POLITECNICO DI TORINO

Master's Degree
in Ingegneria Informatica
(Computer Engineering)

Master's Degree Thesis

Design and prototyping of an Advanced Acoustic Vehicle Alerting System for Electric Vehicles

Supervisor
prof. Marco Carlo Masoero
Company Supervisor
ing. Massimiliano Curti

Candidate
Giulio De Giorgi

Academic Year 2020-2021

Ai miei nonni, Maria e Paolo, Maria e Raffaele, mie radici e mio sostegno.

Summary

In the past decade Electric Vehicles (EV) surely became a trending topic in the automotive world. Most car makers developed this new technology, some of them electrifying models they already produced, such as FIAT with its *500e* and Ford with its *Mustang MACH-e*, some of them adding a supporting electric motor to the Internal Combustion Engine (ICE), such as Toyota with its *Auris* and Jeep with its *Renegade*, and some others again making brand new models, completely electrified from the beginning, like Tesla with its *Model S* and *Model 3*.

This transition is quickly leading to one of the biggest changes in automotive history and, as a result of the exciting interest in electric mobility, brings to the attention of car manufacturer a lot of new challenges. One of these new challenges is to make electric (or hybrid) vehicles emit sounds: this particular problem is crucial, in fact often the sound of electric motors is quite inaudible, both from the cockpit and from the outside.

While external sound production is being subject to regulation from the main regulator authorities - US Congress, Japanese government and European Commission are the main promoters of the regulation -, internal cabin sound is not subject to any rule. For this reason internal sound can be designed not only with the alerting aim in mind, but also with the proposal of giving back the driving feeling to the driver, lost with the transition from the Internal Combustion Engine to the Electric Motor. In addition, internal sound generation can be treated as a branding characteristic and car makers can develop an original sound propriety in order to become more recognizable to the buyers. [25]

With the paramount help and the contribution of Teoresi S.p.A., a leader company in the consulting world, which core area of expertise is the Automotive world, this project has come to the light and led to the prototyping of a working device which is able to make an Electric Vehicle emit actual sound.

In this Master Thesis different design approaches for internal cockpit sound generation will be explored. A sound generation server will be employed in both approaches, using "supercollider" as programming language.

The communication between the vehicle's Electronic Control Unit (ECU) and the sound generation server will be managed via a python script, that will receive and filter

vehicle's CAN bus messages and will send the commands to the sound generation server, using "Open Sound Control" socket (OSC).

All these technologies are open source tools and already widely employed in the automotive world. In the end, a desktop GUI application for test benching the sound generation algorithm without using the real vehicle, will be developed, using "python" and "kivy".

Acknowledgements

Ma s'io avessi previsto tutto questo,
dati causa e pretesto,
le attuali conclusioni

probabilmente rifarei le stesse scelte, gli stessi errori. Questi quasi sei anni non sono stati un percorso banale, tant'è vero che forse non consiglierei a nessuno di prenderne spunto. So solo che in questo breve momento sono felice. Non tanto perchè sono arrivato alla conclusione dei miei anni universitari, piuttosto per aver realizzato negli scorsi giorni quanti siano coloro che, in questi anni, non mi hanno mai abbandonato.

Il primo pensiero va ai miei nonni, nonna Maria della mamma e nonno Paolo, nonna Maria di papà e nonno Raffaele, a cui voglio dedicare l'intero lavoro di tesi, perchè osservando loro che sono riusciti a reallizare il sogno di vedermi diventare grande, alla fine ho capito che stavo realizzando anche il mio di averli accanto.

Vorrei ringraziare anche i miei amici, quelli che sono rimasti con me, ma anche quelli da cui, nel corso del tempo, mi sono allontanato, perchè è anche grazie a loro se sono arrivato fin qui. In particolare ringrazio Alessandro, il mio migliore amico, con cui ho condiviso tutto, Martina e Giulia, le due amiche più belle e divertenti che si possano desiderare, Stefania, amica da una vita e sempre presente al mio fianco, Jurgen ed Emanuel, i miei coinquilini della residenza, miei mentori, Martina, amica, coinquilina, confessore e consigliera, Giulia, amica rara, che è sempre riuscita, con semplicità, a ricordarmi quanto valevo, anche nei momenti più complicati, Valentina e Salvo, i miei due super colleghi, amici sinceri, simbolo che il PoliTo non toglie soltanto. In più vorrei ringraziare tutti gli amici della residenza Mollino e tutti gli amici che non ho nominato, che spero mi perdoneranno.

Inoltre mando un grande abbraccio ai ragazzi della mia associazione studentesca IEEE-HKN e del board, Emanuela, Sara, Ilio, Federico e Gilberto, che mi hanno tenuto a galla durante questi mesi in cui sprofondare nella solitudine delle restrizioni era fin troppo semplice.

Un doveroso ringraziamento va anche al mio relatore, il professor Marco Masoero, e al mio tutor aziendale, ingegner Massimiliano Curti, nonchè all'intera Teoresi S.p.A., per

avermi dato fiducia e avermi dato l'occasione di lavorare a questo progetto.

Infine, ringrazio la mia famiglia, mamma Anna, papà Roberto e mio fratellino Michele: siete stati il mio faro e il mio molo, mi avete dato tutto e non avete mai voluto nulla in cambio, avete accettato ogni mia scelta supportandola come se fosse vostra, permettendomi di realizzare quelli che erano i miei sogni e di rincorrere le mie aspirazioni.

Un brindisi a tutti voi,

Giulio

Contents

List of Tables

List of Figures

"Seguiremo questo sentiero"
"Quale sentiero?"
"Il sentiero che apriamo noi!...Quel sentiero che apriamo noi!"
[Miguel, Tullio, La strada per El Dorado]

Part I

Background technologies

Chapter 1

Introduction

1.1 The reason behind the project

The automotive world is changing. In the past few decades electric mobility became more than a trend and all car makers started to develop their own concept of Electric Vehicle.

Since Electric Motors emit a sound that is quite inaudible, the need of developing a technology that was able to produce sound according to the vehicle's behavior and the driver's actions on the vehicle has became dominant.

The reasons behind this need are various: first of all the driver needs to perceive what is happening in its vehicle. Just think if, while driving, the driver cannot ear any sound, and thus, he cannot perceive the Vehicle Speed nor the Torque generated by the Motor. Especially at low speeds, when the sound of the rolling tyres and the airflow-generated noise are below the audible threshold, producing an artificial sound is crucial. For this reason, this type of technology is called AVAS, which is the acronym of "Acoustic Vehicle Alerting System".

In Europe and in the United States, the use of this technology for reproducing alerting sounds for pedestrians is becoming mandatory for car makers and it's under regulation by the authorities in terms of volume and intensity of the sound.

After that, car makers can use this technology in order to estabilish even more their brand identity, developing a particular sound that reminds the costumer of the specific manufacturer. For this reason, a good development can bring an immense value to the brand. In the end, producing a particular sound via software architecture is convenient by a cost reduction point of view: in fact, as we will discover after in this paper, low performance hardware is required in order to achieve this goal.

1.2 The overall architecture

The main idea is to develop a tool that is able to read all the messages arriving from the Control Unit of the Vehicle, interpret them and let the algorithm produce a particular sound in each instant of the driving, depending on the read messages. In order to do this, a programming language which is able to synthetize sound in real time is needed. On the other hand, a particular hardware which is capable of reading the signals provided by the vehicle and a protocol to interpret them is required.

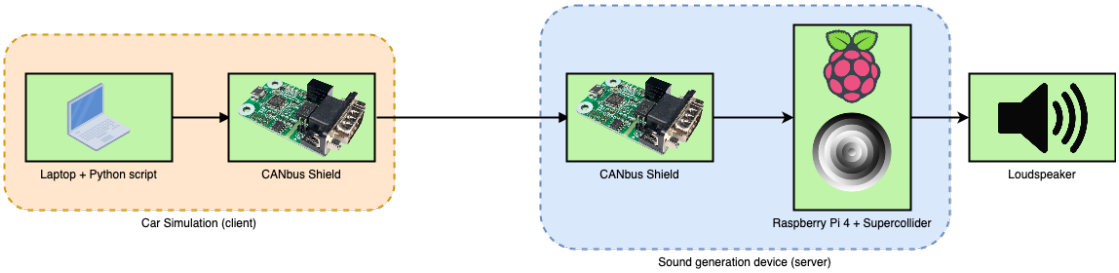


Figure 1.1: Representation of the overall architecture schema

This type of architecture can be modeled as a simple client-server one, in which the vehicle and its ECU act as client and the on-board device acts as server. Nowadays almost all car manufacturers use CAN bus protocol in order to communicate to and to receive informations from the Electronic Control Unit of the vehicle. This particolar tool uses sockets to send and receive messages, via a physical device called "CAN bus Shield".

During the development of this project two CAN bus shield have been used: the first acted as transmitter (Tx), in fact it simulated the message log of the vehicle, and the second one acted as receiver (Rx). The receiver CAN bus Shield is connected to a Raspberry Pi 4, which is the device that runs the python script which filters the message log and sends the directives to the sound generation server. At the end of the chain there's the loudspeaker, which is the actuator of the sound generation.

Since a real vehicle was not available as testbench, a simulation platform has been developed. Its features will be described later in this document.

Chapter 2

The used technologies

In this section all the technologies that have been employed during the development of this project will be described and analyzed.

2.1 Supercollider

Figure 2.1: Supercollider icon [23]

Supercollider is a programming language which allows the programmer to compose sounds algorithmically, that is to reproduce sounds using its integrated sound generation server. In fact, supercollider is made up of three components.

- **scsynth**
  The core of the language. It is the sound generation server that, in real time, is able to reproduce various sounds. The audio engine is accurate and high quality.
- **sclang**
  The language itself. It supports the objected-oriented progamming paradigm and, most important, can be dynamically type. This means that various sections of the code can be compiled and run separately. The flows is not completely sequential but every scope (limited by "") can be run individually and eventually stacked and layered. Its syntax is similar to the C one, which permits an easy usage.
- **scide**
  Supercollider provides also an integrated development environment, which helps the

programmer write its code. Scide is, in fact, a text editor that supports the coding and is directly connected with supercollider's documentation for a faster development [23].

2.2 Controller Area Network

Controller Area Network, also known as "CAN bus", is a vehicle bus standard designed for allowing the various sensors connected to an electronic control unit (ECU) to communicate with each others and with each other's software applications. The protocol has been designed with the possibility to add and to remove any senor's interface (or node) to/from the network at any time without the necessity to act directly on the network.

The main characteristic of the protocol is that the messages don't have a specific destination address but every message is spread among the entire network using the multi-cast method, and every node in the network filters only the packets it needs. In fact a generic CAN-bus packet has only an identifier, which defines the type of datum its payload contains. [20]

CAN bus physical structure is made up of a twisted-pair cable with a 120 Ω characteristic impedance, terminated at both ends with 120 Ω resistors, which match the characteristic impedance of the line in order to prevent signal reflections [8]. The bit value is obtained via differential measurement of the voltage on the two twisted paired lines, called CAN High and CAN Low.

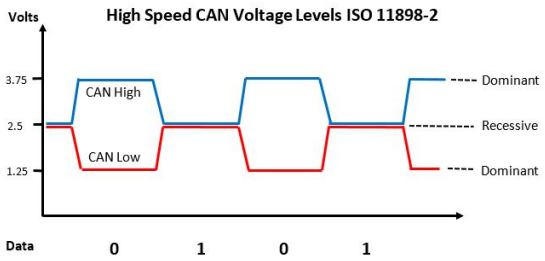


Figure 2.2: CAN frame composition [1]

Bits are encoded with Non-returning Zero (NRZ) method and the two levels employed by the CAN bus protocol are said "dominant" and "recessive". The value that is present on the bus depends on the wiring. For example, in case of a wired-AND implementation of the bus, the "dominant" level would be represented by a logical 0 and the "recessive" level by a logical 1, as showed in Figure 2.2; the opposite is true in the case of a wired-OR implementation [3].

In CAN protocol there are several types of message structures:

- **Data Frame**
  "Data Frame" or simply DF is the most common type of structure and is sent by a so called transmitter node (Tx) and received by all the other nodes of the network (Rx). Each node that receives a Data Frame can choose if it is relevant or not: if the message is relevant, it is read, otherwise it is discarded.
- **Remote Frame**
  "Remote Frame" or simply RF is similar to Data Frame but it does not have the Data field. It is used for requesting a message from a node.
- **Error Frame**
  "Error Frame" is used for signaling a transmission error to the network. If a node spots an error in a packet, it signals it across the network and requests the retransmission of the corrupted packet. Since CAN protocol is highly resilient to faults, a node that receives a certain number of Error Frames, self-excludes itself from the network in order to avoid the blocking of the network.
- **Overload Frame**
  "Overload Frame" is used by a busy node for requesting the delay of the following packets.
- **Interframe Space**
  "Interframe Space" is used to separate the transmission of a Data Frame from the transmission of a Remote Frame

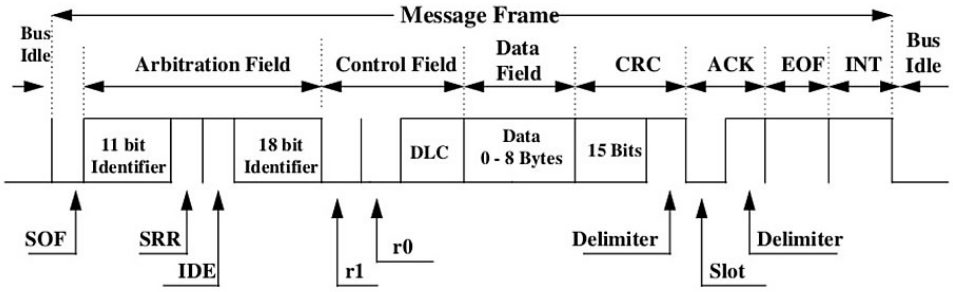


Figure 2.3: CAN frame composition [1]

In figure 2.3 a typical CAN frame composition is showed. A typical CAN message frame is made up of seven fields, that are analyzed below:

- **Start of Frame (SoF)**
  the "Start of Frame" (SoF), which is composed by only 1 dominant bit, indicates the starting of the message frame
- **Arbitration Field**
  "Arbitration Field" indicates if a frame is a Data Frame or a Remote Frame and it is made up of 29 bits.

- **Control Field**
  "Control field" indicates the length of the Data Field, in fact it is composed by 6 bits: 4 of them indicates the length and are called Data Lenghth Code (DLC), while the remaining 2 are reserved for future applications

- **Data Field**
  "Data Field" contains the real informations to be decoded. Its length ranges from 0 to 8 bytes.

- **CRC Field**
  CRC stands for "Cyclic Redundancy Check" and "CRC field" contains the control sequence, which grants to the protocol the rejection of data corruptions. This fields is 16 bits long.

- **ACK Field**
  "ACK field" is made up of 2 bits: the first is called "ACK slot" and the second is called "ACK delimiter". Initially they are both recessives but, when the packet is received, the receiver node overwrites the "ACK slot" bit with a dominant bit.

