Design and Implementation of an engine sound for electric motorcycle

Master Thesis in Computer Engineering

Benjamin Lorin Libby

March 2019

Thesis supervisor:
Prof. Giovanni Malnati

Politecnico di Torino
Istituto Superiore Mario Boella - ISMB

"Art is the lie that enables us to realize the truth"

Pablo Picasso
Artist

Abstract:

The last 20 years have been a period of great and fast paced technological improvements: we are in what many describe as the "Digital Revolution".
We – as human beings – have to react quickly to these changes, and many times we don't even have the time to metabolize a new technology. This is creating many new unexpected problems and challenges, such as: the education towards the use of new technologies, the acceptance of the technology, the morality and the physical or mental danger that the technology can introduce.
Smartphones for example became the new accepted "human assistant" in less than 10 years. Social media became the new worldwide cafe where to make connections. But many aspects of these technologies still have some big unanswered questions: how does the individual react to the fact that he is constantly being reachable and traceable? What is the impact on society of making making less and less human contact, but being always "connected"?

Another reality that is very present in modern life, that is now part of us, is represented by cars or, more in general, vehicles. The invention of these machines allowed people to travel much faster and to reduce the distances.
Since the manufacturing of the first cars in late 1800's/early 1900's, many studies and improvements have been done on the efficiency of the engines, the emissions produced by combustion, the security, and the ergonomics of the interiors.
However, there is one aspect that remained relatively the same for a whole century: the sound of the engine. This is one of the features that defines a vehicle running with a combustion engine: a motorcycle has a different sound than that of a truck or a car. Nonetheless, human beings are able to recognize that typical cyclic engine sound, they identify it as a generic moving vehicle and they become aware of it.

But now, with the recent advent of electric vehicles, this sound feedback is completely gone. What was taken for granted until now, suddenly doesn't exist anymore.
This introduces new legal and security problems: if an incoming vehicle approaches a pedestrian, the absence of acoustic feedback could lead to accidents. Even for the driver it's important to have acoustic feedback and understand how the vehicle is responding.

Abstract

The sight is certainly the first feedback that a pedestrian should have of an incoming vehicle, but sometimes this is not possible due to physical obstacles (e.g. a curve) or just because the pedestrian wasn't looking. This is why the acoustic feedback is also a crucial feature to integrate every electric vehicle.
There is also another aspect that should be considered: the aesthetics and the branding of a sound for every model. Every manufacturer has in fact their own "engine sound", especially in the motorcycle world.

All this led to the realization of this thesis: the implementation of an algorithm that synthesizes an engine sound for electric motorcycle. The request was not to implement a realistic or already existing engine sound, but to create a new one from scratch, more similar to that of a propulsion engine or to an engine coming from the sci-fi world.
This sound is created real-time, it takes as input parameters the velocity and the amount of applied acceleration (the acceleration handle inclination) and it also takes into consideration their derivatives with respect to time, that is "how fast do these input parameters transition".
The reason why also the handle inclination is important is because there could be scenarios where the acceleration that the driver applies doesn't correspond with the actual velocity of the motorcycle, such as when going uphill or downhill.
The final algorithm should run on a small chip that can be installed directly on the motorcycle, and that takes as inputs velocity and handle inclination.

The first stage of this thesis is the analysis of the problem. First, two sound samples are chosen: one is a recording of real dragster engines, the other is a sample coming from the Podracers in Star Wars I. These two samples are analyzed through a spectrogram analyzer tool, which plots out the evolution of the spectrum in time. Through this tool, it's possible to immediately observe the main components of the sound.
The important features to look for are peaks in the spectrum that undergo a frequency shift: these represent periodic components such as sine waves with increasing frequency as the velocity rises. It's also very interesting to observe the relations between these periodic components: sometimes these can be seen as harmonics of a fundamental frequency, thus generating a more harmonic sound, other times a lot of partial components are present, introducing inharmonicity in the sound.
Based on these observations, a first rough and qualitative model can be built.
The other step of this first stage is always a sound analysis, but this time it's based on the studies of jet propulsion engines done by sound designer Andy Farnell in his book "Designing Sound".
In this book, two main features of the analyzed sound stand out: the identification of the typical main periodic components in the sound, and the presence of a loud noise that

comes out when accelerating. The second feature in particular is crucial for the synthesis algorithm used in this thesis: it is a sound that really gives the idea of a combustion process and of air that is forced out through a small hole.
This effect is obtained by using a white noise source and then processing it through a chain of filters and an overdrive component.

The second stage of the thesis is the implementation of a solution on PC. This means many libraries and development environments are available, and almost unlimited resources can be used.
This stage is very important because it allows a sort of trial-and-error approach. Furthermore, the power of implementing a solution on PC is that the results can easily be visualized in a simple GUI, this means plotting for example the waveform and most importantly the spectrum at each time instant to see what's happening.
The development environment that is used is Processing 3 (in Java), and the audio library is Minim. Working in Processing allows to easily program a simple GUI that helps visualizing the work that is being done.
The implementation process is divided in three steps: a solution based on the first model that was built in the first analysis stage, a solution based on Andy Farnell's studies of jet engines, and a hybrid solution combining the previous two.
This last implementation sets the end of this second stage. The sound could still be improved, but its useless to over engineer it at this stage. There is now a basic and stable logical model on which to build the synthesis algorithm. Improving the sound is mainly a matter of tweaking the parameters and adding or removing some components.

The third stage of this thesis consists in actually implementing the algorithm on a chip, by adapting the solution achieved in the previous stage.
The first challenge is to find a board that works well for the purpose: it should have enough resources, it should be small and, if possible, it should have some integrated libraries that help in the process of programming the microprocessor. This led to choosing the Teensy 3.2 board, which has an ARM Cortex-M4 microprocessor, is entirely programmable in Arduino, and has a very well done audio synthesis/processing library. Also, this board can be mounted on an "Audio Shield" containing a female jack connector, so that what is implemented can be heard immediately.
The solution implemented on this board takes a lot of ideas from the hybrid solution in stage 2 of the thesis, but it refines many aspects of it and it perfects the final sound.

Preamble:

Notations and Conventions:

- `Courier New` is used for all source code and in general for everything that would be typed literally when programming: *variables, keywords, constants, method names, classes.*
- Source code appears as follows:

```
#include <cstdlib>

using namespace std;

void main(){
      cout << "Hello World!" << endl;
      return;
}
```

- When referencing to a source code variable/function/keyword in the text, it appears like `'this'`. For example the variable wave in a source code is written as `'wave'`.
- The frequency range of a frequency band is denoted as f1-f2. For example the band between 200 Hz and 400 Hz is denoted as 200-400 Hz
- This thesis is divided into chapters; each chapter is divided into subchapters and each subchapter can be divided in sections
- The Header in each page contains the title of the current chapter. The Footer in each page contains the page number.
- Figures, formulas and tables are numbered inside a chapter. Example: figure '2' of chapter '3' is referenced as "Figure 3.2"
- Formulas are numbered inside a chapter as below:

$$e = m \cdot c^2 \qquad (1.1)$$

- Author references and web references are tagged square brackets. Example: Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua[1].
- Literal quotations are denoted like this *"Lorem ipsum dolor sit amet..."*

Table of Contents:

1 Introduction:

1.1 Thesis objectives:

The use of electric vehicles in everyday life is a reality that will be increasingly present in the future.
While this scenario represents a great step forward in reducing pollution and, more in general, fossil fuels consumption, it does introduce different new and unexplored issues.
This thesis concentrates on one of these issues, the sound, and it tries to answer to the question "What voice should electric vehicles have?".
With the advent of digital audio, it became much easier to process sound and even generate it from scratch through synthesis techniques coming from the DSP world.
The focus is put on the sound for electric motorcycles, even if similar solutions could be used for electric cars.

The main objective of this thesis is to generate a new algorithm that can run on a small chip installed directly on the motorcycle. This chip takes as input the current velocity and how much the accelerator is pressed, and it synthesizes the engine sound based on these parameters.
This means working on basically two aspects:

- **The sound design:** The generated sound should meet both the expectations of the company, and those of the final users, the motorcycle drivers. Since this is a relatively new field, and this particular application on electric vehicles is even more recent, all that companies can do when commissioning this task is to refer to some pre-existing sound, coming either from the real world or from the entertainment world (movies and videogames).
  Also, this sound should be imagined in an everyday life scenario: while it's nice to hear a supersonic sound in a movie scene, the same sound could become disturbing if listened to for more than 10 minutes.

This is why, one of the objectives of this thesis is to generate an enjoyable sound, or to use the same words present in different parts of the thesis: a "comfortable sound".

- **The technical implementation:** This means working with different audio libraries and implementing the chain of processing stages through which the sound passes.
  The first objective is to synthesize a sound in a PC environment, with a nice GUI to help during the development phase, and resources which are de facto limitless.
  The second objective is to implement that same solution (or a similar one) on a small chip that will be installed on the motorcycles. This means not having the possibility to visually debug as with the PC, working in an embedded environment with more low-level libraries, and constantly paying attention to the resources utilization.

1.2 Context in which the work was developed:

This thesis was developed at the LINKS Foundation [1], a modern and stimulating complex of research labs in the IT sector located in the Politecnico di Torino campus. In particular, this work was a project of the ISMB (Istituto Superiore Mario Boella) for a company producing electric vehicles near Torino: NITO srl [2].
A weekly update would occur with the professor following the thesis; this was done to solve some problems, to give hints on the workflow and to change the approach on some implementations. While this support was of great help, it was never a unilateral experience based on just "following instructions": a lot of trust and freedom of expression was given to the student, especially in the more creative parts of the thesis, such as the sound design.

1.3 State of the art in this field:

Giving a voice to electric cars is an extremely new field: a lot of research is being done by manufacturers of research teams around the world, many times the results of these research are not easily accessible and/or semi-secret, because each company wants to "get there first": be the first ones to launch an incredible sound for their car. The fact that this field is still in this primordial stage means that it is open to many possible solutions, each having a different approach: some tend to render a more realistic and traditional sound, some prefer a sound coming from the sci-fi world,

others have a more musical and harmonic approach. There is no "good solution" if not the one that will eventually be accepted by the general public.

An interesting paper written by a research team at Renault in 2018 [3] describes the 3 year process to come up with a suitable sound for electric cars. The interesting point on which it focuses is how many elements come into play when starting to work on a sound for electric vehicles: sound quality, ergonomics (for example the audio feedback that a driver receives from the vehicle), aesthetics (it should be a pleasant sound to listen to, even for long periods of time), acceptability (it should be accepted by the general public as "the sound" for electric vehicles).
The way they approach the work is to consider *"the electric vehicle as a new sound species in a new (urban) ecosystem"*, a concept coming from the acoustic *"ecology theory that states the existence of a sonic organization in nature, in order to ensure the audibility of every species"*.
For what concerns the workflow they used, it is well described by the schema at page 7 of the paper: analysis and research, definition of the specifications, creation of a sound, tests done with users, eventually revising it and coming back over and over to the sound creation step. This approach was broadly used also in this thesis.

Another interesting and extremely original approach to this new challenge was exposed in 2018 at a TED Talk [4] by Renzo Vitale, who defines himself as a "sound geneticist" and worked on this project at BMW. His approach is to consider the sound of electric vehicles not just as noise, but as a voice, a more harmonic solution.
The great intuition was to consider elements such as the character and the identity of the car, creating a sort of transmission of emotions between the vehicle and the pedestrians. He calls this "sound genetics": each car model can be described as a combination of different genes, each one having a different weight. He reduced these genes to 8 main ones: Visionary, Elegant, Dynamic, Embracing, Minimalist, Present, Luminous, Hi-Tech. Depending on the high or low presence of a specific gene in a car model, a "part" of the spectrum of frequencies is generated and evolves with velocity.

1.4 Innovative contributions of this Thesis:

Electric vehicles are just starting to make their way in the monopoly of vehicles running on fuel, and many more years must pass before it becomes mainstream. According to this 2017 NY Times article [5] this conversion will happen somewhere between 2025 and 2040, but it is hard to give a precise date.

Developing and conceptualizing the sound of a "revolution" that still has to happen is itself an innovative objective that looks at the future.
Nobody has a clear idea on how the next generation of vehicles will sound like, but it's clear that at some point this matter will have to be discussed. In fact, two main aspects are pushing towards finding a solution: the awareness of surrounding vehicles for pedestrians, but also the awareness for the driver on how the vehicle is responding.

Typically the people that are working on this topic start by doing a research on similar preexisting known sounds: these can come either from the real world or from the entertainment industry (movies and videogames).
Other solutions aim at creating something completely new and unheard.

The biggest challenge of this thesis was to not fall in one of these categories: the analysis stage is done both on real engines and sounds coming from the sci-fi world, such as the Podracer in Star Wars I. The two approaches are tested individually and then combined to create a sort of hybrid solution.
Also, some components of the synthesized sound are completely invented and don't come from any reference. For example, many LFOs (below 10Hz) are modulating the amplitude of the sine wave components or the central frequency of a filter applied to noise. When this solution was first tried, it improved the sound output considerably, giving the idea of a cyclic process, an engine that is running.

1.5 Thesis organization:

This thesis is organized in three main Chapters:

- The first chapter describes the preliminary analysis that was done on two different sound samples, one from the real world dragster engine and the other from the Podracer sound in Star Wars I. It then discusses an analysis and subsequent sound implementation that was done on jet engines by a renowned Sound Designer: Andy Farnell.
- The second chapter describes three different implementations that were done in a PC environment. The first two are just pieces that come together in the third hybrid solution.
- The third chapter discusses the implementation of the final audio synthesis algorithm on a chip that will be mounted on the motorcycle. The chip that was used is the Teensy 3.2, a very small yet powerful board.

2
Phase 1: Analysis of the problem

Sound is defined as a mechanical wave, caused by a vibrating object that propagates in every direction in a medium through compression and rarefaction.
The simplest sound signal to represent mathematically is the sine wave; it can be described by two parameters: Intensity (loudness) and Frequency.

$$y(t) = A \cdot \sin(2\pi f \cdot t + \phi) \tag{2.1}$$

A = Amplitude (typically from 0.0 to 1.0, from minimum to maximum intensity)
f = Frequency

We usually associate the concept of "pitch" to the word frequency, but this is a wrong association. In fact, real sounds in nature are composed of many frequencies, each one having a certain intensity level.
Depending on the organization of these frequencies, human ear could recognize a pitch (in this case it is defined as a "harmonic sound"), or something more similar to noise, which doesn't have a defined pitch (non-harmonic sound).
In general, every sound in nature can be analyzed by extracting its frequency components. The process through which this is done is called Fourier Transform: it transforms a signal from time domain into frequency domain.

This chapter first discusses the analysis in frequency domain of some sample jet engine sounds that were given by the client. This is a very important step, because it can help creating a qualitative initial model to follow for the sound synthesis; it's not a definitive model, and many changes can be done, but it is a good starting point to see how that type of sounds "is shaped".
The chapter then discusses some studies that were done by sound engineers specifically on jet engine sounds.

