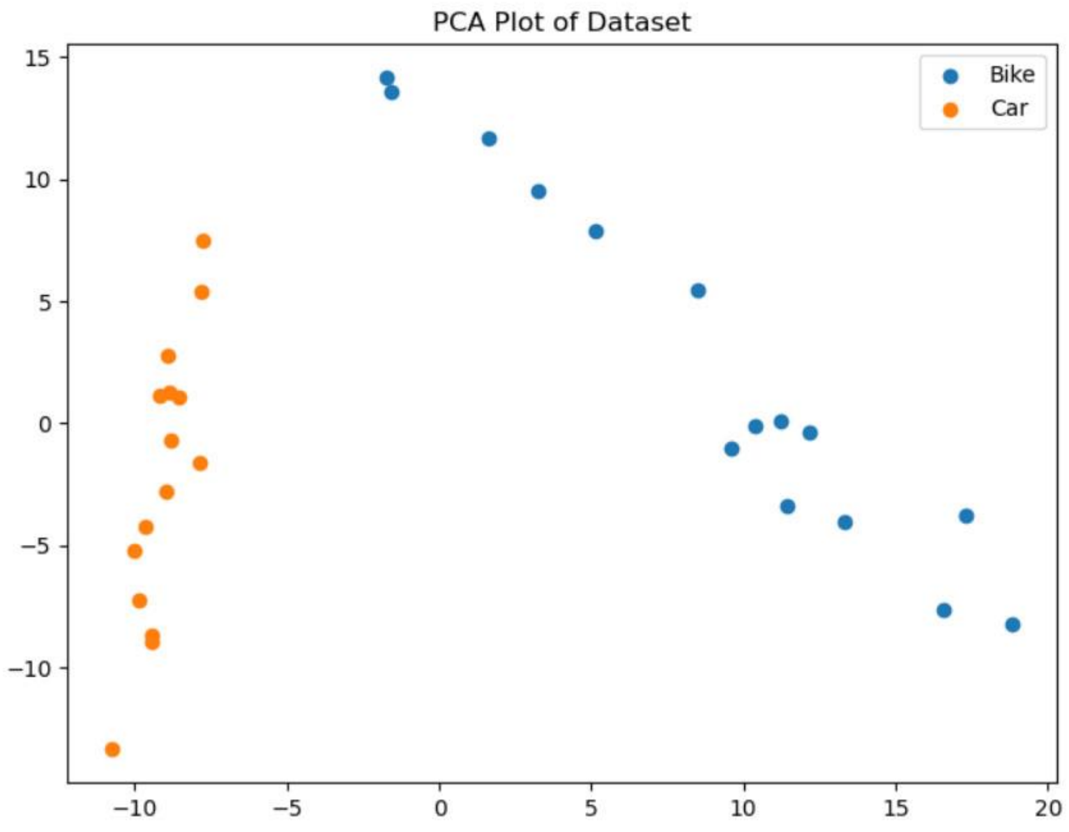


Figure 13. Data distribution on 2d space, after applying PCA.

Also, a suitable pre-processing step, before starting training, is to plot the spectrograms of the two audio samples we are using, to see if they really differ in their frequency component, otherwise they may not be separable.