- **End of Frame (EoF)**
  EoF indicates the end of the message frame. It is made up of 7 bits. [1] [24]

2.2.1 CAN bus Databases

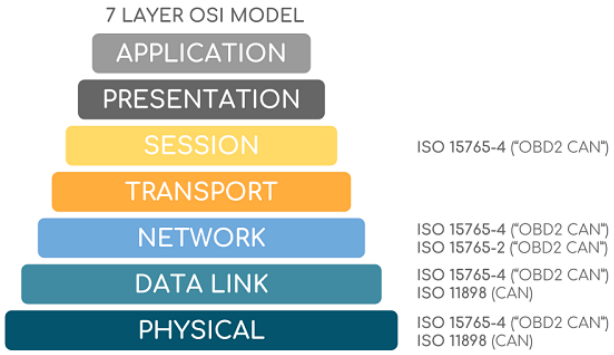


Figure 2.4: CAN OSI model

As we can see in Figure 2.4 only Physical layer, Data Link layer, Network layer and Session layer are standardized. For this reason each car maker can freely make some modifications and adjustments to the Application level, in fact each manufacturer usually defines a set of signals provided by the ECU of each vehicle and assigns them a correspondent CAN frame ID. In this way all the informations provided by the ECU can be easily recognized. All the associations defined by the manufacturer is gathered in a so called "CAN database" or "CAN db".
A single CAN bus frame can contain the description of multiple signals provided by the

Electronic Control Unit: for this reason the CAN bus database stores also the starting bit of the specified signal and its length.

A CAN bus Database is a database in which each entry describes how a signal provided by the Electronic Control Unit is described in a CAN message. Every manufacturer has its own CAN bus Database, since there's not a standard yet. The most used file formats for this type of data are ḋbc, k̇cd and ċdd.

In the following pages the words "frame" and "messages" will be used with the intention of referring to the same object and will be often interchanged, even if they refer to slightly different things.

- **Ignition**
  "Ignition" message indicates if the vehicle is switched on or turned off. Its message description is reported in Table 2.1.

ID	Name	Signal Name	Len	Factor	Original Name
0x1	m01	Ignition	3	1	COMMAND.IgnitionStatus

Table 2.1: Ignition CAN message description

"Ignition" signal can have different meanings, in fact it describes the angle in which the ignition key is placed. Is worth noticing that the ignition device in which the ignition key is inserted - the keyhole on the left or on the right of the steering wheel - is a circuit with a resistance in which the voltage changes every time the key angle changes. This voltage change is detected by the Electronic Control Unit and is encapsulated into a CAN message and sent across the network. Thus, this single CAN frame can assume different values, as described in Table 2.1.

Value	Name	Description
0x0	BEGIN	Initialization
0x1	FIRING_LK	Ignition Lock
0x2	XSRY	Accessory
0x3	SPRINT	Runnable
0x4	GO	Start
0x7	SNA	Signal Not Available

Table 2.2: Different Ignition message values

- **Gas Pedal Position**
  "Gas Pedal Position" signal indicates the pressure level of the gas pedal and it ranges from 0 to 100 with a factor of 0.4, as showed in Table 2.3.

ID	Name	Signal Name	Len	Factor	Original Name
0x2	m02	GasPedalPosition	8	0.4	ENGINE.PedalPosition

Table 2.3: Gas Pedal Position CAN message description

- **Vehicle Speed**
  "Vehicle Speed" signal indicates the actual vehicle speed and it ranges from 0 to 511, with a factor of 0.0625, which means that the maximum speed that can be measured and read by the Electronic Control Unit is 318,8 km/h. If the vehicle can reach a speed grater than that maximum value, the length of the signal should be increased and thus the entry in the CAN db must be modified. In this project the hypothesis of having a vehicle that cannot exceed this maximum value will be taken as valid. The description of the signal is reported in Table 2.4.

ID	Name	Signal Name	Len	Factor	Original Name
0x3	m05	VehicleSpeed	13	0.0625	ENGINE.VehicleSpeed

Table 2.4: Vehicle Speed CAN message description

- **Braking Status & Regenerative Brake**

  "Braking Status" indicates the status of the braking pedal. It is a 2 bit long message. Table 2.5 describes in detail the message encoding. When the message is equal to 0 the pedal is released, when it is equal to 1, the pedal is pressed.

ID	Name	Signal Name	Len	Factor	Original Name
0x4	m04	Braking Status	2	1	BRAKE.BrakingStatus

Table 2.5: Braking Status CAN message description

"Regenerative Brake" is produced when the vehicle is regenerating its batteries through braking. In fact, a peculiar characteristic of an Electric Vehicle is that the driver can allow the batteries to be recharged while braking the vehicle, i.e. pressing the brake pedal. When the vehicle's brakes are actuated, in fact, the electric motor acts as an electric generator, and part of the energy can be used for battery recovering. This is one of the main feeling difference while driving an Electric vehicle: the driver will listen to a sound that indicates when the regeneration is active and thus they can assume a better driving behaviour, that will safe energy and will increase the travel range. As showed in Table 2.6, the Regenerative Brake payload is 12 bit long, in fact it can range from 0 to 8190, with a 2 multiplicative factor.

ID	Name	Signal Name	Len	Factor	Original Name
0x5	m07	RegenBrake	12	2	BRAKE.RegenBrake

Table 2.6: Regenerative Brake CAN message description

2.2.2 Innomaker USB2CAN

A so called "CAN bus Shield" is a device that allows electronic prototyping boards, like Arduino or Raspberry Pi, to poll the ECU of a vehicle for informations. In this way a communication is enstabilished between the vehicle and the board and all the informations can be exploited, stored and used for computations.

This type of device can be employed also as a tool for On-board Diagnostic (OBD), in fact all the data coming from the shield can be used by a software in order to obtain valuable informations about the state of the vehicle and its failure.

For the development of this project, "Innomaker USB2CAN" has been identified as the best alternatives among a various list of CAN bus shields available on the market. Most of the other alternatives on the market needed a board for working, or did not have a USB port for communication, or needed an additional module for USB port, or again were not designed for multi-platform support. Innomaker USB2CAN is a plug-and-play module device that does not need to be powered externally, in fact it's power supplied directly through its USB connection. [14]

In addition, Innomaker USB2CAN does not need any additional hardware component for working properly, in fact the microcontroller is integrated in a single plate where also the USB port for communication is present. The microcontroller is the STM32F0, which grants support for SPI interface.

One of its key feature is that it is fully compatible with Raspberry Pi 4 and, in addition, its on-board microcontroller's symbol rate can range from 20 Kbps to 1 Mbps.

In Table 2.7 the main technical specification are presented.

Connector	
CAN	D-SUB, 9 pins
USB	USB 2.0 Full-Speed, Micro USB
CAN Features	
Specification	ISO 11898-2 High-speed CAN
Data Rate	From 20kbps to 1Mbps
Isolation Voltage	1.5K VDC/min, 3K VDC/1s
Microcontroller	STM32F0, 48 MHz
Termination	120 Ohm resistor selectable jumper
CAN Transceiver	ISO1050DUBR, Texas Instruments

Table 2.7: Innomaker USB2CAN main technical specifications

The pin connectors description are showed in Table 2.8 and the schematic is presented in Figure 2.5 .

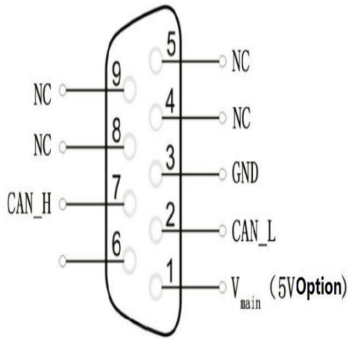


Figure 2.5: Innomaker USB2CAN Connector Pinout [14]

Pin	**Description**
1	5V/150mA Output
2	CANL bus line (dominant low)
3	CAN_GND
4	NC
5	NC
6	NC
7	CANH bus line (dominant high)
8	NC
9	NC

Table 2.8: Innomaker USB2CAN Connectors Pinout description [14]

Since the architecture is a client-server one, two modules have been employed: the first reads all the data coming from the vehicle's ECU and send them to the second one, which selects only the data which are valuable for the sound generation and lets the script do the computations.

2.2.3 can-utils

The devices involved in the development of this project are Linux-based, in fact the Raspberry Pi 4 uses an Ubuntu for Raspberry distribution, while the device employed for the vehicle simulation runs an Ubuntu 20.02 operative system. For this reason a tool for managing CAN messages, both client side and server side, is needed.

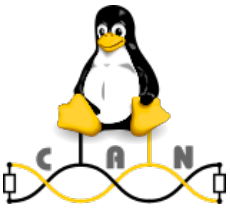


Figure 2.6: can-utils logo [4]

In the linux kernel a set of drivers that are able to interact with the CAN protocol is present and it is called "can-utils". SocketCAN has been originally developed by Volkswagen and then widespread into the linux world, being open source and easily available. The basic operations are to open and close sockets of the so called "PF_CAN" protocol family [2].

In Figure 2.7 the typical communication layers are shown: as we can see, SocketCAN is a bridge between the hardware and the user level, in fact it provides all the tools for the CAN packets exchange.

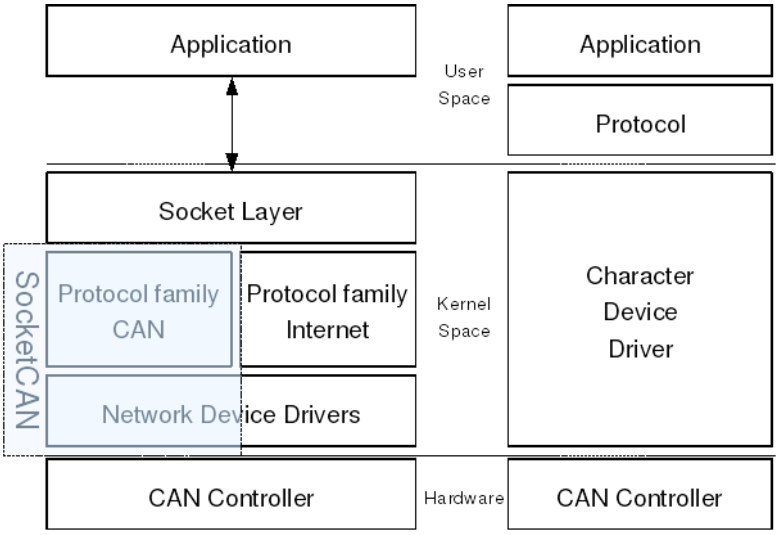


Figure 2.7: SocketCAN communication layers [2]

"Can-utils" is an interface that exploits the SocketCAN drivers, providing a set of commands that the user can use in the shell. Mainly, using can-utils, the users can initialize the CAN shield interface, send and receive CAN packets.

The main commands are described in detail below:

- **cansend**
  cansend allows the user to send a single CAN frame from the specified interface. Using the <can_frame> field the ID, the length and the data can be specified.

- **cangen**
  cangen is a command for sending a flow of CAN frames: by setting various options the user can decide how many frames to send per second, the number of frames to send, the ID of the specific frame to send, the length of the frame and, most importantly, the user can force the sending of frames with random IDs, for debugging reasons.

- **candump**
  candump is a command for receiving CAN bus traffic: adding options directives the user can modify the way the messages are displayed, can impose the message log to be written into a file instead of displayed on the shell, and even filter them, dumping only the desired IDs and dropping all the other packets. The candump command supports the reading of various interfaces at once, with a superior limit of 16 interfaces.

2.2.4 CAN bus tools

In addition to the "can-utils" interface, also "CAN bus tools" or "cantools" has been utilized. Cantools is a library that provides an easy way to use all the can-utils commands

in a python environment. Furthermore, cantools provides also advanced functions, such as the possibility to fetch a CAN database from the most used file formats, even the proprietary ones, such as ḋbc, k̇cd, ċdd and to encode and decode CAN frames. [15]

2.3 Open Sound Control

Figure 2.8: Open Sound Control logo [7]

Open Sound Control is a content format for networking between sound instruments, computers and media devices in general. Its functionality is similar to XLM or JSON, in fact it allows the communication between two interfaces (client and server) encoding time tags, providing some quality of service support, wireless efficiency (in fact it is widely used in musical instruments that requires a wireless environment) and time synchronization [7].

In this project Open Sound Control will be employed in two instances. First of all, it is used internally by supercollider, for allowing the communication between sclang, the language itself, and scsynth, the sound generation server. In addition, Open Sound Control is employed for the communication between the python script that decodes the CAN bus messages and the supercollider script that implement the sound synthesis algorithm.

2.4 Kivy

Figure 2.9: Kivy logo [6]

Kivy is an open-source cross-platform framework for python language for making Graphic User Interfaces in an easy and fast way. Kivy is completely free to use and, in this project, it will be employed for the creation of a desktop application that is able to emulate the behaviour of an electric vehicle, generating a flow of CAN messages.

Kivy's core is written in C language and it is high performance, due to the fact that is built over OpenGL ES 2.0, which grants a fast graphic pipeline [6].

Part II

Designing the Project

Chapter 3

Different design approaches

3.1 The vehicle simulation

The problem of simulating the vehicle behaviour came straightforward: since this Master Thesis project has been developed in remote, the real vehicle was not available. Thus the necessity of finding a way to reproduce all the CAN bus signals needed for the sound generation became clear.

During the development process multiple ways to reproduce the vehicle's behaviour have been explored and in the following sections some of them will be analyzed and compared: the trivial simulation via keyboard, the simulation via proximity sensor and a desktop GUI application.

The trivial simulation via laptop keyboard is a simple type of simulation which involves only the arrow keys of the keyboard for simulating the Vehicle Speed and the Gas Pedal Position signals, without the possibility to increase these signals continuously, while the simulation via proximity sensor involves a ToF proximity sensor for implementing a more plausible behaviour, inertia and the possibility to increase or decrease the value of the signals continuously. In the end, a desktop application has been developed, with sliders and button simulating the variations of the signals.

3.1.1 Trivial simulation via keyboard

The first way to simulate the vehicle's behaviour is the trivial one: using a laptop and its keyboard's arrow keys it is possible to ignite the engine, to shutdown the vehicle and increase or decrease the vehicle speed, encapsulating the informations into a CAN bus frame and sending it through the CAN bus shield.

All the necessary libraries in order to perform input/output terminal operations are imported, in addition to the basic terminal control functions. Furthermore all libraries for allowing the script to send Open Sound Control directives are added: the `OSC message`

`builder` and the `UDP client`, which is the core tool for sending these messages.

The script uses also the libraries related to the Controller Area Network management, such as `socket`, which gives the basic support for opening network socket, `can` and `cantools`, that are the libraries that allow the script to recognize the CAN bus interface and to encapsulate CAN bus messages into packets and send them via such interface.
Right after all the imports, the informations about the CAN bus shield connected to the laptop are stored.

The class `Getch` is defined: when this class is called, it takes the press of an arrow key as input and returns the address of such arrow key as ASCII escape code. The ASCII code mapping of the arrow keys is reported in Table 3.1.