2.1 Qualitative spectral analysis in time of the samples given as examples

In order to give a sense of the type of sound that is wanted, five sample sounds were provided.
Three of them are taken from the Star Wars movies: they are somewhat less realistic because they are synthetized sounds coming from the movie industry, specifically in the sci-fi genre.
The other two are taken from the real world: one is a recording of a jet-pack like suit and the other is a recording of dragsters engines.
When designing a new sound, specially one that will be present in everyday life, it is important to work on two aspects:

- The sound should have an element of innovation that differentiates it from any other existing sound: it should be highly identifiable in order not to confuse it with something else.
  For example a sound of a bird on an electric car could be misleading and potentially dangerous if people make the wrong association.
- It should also be a "comfortable" sound. This basically means two things: it should be a pleasant sound, and avoid for example a big contribution of very high frequencies (>12 kHz), which are typically unpleasant to human ear.
  But having a "comfortable" sound also means that it should have some elements that people are familiar with and that satisfy the criteria of acceptability.
  In the collective imaginary in fact, a sound in nature is always associated to a certain family of sounds, in this case "engine sounds": different engines generate different sounds, but we, as humans, are able to understand if it is an engine or something else.

This is why two types of sample sounds are analyzed: real ones and artificial ones.
When Audio Engineers were designing the sound of Podracers in Star Wars Episode 1, they had to create something new and captivating, but at the same time something that could easily be associated to a propelled vehicle flying at high speeds.

The tools used to analyze the spectrum of these sample sounds is the Spectrum analyzer present in Ableton 9.6 – one of the leading DAWs in today's music industry – and the Spectrogram viewer of 'Sonic Visualizer' [6].
MatLab was also used to plot functions that are useful to describe a model derived from the analysis.

Ableton was chosen because of the ease of use in editing multiple sound samples (cutting and looping), and in applying and shaping different filters to isolate specific frequency bands.
Five audio channels are present – one for each sample sound – and on each one of them a chain of filters (Figure 2.1) is applied: a high-pass, a low-pass and a band-pass. These filters can be activated or deactivated depending on what frequency range is to be analyzed. This approach is useful to focus attention on one frequency band at a time, so that there is an acoustic separation in addition to the visual one.
At the end of this chain of filters there is the Spectrum analyzer component.

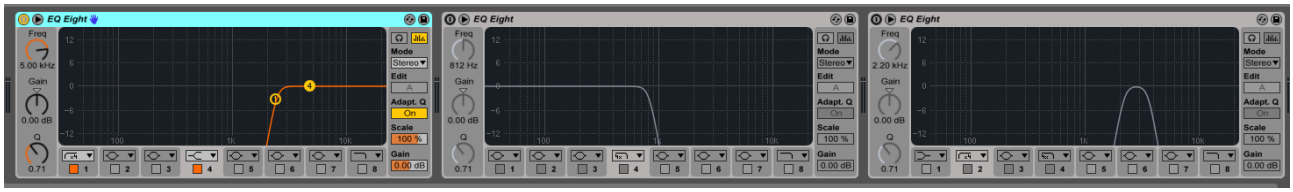


Figure 2.1: A chain of three filters in Ableton. From left to right: HP, LP, BP. Only the first one is activated here.

Sonic Visualizer is then used because of its very nice spectrogram: the provided sample sounds in fact all have a transient during which the engine is accelerating, and therefor it's crucial to study the spectrum evolution in time and the frequency shifts. The application also has many adjustable parameters to focus attention on a specific range of frequencies and intensity values.

This analysis is qualitative, and therefor it is imprecise: the objective is not to obtain precise values, but to get an idea of the sound shape and behavior in time.
In this regard it is needed to extract the following information:

- The spectrum shape. The main things to look for are peaks: these are periodic partial components, with a specific pitch which clearly stands out from the rest of the sound.
- The spectrum evolution in time. This consists in observing which parts of the spectrum undergo a frequency shift, and which instead are static.

The following two sections each analyze a specific frequency band and compare two of the provided sound samples: the Podracers taken from Star Wars I and the real dragster sound.
Note that when referring to the "transient" of the sound, it means the evolution in time, because all of these sounds are recording an accelerating engine.

2.1.1 Low and Mid Frequency Response: 20 – 3000 Hz

The Podracer sound from Star Wars is first analyzed.
The main component in the low frequency band (0-500Hz) is noise, with only one main frequency component that peaks out: it starts around 300 Hz and goes up to 500 Hz during acceleration (Figure 2.2).

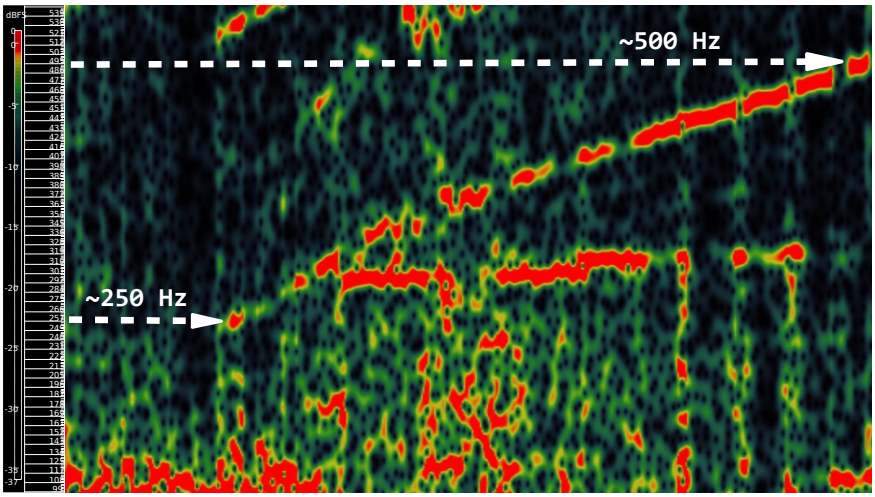


Figure 2.2: Spectrogram of Podracer in Star Wars at low frequencies (0-500 Hz). Mainly noise, with a visible frequency components shifting from 250 Hz to 500 Hz

In the 500-3000 Hz frequency band, the periodic components are more in number and particularly pronounced.
The main ones are 5 and at full regime they are separated from one another by a distance of ~250 Hz: the first one ends at 500 Hz, the last one at around 1750 Hz (Figure 2.3 and number 1 and 2 of Figure 2.5).

From this last analysis, being that at full regime these periodic components are equidistant with a $\Delta f \approx 250$ Hz, they could be modeled as harmonics of a fundamental frequency at 125 Hz which doubles in time. This rule comes from theory of harmonics, which states that given a signal with a fundamental frequency f_0 (the perceived pitch) and many frequency components f_1, f_2, f_3 ect. equispaced by an interval Δf which is a multiple of f_0, these are harmonics of the main frequency f_0: they don't change the pitch, but they enrich the sound (Formula 2.2).

$$f_k = k \cdot f_0 \qquad with \quad k = 1,2,\ldots,n \tag{2.2}$$

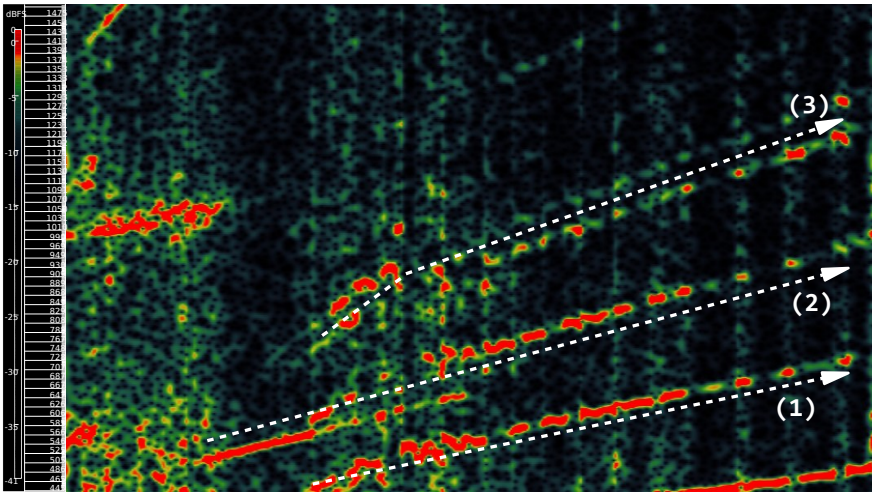


Figure 2.3: Spectrogram of Podracer in Star Wars in a 500-1500 Hz frequency band. Three main frequency components are visible: (1) ~450-750 Hz shift, (2) ~500-1000 Hz shift, (3) ~750-1250 Hz shift.

Increasing frequency of the fundamental also increases the harmonics frequency proportionally: if f_0 doubles, also Δf and all of its harmonics do.
This would also be in line with the results from previous analysis at 20-500 Hz frequency band, where the periodic component makes a shift from 250 to 500 Hz.
The fundamental frequency (the one going from 125 to 250 Hz) is not present in the spectrograms: maybe it is covered by the noise level, but it could also be a so called "virtual picth", where fundamental frequency is not physically present, but all of it's harmonics are, and our ear still perceives a pitch of f_0.
The model is plotted in Figure 2.4.

(a) (b)

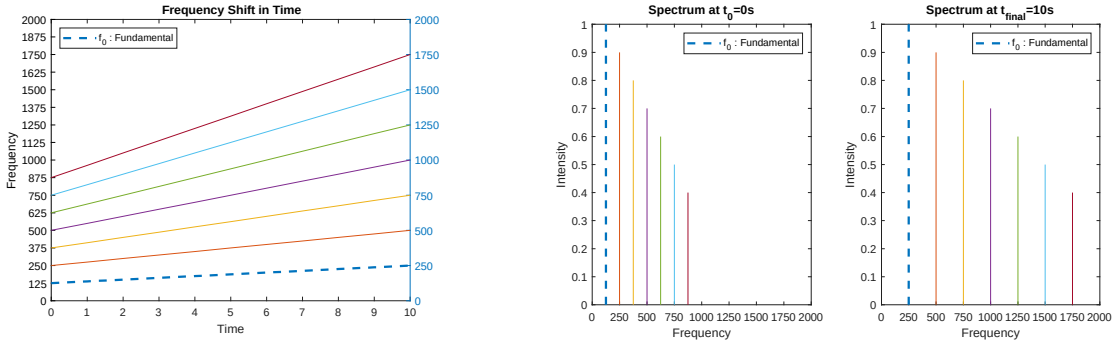


Figure 2.4: (a) Shows the model of 6 harmonics of a virtual fundamental frequency f_0 which shifts in time. (b) Shows the two spectra of the fundamental and its harmonics before and after the frequency shift

Three other components higher in frequency are visible at the beginning of the transient (number 3, 4 and 5 of Figure 2.5), initially spaced of a $\Delta f \approx 100$ Hz. They linearly increase in frequency and the highest reaches ~2250 Hz.
A last peak (number 6 of Figure 2.5) is visible at the end of the transient but is uncorrelated to all of the previously analyzed ones: it has a much broader band, meaning it is very noisy, and a much steeper frequency shift. From listening to the sample it is clear that it's an added sound effect, and doesn't really contribute to the overall sound.

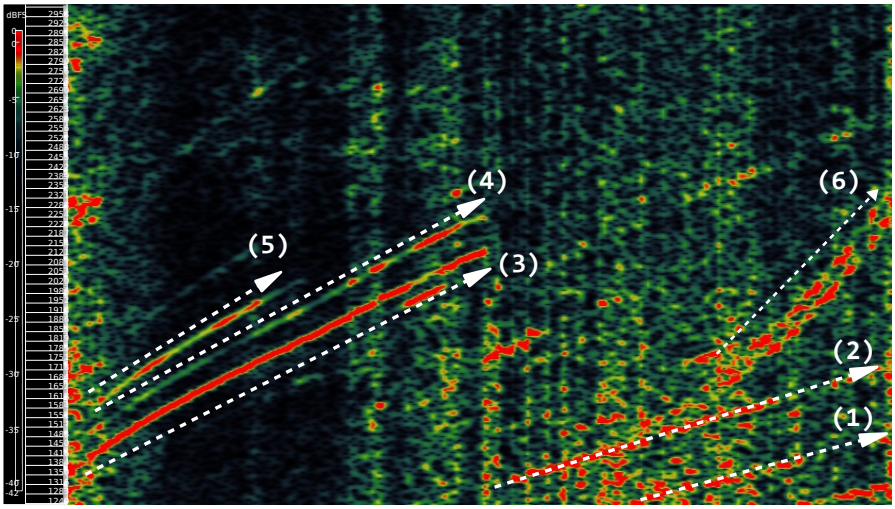


Figure 2.5: Spectrogram of Podracer in Star Wars in a 1250-3000 Hz frequency band. Number (1) and (2) respectively have a ~1250-1500 Hz and ~1250-1750 Hz frequency shift. (3) ~1400-2150 Hz shift, (4) ~1500-2250 Hz shift, (5) ~1600-2000 Hz shift. (6) ~1600-2300 Hz shift

Considering now the real sound of a dragster car, the first noticeable differences are:

- A concentration of the main partial components in the high frequencies domain (>3000 Hz). These will be discussed in the next section.
- Much steeper frequency shifts.

Below 3000 Hz, the number of peaks is very limited, and most of them just have the initial part of their transient in this range.
From 0 to 500 Hz the main component is noise, and only a small frequency shift from 150 to 250 Hz is noticeable when the dragster starts accelerating (Figure 2.6).

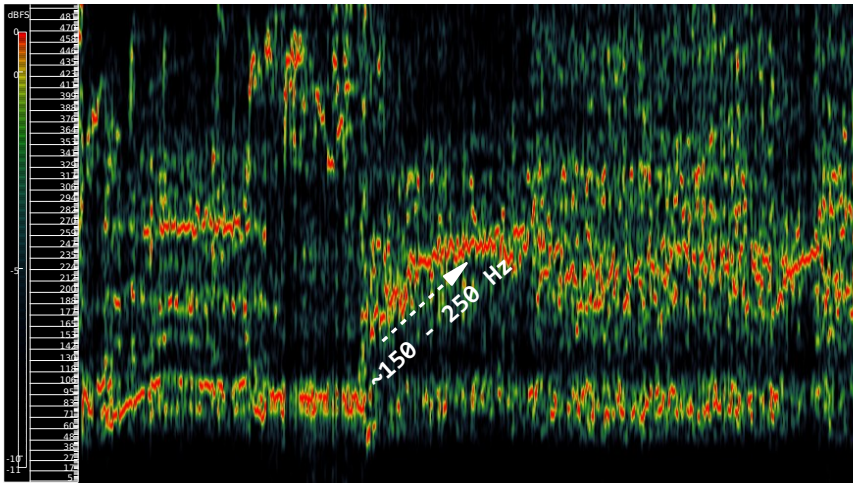


Figure 2.6: Spectrogram of dragster sound in a 0-500 Hz frequency band. Mainly noise, with a slight frequency shift in the central part of the transient

In the 500-3000 Hz frequency band, only one main periodic component is fully contained, and it undergoes a very large frequency shift: from ~830 Hz to ~2700 Hz (Figure 2.7).