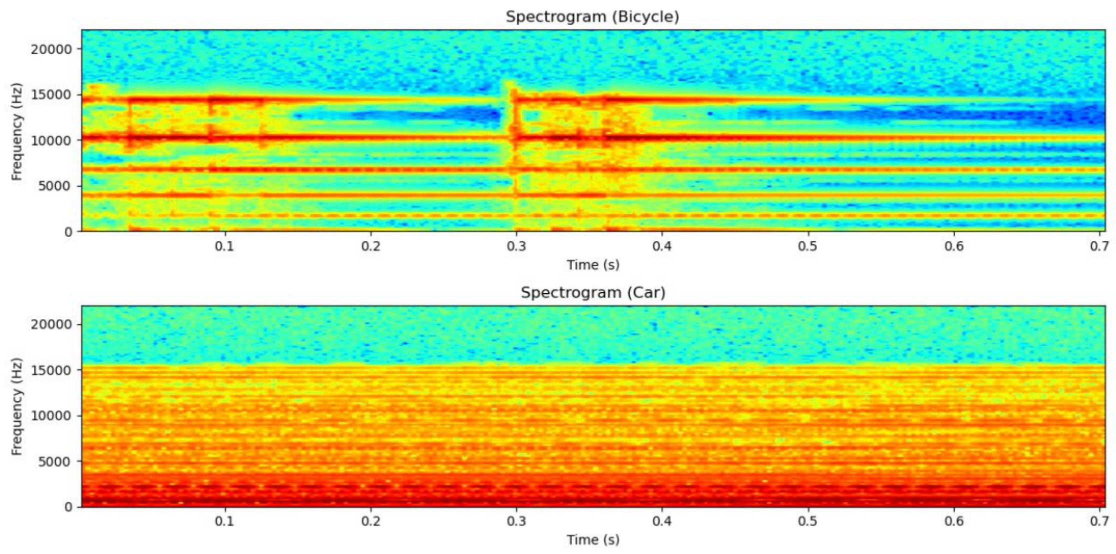
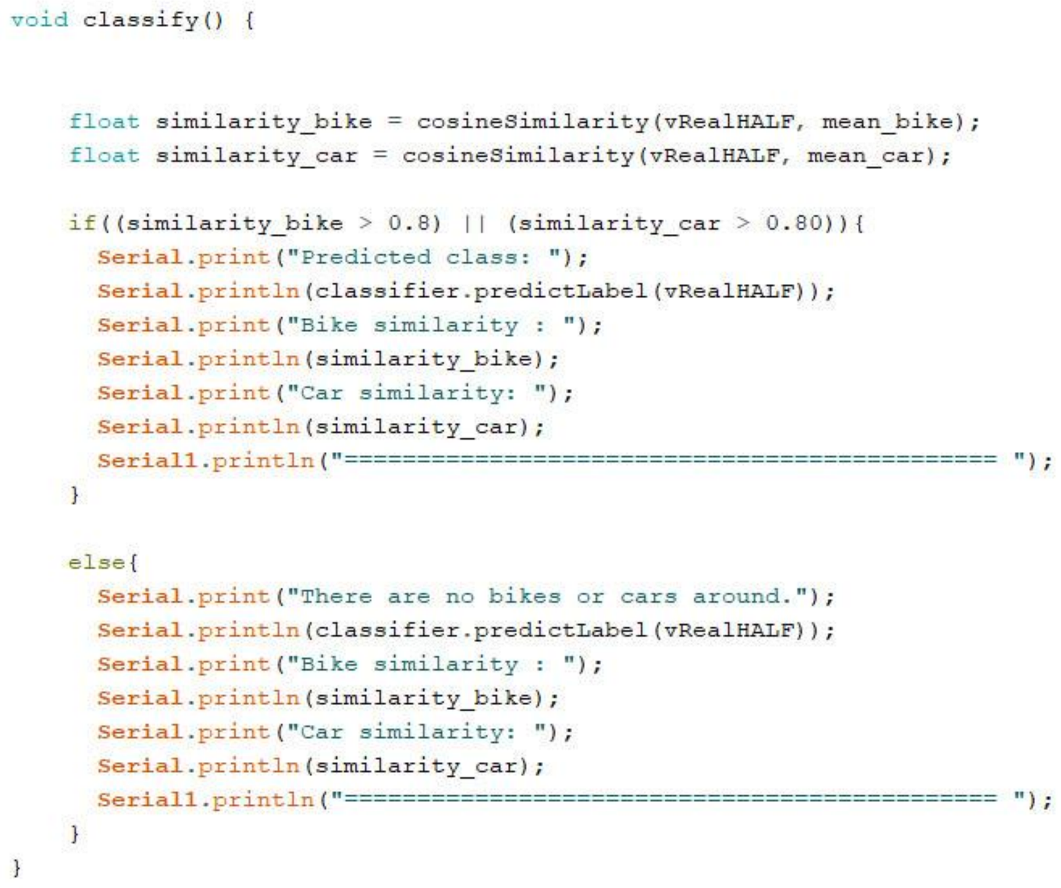


Figure 14. Spectrograms of the two sounds. As we can see they are quite different. For performance reasons, we classify sounds by only taking a thin strip from the bottom of their spectrograms, as we capture only 512 samples of the time signal (meaning only frequencies up to 256 Hz can be represented, according to the Nyquist theorem).

Two of the most commonly used classification methods for such cases are SVM (Support Vector Machines) and Random Forest (we chose the first). After training the model, we use the MicroMLGen package [12], to convert it into a C++ header file and compile it into the microcontroller. Then, extending the original code we use for sampling, we also add the model to it and give the classifier as a parameter, the feature vector (magnitude spectrum) of each recorded sound. The SVM model classifies the given sound into one of two classes. But since the whole process is about classification and **not** about recognizing a sound, what we do is also calculate the average feature vector of each class (in Python code, during the training process), according to the training data set. This helps us ensure that if we give a random sound (neither bike nor car), the model will only classify it by measuring the similarity of the cosine of its feature vector and the average of each class, and if it is more than 80%, then it is classified as recognizable sound.



```
void classify() {


    float similarity_bike = cosineSimilarity(vRealHALF, mean_bike);
    float similarity_car = cosineSimilarity(vRealHALF, mean_car);

    if((similarity_bike > 0.8) || (similarity_car > 0.80)){
      Serial.print("Predicted class: ");
      Serial.println(classifier.predictLabel(vRealHALF));
      Serial.print("Bike similarity : ");
      Serial.println(similarity_bike);
      Serial.print("Car similarity: ");
      Serial.println(similarity_car);
      Serial1.println("============================================== ");
    }

    else{
      Serial.print("There are no bikes or cars around.");
      Serial.println(classifier.predictLabel(vRealHALF));
      Serial.print("Bike similarity : ");
      Serial.println(similarity_bike);
      Serial.print("Car similarity: ");
      Serial.println(similarity_car);
      Serial1.println("============================================== ");
    }
}
```

Figure 15. Part of the Esp32 code, which measures cosine similarity and performs classification.

References

[1] https://playground.arduino.cc/Main/CapacitiveSensor/

[2] https://makeabilitylab.github.io/physcomp/advancedio/smoothing-input.html

[3] https://www.youtube.com/watch?v=40TjjWLljaU&list=LL&index=3&t=1133s

[4] http://highlowtech.org/?p=1963

[5] https://projecthub.arduino.cc/ashraf_minhaj/how-to-use-servotimer2-library-simple-explain-servo-sweep-9bbe4e

[6] https://ai.thestempedia.com/docs/dabble-app/getting-started-with-dabble/

[7] https://eloquentarduino.github.io/2019/12/word-classification-using-arduino/

[8]https://www.youtube.com/watch?v=qeqfoGQs9yo&list=PL9llF8o97zIx1VGLLm1Nk8-qVPAHvWvB7&index=2

[9] https://www.youtube.com/watch?v=Mgh2WblO5_c

[10] https://www.youtube.com/watch?v=hnxYPTehBMk

[11] https://github.com/nathaniel-johnston/arduino-fft

[12] https://eloquentarduino.com/libraries/micromlgen/