Arrow key	**ASCII escape code**
←	\x1b[D
↑	\x1b[A
→	\x1b[C
↓	\x1b[B

Table 3.1: Arrow keys ASCII encoding

The method `get()` is the core method of the entire simulation, in which the CAN bus interface is initially recognized and set up, the CAN bus database is read and subsequently an endless loop for the messages flow generation is started.

Right after the CAN interface is start up and the bit rate is set. Doing this the start up process via the terminal shell is avoided and everything is set up in the script.

The CAN bus interface is finally initialized, through the method `can.interface.Bus(...)` and the CAN bus database in parsed and memorized into the variable `db`.
Subsequently the valuable message are selected from the database and saved into the variables `speed_message`, `ignition_message` and `gaspedal_message`. In this version of the vehicle simulation algorithm, the signal referring to the "Braking Status" has been neglected.

The first operation in the while-loop is to call the function `_Getch()` and to store the return value into the variable `k`, i.e. the value of the arrow key pressed.

When the input corresponds to ← the ignition message is encoded, then encapsulated into the CAN bus message and sent. The parameters of the method `ignition_message.encode(...)` are described in Table 3.2.

Parameter Description	Value
Body Control Module Fire Prevention Status Command	0
Body Control Module Fire Prevention Status Confirm	0
Body Control Module Fire Prevention Fail Status	0
Turn Indicator Status	0
Message Counter Body Control	0
Key Inserted Ignition Status	0
Command Ignition Fail Status	0
Command Ignition Status	4: START
Cyclic Redundancy Check	0

Table 3.2: Parameters of the Ignition message

If value is 0 it means that such parameter is not of interest, while, in case of the parameter "CmdIgnStatus", it is 4 which means "START". For a complete knowledge of what this value means, refer to Table
2.2.

When ↑ is pressed the gas level is changed and then stored into `gaslevel`. The gas level range is between 0 and 100, thus its behaviour follows $y = x$ when in range [0, 40], while it follows $y = 2.5x$ when in range [40, 100]. A check is done when the gas level reaches its maximum value, in this case every later pressure of the arrow will lead to no variations. After that, the speed is updated and then stored into `speed`.

As we previously said, in this version of the simulation, the braking action has not been taken into consideration, in fact the ↓ acts directly on the speed when pressed. When ↓ is pressed, the gas pedal position is set to 0 and the speed is decreased by 3 each pressure.

At the end of each cycle a speed signal is encoded and then sent. The method `speed_message.encode(...)` takes as parameters the informations reported in Table 3.3.

Parameter Description	Value
Anti-spin Braking System Active	0
Electronic Stability Control Active	0
Brake Intervention Status	0
Prefill Active	0
CMM Intervention Acknowledgment	0
Vehicle Speed Signal	speed
Vehicle Speed Failure Status	0
Message Counter	0
Cyclic Redundancy Check	0

Table 3.3: Parameters of the Vehicle speed message

Like the ignition signal, some of the parameters have been set to 0 since they are not

relevant. On the contrary the parameter "VehicleSpeed" is set to `speed`. Right after the speed signal, also the gas pedal position is sent to the network.

In the end, when → is pressed the vehicle is shutdown and a specific signal is encapsulated and sent. The parameters are the same of the ignition case, but, instead of 4, the "CmdIgnStatus" parameter is set to 1.

Advantages and disadvantages of the trivial simulation via keyboard

The use of the vehicle simulation via a keyboard has advantages and disadvantages: the most important advantage is the easy of use, it in fact involves only one input, i.e. the keyboard, and it is simple to code and debug.

Unfortunately this method of simulation has enormous drawbacks: first of all the number of signals that can be reproduced is extremely limited, as a matter of facts only the Vehicle Speed and the Gas Pedal Position are considered, while the Braking Pressure has been neglected, furthermore the signal values' increment is not continuous but it is discrete, corresponding to the pressure of the specific arrow key. In addition, it is complicated to implement inertia, that is the decreasing of the speed due to the friction forces, once the gas pedal is released.

In the end, the messages sending frequency cannot be evaluated or changed dinamically while the process is running: tipically a CAN bus shield is able to send or receive a message every 200 ms, but this frequency can be changed in order to meet some testing needings. With this type of simulation this is not possible.

For these reasons, a new type of simulation is needed, which is the vehicle simulation via proximity sensor.

3.1.2 Simulation via proximity sensor

For the simulation via proximity sensor a so called Time-of-Flight sensor has been employed. This particular device is able to estimate the distance between the camera and the object, measuring the time spent by a light beam to travel from the sensor to the object and to get reflected back to the sensor. This type of sensors is widely used in engineering and it is often employed for allowing the human-machine interaction.

The Time-of-Flight product chosen for this type of simulation is a V53L3 by STMicroelectronics, in combination with the board STM32L476RG by STMicroelectronics, which is able to communicate with the ToF sensor and define its transfer function.
STM32L476RG is an ultra low power device based on ARM Cortex-M4 32-bit RISC core architecture [21], while V53L3 ToF sensor is a proximity and distance low-power sensor which emits 940 nm invisible laser for making its measurements [22].

Using the ToF sensor the Gas pedal position can be simulated, in fact the user can use its hand like it was driving an actual car using its foot to act the real pedal.

The transfer function implemented is a damped step in discrete time, which is represented by the following expression

$$y = \frac{0.002059z + 0.001686}{z^2 - 1.511z + 0.5488}u$$

and generates a step and impulse response as showed in Figure 3.1.

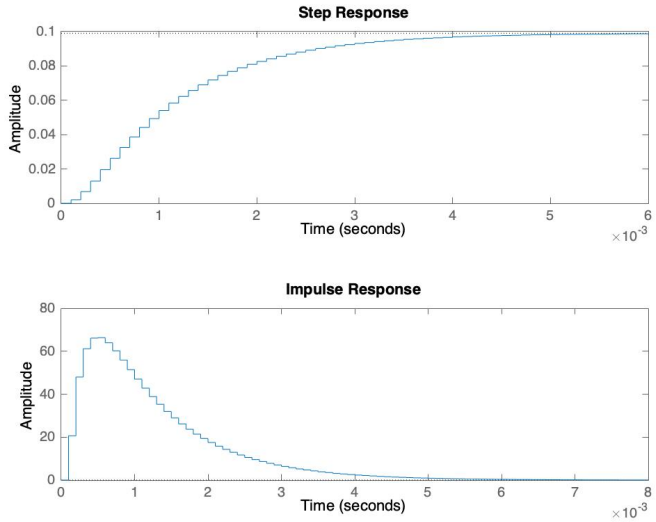


Figure 3.1: Step and impulse response of the ToF sensor's transfer function

The micro controller reads the ToF sensor in polling mode and the polling time is 1 ms. Every time the sensor is polled, an input is obtained and the method that implements the transfer function is called, in order to obtain the speed value.

The function `double tfcnDcMotorStep (double u)` takes as parameter the input `u`, and then implements the transfer function in order to obtain the value of the speed.

In order to obtain and then send the corresponding CAN bus messages to CAN bus shield interface, a python script is employed: the serial port is initialized and the timeout time is set, then an endless while-loop is utilized in order to obtain continuously the information from the serial port, which is directly connected to the micro processor with the ToF sensor. A serial string is read every iteration if the datum is ready, then the information referring to the gas pedal position and the speed are obtained and stored into two variables. In the end, the information are wrapped into CAN bus messages and sent to the CAN interfaces, which will communicate to the Raspberry Pi 4 and its CAN bus shield, in order to transmit these values to the sound-generation algorithm.

Advantages and disadvantages of the trivial simulation via proximity sensor

Simulation using a Time-of-Flight sensor is a better choice for sure: with this method the pressure of a pedal can be simulated, even from a physical point of view, in fact the user can use its hand in order to provide the input to the sensor. In addition, through the implementation of the transfer function, also the inertia can be implemented, in fact when the pedal is released, i.e. when the proximity sensor doesn't see any object in the neighbourhood, the speed gradually decreases, as it would be in a real scenario, where the friction of the asphalt and the air is applied.

On the other hand, some new hardware is required, in addition to the laptop, in fact, a board with a micro controller and the sensor itself are needed in order to perform this simulation. Furthermore some new code has to be developed, which surely increases the complexity of the project.

In the end, in the case of using only one proximity sensor, which is the case of this simulation, only one pedal per time can be simulated. That means that, for simulating the Braking Pedal, another sensor is needed and some other code has to be produced. This is a critical drawback, that must be taken into consideration when approaching to this type of simulation.

3.1.3 TouCAN: a desktop GUI application for virtual test benching

Figure 3.2: TouCAN logo [12]

TouCAN is a multi-platform desktop application with GUI for virtual test benching that has been developed in the context of this project in order to simulate the behavior of the network of a vehicle.

As said previously, the necessity of having a good simulation came straightforward since the entire work has been developed in remote, that is without the availability of a real vehicle. The application has been entirely developed in python and kivy, in order to grant the multi-platform usage.

This application is made up of two threads: the main thread, that builds the graphical user interface, initializes the CAN bus interface and sends the messages for ignition, shutdown and brake, and the thread `t`, that continuously polls the values of Vehicle Speed and Gas Pedal position and sends them across the network.

An operational overview

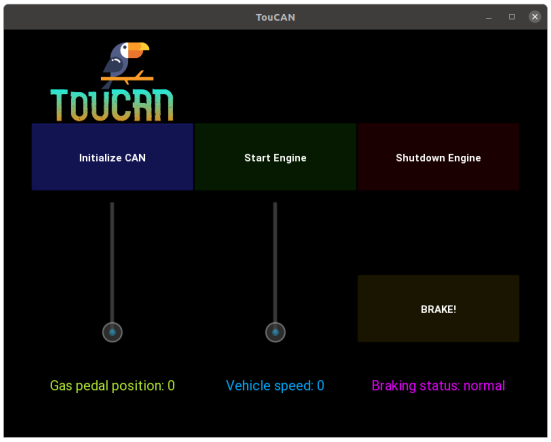


Figure 3.3: TouCAN application main page

In Figure 3.3 the landing page of the application is showed: it presents only one clickable button, in fact all the others are disabled. The reason for this choice is that, in order to enstabilish a communication with the network of the vehicle, the CAN bus interface has to be initialized.

`OnInitializeButtonClick(self, instance)` initializes the CAN bus shield and enables one of the other buttons if the initialization process is successfully completed.

`OnStartButtonClick(self, instance)` describes the actions of the "Start Engine" button: it enables all the other buttons, sends the ignition message into the network and, in the end, it starts the thread `t`.

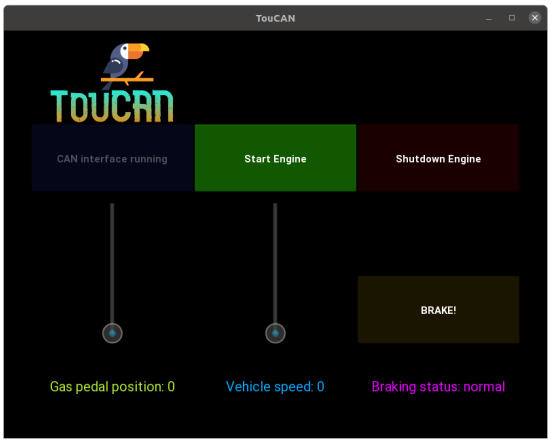


Figure 3.4: TouCAN application start engine page

In Figure 3.4 the look of the application after that the CAN bus interface has been successfully initialized is showed. The blue button on the first column of the first row is now disabled and presents the status of the CAN bus interface. The only clickable button is, in fact, the "Start Engine" one. If clicked, it sends a CAN msg with the ignition signal encoded in it.

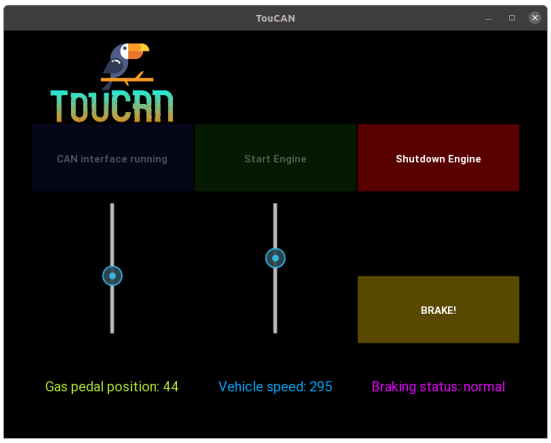


Figure 3.5: TouCAN application running page

Figure 3.5 shows the appearance of the TouCAN application when the "Start Engine" button is pressed: all the buttons and the sliders are enabled and the user can set both Gas Pedal Position and Vehicle Speed arbitrarily, in order to send across the network the desired value of each signal.

Minimum and maximum values are chosen accordingly to the CAN database used in the entire project: each range for each signal is reported in section "CAN bus database".

`pollThread(self)` is the method taken as parameter by thread `t` and reads the Vehicle Speed and Gas Pedal Position signals from the current value of the sliders every 50 milliseconds, accordingly to the real frequency of a vehicle (that ranges from 20 milliseconds to 100 milliseconds) and then encodes them into the CAN messages for sending them across the network.

The Brake signal is generated when the corresponding button is pressed; the label under the button describes when the pedal is pressed and when is released. The signal is, as usual, encapsulated and sent across the network.

In the end, if the "Shutdown Engine" button is pressed the application sends across the network the message that indicates the shutdown of the vehicle, then waits for the thread t to join when it will return. It also sets all the sliders' values to 0 and disables all the other buttons.

Advantages and disadvantages of TouCAN

Being a GUI desktop application, TouCAN is extremely user friendly and complete, in fact it simulates all the signals needed for running an audio generation algorithm. For this reason it has been widely employed for the testing of the project. Unfortunately it does not implement physics, in fact Gas pedal position signal and Vehicle Speed signal can be individually modified without any restriction imposed by the dependence between these two signals.

However, since this application's goal is to implement a simulation that is functional to the test benching of the sound generation algorithm, this does not imply anything crucial. Instead it realizes an ideal situation when each signal can be driven independently, granting an easier way to explore all the different use cases.

3.2 Sound generation

Frequency spectrum Analysis

Frequency spectrum Analysis is a visual representation of signals, in the frequency domain. This type of representation is based on the Fourier Transform but became computationally efficient when the Fast Fourier Transforms were discovered.

Fast Fourier transforms (FFTs) are fast algorithms for the computation of the discrete Fourier transform (DFT) or of its inverse (IDFT). The FFTs are among the most important algorithms in applied sciences and engineering, mathematics and in computer science, in particular for one and multidimensional systems theory and signal processing. An algorithm is called fast if it has low complexity, where the complexity is the number of elementary computation steps necessary to execute it [16].