(a) (b)

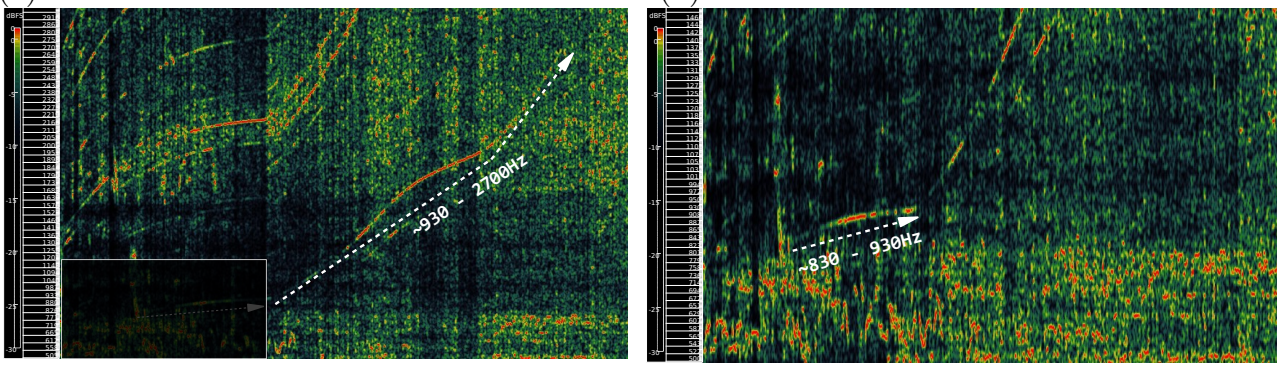


Figure 2.7: Spectrogram of dragster sound in a 500-3000 Hz frequency band. (a) Only one main periodic component is fully contained in this frequency range. The darkened area is zoomed in (b). Other visible periodic components are analyzed in the next section.

2.1.2 High Frequency Response: >3000 Hz

The Podracer sound from Star Wars is first analyzed.
In this frequency band many visible peaks undergo a very steep frequency shift. They can be grouped in two main sets: one lasting for the hole transient duration and going from ~2900 Hz to ~7000 Hz (number 1.1, 1.2 and 1.3 of Figure 2.8), the other in the second half of the transient, from ~3000 Hz to ~5000 Hz (number 2 of Figure 2.8).
These components are much higher in frequency, and lower in intensity (a gain of ~3dB was applied to make them visible).
Although these are part of the sample, in a practical implementation and a scenario of a prolonged exposure to this sound, their contribution should be lower in order to obtain a sound that is concentrated in a comfortable range of frequencies of ~30-4000 Hz (consider a standard 88-key piano ranges from 27.5 Hz to 4186 Hz, and the last octave is rarely used [7]) and avoid the so called "listener fatigue" phenomenon.

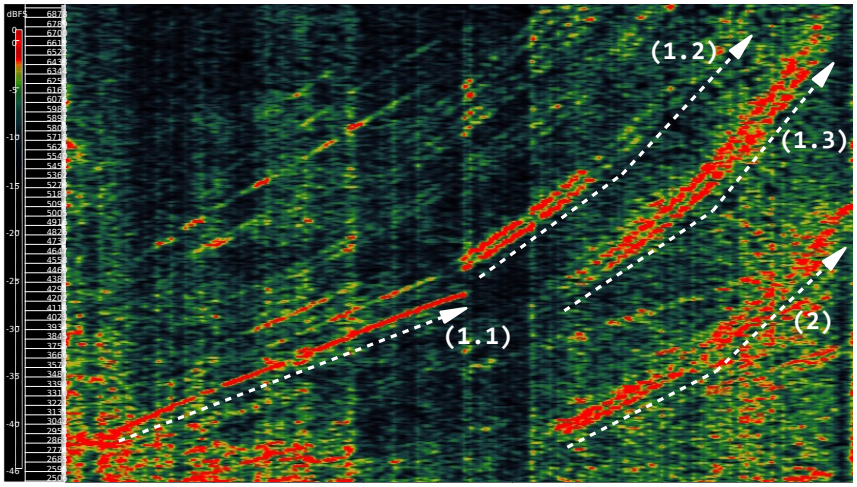


Figure 2.8: Spectrogram of Podracer in Star Wars in a 2500-7000 Hz frequency band. (1.1), (1.2) and (1.3) can be grouped as one component undergoing a ~2900-7000 Hz shift. (2) ~3000-5000 Hz shift

In the dragster sound, many more periodic components are present and they undergo a much larger frequency shift. They also tend to decrease in intensity after ~12 kHz.
Their behavior is similar, and they are therefor all grouped in Figure 2.9.
Also note how a little before the middle of the transient there is a global change in the behavior: this is because at that point the dragster changes its state from warming up the engine to accelerating.

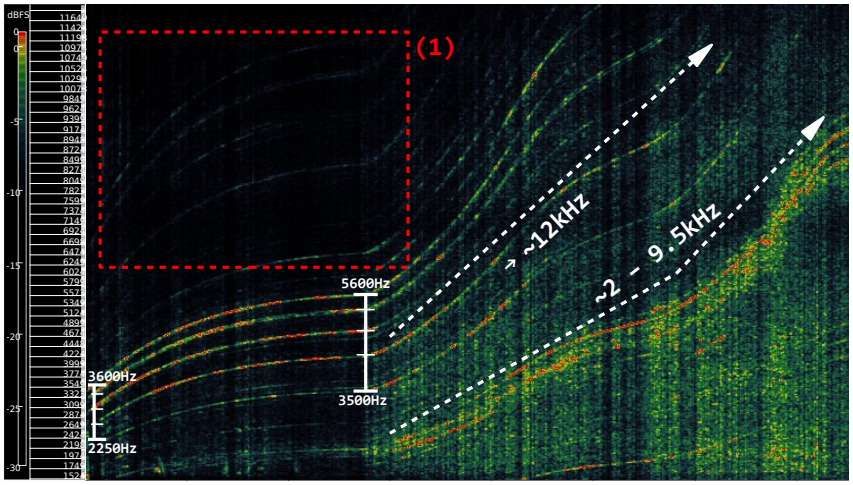


Figure 2.9: Spectrogram of dragster sound in a 1500-12000 Hz frequency band. Most of the periodic components are grouped together: they start at an average of ~3000 Hz and all terminate around 12 kHz. One lower one stands out and reaches ~9.5 kHz. The red quadrant (1) shows other minor periodic components which are much lower in intensity.

2.2 Spectral Analysis of Jet Engines

This subchapter aims at presenting the analysis and synthesis of a typical jet engine sound done by Sound Designer Andy Farnell in his book "Designing Sound" [8].
Many aspects of this work contributed to the implementation of the final solution presented in this thesis.

The study divides the sound in two big components and creates a model for the reproduction:

- An overdriven noise sound, which comes from the turbulent forced flame of the engine
- The periodic partial components, typical of a turbine

This subdivision is based on a mechanical analysis of a jet engine, which has a rotating element (the turbine) that creates many inharmonic periodic sine waves, and the combustion process in the chamber, which creates the noise-like sound.

The tool used by the author is PureData, a graphical programming language that makes sound synthesis much more immediate and gives an intuitive view of the processing stages that the sound undergoes.

Overdriven noise:
When the engine is at minimum speed, the noise component is little, and it is mainly concentrated in the low frequency band.
As the engine starts speeding up, the spectrum of the noise flattens out, and the contribution from high frequencies starts increasing.
At full thrust, the spectrum becomes basically flat, but it's not a simple white noise: the sound should in fact give an idea of "ripped air" forced out of a small hole; white noise instead is acoustically very uniform.
This "ripping" effect is obtained by increasing enormously the gain on noise and then applying a harsh clipping on the samples: this audio processing technique is called "overdrive". It bounds the amplified signal in a chosen range, and when a sample value is greater then the threshold, it is forced back to that threshold.
Through this technique, discontinuities of type C^1 (derivatives) are created in the points where the signal is clipped. If a simple sine wave is considered, this introduces many harmonics in the spectrum, and the signal in time resembles a square wave (Figure 2.10).

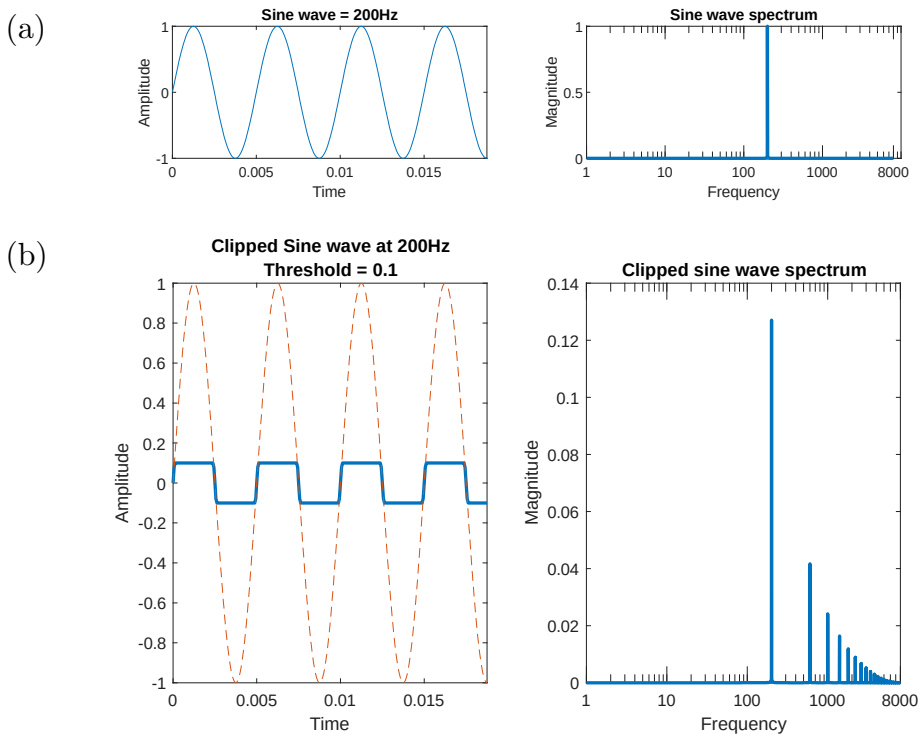


Figure 2.10: (a) Sine wave at 200 Hz with Amplitude=1; the spectrum has a single peak at f=200 Hz. (b) Sine wave at 200 Hz, clipped at a threshold of 0.1; the spectrum now has many harmonics. Also note how the clipped signal resembles a square wave.

When considering white noise instead, the signal that is generated from high gain followed by harsh clipping is a non-periodic square wave (Figure 2.11). This also introduces high frequencies in the sound, but in white noise this would result again in white noise.
In order to hear the high components that are introduced, a low pass filter has to be applied before the clipping process: this leaves some headroom and the frequencies generated by clipping are heard distinctively, creating that effect of ripping air.
This is the logic that is used by Andy Farnell to simulate the "forced flame" sound. By changing the parameters of cutoff frequency of the filter, gain amount and clipping threshold, many variants of the same sound can be synthesized.

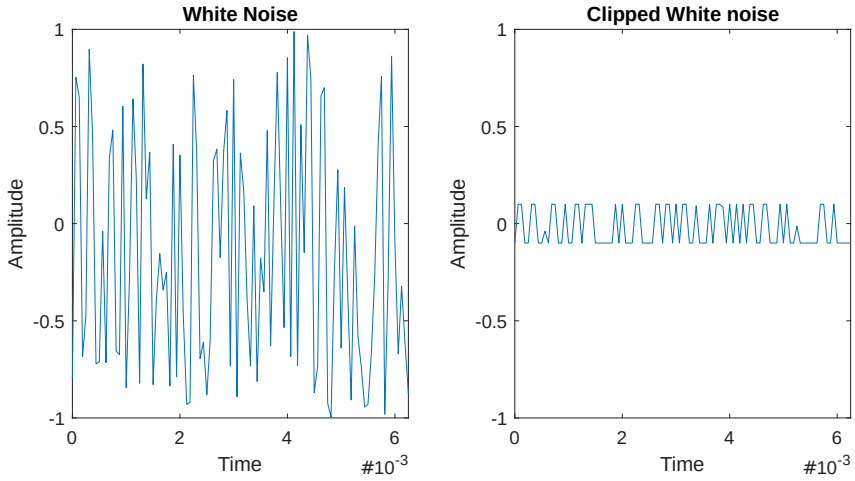


Figure 2.11: Normal white noise on the left and clipped white noise on the right. Note how the clipped one resembles a non-periodic square wave.

Figure 2.12 shows the 'patch' (the name of a PureData program) used to produce the sound. Without going into details of each component or into PureData's syntax, now the logical chain of processing stages will be described.
Note that the filters used here are of first order, so they have a very smooth attenuation.

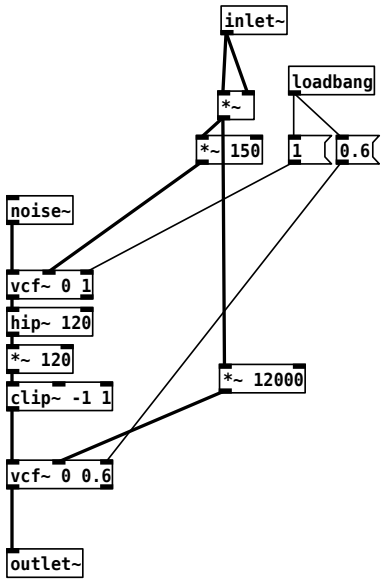


Figure 2.12: PureData patch for the engine "forced flame" synthesis

The highest block [inlet~] is an input changing from 0.1 to 1.0, which simulates the engine acceleration. It comes from an external Pd patch.
This input goes into a multiplication block [*~] which squares the value, just to have a non-linear model.
The output from this block – which is still between 0.1 and 1 – enters two other multiplications, one by 150 and the other by 12000. These values will be used later.
On the other side a [noise~] block produces white noise, the basis for the sound.
This noise goes through a first dynamic band-pass filter, the [vcf~] block, with a central frequency f_c shifting from 15 to 150 Hz (depending on the value coming out of the [*~ 150] block previously described), and Q=1 (BW=f_c following Formula 2.2). This is the filter which provides headroom for later frequencies introduced by overdrive.
The signal then goes through a fixed high-pass filter block [hip~] with cutoff frequency at 120 Hz, to avoid very low frequencies.
At this point the overdrive effect is applied: first the signal is amplified by a factor of 120, then it is clipped back to the standard amplitude interval [-1, 1]. This produces the high frequencies which create the "ripping" effect.
Finally, a dynamic band-pass filter with Q=0.6 and central frequency f_c shifting from 120 Hz to 12 kHz is used to tune the "amount" of overdrive effect: when 120 Hz, the bandwidth is very small and attenuation starts basically from the beginning of the spectrum (it acts as a very smooth low-pass filter with cutoff at f=220 Hz); when 12 kHz, BW=12000/0.6=20000, so attenuation starts outside of the interval [(12000 – 10000), (12000 + 10000)]→[2000, 22000]. In the latter case a lot of the overdrive effect is heard.

$$Q=\frac{f_c}{BW} \quad \rightarrow \quad BW=\frac{f_c}{Q} \tag{1.2}$$

Turbine sound:
The partial periodic components of the turbine are calculated experimentally, and at full thrust five main ones are chosen: 3097Hz, 4495Hz, 5588Hz, 7471Hz and 11000Hz.