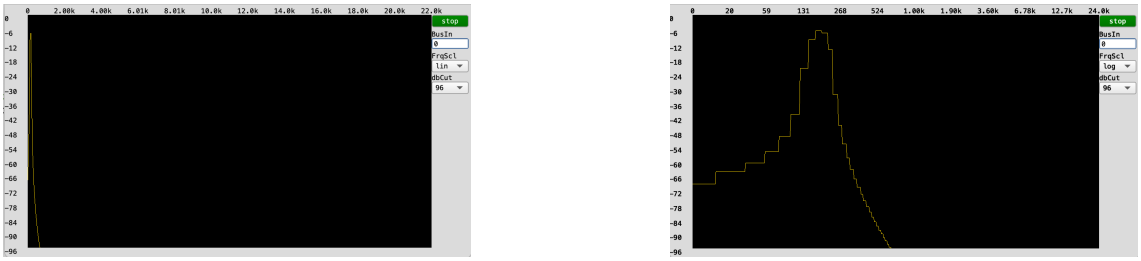
Supercollider offers an integrated graphical user interface for showing this type of analysis.

This tool is widely used for comparing signals, especially sounds, in a two-dimensional graph, with frequency on the horizontal axis that varies from the left to right, measured in Hz, and the gain on the vertical axis, ranging from the bottom to the top, measured in dBm.

For example, a sin wave at 200 Hz, starting with a phase 0 and an amplitude of 0.5, in supercollider is easily obtained using this command

```
{ SinOsc.ar(200, 0, 0.5) }.play;
```

And generates a two-dimensional frequency spectrum graph as showed in Figure 3.6:



(a) Linear scale (b) Logarithmic scale

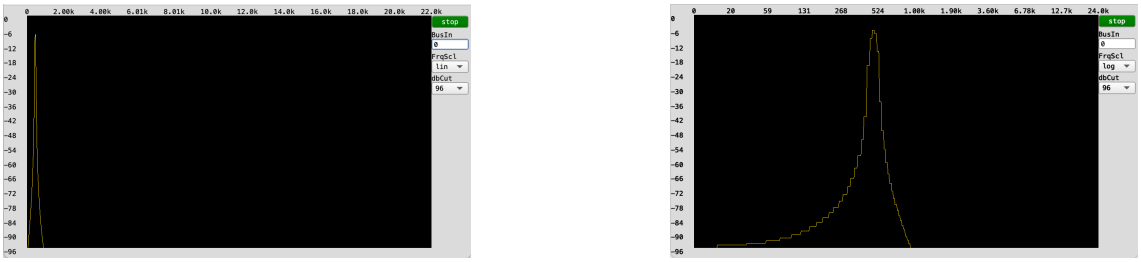
Figure 3.6: Frequency spectrum graph of a sin wave at 200 Hz and amplitude 0.5

Figure 3.6a shows a Dirac's delta centered in 200 Hz, which is difficult to read because of the high full scale value; for this reason in Figure 3.6b a more readable representation of the same signal, in logarithmic scale, is showed.

While, a sin wave at 500 Hz, starting with phase 0 and an amplitude of 0.5, is obtained with the command

```
{ SinOsc.ar(500, 0, 0.5) }.play;
```

And generates the frequency spectrum graph showed in Figure 3.7

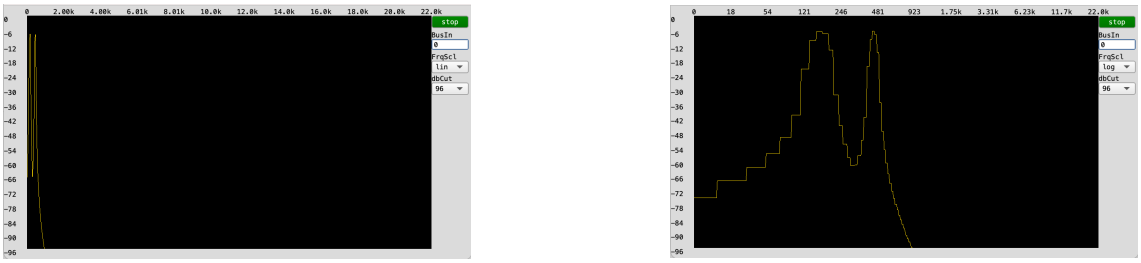


(a) Linear scale (b) Logarithmic scale

Figure 3.7: Frequency spectrum graph of a sin wave at 500 Hz and amplitude 0.5

In order to understand how easy is to read a frequency spectrum graph and, thus, how

powerful this tool is, the Figure 3.8 shows the frequency spectrum graph of a sin wave at 200 Hz and a sin wave at 500 Hz while playing at the same time.



(a) Linear scale

(b) Logarithmic scale

Figure 3.8: Frequency spectrum graph of a sin wave at 200 Hz and a sin wave at 500 Hz played simultaneously

Spectrogram

A spectrogram is a visual representation of data, usually sounds, voices or waves in general, in both time and frequency domain. For obtaining a spectrogram usually a certain wave is subject to successive Fourier transforms of data using a sliding window. A spectrogram indicates also the signal strength or, in general, its loudness.

This tool is widely used for wave visualization: it consists in a two-dimensional graph, with an optional third dimension represented by colors. Time ranges from left to right along the horizontal axis, while frequency varies from bottom to the top along the vertical axis. The scale can be both logarithmic or linear. In this section only spectrograms with logarithmic scale will be employed. [11]

In this section, spectrograms will be often used to show how a certain sound is represented, in addition to classical frequency-only domain representation.

For example, a white noise with frequency 1 Hz, subject to a Low Pass Filter which cut-off frequency is a linear ramp ranging from 20 to 12250, is obtained from this supercollider script [13]:

```
{ LPF.ar(WhiteNoise.ar(1), LFNoise1.kr(1).range(20,12250)) }.play
```

And, produces the spectrogram reported in Figure 3.9 when played for about 20 seconds.

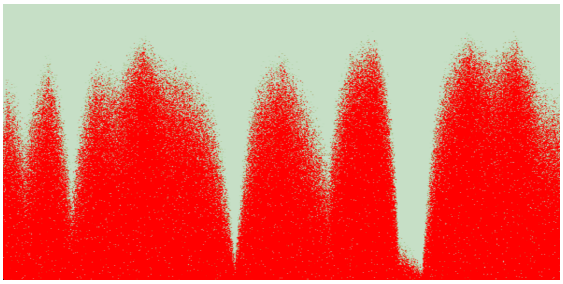


Figure 3.9: Spectrogram of a White Noise subject to a Low-Pass filter

Also the spectrogram graphs generator is available on supercollider IDE as a plug-in and will be taken into consideration during the analysis of the different sound generation approaches.

3.2.1 Device-side decoding of CAN messages

In order for the sound generation device to do its computations via the algorithm implemented in supercollider, the device has to receive the CAN messages that are coming across the network, to filter them and to send the informations the contained to the supercollider script.

In order to do so, a script has to be run device-side.

First of all, the script has to parse the CAN bus database, that must be coherent with the one used by the Electronic Control Unit of the vehicle. As a matter of fact, if the two databases were different, the communication between the ECU and the Raspberry Pi 4, i.e. the device which used for sound generation, would not be possible at all.

A function for wrapping the `send_message()` method of "pythonosc" library has been defined. Using this new function, the script is able to send all the different type of messages using only one method, generalizing the entire code.

The method `receive_all` implements the set-upping process of the Open Sound Control network and the initialization of the CAN bus shield interface. After that, it implements a endless while-loop that receive all the messages that are transmitted on the network, filter them, extracts the information needed, decodes them using the CAN bus database that has been parsed before and, in the end, send all these informations to the supercollider server. In the case the user interrupts the process while it is still running (Ctrl+C is pressed), an exception is raised and a message containing the "Shutdown" is sent.

The UDP client, which is relative to the Open Sound Control communication, is then initialized using "127.0.0.1" as default IP address and "57120" as default port. After this operation the UDP client will be setup and up to run.

After setting up the UDP client for OSC communication, the CAN bus shield interface must also be initialized specifying the bus type, channel and bit rate.

The first operation to perform is the receiving of the messages from the CAN bus shield interface and to make sure that the message is not empty. Sometimes, when the communication is just set-upped, a CAN frame can be received as empty, for this reason the script makes sure to skip that meaningless message and to go on receiving the others. After that, a decoding operation is performed and the "arbitration_id" and the "data" are decoded from the message. After that all the items of the message are stored. All of them will be filtered in the next operations.

The "arbitration_id" is the first field to be checked, in fact it is compared to the ID reported in Table 2.1 and, after that, its content is checked: if the data contained in the frame corresponds to "IGN_LK", it means that the vehicle has just been shutdown and, thus, an OSC message corresponding to this action must be sent. Instead, if the "START" message is obtained, it means that the ignition key has been moved to angle for igniting the vehicle and, as a consequence, a OSC message containing the ignition directive has to be send, in order to make the supercollider script start reproducing sound.
A string containing the timestamp is printed on the console, for debug and visualization purposes only.

For what concerns the vehicle speed, the message with "arbitration_id" corresponding to "0x3" is filtered and, then, the speed information is retrieved. This information is refactored, according to the ranges of vehicle speed signal, showed in Table 2.4. After retreiving the information, a Open Sound Control message is sent through the method `send_osc()`.

The same steps are followed for what concerns the Gas Pedal Position information: the message is retrieved comparing its "arbitration_id" and then it is sent to the supercollider script.

3.2.2 The algorithmic composition

Algorithmic composition is a technique for sound generation that consists in taking a variety of signals and combining them to obtain a brand new sound, much like it is done with canonical composition with real instruments.

Taking as an example a guitar, or a piano, in order to obtain a sound, which can be a chord, two or more keys or two or more strings have to be played at the same time.

In algorithmic composition the principle is the same, the only difference is that we act directly on signals and their waveforms, using a programming language, which is supercollider, to make the computer emit sounds through an actuator, tipically a loudspeaker. From a computational point of view this type of composition is quite efficient, due to the fact that all the sound generation are based on the Fast Fourier Transform (FTT)

algorithm, which grants a light execution of every sound-related function.
In order to obtain a sound capable of emulating an Internal Combustion Engine sound, a four stroke engine design has been taken into consideration, starting from the description in [9] and [26].

The first thing to do is to declare a set of OSC listeners, which are a objects, that will be often called improperly "socket", that receive the messages from the python scripts that decodes the CAN bus msg flow. The first one listens to the ignition messages from the python script. Once an ignition message is received, declares a new instance of the engine sound, setting the speed at 0.

Subsequently a socket for listening the speed message is declared, that will set the new speed each time it receives a new speed message, in order to change the value of one of the input of the algorithm, that will adapt its behaviour and, in the end, will emit a different sound according to the value received.

In addition an OSC listener is declared in order to receive the informations concerning the gas pedal position. When the algorithm receives this type of messages as input, it changes the value of `wguideFeedback`, `mixParabolic`, `parabolicDelay`, `warpDelay` and `waveguideWarp`, which are some parameters that affect the delay with which each buffer is read.

In the end, an OSC socket is opened in order to listen when a shutdown message is received. This action leads to the interruption of the sound generation.

The outline schema of the algorithmical composition method implied into this section is described in Figure 3.10 .

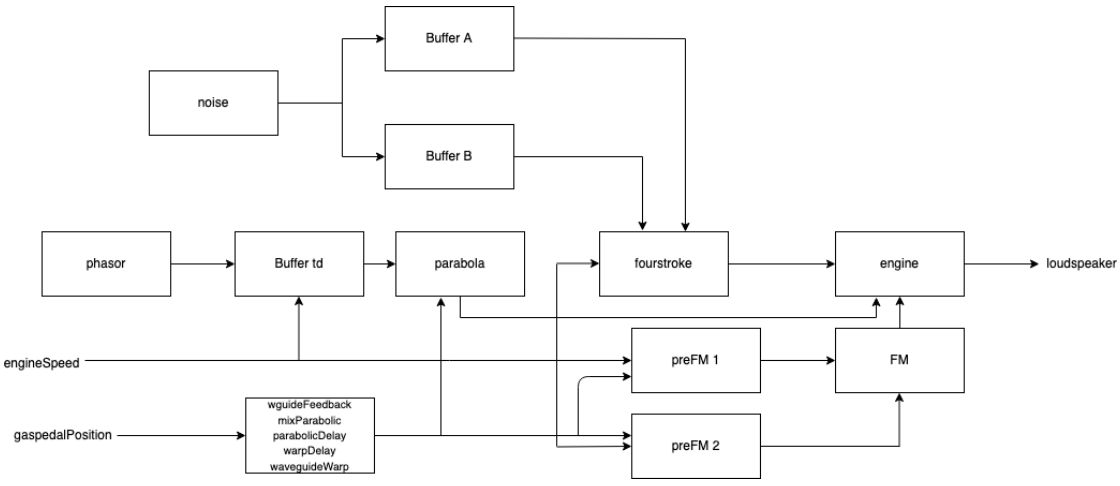


Figure 3.10: Outline schema of the algorithm

First of all, three local buffers have been defined, having 44100 frames each and only one channel.

Then, a white noise is defined and filtered by two One Pole filters which coefficient is $e^{-2\pi*20*Sample\_Duration}$. The typical use of a One Pole Low-pass filter, also known as One Pole filter, is to pass signals with a frequency lower than a selected cutoff frequency and attenuate all the other signals. It's extremely cheap, computationally, so it's the perfect choice when just a bit of smoothing is needed. In audio composition One Pole filters are usually employed for filtering on parameters instead of on the audio directly. [17]
A One Pole filter implements the formula

$$out(i) = ((1 - abs(coef)) * in(i)) + (coef * out(i-1))$$

where "in" is the input signal and "coef" is the second parameter of the function.

In this way the generated signal is represented in Figure 3.11 in the Frequency spectrum graph. At the end, the signal is written onto `bufferA` and `bufferB`.

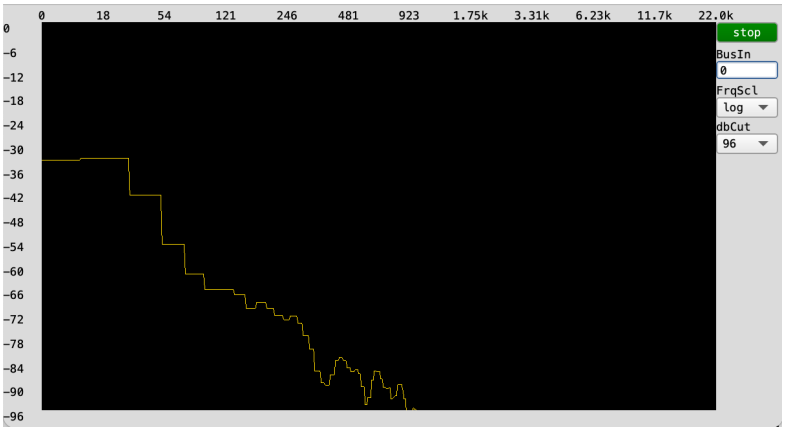


Figure 3.11: Frequency spectrum graph of a White noise subject to two One Pole filters

Also a phaser is defined. A phaser, in electronic sound processing, is a filter that consists in a series of peaks and troughs in the frequency domain, as it is shown in Figure 3.12. In this case it is realized using a sawtooth oscillator which frequency depends on `engineSpeed` variable. `engineSpeed` is the variable that changes when the speed of the vehicle changes, in fact it is directly modified by the OSC messages. The phaser wave is then stored in `bufferTd` variable.

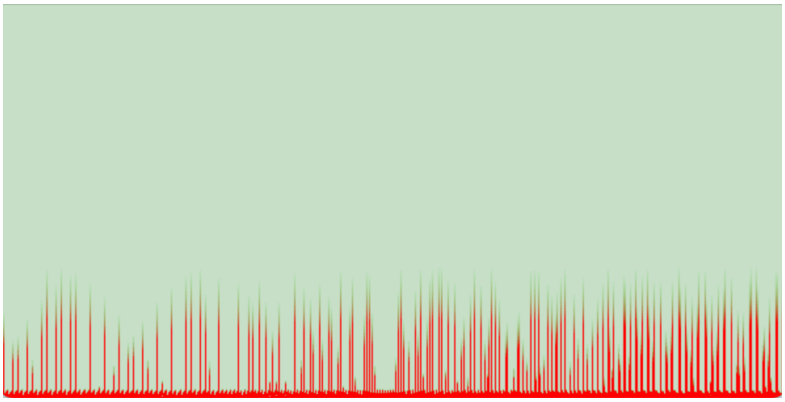


Figure 3.12: Spectrogram of a sawtooth signal

In Figure 3.13 the Frequency spectrum graph of the sawtooth signal is showed.

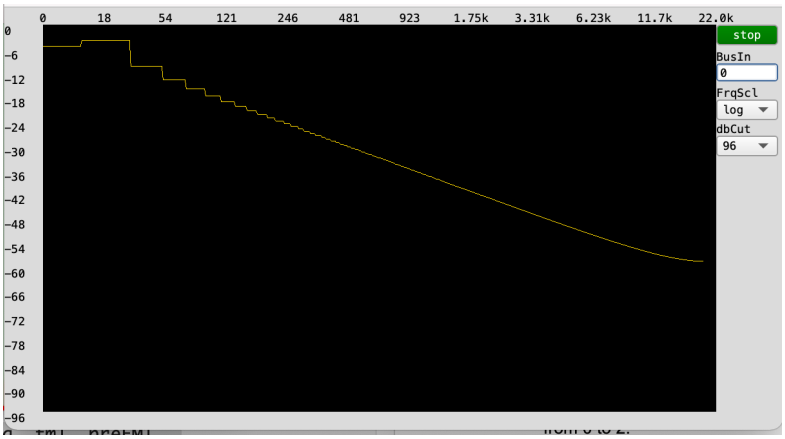


Figure 3.13: Frequency spectrum graph of a sawtooth signal

`fourstroke` variable reads from `bufferA`, containing the informations about the white noise, with a delay of $[5, 10, 15, 20]/1000$ and a cubic interpolation. Subsequently the saw tooth contained in the `phaser` variable is added.

After that, the signal is multiplied by a scale factor, then multiplied by the white noise contained in `bufferB` variable, which is read with a delay depending from `engineSpeed` variable and a One Pole filter is applied.

Now the sound is starting to resamble the one of a internal combustion engine. The frequency spectrum graph in Figure 3.14 shows the behaviour of the sound wave when `engineSpeed` corresponds to a value which is near half of the maximum speed value. It is worth to notice how low frequencies are enhanced while the high frequencies' gain is pulled down.

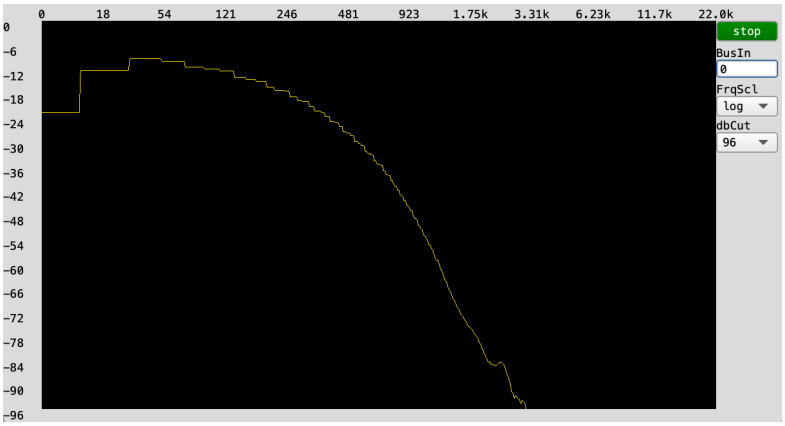


Figure 3.14: Frequency spectrum graph of the "fourstroke" signal

In this last part of the algorithm, `bufferTd` is read and scaled by 2π, then shifted, in addition the input `engineSpeed` is taken and, then, a One Pole filter is applied to the signal, with a pole in $0.2 * SampleDur$. The frequency spectrum graph of the obtained signal is showed in Figure 3.15.

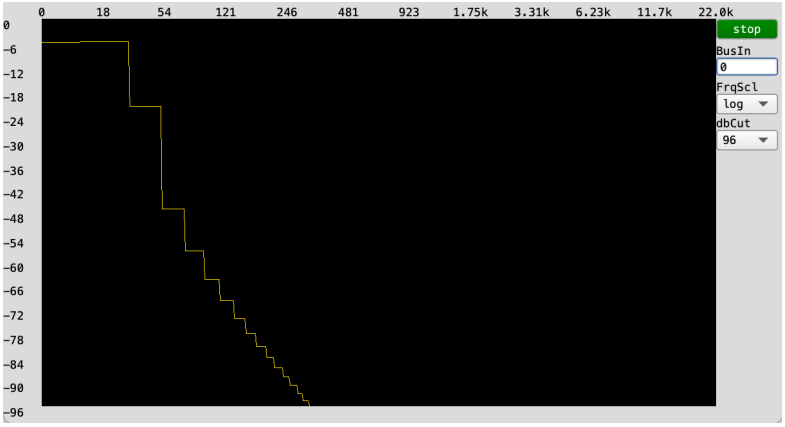
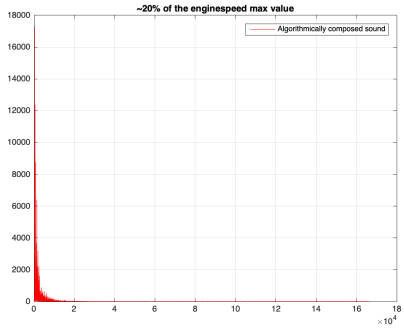
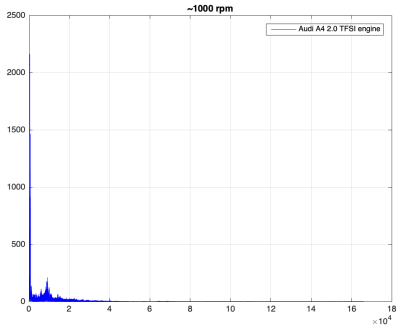


Figure 3.15: Frequency spectrum graph of the "fm1" and "fm2" signals

In order to compare the results obtained with the algorithmic composition, the sound of an Audi A4 Allroad equipped with a 2.0 Turbocharged Fuel Stratified Injection (or TFSI) petrol engine has been taken into account.



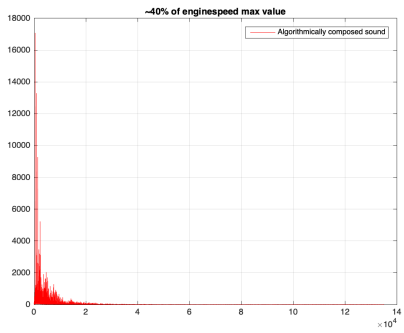
(a) Algorithmically composed sound



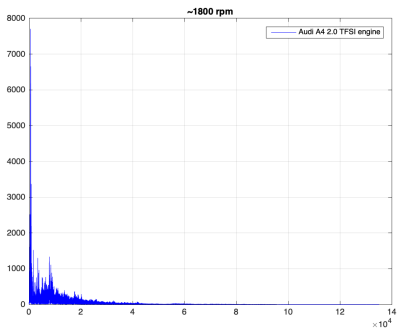
(b) 2.0 TFSI engine at around 1000 rpm

Figure 3.16: Frequency spectrum graph of the engine sound at low speed

As we can see from Figure 3.16 the gain peak is at low frequencies (around 500-700 Hz), the behaviour in frequency of the two signals is quite similar. The gain of the algorithmical composed sound is significantly greater with respect to the sound of real engine because of the recording conditions of the second.



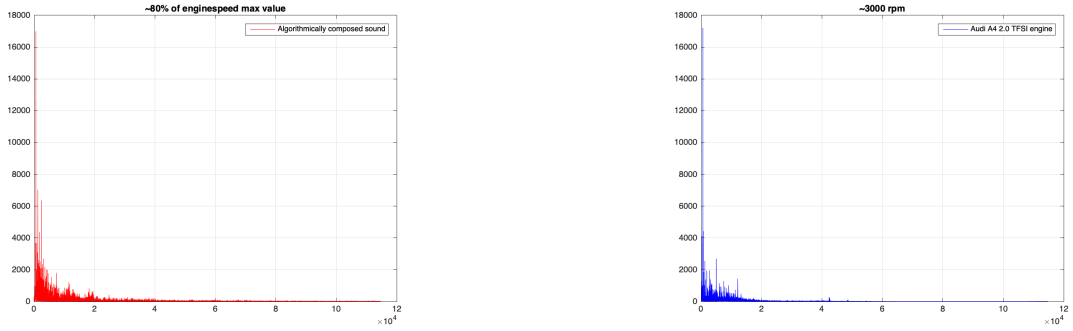
(a) Algorithmically composed sound



(b) 2.0 TFSI engine at around 1800 rpm

Figure 3.17: Frequency spectrum graph of the engine sound at medium speed

In Figure 3.17 we can see that also frequencies in the range between 1000 Hz and 1500 Hz start to become significant. However the overall behaviour of the algorithmically composed sound remains consistent with the behaviour of the real sound, even if the gain is much more greater.



(a) Algorithmically composed sound

(b) 2.0 TFSI engine at around 3000 rpm

Figure 3.18: Frequency spectrum graph of the engine sound at high speed

In the end, when the speed is high, the frequencies of the algorithmically composed sound start to spread all across the spectrum, in fact as we can observe in Figure 3.18, now also the frequencies in the range between 1500 and 2200 Hz are significant. In addition, also the gain of the real engine sound is now similar to the sound of the algorithmically composed one.

Another comparison is given by the spectrogram visualization of the real sound and the algorithmic generated one: the overall result of the superposition of each of the signal described before is reported in the spectrogram in Figure 3.19.

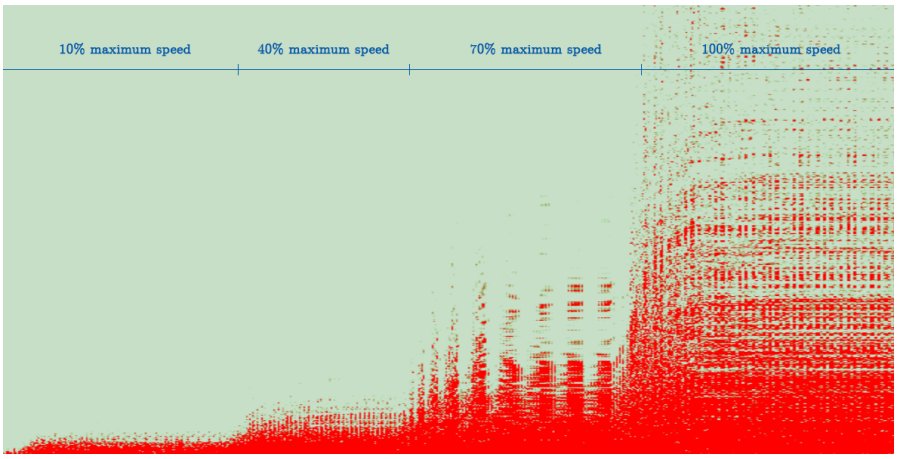


Figure 3.19: Spectrogram of the final results

Figure 3.19 is divided in four section, each one corresponds to a different range of vehicle speed. In fact it has been generated when the vehicle is at low speed, then the speed has been gradually increased until it reached almost the maximum value.

In Figure 3.20 the spectrogram of the soundwave of another real internal combustion engine is showed. The graph is divided in sections, each one corresponding to a specific

state of the engine, 800 rpm, 1800 rpm and 2700 rpm.

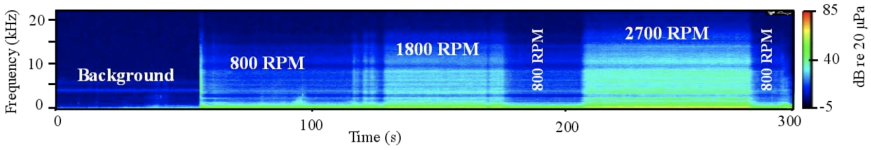


Figure 3.20: Spectrogram of the sound of a real Internal Combustion Engine [19]

For what concerns the braking alerting sound for the regenerative brake notification, a Sawtooth oscillator with frequency of around 100 Hz has been employed. This sound is generated every time the script detects that the pedal has been pressed and reproduces a sound that fades out after two seconds and finishes completely after five seconds.

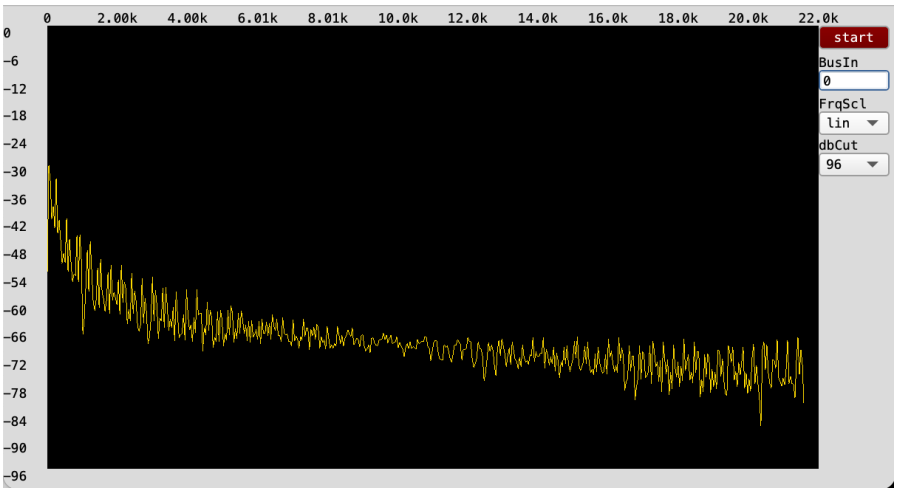


Figure 3.21: Frequency spectrum graph of the sound generated after braking in linear scale

In Figure 3.21 the behaviour in the frequency domain of the sound generated after the pressure of the braking pedal is showed, one instant before starting fading out.

Advantages and disadvantages of classical algorithmic composition

Algorithmic composition is a powerful tool for sound generation, in fact, through the superposition and the refactoring of different signal a decent sound emulating the real sound of an Internal Combustion Engine has been obtained. The spectrogram of the overall result shows how good the simulation, in fact its behaviour in time-frequency is very similar to the one of the real internal combustion engine.