These partials shift linearly in frequency as the engine speed changes. They have different amplitudes from one another.
Figure 2.13 shows the PureData patch used to produce the turbine sound.
Below, a description of the logical processing stages that the sound undergoes.

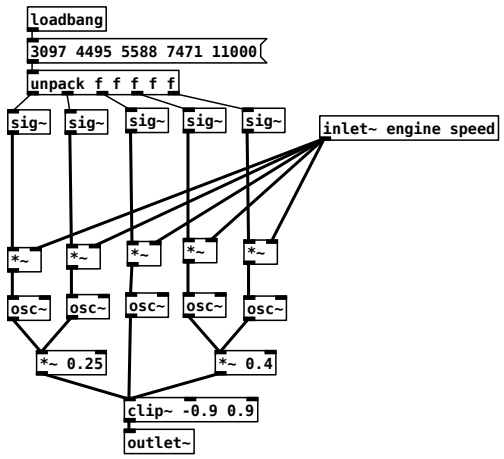


Figura 2.13: PureData patch for the engine turbine sound synthesis

First five [osc~] blocks are created, to model the five partials. Their frequency is given by multiplying the frequency value at full thrust by a factor in the range [0.1, 1.0] coming from outside, and corresponding to the engine speed.
For example, the first partial will shift linearly from 309.7 Hz to 3097 Hz.
Next, their amplitude is regulated: the first two are summed together (thus potentially reaching a maximum amplitude of 2), and this resulting signal is attenuated by a factor 1/4; the third partial is unchanged, and the last two are summed and then attenuated by 2/5.
The resulting three signals are summed together, potentially reaching a maximum amplitude of 0.25+1.0+0.4=1.65.
Finally a clipping to the range [-0.9, 0.9] is applied to create some harmonic components and add some color to the sound.

Note that although this is a good starting point for the turbine sound, these are experimental results coming from a specific type of jet engine sound analysis; the values of frequency or amplitude can be tweaked to better suit the desired sound.

3

Phase 2: Design of a solution on PC

Sound design is the art of creating or modifying sounds to employ in a variety of disciplines such as filmmaking, television production, video game development, sound art and many others [9].
The critical role of sound in performing arts can be traced back to ancient Greece (5^{th} century BC), where theaters had a precise semi-circular architecture to enhance acoustic quality in every point. Besides the actors in fact, there was also the choir, an essential component of the ancient Greek theater, which performed in the center part of this circle, the orchestra [10].
Modern sound design as we intend it today, however, has mainly developed over the last century, especially after the advent of digital technology. Sound started being recorded and represented as a discrete signal in time, an array of numbers each corresponding to a sample value.
This allowed manipulating sound in a totally different and revolutionary manner: sound could now be "processed" much more easily as a set of numbers and, most importantly, it could be created from scratch through an algorithm that generated those numbers.
This paved the way for an infinity of applications, some of which are an evolution of the classic performing arts, others are totally new and innovative fields, such as the subject of this thesis: giving a voice to electric vehicles.

Chapter 3 describes the second phase of this thesis work: the actual design and a first synthesis implementation of the sound for electric motorcycles, based on the previous analysis of general behaviors in a typical jet engine sound.
This stage consists in working on two main aspects:

- The sound design. Great care should be taken when creating this sound: it should be captivating, it should comply with the main criteria requested by the client and with the acceptability principles of a "comfortable" sound discussed in Chapter 2.
- The algorithm implementation. The sound is procedural, meaning it is generated real-time by a program taking different input parameters (velocity, acceleration). A

model should be built, first by choosing the relevant inputs for the purpose, and then by implementing the black box that outputs sound.

All of this was a sort of preliminary work for the final implementation on chip (Chapter 4): it was necessary to first have a fast way to synthesize different sounds by applying a trial and error methodology. It was done on PC in order to fully exploit the potential of a modern work environment, such as visualizing the sound spectrum and the waveform in a simple GUI or connecting any hardware interfacehaving faders to simulate inputs.

The first two subchapters discuss the different work environments and synthesis techniques that were considered. The actual implementation description starts at Subchapter 2.3.

3.1 Evaluation of different sound synthesis/sound design environments

When speaking of digital sound synthesis, a very large number of tools and software solutions come into play, each having different strong points: some concentrate their attention on sound synthesis, others on audio processing and feature extraction; some are used in a more scientific environment for statistical and signal analysis, others have more of a "musical" approach; some can provide really good and precise results, but are not real-time.
Given the amount of possible solutions that can be used, it is critical to focus on the technical requirements and on the aspects of sound processing that are most useful for this specific purpose. The development environment should be a good compromise between ease of use, flexibility and quality of the final result.

The first aspect to consider is the fact that the sound must be procedural. This means it should run based on an algorithm, thus drawing the attention towards solutions that are more "programming oriented" (such as Audio Libraries), and excluding typical musical synthesis environments such as DAWs [Digital Audio Workstation].
A procedural sound also means it should be generated real-time, maybe by sacrificing a little precision in return for a faster result.
A tool like Matlab for example is great for digital signal processing in general: it has a very rich library, sound can be manipulated very precisely and many statistical measures can be done. The problem is that it is more of a scientific tool, with great computational power to use in demanding analyzes where precision is critical: Matlab in fact is typically not used for real-time applications. Also, more high-level structures such as a chain of effects (series

of filters) or an LFO controlling amplitude modulation must be hard-coded and are not directly managed by the library, which concentrates on the processing computation optimization.

The second aspect to be aware of is that the library that is used should have both synthesis and processing functionalities, and it should be general purpose, not for specific applications. This excludes many audio libraries that specialize in sound analysis: Aquila for example focuses its attention on frequency domain analysis, Aubio instead is great for feature extraction (such as pitch and beat detection).
JUCE is a great synthesis library, but it is used mainly for audio plugins that are to be used in DAWs.
All of these libraries (and many more) can be found here: https://superpowered.com/audio-library-list

This research eventually led to two main solutions: PureData and the Processing 3.0 environment, specifically the Minim library.
PureData was already briefly presented in Chapter 2: it is a graphical programming language that aims at creating an intuitive and fast workflow for sound synthesis. It works with basic blocks connected to each other. Every block has a specific function: some generate numbers, others signals (the oscillator block [osc~]), some also have inputs and process what enters (the low-pass filter block [lop~]).
Through these blocks, small programs can be created (they are called 'patch' in the PureData world): they generate sound that can be controlled through input parameters given by faders, numbers or check boxes.
Given these features, PureData was mainly used to quickly try a solution before actually implementing it in code.

Processing 3.0 overview:
Processing 3.0 was instead the main development environment in this phase. It is an open-source graphical library and integrated development environment (IDE) developed by the MIT Media Lab and built mainly for electronic arts, new media art and visual design (definition from r).
The main point of strength of Processing is the ability to implement simple GUI and computer graphics in a very immediate way. Given that in this case the graphical part is just a support to the synthesis and not a primary need, Processing was a great compromise.
The language of Processing is Java, but other versions adapt the environment to JavaScript (p5.js) or Python (Processing.py).

A program in Processing is called 'sketch', and every sketch is a subclass of the PApplet Java class, where some methods are re-implemented by the user [11], and all other classes defined by the programmer are treated as inner classes.
The two main methods are:

- **`setup()`** - This is used to initialize variables that are declared before the `setup()` call.
  NOTE: variables declared in the `setup()` are not visible from outside.
- **`draw()`** - It is a method that is called just after the `setup()` and continuously after that, in a loop at a certain frequency (default is 60fps), until the program is stopped.
  This is the method that is usually used for the graphics displaying, which needs to be updated continuously.

Besides all the graphical libraries that Processing incorporates, there is an audio library called 'Minim' and developed by Damien DiFede [12] which provides real-time synthesis functionalities and a great high-level management of audio processing chains: every object can be connected to another object through a simple call of the `'patch()'` method.
The object from which the connection starts is the one calling the method; the other end of the connection is the object passed inside the `'patch()'` method.

UGen and Oscil classes:
The basic building block of Minim is the `'UGen'` object: almost everything is a UGen, from the oscillator to the processing effects (a delay for example). A UGen is a 'Unit Generator', in the sense that it creates or modifies a single sample value.
`'Oscil'` is the UGen used to create a simple oscillator; its constructor takes as input parameters the frequency, the amplitude (from 0.0 to 1.0) and the waveform.
A UGen can be patched to another UGen, or to a so called UGenInput; every UGen can in fact have different UGenInputs, which are basically its controlling parameters. An oscillator for example has four main UGenInputs: amplitude, frequency, phase and offset.
Considering the following example, a Low Frequency Oscillator (LFO) controls the amplitude of another oscillator:

```
wave = new Oscil(440, 0.1f, Waves.SINE);
LFO = new Oscil(5, 0.1f, Waves.SINE);

LFO.patch(wave.amplitude);
```

Code snippet 3.1: 'patch' demonstration in Minim

In *Code snippet 3.1* the two objects `'wave'` and `'LFO'` are `'Oscil'` objects, which inherits from the UGen class. The `'wave.amplitude'` instead is of type UGenInput.

The Minim, AudioOutput and Summer classes:

The `'Minim'` class is the starting point for all of the audio input/output in the program. Whether it is reading from an audio file, or from an in-line microphone, or outputting to external monitors, everything is managed by the Minim class. In fact, Minim has methods that return objects for each one of these functionalities.

To output sound for example, an `'AudioOutput'` object is needed. This object is returned by the `'getLineOut()'` method of the `'Minim'` class, and it receives all of the parameters that are necessary to characterize the technical aspects of the digital audio that is used.

Here is a simple sketch that outputs a 440 Hz and an 880 Hz sine wave:

```
Minim minim;
AudioOutput out;

Oscil wave1, wave2;
Summer mixer;

void setup(){
  minim = new Minim(this);
  out = minim.getLineOut(Minim.STEREO, 512, 44100);

  Mixer = new Summer();
  wave1 = new Oscil(440, 0.1f, Waves.SINE);
  wave2 = new Oscil(880, 0.1f, Waves.SINE);

  wave1.patch(mixer);
  wave2.patch(mixer);
  mixer.patch(out);
}

void draw(){
}
```

Code snippet 3.2: Minim sketch of a 440 Hz and an 880 Hz sine wave that are sent to audio output

In the *Code snippet 3.2* the `'getLineOut()'` method receives as input parameters: the audio mode (either MONO or STEREO), a buffer size of 512 samples and a sampling frequency of 44100 Hz.

When patching the wave to the AudioOutput `'out'` object, what the Minim library does is to first fill a buffer of samples (of size 512 in the example); as soon as this buffer is full, it sends it to the AudioOutput, which will manage the playback on the computer.

The `'Minim'` class could also return other audio inputs/outputs, for example the `'AudioIn'` object, which receives sound from the microphone.

Finally, the `'Summer'` class works as a mixer: multiple oscillators can be patched to it and it is then patched to the AudioOutput `'out'`.

3.2 Evaluation of synthesis techniques:

In order to create a sound, many different synthesis techniques can be used, each one with their own advantages and drawbacks: some for example can be more CPU intensive but give a better final sound, others are computationally less demanding but the quality of the resulting sound is not so precise. Some are harder to implement and optimize the CPU usage, others are immediate but could require more CPU.
Once the development environment is defined, the next step is to explore all of the different synthesis techniques in order to have a rough idea of the quality of synthesized sound vs the CPU usage. It's important to consider this aspect already at this stage, in view of a future implementation of the algorithm on a chip, whose resources are much more limited than those of a PC.

One of the main aspects to work on when synthesizing a sound is the "timbre". This can be defined as the frequency envelope of the spectrum resulting from all of the harmonics and the inharmonic partials (Figure 3.1).

Perceptively, the 'timbre' is the property of sound that differentiates a sound from another when they have equal pitch.
In reality timbre is given by a combination of the spectrum envelope and the time envelope (the evolution of a sound amplitude in time): in fact if the sound of a trumpet is played backwards it sounds differently because the time envelope is different, even if the spectrum and the pitch are equal.
In this case though, only spectrum envelope is considered because the time envelope can be controlled later on, independently from the synthesis technique.

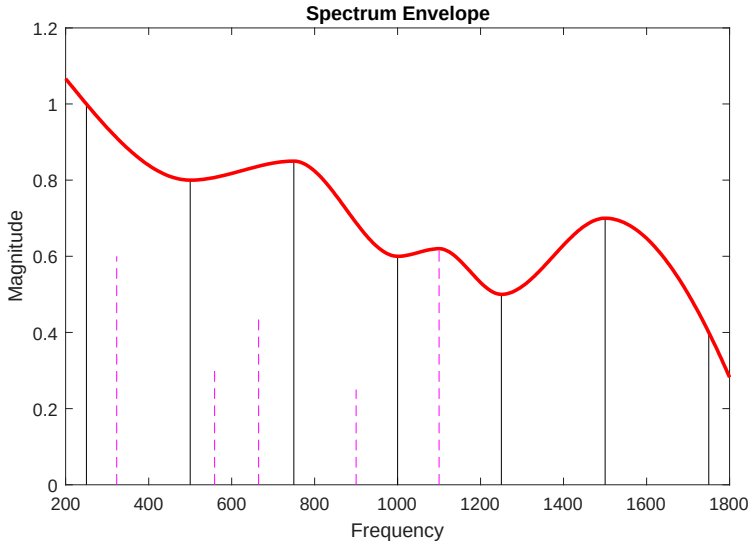


Figure 3.1: Spectrum envelope of the frequency components. It contributes to define the 'timbre' of a sound.

One of the main sound synthesis techniques is the "additive synthesis". It consists in adding sine waves together in order to precisely shape the spectrum and obtain the desired timbre: each sine wave in fact ideally represents a single peak in frequency domain.
This approach is intuitive and it allows a very fine tuning of the synthesized sound, but if the components become too many the CPU load can increase a lot: each sine wave is in fact processed independently.
When creating complex sounds, the components are generally a lot (in the order of a few hundreds), so in these cases additive synthesis is usually used in combination with other techniques.

Another very common synthesis technique is the "subtractive synthesis". The approach is opposite to that of the additive one: it consists in attenuating the partials and harmonics of a sound that is rich in frequency components, possibly white noise.
The attenuation is done through filters, which shape the spectrum in order to obtain the desired timbre and pitch.
This approach in general is computationally more efficient as compared to additive synthesis, but the resulting sound cannot be tuned as precisely. Digital filters in fact have a limited roll-off, depending on the number of poles that are used in that filter (and consequently the complexity): a one pole filter (filter of first order) has a -20dB/decade roll-off, and each additional pole changes the roll-off of -20dB/decade.
This limits the precision when defining beginning and end of a band, especially when the band is very narrow and should tend to a peak shape. To increase precision, filters of higher order must be used, and this increases the computational load.

Very often a combination of additive and subtractive synthesis is used, in order to make the most of the advantages of each technique. Additive synthesis is well suited for the modeling of deterministic components of a sound (sinusoids), while subtractive synthesis is most effective for the stochastic components (noise).