On the other hand, debugging this code is very hard and complex: adding new features might be complicated, in particular in the case when a car manufacturer wants to obtain a

brand new sound starting from something that does not resamble the exact characteristics of a classical Internal Combustion Engine, but instead wants to create a brand new sound that only follows the behaviour of the engine but not its sound footprint. A huge work in reverse engineering has to be done in order to obtain something like that and the impossibility to use physical instruments makes the composition method even more inaccessible.

For this reason a new method has to be analyzed, in order to increase the possibility to explore new sounds and, more important, to make the debugging of the code easier.

3.2.3 The granular synthesis

Granular synthesis is a completely different approach to sound generation. In sound synthesis, the generated timbre depends on a set of user defined parameter. In most implementations, these parameters can be manipulated at runtime to produce dynamic variations of the timbre. Common parameters in granular synthesis implementations are the size of audio grains, the amplitude envelope, the grain density, and the grain sequencing mode. The method of generating grains or extracting these from selected sources also has a significant impact on the sound, however. [10]

Granular synthesis consists in choosing a short interval from an existing sound and looping it [18], acting on its phase, pitch, frequency and duration in order to obtain something that can follow a trend, specified by the CAN bus messages as the previous approach.

Each of these intervals is called "grain", for this reason the technique is called "granular". Each grain contributes to the sound synthesis with its magnitude spectrum only. Phase is reconstructed using spectrogram inversion techniques which support real-time computation. With this method, windowing and overlapping grains of regular size is no longer required. [10] Through this method we can obtain a sound that is more rich and alive [25], possibly combining it with the previous approach for obtaining brand new sounds.

For this approach, three different synthesizers have been defined, each one taking an already existing sound as a source. These sounds must have particular characteristics in order to be chosen: first of all, they must not contain part when the volume is zero or is particularly low, secondly they are preferred to be low tuned.

The first sound is the trumpet of an elephant, which frequency spectrum graph is presented in Figure 3.22, this time in linear scale.

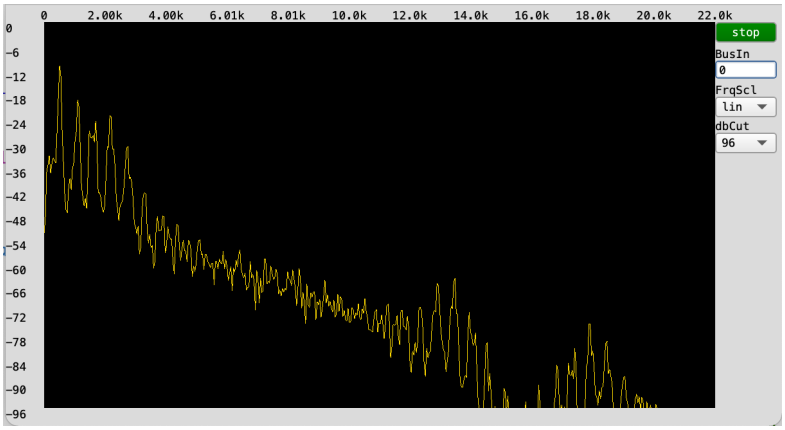


Figure 3.22: Frequency spectrum graph of an elephant trumpet

The second sound that has been employed is a radio interference, which frequency spectrum graph is presented in Figure 3.23, again in linear scale.

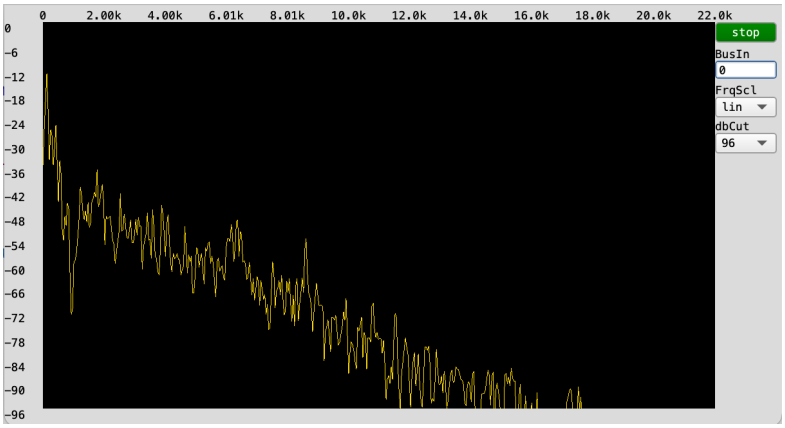


Figure 3.23: Frequency spectrum graph of a radio interference

These two sounds will be employed to form the output commanded by the speed of the vehicle, and will blend.

In the end, the third sound that will be employed is a C chord played by a stringed instrument, like a violin, with some reverb, which frequency spectrum graph is showed in Figure 3.24.

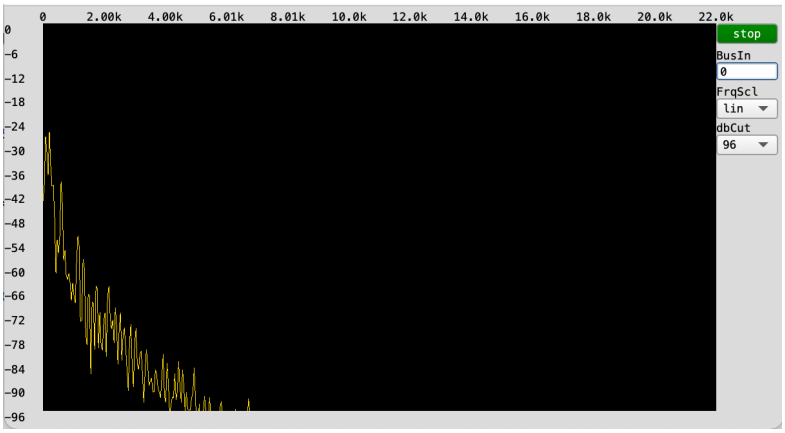


Figure 3.24: Frequency spectrum graph of a C chord

Also this last sound will be stored in a variable, and will be employed to represent the torque generated by the electric vehicle, taking as input the gas pedal position. Starting from these three sound files, three synthesizer has been defined, two referring to the speed of the vehicle, and one referring to the gas pedal position. The first sound is played simply in loop, without any modification, while the second sound is taken as a grain, which is triggered by an Impulse oscillator. The last synthesizer will be based on the gas pedal position, and takes the third sound as a grain and stores it into a buffer which will be triggered by an Impulse oscillator.

In this approach, also the action corresponding to the receiving of a message by the OSC socket are significantly changed, in fact, the core of the algorithm is moved inside the definition of the OSC socket.

The first OSC socket listens to the ignition signal, and after the receiving of that specific signal, instances all the new Synthesizers and associates each of them to the corresponding buffer.

The variable `msg` contains the information about the vehicle speed sent by the python script that decodes the CAN messages from the CAN interface. The method `LinLin` generates a linear mapping between the value contained in `msg` and `speedrate`, `duration`, `playbackrate` and `startPos`. `LinLin` transforms the value it receives in an output value in the specified range. `rate`, as an example, can range between 1 and 1.5: when the value increases the playback rate of the sampled sound increases. The same is for `duration`, which causes the duration of the sample to increase, or the `startPos`, which shifts forward the playback position for the grain to start along the wav file (0 is the beginning, 1 is the end of file).

In the end, also the gas pedal position value is contained into the variable `msg`, and then is mapped linearly in order to modify the behaviour of `speedrate`, `duration`, `playbackrate` and `startpos` of the synthesizer.

In this way we can reproduce a sound that is similar to the one of a real Internal combustion engine, even with some peculiarities, given by the fact that the grain used as a starting point for the generation of the sound are sampled by already existing sound signals, which characteristics varies with the behaviour of the vehicle.

Advantages and disadvantages of granular synthesis

Granular synthesis is surely a method with which creativity can be fully expressed, in fact starting from any existing sound and isolating a short interval, brand new sound effects can be obtained. Even changing the mapping of signal new results can be explored, for example using a linear to exponential mapping or, again, dividing the speed ranges in sections in which the mapping can vary for giving the sound generation some brand new characteristics.

However, using this method, multiple synthesizer have to be used, while in the algorithmical composition only one was used. This can lead to an increasing complexity in the sound generation that can stress the computation capabilities of the device and that can get in the way of scalability.

Part III

Conclusions

Chapter 4

Conclusions

At the beginning the goal of this Master Thesis project was to design and implement a sound generation algorithm that was able to reproduce the sound of an Internal Combustion Engine into the cockpit of an Electric Vehicle, in order to give back to the driver all the feelings it lost during the transition from the traditional fossil fuel powered vehicles to the electric ones, that are quite inaudible and silent, in particular at higher speeds.

The first phases of the development consisted in doing some researches looking for the most suitable hardware to use in order to implement the project: a Raspberry Pi 4, an Innomaker USB2CAN and a STM32L476RG by STMicroelectronics have been selected.

After that, the programming language have been chosen: python, for its simple usage and variety of library, supercollider, for its sound generation performance, kivy, for its integration with python, have been pick up.

In the end, the communication protocols have been identified: CAN, which is the standard in the automotive world and is widely implemented in various plug-in libraries in Linux and python, and OSC, for the communication to and from the sound generating server.

The most significant challenge was to establish the communication between the sound generating server and the vehicle's informations provided by the Electronic Control Unit of the car.

However, a new challenge gradually came to the light: while developing all the various scripts for making the connection between the device for generating sound and the ECU, the non availability of a real vehicle quickly became a paramount problem. For this reason the focus shifted towards the development of a method that was able to reproduce the behaviour of the vehicle, granting a valuable method for emulating all the relevant signals provided by the ECU.

Most already existing software for vehicle simulation are closed source, proprietary,

single-platform and, most important, expensive; for this reason in this project a considerable amount of time has been spent trying to implement brand new methods in order to get a relevant result.

A variety of method, in fact, has been explored: first of all, a trivial simulation via keyboard, that rapidly evolved into a simulation using a Time-of-Flight sensor and became, in the end, a complete desktop application with GUI, able to emulate all the needed signals.

For what concerns the sound generation algorithm, two methods have been described: the first one is the algorithmic composition, which script has been originally taken from [26] and adapted in order to work with CAN bus signals as input, and the granular synthesis, a brand new technique that can lead to yet unexplored scenarios in sound generation.

In the end, all these goals have been reached with success, and the project has achieved good results, becoming a nice starting point over which further developments can be explored.

4.1 Future developments

As showed in the previous chapters, especially Granular Synthesis technique can lead to a huge panorama of possibilities: in this project the goal has been to emulate the behaviour, and thus the sound, of an Internal Combustion Engine, neglecting completely the branding purpose. However from a car maker point of view branding can be crucial. For this reason, using Granular Synthesis each automotive manufacturer can explore new possibilites of sound, ranging from the most classical ones to more exotic sounds, that nothing have in common with the sounds we are used to, but that can get closer to that kind of sounds we ear from sci-fi movies.

From the point of view of validation, huge progress can be made: in this project the final designed has not been validated by beta testers, since the entire project has been developed in remote. However, testing and validation is necessary, in order to know directly from the driver's experiences if it perceives the correct feelings while pushing the gas pedal, or while decelarating or while braking. The next step, in fact, should be to make all the scripts run while the Raspberry Pi 4 is present in the vehicle, in order to get validated by the users. Of course, if perception were not as good as desired, design should be reconsider and more analysis should be done, changing parameters as frequency, volume and pitch of the sounds.

Also safety and working conditions should be considered: of course, vehicles must be usable in the most various scenarios, and this entire project is based on loudspeaker as actuator, which emitted sound can vary in function of temperature or pressure [5]. For this reason the operation and the working conditions must be evaluated using a real vehicle in all the different use cases scenarios, in order to explore all the possible devices that can

be employed as actuator, and, finally, chose the proper one, valuating the most efficient.

Furthermore, miniaturizing can be also taken into consideration: a Raspberry Pi 4 can be quite invasive to place into a vehicle cockpit, especially in those vehicles that are dedicate to city driving, in which every centimeter counts, considering also that Raspberry Pi 4 is also equipped with its cooling system made up of a fan and an aluminum enclosure, which increments again its dimensions. For this reason, all the script can be ported to the system infotainment built into the car, in order to avoid other useless obstructions.

Appendices

Appendix A

Scripts for vehicle simulation

A.1 Simulation via keybord

```
import sys,tty,termios
import time
from pythonosc import osc_message_builder
from pythonosc import udp_client
import socket
import can
import cantools
from pprint import pprint
from datetime import datetime


bustype = 'socketcan'
channel = 'can0'

class _Getch:
    def __call__(self):
            fd = sys.stdin.fileno()
            old_settings = termios.tcgetattr(fd)
            try:
                tty.setraw(sys.stdin.fileno())
                ch = sys.stdin.read(3)
            finally:
                termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
            return ch

def get():
    can_bus = can.interface.Bus(channel, bustype=bustype)
    db = cantools.database.load_file('/path/to/dir/Can_db.dbc')

    speed_message = db.get_message_by_name('BRAKE')
    ignition_message= db.get_message_by_name('COMMAND')
    gaspedal_message=db.get_message_by_name('ENGINE')
```

```
bus = can.interface.Bus(channel=channel, bustype=bustype)

global speed
global ignited
global gaslevel

try:
    while True:
        inkey = _Getch()
        k=inkey()

        if ignited == 1:
            if k=='\x1b[A':
                if(gaslevel == 0):
                    gaslevel = gaslevel + 0.5
                if (gaslevel > 0 and gaslevel < 40):
                    gaslevel = gaslevel + 1
                if (gaslevel >=40 and gaslevel <100):
                    if(gaslevel + 2.5 > 100):
                        gaslevel=100
                    else:
                        gaslevel = gaslevel + 2.5

                if ((speed+1)<512):
                    speed=speed+gaslevel/100

            elif k=='\x1b[B':
                if (gaslevel > 0 ):
                    gaslevel = 0

                if((speed-3)>0):
                    speed=speed-3

            elif k=='\x1b[C':
                shutdown_data = ignition_message.encode({[redacted]: 0,
                                                         [redacted]: 0,
                                                         [redacted]: 0,
                                                         [redacted]: 0,
                                                         [redacted]: 0,
                                                         [redacted]: 0,
                                                         [redacted]: 0,
                                                         [redacted]: 1,
                                                         [redacted]: 0}
                                                         )
                message = can.Message(arbitration_id=ignition_message.frame_id,
                                      data=shutdown_data)
                print(datetime.now(), "- Engine shutdown")
                can_bus.send(message)
                speed=0
                return

            data = speed_message.encode({[redacted]: 0,
                                         [redacted]: 0,
```

```
                                        [redacted]: 0,
                                        [redacted]: 0,
                                        [redacted]: 3,
                                        [redacted]: speed,
                                        [redacted]: 0,
                                        [redacted]: 0,
                                        [redacted]: 0}
                                        )
                    message = can.Message(arbitration_id=speed_message.frame_id,
                                          data=data)
                    print(datetime.now(), "- Speed = ", speed, "km/h - Gas pedal
                                          position: ", gaslevel)
                    can_bus.send(message)

                    data = gaspedal_message.encode({[redacted]: gaslevel})
                    message= can.Message(arbitration_id=gaspedal_message.frame_id,
                                          data=data)
                    can_bus.send(message)



                if k=='\x1b[D':
                    ignition_data = ignition_message.encode({[redacted]: 0,
                                                            [redacted]: 0,
                                                            [redacted]: 0,
                                                            [redacted]: 0,
                                                            [redacted]: 0,
                                                            [redacted]: 0,
                                                            [redacted]: 4,
                                                            [redacted]: 0}
                                                            )
                    message = can.Message(arbitration_id=ignition_message.frame_id,
                                          data=ignition_data)
                    print(datetime.now(), "- Engine ignited")
                    can_bus.send(message)
                    ignited = 1
        except KeyboardInterrupt:
            pass



if __name__=='__main__':
    gaslevel=0
    speed=0
    ignited = 0
    get()


    sys.exit(0)
```