Two other synthesis techniques that are often used are "frequency/amplitude modulation" (FM/AM). Amplitude modulation consists in summing to the amplitude of a signal (typically a sine wave called "carrier") the amplitude of another sinusoid ("modulator") following the relation 3.1.

$$A_c = m \cdot A_m \qquad with: \quad 0 < m < 1.0 \tag{3.1}$$

$$A_c = Carrier\ amplitude$$
$$A_m = Modulator\ amplitude$$

When the frequency f_m of the modulator is below the audible frequency (<20Hz), the effect that is produced is called "tremolo": it's a periodic change in amplitude that is perceivable by human ear.
When f_m increases above 20 Hz, two partials are created in the spectrum, adjacent to the central carrier frequency f_c at a distance of f_m.
Frequency modulation instead consists in summing to the frequency of the carrier signal a value that is equal to the amplitude of the modulator in that time instant: the frequency will then shift back and forth depending on whether the modulator amplitude A_m is positive or negative.
If the frequency f_m is below 20Hz, the produced effect is called "vibrato". When the frequency f_m increases above 20Hz, an infinite number of adjacent partials is generated, with magnitude decreasing following the Bessel functions. The resulting modulated signal is described by the Chowning Formula 3.2, which will not be discussed but is showed to give an idea of the components that contribute to the final signal:

$$FM(t) = A_c \cdot \sum_{k=-\infty}^{+\infty} \underbrace{J_k\left(\frac{A_m}{\omega_m}\right)}_{Bessel\ function} \cdot \sin(\underbrace{(\omega_c + k \cdot \omega_m)}_{Freq\ of\ harmonics} t) \tag{3.2}$$

The main point of force of AM/FM synthesis is that it allows creating complex sounds having many harmonics with only two simple sine waves, so a very low computational complexity.

In this thesis work, amplitude modulation was used a lot, but only in the "tremolo" version, where modulator signal has frequency <20Hz, taking the name of LFO (Low Frequency Oscillator).

The last synthesis technique that is discussed in this subchapter consists in modeling directly the spectrum and then applying the Inverse Fourier Transform to generate the sound.
In reality this is a subclass of the additive synthesis, because also here different components must be added to shape the spectrum. Differently from additive synthesis though, each component is described as a mathematical formula, and not as a separate oscillator object that runs in loop.
The complexity here lies in the implementation: finding the correct mathematical formula to best approximate each spectral component that is wanted. A sine wave for example can be modeled in frequency domain as a very narrow Gaussian function.
Once all the functions are defined, the only remaining step is to apply an Inverse Fourier Transform algorithm such as IFFT, and generate a buffer of samples.
A frequency shift can thus be modeled as a simple translation on the x-axis of the spectrum (the frequency), and the amplitude can be adjusted by scaling the magnitude y-axis.
This approach was tested a lot in the initial stage of the design, and the results were not bad: it wasn't computationally heavy, as most of the power was dedicated to the IFFT, and the sound could be precisely tuned.
The main limitation of this technique occurs when trying to simulate effects that usually happen in the time domain. The clipping of a sine wave for example introduces many higher partial and harmonic components, but it's very hard to even predict the behavior or the distribution of these components in the spectrum.
This led to discard this approach as the main one, although keeping it in mind in the future for some parts of the sound synthesis that didn't need time-domain processing.

All of the tests that were carried out in this phase made the choice converge mainly towards additive and subtractive synthesis, in conjunction with some low frequency amplitude modulation to give more "movement" to the sound. By using simple sine waves where a peaking frequency was needed, and waves with a more complex spectrum (square wave) or noise combined with filters, a satisfying sound could be generated with a few dozens of oscillators. This is absolutely acceptable in terms of CPU load and it allows fine tuning and regulation of the sound.
Also, most of the synthesis audio libraries very well support additive and subtractive synthesis.

3.3 Modeling of the motorcycle system and input parameters:

As already stated, the sound of the electric motorcycle must be procedural, meaning it should change dynamically as the input parameters change.
It is thus crucial to choose these inputs carefully and obtain a reasonable model of a motorcycle.

The first parameter that immediately comes to mind is the motorcycle velocity, the speed. Intuitively in fact any engine varies its sound when working at a higher rate, typically introducing more noise and louder sounds.
One could think velocity is directly controlled by the acceleration amount applied by the motorcyclist (the angle of the handle in a motorcycle, or the inclination of the pedal in a car), but if considering the motorcycle purely from a sound perspective, the two parameters are independent.
In fact, another way of interpreting the handle inclination is as a calculation of the engine power (kW) applied in each time instant: when handle is at 0° no power is supplied, when it is at maximum inclination instead, the engine works at maximum power.
For example, there could be a situation in which the motorcycle is not moving but the handle is fully turned just to generate the roar of the engine. In this case the sound should be loud and powerful, but shouldn't give an idea of acceleration.
There could also be the opposite situation, in which the motorcycle is moving downhill but the acceleration applied by the engine is close to null. In that case the sound should give an idea of acceleration and speed, without however giving that sense of power from the engine.
Besides these extreme cases, any intermediate scenario should be considered, such as the motorcycle going uphill, thus using more engine power (more noise) for a lower speed.
This is the reason behind the choice of keeping these two input parameters separate: speed and inclination of the handle can be captured by two different sensors on the motorcycle.

Now that the two main parameters are defined, it is important to note that also the way they vary is important: turning the acceleration handle very fast produces a sound that is different from the one generated by a slow power increase. In the same way, a slow acceleration has a different sound than that of an immediate one.
This can be modeled as the derivative of the two input parameter: how fast they change in a time interval, not only the value at each time instant.
The derivative of velocity is acceleration, while the derivative of the handle inclination is the angular velocity: how fast the angle changes.

In order to have a realistic model, all of these four parameters must be considered. The model can be seen in Figure 3.2.

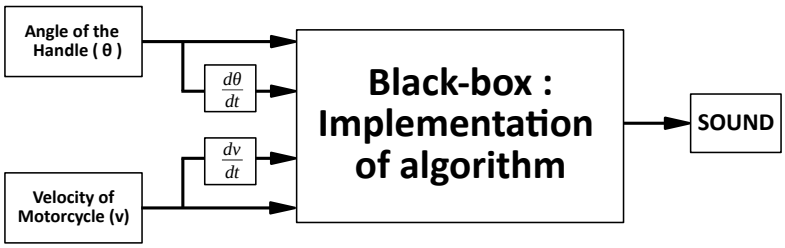


Figure 3.2: Model of the motorcycle for the engine sound. It receives four input parameters: angle of the handle, velocity of the motorcycle and their derivatives.

3.4 First algorithm implementation based on qualitative analysis

This subchapter discusses the first sound implementation that was done in the Processing 3 environment with the Minim library. It is based uniquely on the qualitative analysis that is discussed in Subchapter 2.1, and it is just a first step towards the final sound.
Before going into the details of the code, a brief general overview of the different components of the algorithm is here presented.

The first thing to note is that at this stage only two input parameters were considered (one of the two main ones and its derivative) in order to simplify the problem. Being this the case, it's irrelevant whether it's the velocity or the inclination of the handle, but for sake of consistency with the next chapters – where this implementation is resumed and updated with other parts – the input parameters will be considered as the handle inclination and its derivative.

The general behavior of the sound, as seen in Subchapter 2.1, has many sinusoidal components that undergo a frequency shift in time. In the actual implementation this translates into many oscillators that change their frequency depending on the handle input parameter.

To give more the idea of an engine that works following a cyclic process, each of these sine waves are modulated in amplitude with an LFO, whose frequency also increases with the handle input parameter; this behavior simulates the increase in cycles/second of the engine when taking speed.
The other main component in the sound is noise. This noise should not be uniformly present throughout the whole spectrum, but it should be shaped and modulated.
A series of filters are used to shape basic white noise in the spectrum. Some of these filters are static, others are dynamic and change some of their parameters – such as cutoff frequency – following the handle input.

3.4.1 GUI implementation for direct analysis and visual debug

When synthesizing a new sound from scratch, being able to see the waveform and – most importantly – the spectrum is very important. It gives a feedback other than the acoustic one on the sound that is created and it works as a sort of visual debug: through the spectrum one can immediately see if the frequency components behave as expected following the implementation.

In the Processing environment, setting up simple GUIs is pretty straightforward and, because of this, it was decided to integrate one with the following essential components: the waveform, the spectrum and two faders, one for handle angle and one for velocity.
This section goes through the implementation of this GUI, which can be seen in Figure 3.3.

The basic parameters to set when creating a window in Processing are: window size, refresh time and title of the window. This is implemented in the `'setup()'` function:

```
void setup(){
  size(1600, 900);
  frameRate(45);
  frame.setTitle("Sound Prototype");
}
```

Code snippet 3.3: Setup of a 1600x900 window in Processing

Everything that is drawn (from the waveform to the spectrum) in the window will be refreshed with a frequency of 45Hz.
The implementation of the objects that are drawn is in the `'draw()'` method.

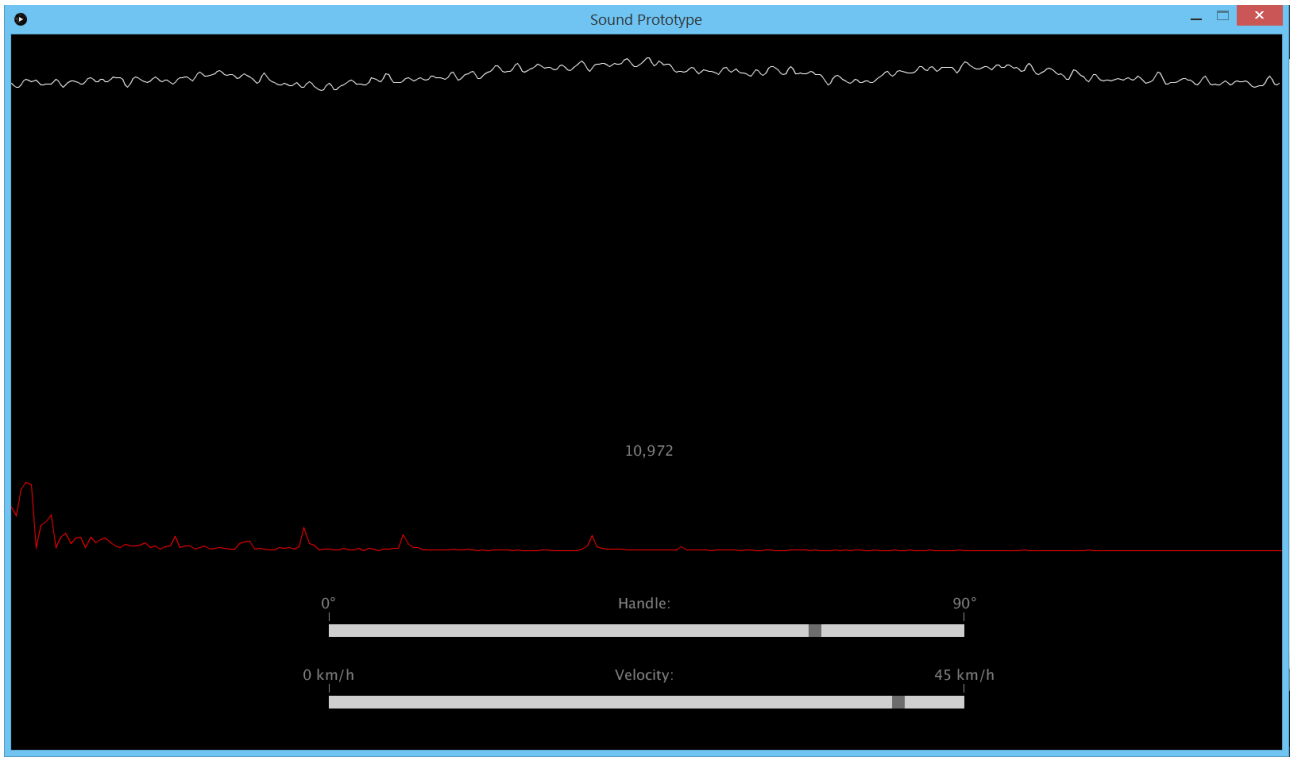


Figure 3.3: GUI screenshot implemented for visual feedback. From bottom to top: the speed bar from 0 km/h to 45 km/h, the handle inclination from 0° to 90°, the spectrum in red, a number outputting current speed in meters/s, the waveform in white.

The waveform:

The waveform samples are taken from the `'AudioOutput out'` object seen at the end of Subchapter 3.1. Before sending the samples to the loudspeakers, this object collects them into a buffer of a given length set by the programmer; this buffer is accessible, and the sample values can thus be drawn with this code:

```
void draw(){

  background(0); // Sets background color to BLACK
  stroke(255);   // Sets drawing line color to WHITE
  strokeWeight(1); // Sets drawing line weight to 1

  // DRAW WAVEFORM
  for(int i = 0; i < out.bufferSize() - 1; i++)
  {
    // find the x position of each buffer value
    float x1  =  map( i, 0, out.bufferSize(), 0, width );
    float x2  =  map( i+1, 0, out.bufferSize(), 0, width );
  // Draw a line from one buffer position to the next for both channels
    line( x1, 50 + out.left.get(i)*50, x2, 50 + out.left.get(i+1)*50);
  }
```

Code snippet 3.4: Drawing the waveform in Processing

After setting basic colors and weights, a loop explores each sample present in the buffer of `'out.left'`, the left audio channel; in this case the audio is mono, so it doesn't make a difference whether the left or right channel is considered.
At each iteration, the current and next position in the buffer are first mapped to a pixel row number in the window and then stored in `'x1'` and `'x2'` respectively. The mapping is done through the `'map()'` function: it takes the current value which is between 0 and buffer size and maps it to a new value between 0 and the width of the window in pixels.
The corresponding two 'y' values are obtained by reading the sample value at the current position in the buffer. Note how they are scaled by a multiplicative factor of 50, and then translated of 50 pixels down (y=0 corresponds to the top pixel row of the window).
Now that the coordinates of two points are known, a line connecting them can be drawn. This process is repeated until all samples are read.

The spectrum:
In order to draw the spectrum, a Fourier Transform must be done at each refresh cycle of the screen. This operation is done through the `'FFT'` class in Minim.
First an FFT instance is declared, and initialized inside the `'setup()'`:

```
FFT fft;
int bsize = 512;
.
.
setup(){
  fft = new FFT(bsize, 44100);
}
```

Code snippet 3.5: 'FFT' class initialization in Minim

The `'bsize'` is important: it's the number of frequency bands in which the spectrum is divided and it must be a power of 2, as requested by any FFT algorithm. In this case a spectrum that considers frequencies from 0 Hz to 44100/2=22050 Hz is divided in 512 equal parts, so each band is 22050/512=43.07 Hz wide.
In order to perform the algorithm, the method `'forward(out.left)'` is called. This takes as input the sample buffer and it fills up another buffer of frequency bands having the magnitude value in each position.
At this point, the spectrum can be drawn like this:

```
// DRAW THE SPECTRUM
  noFill(); // Disables geometric figure color filling
  stroke(255, 0, 0); // Red line
  beginShape();

  for(int i = 0; i < fft.specSize(); i++)
  {
    float y = map(fft.getBand(i), 0, fft.specSize(), height-250, 0);
    float x = map(i, 0, fft.specSize()-1, 0, width);
    vertex(x, y);
  }
  endShape();

  fft.forward(out.left);
```

Code snippet 3.6: Drawing the spectrum in Processing

The magnitude value of each *i-th* frequency band is read through the `'fft.getBand(i)'` and the spectrum size of 512 is returned by `'fft.specSize'`.
Variable `'x'` is the frequency band number scaled to the window's width, and `'y'` is the magnitude value of that frequency band scaled to the interval [0, (height - 250)].
The vertex having those coordinates is then drawn.