A.2 Simulation via ToF sensor

```
import serial
import argparse
import random
import time
from pythonosc import osc_message_builder
from pythonosc import udp_client
import socket


# This function wraps send_message in OSC
def send_osc(client, listener, value):
    client.send_message(listener, value)




PORT = '/dev/ttyACM0'
ser = serial.Serial(PORT, 115200, timeout=10)
parser = argparse.ArgumentParser()
parser.add_argument("--ip", default='127.0.0.1',
                    help="The ip of the OSC server")
parser.add_argument("--port", type=int, default='57120',
                    help="The port of the OSC server")
args = parser.parse_args()
client = udp_client.SimpleUDPClient(args.ip, args.port)

while (1==1):
    if(ser.in_waiting >= 4):
        serialString = ser.read(4)

        if (serialString[0]==171):
            gaspedalpos = serialString[1]
            speed = serialString[2]

            print("Gas pedal position: ", gaspedalpos, " - Speed: ", speed)
            data = speed_message.encode({[redacted]: 0,
                        [redacted]: 0,
                        [redacted]: 0,
                        [redacted]: 0,
                        [redacted]: 3,
                        [redacted]: speed,
                        [redacted]: 0,
                        [redacted]: 0,
                        [redacted]: 0}
                        )
            message = can.Message(arbitration_id=speed_message.frame_id, data=data)
            can_bus.send(message)

            data = gaspedal_message.encode({[redacted]: gaslevel})
            message= can.Message(arbitration_id=gaspedal_message.frame_id, data=data)
```

```
            can_bus.send(message)
```

A.2.1 Transfer function implementation

```
double tfcnDcMotorStep(double u)
{
  static double ym2 = 0.0;  /* y(n-2) */
  static double ym1 = 0.0;  /* y(n-1) */
  static double y = 0.0;    /* y(n)   */

  static double um2 = 0.0;  /* u(n-2) */
  static double um1 = 0.0;  /* u(n-1) */

  y = (1.511*ym1 - 0.5488*ym2) + (0.002059*um1 + 0.001686*um2);

  ym2 = ym1;
  ym1 = y;

  um2 = um1;
  um1 = u;

  return y;
}
```

A.3 TouCAN desktop application

```
from logging import disable
from posixpath import expanduser
from kivy.core import text
from kivy.uix.boxlayout import BoxLayout
from kivy.app import App
from kivy.uix.gridlayout import GridLayout
from kivy.uix.relativelayout import RelativeLayout
from kivy.uix.label import Label
from kivy.uix.image import Image
from kivy.uix.button import Button
from kivy.uix.textinput import TextInput
from kivy.uix.slider import Slider
import os
import can
from threading import Thread
import time
from pythonosc import osc_message_builder
from pythonosc import udp_client
```

```
import socket
import can
import cantools

class TouCAN(App):
    def build(self):
        self.db = cantools.database.load_file('/path/to/dir/Can_db.dbc')
        self.speed_message = self.db.get_message_by_name('BRAKE')
        self.ignition_message= self.db.get_message_by_name('COMMAND')
        self.gaspedal_message= self.db.get_message_by_name('ENGINE')
        self.brake_message= self.db.get_message_by_name('BRAKE')

        self.t= Thread(target=self.pollThread)
        self.running=False
        self.can0 = None
        self.window = GridLayout()

        #add widgets to window
        self.window.cols = 3
        self.window.size_hint = (.9, .95)
        self.window.pos_hint = {"center_x": 0.5, "center_y":0.5}

        canRow = BoxLayout(orientation='vertical')

        # image widget
        canRow.add_widget(Image(source="toucan.png"))

        # initialize can button
        canRow.initializebutton=Button(
                            text="Initialize CAN",
                            size_hint = (1,None),
                            bold = True,
                            background_color = '#3437eb'
        )
        canRow.initializebutton.bind(on_press=self.OnInitializeButtonClick)
        canRow.add_widget(canRow.initializebutton)


        self.window.add_widget(canRow)


        # start engine button
        self.startengine = Button(
                            text="Start Engine",
                            size_hint = (1,None),
                            bold = True,
                            background_color = '#33ff00',
                            disabled=True
        )
        self.startengine.bind(on_press=self.OnStartButtonClick)
        self.window.add_widget(self.startengine)

        #shutdown engine button
```

```
        self.shutdownengne = Button(
                            text="Shutdown Engine",
                            size_hint = (1,None),
                            bold = True,
                            background_color='#ff0000',
                            disabled = True
        )
        self.shutdownengne.bind(on_press=self.OnShutdownButtonClick)
        self.window.add_widget(self.shutdownengne)

        # gas pedal position slider
        self.gaspedalslider = Slider(
                            min=0,
                            max = 100,
                            value = 0,
                            orientation = 'vertical',
                            disabled = True
        )
        self.gaspedalslider.bind(value=self.OnGasPedalSliderValueChange)
        self.window.add_widget(self.gaspedalslider)

        # speed slider
        self.speedslider = Slider(
                            min=0,
                            max = 511,
                            value=0,
                            orientation = 'vertical',
                            disabled = True
        )
        self.speedslider.bind(value=self.OnSpeedSliderValueChange)
        self.window.add_widget(self.speedslider)

        # braking pedal button
        self.brakepedalbutton = Button(
                            text="BRAKE!",
                            size_hint = (1,None),
                            bold = True,
                            background_color='#ffd000',
                            disabled = True
        )
        self.brakepedalbutton.bind(on_press=self.OnBrakingPedalButtonClick)
        self.brakepedalbutton.bind(on_release=self.OnBrakingPedalButtonRelease)
        self.window.add_widget(self.brakepedalbutton)

        # gas pedal position label
        self.gaslabel = Label (
                        text = 'Gas pedal position:
                                    '+str(int(self.gaspedalslider.value)),
                        font_size=20,
                        color = '#b4eb34'

                        )
        self.window.add_widget(self.gaslabel)
```

```
        # vehicle speed label
        self.speedlabel = Label (
                        text = 'Vehicle speed: ' + str(int(self.speedslider.value)),
                        font_size=20,
                        color = '#00aaff'
                        )
        self.window.add_widget(self.speedlabel)

        # braking level label
        self.brakelabel = Label (
                        text = 'Braking status: '+ str(self.brakepedalbutton.state),
                        font_size=20,
                        color = '#e600ff'
                        )
        self.window.add_widget (self.brakelabel)

        return self.window


    def OnGasPedalSliderValueChange(self,instance, value):
        self.gaslabel.text = 'Gas pedal position: ' +
                                            str(int(self.gaspedalslider.value))

    def OnSpeedSliderValueChange(self,instance, value):
        self.speedlabel.text = 'Vehicle speed: ' + str(int(self.speedslider.value))


    def OnBrakingPedalButtonClick(self, instance):
        self.brakelabel.text = 'Braking status: ' + str(self.brakepedalbutton.state)
        braking_data = self.brake_message.encode({[redacted]: 1)
        message = can.Message(arbitration_id=self.brake_message.frame_id,
                                            data=braking_data)
        self.can0.send(message)


    def OnBrakingPedalButtonRelease(self, instance):
        self.brakelabel.text = 'Braking status: ' + str(self.brakepedalbutton.state)



    def OnStartButtonClick(self, instance):
        self.shutdownengne.disabled=False
        self.speedslider.disabled=False
        self.brakepedalbutton.disabled=False
        self.gaspedalslider.disabled=False
        self.startengine.disabled=True
        if(self.running==False):
            ignition_data = self.ignition_message.encode( {[redacted]: 0,
                                            [redacted]: 0,
                                            [redacted]: 0,
                                            [redacted]: 0,
                                            [redacted]: 0,
```

```
                                        [redacted]: 0,
                                        [redacted]: 0,
                                        [redacted]: 4,
                                        [redacted]: 0}
                                        )

        message = can.Message(arbitration_id=self.ignition_message.frame_id,
                                        data=ignition_data)
        self.can0.send(message)
        self.running=True
        self.t.start()


def OnInitializeButtonClick(self, instance):
        os.system('sudo ip link set can0 type can bitrate 1000000')
        os.system('sudo ifconfig can0 up')
        self.can0=can.interface.Bus(channel='can0', bustype='socketcan')

        if(self.can0):
            instance.text = "CAN interface running"
            instance.disabled=True
            self.startengine.disabled=False

def OnShutdownButtonClick(self, instance):
    self.shutdownengne.disabled=True
    self.speedslider.disabled=True
    self.speedslider.value=0
    self.brakepedalbutton.disabled=True
    self.gaspedalslider.disabled=True
    self.gaspedalslider.value=0
    self.startengine.disabled=False
    if(self.running==True):
        self.running=False
        self.t.join()
        shutdown_data = self.ignition_message.encode({[redacted]: 0,
                                        [redacted]: 0,
                                        [redacted]: 0,
                                        [redacted]: 0,
                                        [redacted]: 0,
                                        [redacted]: 0,
                                        [redacted]: 1,
                                        [redacted]: 0}
                                        )
        message = can.Message(arbitration_id=self.ignition_message.frame_id,
                                        data=shutdown_data)
        self.can0.send(message)


def pollThread(self):
    while(self.running):
        print(self.speedslider.value)
        data = self.speed_message.encode({[redacted]: 0,
```

```
                                    [redacted]: 0,
                                    [redacted]: 0,
                                    [redacted]: 0,
                                    [redacted]: 3,
                                    [redacted]: self.speedslider.value,
                                    [redacted]: 0,
                                    [redacted]: 0,
                                    [redacted]: 0}
                                    )
            message = can.Message(arbitration_id=self.speed_message.frame_id,
                                    data=data)
            self.can0.send(message)

            print(self.gaspedalslider.value)
            data = self.gaspedal_message.encode(
                                    {[redacted]:self.gaspedalslider.value}
                                    )
            message= can.Message(arbitration_id=self.gaspedal_message.frame_id,
                                    data=data)
            self.can0.send(message)
            time.sleep(0.05)

        return


if __name__ == "__main__":
    TouCAN().run()
```

Appendix B

Scripts for sound generation

B.1 Filtering and decoding CAN msg flow

```
import argparse
import random
import time
from pythonosc import osc_message_builder
from pythonosc import udp_client
import socket
import can
import cantools
from pprint import pprint
from datetime import datetime

db = cantools.database.load_file(
    '/path/to/dir/CAN_db.dbc')


# This function wraps send_message in OSC
def send_osc(client, listener, value):
    client.send_message(listener, value)

def receive_all():
    """Receives all messages and prints them to the console until Ctrl+C is
                                        pressed."""
    parser = argparse.ArgumentParser()
    parser.add_argument("--ip", default='127.0.0.1',
                        help="The ip of the OSC server")
    parser.add_argument("--port", type=int, default='57120',
                        help="The port of the OSC server")
    args = parser.parse_args()
    client = udp_client.SimpleUDPClient(args.ip, args.port)


    with can.interface.Bus(
        bustype="socketcan", channel="can0", bitrate=125000
```

```
    ) as bus:


        try:
            while True:
                message = bus.recv(1)
                if(message is None):
                    continue
                decoded = db.decode_message(message.arbitration_id, message.data)
                items = decoded.items()

                if(message.arbitration_id == 0x1):
                    search_key = [redacted]
                    res = [val for key, val in items if search_key in key]

                    if res[0] == [redacted]:              #shutdown
                        send_osc(client, "/shutdown", 1)
                        break
                    elif res[0] == [redacted]:          #ignition
                        send_osc(client, "/ignition", 0.0)



                elif(message.arbitration_id == 0x3):          #vehicle speed
                    search_key = [redacted]
                    res = [val for key, val in items if search_key in key]
                    real_speed = res[0]
                    send_osc(client, "/speedEngine", real_speed*0.9/512)

                elif(message.arbitration_id == 0x4):          #braking status
                    search_key = [redacted]
                    res = [val for key, val in items if search_key in key]
                    brake_status = res[0]
                    send_osc(client, "/brakepedal", brake_status)

                elif(message.arbitration_id == 0x2):
                    search_key=[redacted]          #gas pedal pos
                    res = [val for key, val in items if search_key in key]
                    gas_position = res[0]
                    send_osc(client, "/gaspedal", gas_position)

                else:
                        print("Something strange happened")


        except KeyboardInterrupt:
            pass  # exit normally
            send_osc(client, "/shutdown", 1)


if __name__ == "__main__":
    receive_all()
```

B.2 Algorithmic generation

```
(
e = SynthDef(\engine, {

    | // arguments range: 0.0 — 1.0
    mixCylinders   = 0.6,
    mixParabolic   = 0.3,
    engineSpeed    = 0.0,
    parabolaDelay  = 0.15,
    warpDelay      = 0.4,
    waveguideWarp  = 0.35,
    wguideFeedback = 0.35,
    wguideLength1  = 0.2,
    wguideLength2  = 0.2,
    wguideWidth1   = 0.5,
    wguideWidth2   = 0.2
    |


    var transDelay = NamedControl.kr(\transDelay, [0.2, 0.3, 0.45]);
    var overtonePhase = NamedControl.kr(\overtonePhase, [0.25, 0.35,
   0.5]);
    var overtoneFreq = NamedControl.kr(\overtoneFreq, [0.3, 0.47, 0.38]);
    var overtoneAmp = NamedControl.kr(\overtoneAmp, [0.1, 0.2, 0.2]);

    var noise, bufferA, bufferB, bufferTd, fourstroke, phasor, td,
   parabola, fm1, preFM1,
    fm2, preFM2, overtone, overtoneDrive, e1b, e2a, e2b, e1a, spacewarp,
   engine;

    bufferA = LocalBuf(44100, 1);
    bufferB = LocalBuf(44100, 1);
    bufferTd = LocalBuf(44100, 1);

    noise = WhiteNoise.ar;
    noise = OnePole.ar(noise, exp(-2pi * (50 * SampleDur.ir)));
    noise = OnePole.ar(noise, exp(-2pi * (20 * SampleDur.ir)));
    noise = (DelTapWr.ar([bufferA, bufferB], [noise * 0.5, noise * 10]));

    phasor = LFSaw.ar(
        OnePole.ar(K2A.ar(engineSpeed) * 30, exp(-2pi * (0.8 *
   SampleDur.ir))),
        1, 0.5, 0.5);
```

```
td = DelTapWr.ar(bufferTd, phasor);

fourstroke = DelTapRd.ar(bufferA, noise[0.5], [5, 1, 15, 20]/1000, 4);
fourstroke = phasor + fourstroke — [0.75, 0.5, 0.25, 0];
fourstroke = (fourstroke * 2pi).cos;
fourstroke = fourstroke * (DelTapRd.ar(bufferB, noise[1], [5, 10, 15,
20]/1000, 4) + ((1 — engineSpeed) * 15   + 7));
fourstroke = 1 / ((fourstroke * fourstroke) + 1);
fourstroke = fourstroke.sum * mixCylinders;
fourstroke = fourstroke — OnePole.ar(fourstroke, exp(—2pi * (4 *
SampleDur.ir)));

parabola = DelTapRd.ar(bufferTd, td, (parabolaDelay * 100)/1000, 1)
— 0.5;
parabola = parabola * parabola * (—4) + 1 * 3 * mixParabolic;

preFM1 = DelTapRd.ar(bufferTd, td, (warpDelay * 100)/1000, 1);
preFM1 = (preFM1 * 2pi).cos;
preFM2 = K2A.ar(engineSpeed * waveguideWarp);
preFM2 = OnePole.ar(preFM2, exp(—2pi * (0.2 * SampleDur.ir)));
fm1 = (1 — preFM1) * preFM2 + 0.5;
fm2 = (preFM2 * preFM1) + 0.5;

overtoneDrive  = overtoneDrive!3;
overtone = overtone!3;

3.do{|i|

    overtoneDrive[i] = DelTapRd.ar(bufferTd, td,
(transDelay[i]*100)/1000) * (0.5**(i+1)*36);
    overtoneDrive[i] = Wrap.ar(overtoneDrive[i]);

    overtone[i] = overtoneDrive[i].max(overtonePhase[i]) —
overtonePhase[i];
    overtone[i] = overtone[i] * (1 — overtonePhase[i]).reciprocal;
    overtone[i] = overtone[i] * ((overtoneFreq[i] * 12) *
overtonePhase[i]);
    overtone[i] = Wrap.ar(overtone[i]) — 0.5;
    overtone[i] = (overtone[i] * overtone[i]) * (—4) + 1 * 0.5;
    overtone[i] = (overtone[i] * (1 — overtoneDrive[i])) *
(overtoneAmp[i] * 12);
};


# e1b, e2b, e2a, e1a = DelayC.ar(
```

```
        in: InFeedback.ar(bus:(10..13)),
        maxdelaytime: 0.15,
        delaytime:
    ((([wguideLength1,wguideWidth1,wguideLength2,wguideWidth2] * 40)
            * [fm1,fm1,fm2,fm1])/1000)
    );

    OffsetOut.ar(13, e1b + overtone[1.5]);

    e2b = e2b + overtone[2];
    OffsetOut.ar(13, e2b);

    e2a = e2a + overtone[0];
    OffsetOut.ar(10, e2a);

    OffsetOut.ar(10, e1a * wguideFeedback + (parabola -
    OnePole.ar(parabola, exp(-2pi * (30 * SampleDur.ir)))));

    spacewarp = e1b + e2b + e2a + e1a;
    spacewarp = spacewarp - OnePole.ar(spacewarp, exp(-2pi * (200 *
    SampleDur.ir)));

    engine = (spacewarp + fourstroke)!2 * 0.5;

    if(engineSpeed!=0.00){
        Out.ar(0, engine);
    }
}).add;

SynthDef(\regbrake, {arg out = 0, release_dur, gate =1, amp = 0.2;
    var saw, env;
    env = EnvGen.kr(Env.asr(0.01, amp, release_dur), gate, doneAction:2);
    saw = Clip.ar(LFSaw.ar([99.8, 100.2], 1, 0.5, 0.5).sum - 1, -0.5,
    0.5);

    Out.ar(out, saw * env);
}).add
)

(
OSCdef ('ignitionlistener', {
   arg msg;
   y=Synth.new(\engine, [\engineSpeed, 0.0]);
}, "/ignition");
)
```

```
(
OSCdef ('gaspedalposlistener', {
   arg msg;
   e.set(\wguideFeedback, (msg/100).range(0,0.7));
   e.set(\mixParabolic, (msg/100).range(0,0.5));
   e.set(\parabolicDelay, (msg/100).range(0,0.3));
   e.set(\warpDelay, (msg/100).range(0,1));
   e.set(\waveguideWarp, (msg/100).range(0,1));
}, "/gaspedal");
)

(
OSCdef ('speedlistener', {
   arg msg;
   y.set(\engineSpeed, msg);
}, "/speedEngine");
)

(
OSCdef ('shutdownlistener', {
   arg msg;
   y.run(false)
}, "/shutdown");
)

(
OSCdef ('brakinglistener', {
   t = Task({
    var a;
    a =  Synth.new(\regbrake, [\release_dur, 5, \out, 0, \amp, 0.2,
    \gate, 1]);
    1.wait;
    a.set(\gate, 0);
    1.wait;
   });

   t.play;
}, "/braking");
)
```

B.3 Granular synthesis

```
(
var envtri, sig, envsq, envsaw;
~sin = Signal.sineFill(1024,1!1,0!1).asWavetable;
envtri = Env([0,1,0,−1,0]);
sig=envtri.asSignal(1024);
~tri=sig.asWavetable;
envsq = Env([1,1,−1], curve: \step);
sig=envsq.asSignal(1024);
~square=sig.asWavetable;
envsaw = Env([0,1,−1,0],[1,0,1], curve: \linear);
sig=envsaw.asSignal(1024);
~saw=sig.asWavetable;

~triBuf=Buffer.loadCollection(s, ~tri);
~squareBuf=Buffer.loadCollection(s, ~square);
~sawBuf=Buffer.loadCollection(s, ~saw);
~sinBuf=Buffer.loadCollection(s, ~sin);
~tri2Buf=Buffer.loadCollection(s, ~tri);
~saw2Buf=Buffer.loadCollection(s, ~saw);
~sin2Buf=Buffer.loadCollection(s, ~sin);
~square2Buf=Buffer.loadCollection(s, ~square);
~spark=Buffer.read(s, Platform.resourceDir +/+"/interference.wav",
    channels:[0]);
~fourstroke=Buffer.read(s,Platform.resourceDir +/+"/elephant.wav");
~torque=Buffer.readChannel(s,Platform.resourceDir +/+"/string.wav",
    channels:[0]);

OSCdef ('ignitionlistener', {

    arg msg;
    var buftri, first, second, third, fourth, fifth, bufsq, bufsaw,
    bufsin, buf2tri, buf2saw, buf2sin, buf2sq, scoppiosynth, grainsynth;
    buftri=~triBuf.bufnum;
    bufsq=~squareBuf.bufnum;
    bufsaw=~sawBuf.bufnum;
    bufsin=~sinBuf.bufnum;
    buf2tri=~tri2Buf.bufnum;
    buf2saw=~saw2Buf.bufnum;
    buf2sin=~sin2Buf.bufnum;
    buf2sq=~square2Buf.bufnum;
    ~first = Synth.new(\blend, [\bufpos, buftri, \start, 0]);
    ~second = Synth.new(\blend, [\bufpos, bufsq + 0.4, \start, 0]);
    ~third = Synth.new(\blend, [\bufpos, bufsq + 0.7, \start, 0]);
    ~fourth = Synth.new(\blend, [\bufpos, bufsaw + 0.68, \start, 0]);
```

```
    ~fifth = Synth.new(\blend, [\bufpos, bufsaw + 0.6, \start, 0]);
    ~sparksynth=Synth.new(\scoppio, [\bufnum, ~scoppio.bufnum, \startPos,
    0, \loop, 1]);
    ~grainsynth=Synth.new(\granularspeed, [\bufnum, ~grain.bufnum,
    \startPos, ~grain.numFrames*0.5]);
   ~grainpedal=Synth.new(\granularpedal, [\bufnum, ~grainele.bufnum,
    \startPos, ~grainele.numFrames*0.5]);

}, "/ignition");

OSCdef ('speedlistener', {
    arg msg;

   ~first.set(\start, LinLin.kr(msg,0, 150, 198, 492));

   ~second.set(\start, LinLin.kr(msg,0, 150, 224, 493));
    ~third.set(\start, LinLin.kr(msg,0, 150, 263, 498));
    ~fourth.set(\start, LinLin.kr(msg,0, 150, 298, 495));
    ~fifth.set(\start, LinLin.kr(msg,0, 150, 99, 489));
   ~sparksynth.set(\rate, LinLin.kr(msg,0, 150, 1, 1.5));
   ~grainsynth.set(\speedrate, LinLin.kr(msg,0, 150, 200, 400));
   ~grainsynth.set(\duration, LinLin.kr(msg, 0, 150, 0.5, 0.2));
   ~grainsynth.set(\playbackrate, LinLin.kr(msg, 0,150, 1, 1.5));
   ~grainsynth.set(\startPos, LinLin.kr(msg, 0,150, ~grain.numFrames*0.5,
    ~grain.numFrames*0.2));

}, "/speedEngine");

OSCdef ('gaspedallistener', {
    arg msg;

   ~grainpedal.set(\speedrate, LinLin.kr(msg,0, 100, 200, 400));
   ~grainpedal.set(\duration, LinLin.kr(msg, 0, 100, 0.5, 0.2));
   ~grainpedal.set(\playbackrate, LinLin.kr(msg, 0,100, 1, 1.5));
   ~grainpedal.set(\startPos, LinLin.kr(msg, 0,100,
    ~grainele.numFrames*0.5, ~grainele.numFrames*0.2));

}, "/gaspedalpos");

OSCdef ('shutdownlistener', {
    arg msg;
    ~first.free;
    ~second.free;
    ~third.free;
    ~fourth.free;
```

```
    ~fifth.free;
    ~sparksynth.free;
   ~grainsynth.free;
   ~grainpedal.free;
}, "/shutdown");

SynthDef.new (\spark ,{
    arg out=0, bufnum=0, rate=1, startPos=0, loop=1;
    var signal;
   signal = PlayBuf.ar(1, bufnum: 0, rate:1, trigger: 1.0, startPos: 0,
    loop: 1);
    Out.ar(out, 0.08*signal!2);
}).add;

SynthDef.new (\granularspeed ,{
    arg out=0, bufnum=0, startPos=0, speedrate=200, duration=0.5,
    playbackrate=1;
    var signal;
   signal = GrainBuf.ar (1, Impulse.kr(speedrate), duration +
    LFNoise1.kr(100).exprange(0.001,0.005) , bufnum, playbackrate,
    startPos + LFNoise1.kr(100).exprange(1,5), 2);
    signal = LeakDC.ar(signal);
    Out.ar(out, 0.08*signal!2);
}).add;


SynthDef.new (\granularpedal ,{
    arg out=0, bufnum=0, startPos=0, speedrate=200, duration=0.5,
    playbackrate=1;
    var signal;
   signal = GrainBuf.ar (1, Impulse.kr(speedrate), duration +
    LFNoise1.kr(100).exprange(0.001,0.005) , bufnum, playbackrate,
    startPos + LFNoise1.kr(100).exprange(1,5), 2);
    signal = LeakDC.ar(signal);
    Out.ar(out, 0.08*signal!2);
}).add;


SynthDef.new (\blend, {
    arg out=0, bufpos=0, freq=440, start= 180, warp=0, end= 500, mul=0.01;
    var signal;
    signal = VOsc.ar (bufpos, start, mul: mul);
    signal = LeakDC.ar(signal);
    Out.ar(out, signal!2);
    }).add;
```

```
)
```

Bibliography

[1] N. Abid Ali Khan, M. Shyam Sundar, S. Sambiah, and P.A. Govindacharyulu. Low-cost usb2.0 to can2.0 bridge design for automotive electronic circuit. *International Journal of Electronics Engineering*, 2010.

[2] The Linux Kernel Archives. Socketcan documentation, 2021. URL https://www.kernel.org/doc/Documentation/networking/can.txt.

[3] Bosch. *CAN Specifications 2.0*. Bosch, 1991.

[4] can utils. can-utils, 2021. URL https://github.com/linux-can/can-utils.

[5] Rafael Carbo and Adriana C. Molero. The effect of temperature on sound wave absorption in a sediment layer. *The Journal of the Acoustical Society of America*, 2000.

[6] Kivy community. Kivy: Cross-platform python framework for nui development, 2021. URL https://kivy.org/#home.

[7] Open Sound Control. Open sound control, 2021. URL http://opensoundcontrol.org/.

[8] Steve Corrigan. *Controller Area Network Physical Layer Requirements, Application Report*. Texas Instruments, 2008.

[9] Andy Farnell. *Designing Sound*. The MIT Press, 2010.

[10] Stefano Fasciani. Spectral granular synthesis. *International Computer Music Conference 2018*, 2018.

[11] Mark French and Rod Handy. Spectrograms: Turning signals into pictures. *Journal of Engineering Technology*, 2007.

[12] Flat icon. Toucan flat icon, 2021. URL https://www.flaticon.com/free-icon/toucan_2152476.

[13] ixi audio. Spectrogram quark plug-in for supercollider, 2020. URL https://github.com/supercollider-quarks/Spectrogram.

[14] INNO MAKER. innomaker can2usb can bus shield documentation, 2021. URL https://github.com/INNO-MAKER/usb2can.

[15] Erik Moqvist. Can bus tools documentation, 2020. URL https://cantools.readthedocs.io/en/latest/.

[16] Ulrich Oberst. The fast fourier transform. *SIAM Journal on Control and Optimization*, 2007.

[17] Nigel Redmon. A one-pole filter, 2012. URL https://www.earlevel.com/main/2012/12/15/a-one-pole-filter.

[18] Curtis Roads. Introduction to granular synthesis. *Computer Music Journal*, 1988.

[19] James Sabatier and Alexander Ekimov. Detection of humans and light vehicles using acoustic-to-seismic coupling. *Army Research Office*, 2009.

[20] A. A. Salunkhe, Pravin P Kamble, and Rohit Jadhav. Design and implementation of can bus protocol for monitoring vehicle parameters. *2016 IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, 2016.

[21] STMicroelectronics. stm32l476rg datasheet, 2016. URL https://www.st.com/en/microcontrollers-microprocessors/stm32l476rg.html.

[22] STMicroelectronics. V53l3 datasheet, 2016. URL https://www.st.com/en/imaging-and-photonics-solutions/vl53l3cx.html.

[23] supercollider. Supercollider, 2021. URL https://supercollider.github.io/.

[24] Frank Vahid and Tony D. Givargis. *Embedded System Design: A Unified Hardware/-Software Introduction*. John Wiley & Sons, 2002.

[25] Bart Verrecas. How to make remarkable sound of electrified vehicles inside out with active sound design. *Siemens Automation*, 2020.

[26] WikiBooks. *Designing Sounds in Supercollider*. WikiBooks, 2019.