The scrollbars:
The scrollbars are used to simulate the value of the input parameters. They can be operated by either using the mouse or by connecting an external hardware fader that works with the MIDI protocol.
The scrollbar component is a separate class that was programmed following an example implementation on the Processing official site [13]. The class name in this project is `'Hscrollbar'`.
The class fields and the constructor are:

```
class HScrollbar {
  int barWidth, barHeight;  // width and height of bar
  float barPos_x, barPos_y;  // x and y position of bar
  float sliderPos, newSliderPos;  // x position of slider
  float minSliderPos, maxSliderPos; // max and min values of
                                    // slider
  int loose;  // how loose/heavy
  boolean isOver;  // is the mouse over the slider?
  boolean isLocked;  // Mouse click is pressed on slider
  String title, initValue, endValue;
```

```
HScrollbar (float xp, float yp, int barW, int barH, int l,
            String name, String start, String end) {
  barWidth = barW;
  barHeight = barH;
  barPos_x = xp - barW/2;
  barPos_y = yp - barHeight/2;
  sliderPos = barPos_x  ;
  newSliderPos = sliderPos;
  minSliderPos = barPos_x;
  maxSliderPos = barPos_x + barWidth - barHeight;
  loose = l;

  title = name;
  initValue = start;
  endValue = end;
}
```

Code snippet 3.7: Internal fields and constructor of the 'HScrollbar' class

Most of the fields are immediate and describe geometrically the object in the window: width and height of the scrollbar, position of it in the window slider position within the scrollbar.
The important field here is `'loose'`: it's a number controlling how smooth the slider should transition from a position to another. It acts as a low-pass filter for float numbers by filtering fast transitions and smoothing them out. The tuning of this parameter is key to controlling the response of the model to the derivative for example of handle inclination, that is "how fast does this handle transition from one angle θ_1 to another one θ_2".

The `'HScrollbar'` class then has different methods to control it, the most important being `'update()'`. This method is called repeatedly from the `'draw()'` function: it checks if the user is clicking on the slider and moving it, and it updates the slider position accordingly.

Here is the implementation:

```
void update() {
  if (overEvent()) {
    isOver = true;
  } else {
    isOver = false;
  }
  if (mousePressed && isOver) {
    isLocked = true;
  }
  if (!mousePressed) {
    isLocked = false;
  }
  if (isLocked) {
    // constrain() -> Constrains a value to not exceed a
    // maximum and minimum value
    // This can happen for example if the mouse goes
    // outside of the window
    newSliderPos = constrain(mouseX-barHeight/2,
                                    minSliderPos, maxSliderPos);
  }
  if (abs(newSliderPos - sliderPos) > 1) {
    sliderPos = sliderPos + (newSliderPos-sliderPos)/loose;
  }
  if(newSliderPos - sliderPos < -1){
    sliderPos = sliderPos + (newSliderPos-sliderPos)/16;
  }
}

float constrain(float val, float minv, float maxv) {
  return min(max(val, minv), maxv);
}

boolean overEvent() {
  if (mouseX > barPos_x && mouseX < barPos_x+barWidth &&
     mouseY > barPos_y && mouseY < barPos_y+barHeight) {
    return true;
  } else {
    return false;
  }
}
```

Code snippet 3.8: 'update()' method of 'HScrollbar'

First, the condition whether the mouse is hovering over the scrollbar area is checked: this is done through the `'overEvent()'` method, which compares the mouse coordinates with the area.

Then, if this is true and the mouse is pressed down, it sets a "lock" on the slider which indicates that the slider is selected and could potentially move.
Note how `'mousePressed'` is a function that is automatically called by the Processing environment when it senses a mouse button pressed.
If the "lock" is active, then it proceeds to calculate the new slider position depending on the mouse position, after constraining it in the scrollbar pixel interval.
At the end, it updates the actual slider position, but instead of adding directly the difference between current position and new position, it adds a portion of this difference (a portion based on the loose or on the number 16). Each time the `'update()'` method is called, if mouse is still pressed, it will decrease the gap between current position and new position of the slider.
This results in a smoothed movement of the slider in the scrollbar: it regulates the speed of transition, acting on that derivative parameter that was introduced in Subchapter 3.3. If the motorcycle driver in fact turns the handle very fast, the resulting sound shouldn't have a fast transition from 0 to full thrust, but it should change slowly.
Also note how this transition differs according to whether it is positive or negative (accelerating or decelerating).

The position can then be obtained from the main file, by calling the `'getPosPrecise()'` method of `'Hscrollbar'`:

```
float getPosPrecise(){
  return (sliderPos-barPos_x);
}
```

Code snippet 3.9: 'getPosPrecise' method of the 'HScrollbar' class

Another way to update slider position is through a MIDI interface with a slider that sends values between 0 and 127. Of course, in order to use and recognize the external hardware interface, a MIDI library must be included in the project; "The MidiBus" library was chosen because it is included in the Processing framework.
The external device that was chosen is an "Akai APC Mini", which has 9 different faders and can be connected via USB to the computer.
The `'updateByMIDI()'` method is:

```
void updateByMIDI(int value){
  newSliderPos = map(value, 0, 127, barPos_x, barPos_x + barWidth -
                          barHeight);
}
```

Code snippet 3.10: 'updateByMIDI' method of 'HScrollbar' class

As it can be seen, it just calculates the 'newSliderPosition'. The normal 'update()' method will in fact still be called repeatedly from 'draw()': it will skip the mouse checks and directly start applying the smooth transition.
This is how the 'MidiBus' object is created and how the 'updateByMIDI()' method is called in the main:

```
MidiBus myBus;

void setup(){

  myBus = new MidiBus(this, 0, 1); // Parent 'this',
         // device input number '0', device output '1'
}

void draw(){…}

void controllerChange(int channel, int number, int value){

  if(number == 48){
    handleBar.updateByMIDI(value);
  }
  if(number == 49){
    speedBar.updateByMIDI(value);
  }
}
```

Code snippet 3.11: 'MidiBus' initialization and external MIDI device monitoring

Note how the function 'controllerChange' is automatically called by the library when it senses a parameter has changed on the MIDI device. The 'number' is the physical parameter id (which fader was moved), and 'value' is the value that the parameter has assumed.

The last method that remains to analyze is the 'display()'. It is called repeatedly from the 'draw()' function and it draws the scrollbar on the window, based on its basic parameters.

```
void display() {
  noStroke();
  fill(204);

  rect(barPos_x, barPos_y, barWidth, barHeight);
  if (isOver || isLocked) {
    fill(0, 0, 0);
  } else {
    fill(102, 102, 102);
  }
  rect(sliderPos, barPos_y, barHeight, barHeight);

  fill(128);
  textSize(18);
  text(title, barPos_x + barWidth/2, barPos_y - 20);
  textAlign(CENTER);
  text(initValue, barPos_x, barPos_y - 20);
  text(endValue, barPos_x + barWidth, barPos_y - 20);
  stroke(128);
  line(barPos_x, barPos_y - 15, barPos_x, barPos_y - 5);
  line(barPos_x + barWidth - 1, barPos_y - 15, barPos_x +
            barWidth - 1, barPos_y - 5);
}
```

Code snippet 3.12: 'display()' method of the 'HScrollbar' class

This method first fills the background of the scrollbar with a color value of 204 (almost white), then it colors the slider rectangle with either 0 (black) or 102 (grey), depending on whether the user is displacing the slider with the mouse or not.
Finally some text is added, with the appropriate text formatting, and two small vertical segments are added at the beginning and end of the scrollbar.

3.4.2 MyWave class and Why:

Most of the oscillators that are used for the additive synthesis are subject to an amplitude modulation, to give that dynamic and cyclic sense of a running engine.
Already at an early stage into implementing the algorithm, it was clear that creating two oscillators (the carrier and the modulator) in the main sketch for each component to add could become confusing, especially when giving names and patching each modulator to the carrier amplitude.
This is why the `'MyWave'` class was created: it contains a normal oscillator that will be the audible one and a modulator oscillator (an LFO).

It then internally patches the LFO oscillator to the carrier amplitude, and it contains all of the methods that are useful to set amplitude and frequency of the two oscillators.

Here below are the class fields and the constructor:

```
class MyWave{

  private
    Oscil LFO_AmplModulation;

  public
    Oscil wave;

   MyWave(float f, float a, Waveform wf, float f_LFO, float a_LFO){
      wave = new Oscil(f, a, wf);

      LFO_AmplModulation = new Oscil(f_LFO, a_LFO, Waves.SINE);
      LFO_AmplModulation.offset.setLastValue(a);

      LFO_AmplModulation.patch(wave.amplitude);
    }
```

Code snippet 3.13: Fields and constructor of 'MyWave' class

The constructor receives as inputs the: initial frequency of carrier wave, initial amplitude of carrier, the waveform of carrier, the initial frequency of the modulator and the initial amplitude of the modulator.
Given all of these values, the two `'Oscil'` object are created, and the patching between the two is done.
Note how the offset of modulator (the DC component) is set equal to the carrier amplitude: the result that is wanted in fact is to have variation from the peak value of the carrier amplitude, and not from 0

The methods present in this class are:

```
   void setFrequency(float newFreq){
      wave.setFrequency(newFreq);
    }

    void setAmplitude(float newAmpl){
      wave.setAmplitude(newAmpl);

      LFO_AmplModulation.offset.setLastValue(newAmpl);
    }
```

```
    void setLFO_Frequency(float newFreq){
  LFO_AmplModulation.setFrequency(newFreq);
}

void setLFO_Amplitude(float newAmpl){
  LFO_AmplModulation.setAmplitude(newAmpl);
}

void setLFO_Ampl_Freq(float newAmpl, float newFreq){
  LFO_AmplModulation.setAmplitude(newAmpl);
  LFO_AmplModulation.setFrequency(newFreq);
}

void changeLFO_Shape(Waveform wf){
  LFO_AmplModulation.setWaveform(wf);
}
```

Code snippet 3.14: Methods of the 'MyWave' class

The first two methods are used to set frequency and amplitude of the carrier wave to a new value. Note how in the `'setAmplitude()'` also the offset is updated accordingly.
The other three methods set the frequency and amplitude of the LFO modulator: the first two each set one of these parameters, the third one sets both (it avoids calling two methods in some situations).
The last method just changes the LFO modulator waveform.

3.4.3 Oscillators:

This section discusses the periodic components of the first implementation, which can also be seen as the deterministic part of the sound.
It is based on the analysis made in Chapter 2, particularly on the model that was built in Section 2.1.1. In that model, at the beginning of the transient, there is a harmonic sound with virtual pitch at 125 Hz and 6 different harmonics (equispaced of a Δf=125Hz); at full thrust these frequencies have all doubled.

In the following implementation it was decided to reduce the number of oscillators and work on a wider frequency band: the fundamental frequency is at 250 Hz and it has 3 harmonics (500Hz, 750Hz and 1000Hz). At full thrust all of these frequencies double, so the highest one reaches 2000Hz. This is basically a sound that has double the pitch (1 octave higher) compared to that of the original model.

Also, to add even more harmonics, some of these oscillators are triangular instead of being sinusoidal: a triangular wave contains all odd harmonics with magnitude decreasing of $1/k^2$.
The sound that is thus synthesized is good enough to get an idea for this prototyping phase on Processing 3. The number of harmonics and their values are however changed in the definitive implementation.
Some partial components with different are also added to introduce some inharmonicity.
Here below the declaration and initialization of all the oscillators used and their connections to the AudioOutput object `'out'` through the `'mixer'`:

```
MyWave wave1_handle, wave2_handle, wave3_handle, wave4_handle;
MyWave waveLowFreq;
MyWave waveHighFreq;
MyWave subWave;

LowPassSP LP_waveLowFreq_STATIC;

void setup(){
  …
  Waveform triangle = Waves.TRIANGLE;
  Waveform sine = Waves.SINE;
  wave1_handle = new MyWave(250, 0.01f, sine, 0.0f, 0.00f);
  wave2_handle = new MyWave(500, 0.01f, triangle, 0.0f, 0.00f);
  wave3_handle = new MyWave(750, 0.01f, sine, 0.0f, 0.00f);
  wave4_handle = new MyWave(1000, 0.01f, triangle, 0.0f, 0.00f);

  LP_waveLowFreq_STATIC = new LowPassSP(200, 44100);
  waveLowFreq = new MyWave(116, 0.1f, Waves.SQUARE, 2.5, 0.05);
  waveHighFreq = new MyWave(3000, 0.01f, Waves.TRIANGLE, 0, 0);

  subWave = new MyWave(80, 0.15, Waves.SINE, 2.5, 0.05);

  // Patching
  wave1_handle.wave.patch(mixer);
  wave2_handle.wave.patch(mixer);
  wave3_handle.wave.patch(mixer);
  wave4_handle.wave.patch(mixer);

  waveLowFreq.wave.patch(LP_waveLowFreq_STATIC).patch(mixer);
  waveHighFreq.wave.patch(mixer);
  subWave.wave.patch(mixer);
  mixer.patch(new LowPassSP(11000, 44100)).patch(out);
  …
}
```

Code snippet 3.15: Declaration, initialization and patching of the handle waves in first Processing implementation

Four harmonic waves are first created, they take the name of `'waven_handle'`, with 'n' being the number of the wave and the word 'handle' to differentiate them from the waves that are used for the velocity input parameter.
They are all initialized with amplitude 0.01, and their LFO is initialized at 0 frequency and amplitude (no amplitude modulation). Also `'wave2_handle'` and `'wave4_handle'` are triangular waves.
Two partial waves are then added:

- A low frequency wave: it starts at 116 Hz, with amplitude 0.1. The waveform is square, to get some harmonics out of this oscillator: these harmonics are then gently filtered by a low-pass filter of first order, `'LP_waveLowFreq_STATIC'` with cutoff frequency at 200Hz. The LFO has initial frequency 2.5 Hz and amplitude 0.05.
  This wave has a much higher initial amplitude than the one of the harmonics (x10), this is because human ear is less sensitive to low frequencies. From Figure 3.4 it can be seen that two frequencies at 116 Hz and 250 Hz have the same loudness (fall on the same isophonic curve) if the first one is increased of ~+10dB.
  An increase of +10dB means to multiply signal intensity by 10, this can be seen in Formula 3.3.

$$10\log\left(\frac{10\cdot I}{I_0}\right) = 10\log\left(\frac{I}{I_0}\right)+10\log(10) = n\,dB + 10\,dB \tag{3.3}$$

- A high frequency wave: it starts at 3000Hz and 0.01 amplitude. It's a triangle wave to have some harmonics. Its LFO is initially set to 0.

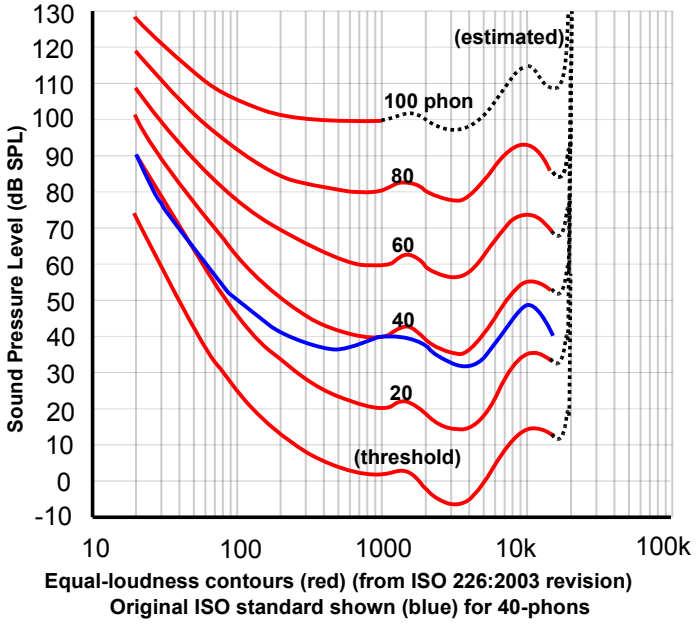


Figure 3.4: Equal loudness curves. From Wikipedia: r

To fill the sub-bass frequencies (<100Hz), the 'subWave' is created: it has initial frequency of 80Hz, amplitude of 0.15. Its LFO is initialized at 2.5Hz and amplitude 0.05. Finally all the necessary patches are done.

At this point all of the oscillators are initialized. The next step is the processing of them inside the 'draw()' function.
The handle inclination input parameter is read from the handle scrollbar, called 'handleBar', through the method 'getPosPrecise()' seen in Section 3.4.1; this value is then mapped to interval [0, 1000] and stored in 'modFactor_MASTER_HANDLE'.

The first processing that occurs is the frequency shift of the waves:

```
float modFactor_MASTER_HANDLE = map(handleBar.getPosPrecise(), 0,
                                    784, 0, 1000);

float wave1_Freq = map(modFactor_MASTER_HANDLE,0, 1000, 250, 500);
float wave2_Freq = map(modFactor_MASTER_HANDLE,0, 1000, 500, 1000);
float wave3_Freq = map(modFactor_MASTER_HANDLE,0, 1000, 750, 1500);
float wave4_Freq = map(modFactor_MASTER_HANDLE,0, 1000,1000, 2000);

wave1_handle.setFrequency(wave1_Freq);
wave2_handle.setFrequency(wave2_Freq);
wave3_handle.setFrequency(wave3_Freq);
wave4_handle.setFrequency(wave4_Freq);

waveLowFreq.setFrequency(116 + modFactor_MASTER_HANDLE*0.15);
waveHighFreq.setFrequency(3000 + modFactor_MASTER_HANDLE*3);

float subWaveFreq = map(modFactor_MASTER_HANDLE, 0, 1000, 80, 150);
subWave.setFrequency(subWaveFreq);
```

Code snippet 3.16: Frequency shift of waves in first Processing implementation

Once the input parameter is read from scrollbar, it is used to tune all the frequencies of the waves. For example to set frequency of 'wave1_handle' the modulation factor is read (values between 0 and 1000), this value is mapped to fit in the interval [250, 500].
The 4 main ones all end with twice their initial frequency.
The 'waveLowFreq' transitions from 115Hz to 150Hz (=1000*0.15), while the 'waveHighFreq' transitions from 3000Hz to 6000Hz (=1000*3).
Finally 'subWave' transitions from 80Hz to 150Hz.

The next processing stage is the change in amplitude of the waves:

```
float wavesHandle_Ampl = map(modFactor_MASTER_HANDLE, 0, 1000,
                             0.01, 0.1);

wave1_handle.setAmplitude(wavesHandle_Ampl);
wave2_handle.setAmplitude(wavesHandle_Ampl);
wave3_handle.setAmplitude(wavesHandle_Ampl);
wave4_handle.setAmplitude(wavesHandle_Ampl);


float waveHighFreq_Ampl = map(modFactor_MASTER_HANDLE,0,1000, 0.01, 0.1);
waveHighFreq.setAmplitude(waveHighFreq_Ampl);

float waveLowFreq_Ampl = map(modFactor_MASTER_HANDLE,0, 1000, 0.1, 0.12);
waveLowFreq.setAmplitude(waveLowFreq_Ampl);

float subWaveAmpl = map(modFactor_MASTER_HANDLE, 0, 1000, 0.15, 0.3);
subWave.setAmplitude(subWaveAmpl);
```

Code snippet 3.17: Amplitude shift of waves in first Processing implementation

These are all average amplitudes: these numbers should be seen as amplitudes still without the effect of the LFO modulation.
The four main harmonic waves all change amplitude linearly from 0.01 to 0.1, and all of the other oscillators also increase their level in order to be audible at full thrust.
Note how `'subWave'` increases a lot: this is to overcome the low frequency noise (discussed in the next Section 3.4.4) that becomes more present with acceleration.

Now that the behavior of the oscillators' main parameters (frequency and amplitude) are implemented, also the evolution of the LFOs should be defined.
It is in fact important to modulate the sound differently as the handle input parameter changes: at the beginning it should give an idea of an engine that is resting, with very low frequency cycles; as the driver turns the accelerator handle, these cycles should increase, and so should the amplitude shift range (the LFO amplitude).

Here is the all of the LFO processing part of the implementation:

```
float lfoAmpl_wavesHandle = map(modFactor_MASTER_HANDLE, 0, 1000, 0.005,
                                    0.05);
float lfoFreq_wavesHandle = map(modFactor_MASTER_HANDLE, 0, 1000, 4, 6);
wave1_handle.setLFO_Ampl_Freq(lfoAmpl_wavesHandle, lfoFreq_wavesHandle);
wave2_handle.setLFO_Ampl_Freq(lfoAmpl_wavesHandle, lfoFreq_wavesHandle);
wave3_handle.setLFO_Ampl_Freq(lfoAmpl_wavesHandle, lfoFreq_wavesHandle);
wave4_handle.setLFO_Ampl_Freq(lfoAmpl_wavesHandle, lfoFreq_wavesHandle);

float lfoFreq_waveLowFreq = map(modFactor_MASTER_HANDLE, 0, 1000, 2.5, 10);
float lfoAmpl_waveLowFreq = map(modFactor_MASTER_HANDLE,0,1000, 0.05, 0.06);
waveLowFreq.setLFO_Ampl_Freq(lfoAmpl_waveLowFreq, lfoFreq_waveLowFreq);

float lfoFreq_subWave = map(modFactor_MASTER_HANDLE, 0, 1000, 2.5, 5);
float lfoAmpl_subWave = map(modFactor_MASTER_HANDLE, 0, 1000, 0.05, 0.15);
subWave.setLFO_Ampl_Freq(lfoAmpl_subWave, lfoFreq_subWave);
```

Code snippet 3.18: LFO modulation of first Processing implementation

First the main waves' LFOs are modified. Their frequency goes from 4 Hz to 6 Hz: this doesn't seem like a big range, but human ear is very sensitive to variations under audible frequencies (<20Hz).
The LFO amplitudes instead go from 0.005 to 0.05, which is also half of the carrier amplitude in every instant: this rule was kept also for the other oscillators.
For the `'waveLowFreq'` and `'subWave'` LFO frequency, the values are experimental and were chosen by tweaking them to get a good sound result.

3.4.4 Noise component:

This section discusses the noise component of the sound, that is the stochastic part of the sound.
The basic concept behind this is to start from white noise and filter it; these filters can be either static or dynamic, meaning their control parameters (such as cutoff frequency or bandwidth) can change and be modulated.
The noise sources used in this first implementation are two: one at low frequencies and one at mid-high frequencies. The latter one also has a cyclic effect applied to it through a band-bass filter with undulating central frequency.

Here below the declaration, initialization and patching of all the components used at this stage:

```
Noise noiseHighFreq, noiseLowFreq;
Constant HighNoiseLVL;
BandPass BP_Noise_DYN;
Oscil LFO_BPCutoff;
LowPassSP LP_Noise_DYN;

void setup(){
…
  noiseHighFreq = new Noise(0.05f, Noise.Tint.PINK);
  noiseLowFreq = new Noise(0.5f, Noise.Tint.BROWN);

  HighNoiseLVL = new Constant(0.05);
  HighNoiseLVL.patch(noiseHighFreq.amplitude);
  BP_Noise_DYN = new BandPass(5000, 2000, 44100);
  LP_Noise_DYN = new LowPassSP(200, 44100);

  LFO_BPCutoff = new Oscil(5, 2000.0, Waves.SINE);
  LFO_BPCutoff.offset.setLastValue(5000);
  LFO_BPCutoff.patch(BP_Noise_DYN.cutoff);

  // Patching
  noiseHighFreq.patch(BP_Noise_DYN).patch(mixer);
  noiseLowFreq.patch(LP_Noise_DYN).patch(mixer);
  mixer.patch(new LowPassSP(11000, 44100)).patch(out);
}
```

Code snippet 3.19: Declaration, initialization and patching of the noise components and filters in first Processing implementation

The first two objects that are created are the two noise sources, of Minim class `'Noise'`. The `'noiseHighFrequency'` is initialized at 0.05 amplitude, and it can be noted that its amplitude is controlled by a `'Constant'` object through a patch: `'HighNoiseLVL'` is a DC signal that changes its value and thus sets the noise amplitude.
The `'noiseLowFrequency'` has a very high level compared to the rest of the components, because of the minor sensitivity of human ear to low frequency levels (Figure 3.4). This noise is filtered by the low-pass filter `'LP_Noise_DYN'`, with initial cutoff frequency at 200Hz.
The next objects are the band-pass filter `'BP_Noise_DYN'` and the LFO `'LFO_BP_cutoff'`; the latter controls the filter's central frequency through a patch. The filter is initialized with central frequency f_c=5000Hz and bandwidth BW=2000Hz, meaning it lets through white noise only in the [4000Hz, 6000Hz] band. The LFO is an `'Oscil'` initialized at 5 Hz, with offset=5000 and amplitude of 2000; these two last values are huge

because they represent the displacement that this filter should undergo in frequency domain: it undulates between 4000 and 6000 Hz.
The last part of the code is dedicated to the patches that compose the processing chain from noise sources to AudioOutput and passing through the filters.
Now the evolution of these objects with respect to handle inclination is presented. This code is inside the `'draw()'` function:

```
…
void draw(){
  float modFactor_MASTER_HANDLE = map(handleBar.getPosPrecise(),
                                          0, 784, 0, 1000);

  float lfoFreq_BP = map(modFactor_MASTER_HANDLE, 0, 1000, 4, 8);
  LFO_BPCutoff.setFrequency(lfoFreq_BP);

  float noiseLVL = map(modFactor_MASTER_HANDLE, 0, 1000, 0.05, 0.3);
  HighNoiseLVL.setConstant(noiseLVL);

  float freqLP = map(modFactor_MASTER_HANDLE, 0, 1000, 200, 1000);
  LP_Noise_DYN.setFreq(freqLP);
}
```

Code snippet 3.20: Modulation of noise components and filters in first Processing implementation

First, the LFO controlling band-pass filter `'BP_Noise_DYN'` is modified: its frequency goes from 4Hz to 8Hz. This filter acts on the `'noiseHighFreq'`, and with this LFO applied to it, the noise gets a cyclic effect similar to that applied to the waves in previous Section 3.4.3.
Also the amplitude of high frequency noise is changed through the `'Constant'` object `'HighNoiseLVL'`: when this value is changed, then the amplitude of noise is set accordingly (thanks to previous patch).
The last step is the "opening" of low-pass filter applied to the other noise source `'noiseLowFreq'`, from 200Hz to 1000Hz. This lets through some higher frequency components when turning the accelerator handle.

3.5 Second algorithm implementation based on Andy Farnell's book

This subchapter discusses the second implementation in the Processing 3 environment with the Minim library. It is based on Andy Farnell's solution discussed in Subchapter 2.2, and it's another building block towards the final sound implementation.

Before going into the details of the implementation, a brief general overview of the different components of the algorithm is here presented.

It's important to first note that in this implementation only two input parameters are considered: the motorcycle velocity and it's derivative, the acceleration.
The next Subchapter 3.6 discusses a hybrid solution between the ones presented in this subchapter and the previous one (3.4), where all of the four input parameters are used.

This implementation follows a slightly different reasoning from the one used in subchapter 3.4. It divides in fact the sound in two main parts:

- The turbine sound, composed of inharmonic partial components
- The engine noise, composed of a processed and overdriven noise

The turbine sound is made of 5 main sinusoidal components, coming from an experimental analysis done by Andy Farnell on jet propulsion engines.
The engine noise instead contributes to the stochastic component of the sound: it gives the idea of forced and burning air coming out of the propulsion engine. This is done through the application of a chain of filters and an overload effect.

3.5.1 Turbine

This part of the implementation is relatively simple: it consist of 5 main oscillators that shift in frequency of a factor x10 with respect to the velocity input parameter and its derivative. In this solution, differently from the previous one in Subchapter 3.4, no amplitude modulation LFO is applied to these oscillators. This is because it should give the idea of a continuous rising speed.
In Andy Farnell's original solution, after the oscillators were summed together, a slight clipping was applied to introduce some harmonics. This clipping is not implemented at this stage, because it wasn't considered as a critical feature of the sound; it will be implemented in the final solution on chip discussed in Chapter 4 to perfect the sound.
The five oscillators end respectively at: 3097 Hz, 4495 Hz, 5588 Hz, 7471 Hz and 11000 Hz.

Below is their declaration, initialization and patching:

```
MyWave wave1_vel, wave2_vel, wave3_vel, wave4_vel, wave5_vel;

void setup(){
  wave1_vel = new MyWave(309.7, 0.0, Waves.SINE, 0.0, 0.0);
  wave2_vel = new MyWave(449.5, 0.0, Waves.SINE, 0.0, 0.0);
  wave3_vel = new MyWave(558.8, 0.0, Waves.SINE, 0.0, 0.0);
  wave4_vel = new MyWave(747.1, 0.0, Waves.SINE, 0.0, 0.0);
  wave5_vel = new MyWave(1100.0, 0.0, Waves.SINE, 0.0, 0.0);

  // Patching
  wave1_vel.wave.patch(mixer);
  wave2_vel.wave.patch(mixer);
  wave3_vel.wave.patch(mixer);
  wave4_vel.wave.patch(mixer);
  wave5_vel.wave.patch(mixer);

  mixer.patch(new LowPassSP(11000, 44100)).patch(out);
}
```

Code snippet 3.21: Waves declaration, initialization and patching in second Processing implementation

Note how they are all sines and their LFO have 0 amplitude and frequency.

Now that the `'MyWave'` objects are created, their evolution with respect to velocity input parameter (implemented in the `'draw()'` function) is described. They undergo a frequency and an amplitude shift:

```
void draw(){

  // Frequency shift
  float modFactor_MASTER_VELOCITY = map(speedBar.getPosPrecise(), 0, 784,
                                        0, 1000);

  float wavesVel_freqShiftFactor_VEL = map(modFactor_MASTER_VELOCITY, 0,
                                           1000, 1, 10);
  wave1_vel.setFrequency(309.7*wavesVel_freqShiftFactor_VEL);
  wave2_vel.setFrequency(449.5*wavesVel_freqShiftFactor_VEL);
  wave3_vel.setFrequency(558.8*wavesVel_freqShiftFactor_VEL);
  wave4_vel.setFrequency(747.1*wavesVel_freqShiftFactor_VEL);
  wave5_vel.setFrequency(1100.0*wavesVel_freqShiftFactor_VEL);
```

```
  // Amplitude shift

  // Here a parabolic behavior is modeled
  float wavesVel_Ampl = map(modFactor_MASTER_VELOCITY, 0, 1000, 0, 0.1);
  wave1_vel.setAmplitude(wavesVel_Ampl*wavesVel_Ampl*10/3);
  wave2_vel.setAmplitude(wavesVel_Ampl*wavesVel_Ampl*10/3);
  wave3_vel.setAmplitude(wavesVel_Ampl*wavesVel_Ampl*10/1.5);
  wave4_vel.setAmplitude(wavesVel_Ampl*wavesVel_Ampl*10/2);
  wave5_vel.setAmplitude(wavesVel_Ampl*wavesVel_Ampl*10/2);
}
```

Code snippet 3.22: Frequency and amplitude shift of waves in second Processing implementation

This time the master parameter `'modFactor_MASTER_VELOCITY'` is taken from the `'speedBar'` scrollbar: it's a value going from 0 to 1000 and it corresponds to time instant velocity. The derivative (that is the acceleration, or how fast the speed varies) is regulated internally by the `'Hscrollbar'` class, as described in Section 3.4.1 when introducing the scrollbar variable `'loose'`: basically there is low-pass filter applied to this scrollbar to make transitions smoother.

All the frequencies of the waves are multiplied by a factor that is 1 at the beginning (unchanged) and 10 at full thrust.

The amplitude variation instead has a parabolic behavior: leaving apart for a moment the final divisions that are applied, the unknown 'x' is `'wavesVel_Ampl'` and it's given by the scrollbar input value scaled to interval [0, 0.1]. This value is squared when applied to the wave amplitude, and when it's at maximum (0.1), the 'y' (amplitude) should also be 0.1. The function and the formula that satisfy this relation are showed in Figure 3.5.

These amplitudes are then divided by some numbers to give different weights to each wave, as it was also done in PureData.

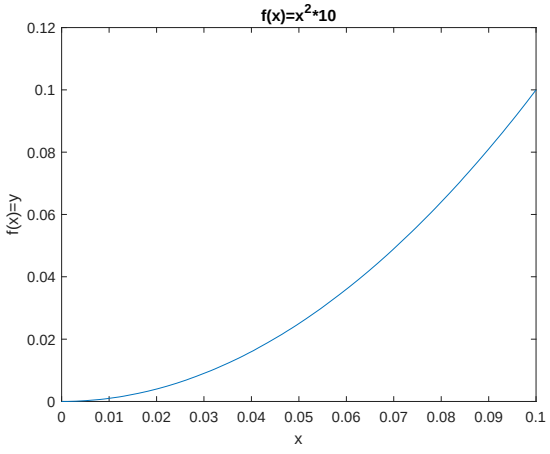


Figure 3.5: Function that describes the amplitude (y) variation of waves with respect to the velocity scrollbar values (x), scaled to an interval [0, 0.1]

3.5.2 Engine noise

In this part of the implementation, the stochastic component of the sound is introduced: at 0 acceleration it should only be a low level disturbance concentrated at low frequencies; at full speed instead it should give the idea of air forced out of the burning engine.
In order to synthesize this sound, simple white noise is sent in a chain of processing stages: dynamic filters and an overdrive effect.

Since the "overdrive" component doesn't exist in the Minim library, and given that Minim is an open source library, a new class called `'Overdrive'` was implemented as a workaround.
This class extends `'UGen'`, and it has only one audio UGenInput so that it's possible to patch an audio stream to it. The implementation was done by taking as a reference the simple `'Gain'` class, which only applies a multiplication factor to each sample. Here, a gain should be applied and then the sample should be clipped if it assumes a value greater than the threshold (as shown in Figure 3.10).
Here below are the fields and the constructor:

```
public class Overdrive extends UGen
{
  public UGenInput audio;
  private float multFactor;
  private float threshold;

  public Overdrive( float multFactor, float clipping )
  {
     this.multFactor = multFactor;
     threshold = clipping;

     audio = new UGenInput(InputType.AUDIO);
  }
  …
```

Code snippet 3.23: Fields and constructor of Overdrive class

The constructor takes as input parameters the multiplication factor and the threshold value, in the interval [0.0, 1.0].
Every time a UGen needs to generate or process a sample, it calls `'uGenerate()'`, an abstract method that needs to be implemented in every class that extends UGen.

Here below is the implementation of 'uGenerate()' inside 'Overdrive', the core of the overdrive algorithm:

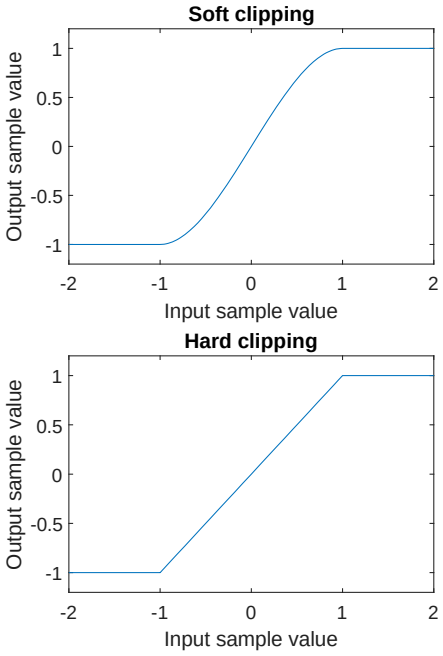
```
@Override
protected void uGenerate(float[] channels)
{
  for(int i = 0; i < channels.length; ++i)
  {
    float overdriveSample = audio.getLastValues()[i] * multFactor;

    if(overdriveSample <= threshold && overdriveSample >= -threshold){
      channels[i] = overdriveSample;
    }
    else if(overdriveSample > threshold){
      channels[i] = threshold;
    }
    else if(overdriveSample < -threshold){
      channels[i] = -threshold;
    }
  }
}
```

Code snippet 3.24: Overdrive algorithm inside the uGenerate method of Overdrive class

The for loop is done because every channel should be processed independently: if for example there is a stereo audio and left channel is lower than the right one, than – with equal threshold – the overdrive effect will clip more on the right channel, the louder one.
'channels' is the array in which generated samples should be stored. Its length is equal to the number of channels: in each position there is the current sample value on that channel. If dealing with a monophonic audio (as in this case), than channels has length equal to 1.
Instead, audio samples coming in (before the processing) can be read from the 'audio' UGenInput through 'getLastValues()'; this method returns an array of length equal to the number of channels, each containing the last sample value.
First of all, the sample is read and multiplied with the 'multFactor', in order to obtain the 'overdriveSample'.
Next, a series of 'if' statements take care of the sample clipping: if the 'overdriveSample' is in the interval [-threshold, +threshold] than it remains unchanged and it's directly assigned to 'channel[i]'; instead, if it falls outside of the interval, then it is forced back to the threshold value, by preserving the sign.
Note that here a harsh clipping is applied, thus discontinuities of type C1 are created. Other smoother clipping algorithm could be used: instead of having a harsh transition from normal to clipped, samples undergo a smooth transition as they approach the

threshold value. This means that if the sample is inside the accepted interval, instead of leaving it unchanged, a function is applied to it, generally a polynomial of 3rd degree.
This is known as "soft clipping", and it introduces less harmonics in the sound with respect to a hard clipping algorithm. The difference between the two can be seen in Figure 3.6.



$$f(x)=\begin{cases} -1, & x<-1 \\ \frac{3}{2}\left(x-\frac{x^3}{3}\right), & -1<x<1 \\ 1, & x>1 \end{cases}$$

$$f(x)=\begin{cases} -1, & x<-1 \\ x, & -1<x<1 \\ 1, & x>1 \end{cases}$$

Figure 3.6: Soft and hard clipping

Now that the `'Overdrive'` class is defined, the discussion on the synthesis of engine noise can be resumed.
Here below the declaration, initialization and patching of all the components used in the algorithm:

```
Overdrive overdrive;

Noise noiseFlame_PRE_PROCESSED;
LowPassSP LP_Flame_DYN_1, LP_Flame_DYN_2;
HighPassSP HP_Flame_STATIC;

void setup(){
  noiseFlame_PRE_PROCESSED = new Noise(0.1f, Noise.Tint.WHITE);
  overdrive = new Overdrive(120, 0.3);
```

```
  LP_Flame_DYN_1 = new LowPassSP(0, 44100);
  LP_Flame_DYN_2 = new LowPassSP(50, 44100);
  HP_Flame_STATIC = new HighPassSP(120, 44100);

  // Patching
  mixer = new Summer();
  noiseFlame_PRE_PROCESSED.patch(LP_Flame_DYN_1).patch(HP_Flame_STATIC)
                          .patch(overdrive).patch(LP_Flame_DYN_2).patch(mixer);
  mixer.patch(new LowPassSP(11000, 44100)).patch(out);
}
```

Code snippet 3.25: Declaration, initialization and patching of components in second Processing implementation

It can immediately be seen that 3 filters are used:

- `'LP_Flame_DYN_1'` is the first low-pass filter applied on noise, before overdrive. It is used to create some headroom for the high frequency harmonics that will be generated by clipping. The filter is dynamic, so its cutoff frequency will move from 0 to a maximum around 150 Hz, depending on the velocity scrollbar position.
- `'HP_Flame_STATIC'` is a static high-pass filter, made to avoid very low frequencies. It is set to 120 Hz.
- `'LP_Flame_DYN'` is a dynamic low-pass filter applied on the noise after the overdrive effect is applied. It is used to filter out the audible high frequency harmonics that are introduced by clipping, depending on the velocity scrollbar position: when velocity is 0 then none of these harmonics are present, when velocity is at maximum, cutoff frequency shifts and so most of them are audible.

The `'overdrive'` object is initialized with a multiplication factor of 120 and a threshold of 0.3. This means that the noise that goes into this block will have a really high gain applied to it followed by a harsh clipping. The resulting signal is a non-periodic square wave, as shown in Figure 2.11.
In this case it's important to filter the signal before sending it to the overdrive stage, because just clipping unfiltered white noise would introduce harmonic components which are indistinguishable from the constant and random components of a flat spectrum: it would result again in white noise. Instead, by filtering it, the high frequencies that are introduced stand out and give that sense of forced and ripped air.

The evolution of the two dynamic low-pass filters with respect to velocity input parameter is pretty straightforward, and its implementation is presented below:

```
float cutoff_FLAME_1 = map(modFactor_MASTER_VELOCITY, 0, 1000, 0, 150);
LP_Flame_DYN_1.setFreq(cutoff_FLAME_1);

float cutoff_FLAME_2 = map(modFactor_MASTER_VELOCITY, 0, 1000, 50, 1000);
LP_Flame_DYN_2.setFreq(cutoff_FLAME_2);
```

Code snippet 3.26: Evolution of the two low-pass filters used for engine noise in second Processing implementation

The first filter has a cutoff frequency that can vary from 0 to 150 Hz, the second one from 50 to 1000 Hz. This last value can be tweaked depending on how extreme the ripping effect is wanted.

3.6 Third algorithm implementation: Hybrid solution

The two previous implementations both have some interesting aspects and strong points: the first one gives the impression of a running engine, with a cyclic sound increasing in frequency as the accelerator is turned. The second sound renders really well the roar of a propulsion engine when air burns and is forced out of a small hole.
This is why the two solutions were eventually merged together, to obtain a full and complete sound working on all four input parameters: handle inclination, velocity and their derivatives.
This is not equal to the final algorithm implemented on chip – whose sound is further perfected – so the two sounds differ, but it was a good arrival point and it marked the achievement of a good base model: after this third implementation, the second phase of the thesis work ended and the efforts moved to on-chip implementation.

The algorithm basically keeps all of the components that were used in the previous two implementations and it merges them, but it also manages some interactions and collisions between the two, especially considering amplitudes of the sine components, where constructive interference between them should be reduced.

Also a major change was made: the overdrive noise of the engine is now associated to the handle inclination, leaving velocity with only the 5 sine waves that have no LFO modulating their amplitudes. This choice was made because it makes sense to have the engine roaring when at full thrust (when accelerator handle is at maximum inclination), but that doesn't necessarily means that the motorcycle is moving: it simulates the process of warming up the engine, and all that is wanted when actually picking up speed is just a continuous sound shifting up in pitch.

As already stated, the sound keeps all the components of the previous implementation and also their variable names are unchanged; therefor they are just listed here, but their declaration, initialization and patching is not showed. For more details, refer to Subchapters 3.4 and 3.5.
From first implementation:

- Four main harmonic oscillators starting respectively at 250Hz, 500Hz, 750Hz, 1000Hz and doubling in frequency as accelerator is pressed. The LFOs modulating their amplitude behave just as described in Section 3.4.3.
- One low frequency oscillator shifting from 116Hz to 266Hz, one sub-frequency wave shifting from 80Hz to 150Hz and one high frequency wave from 3000Hz to 9000Hz. Also here the LFOs modulating their amplitude behave as described in Section 3.4.3.
- One low frequency noise: just a normal white noise source that is filtered by a dynamic low-pass filter, with cutoff frequency shifting from 200Hz to 1000Hz. One medium-high frequency noise, made of another white noise source filtered by a band-bass filter with central frequency oscillating between 4000Hz and 6000Hz and bandwidth of 2000Hz. This is done through an LFO (a 'Oscil' object) having offset=5000, amplitude=2000Hz and patched to the filter's central frequency.

From second implementation:

- One noise source that goes through a chain of dynamic filters and an overdrive stage (Section 3.5.2). This sound is now a function of handle inclination and its derivative, and not of velocity anymore.
- Five main inharmonic sine waves starting respectively at 309.7Hz, 449.5Hz, 558.8Hz, 747.1Hz, 1100.0Hz and shifting up of a factor x10, coming from experimental analysis done by Sound Designer Andy Farnell (Section 3.5.1). There are no LFOs modulating their amplitudes. These are the only components left that are a function of velocity and its derivative, acceleration.

The first important thing to note is that now input parameters are taken from both of the scrollbars: `'speedBar'` is used to read velocity and acceleration of the motorcycle (how fast this scrollbar moves), `'handleBar'` is used for handle inclination and its derivative (how fast this scrollbar moves).
The two derivatives are managed internally in the `'Scrollbar'` class, as explained in Section 3.4.1 when discussing the Scrollbars. When there is a fast transition in the scrollbar, then the two variables `'modFactor_MASTER_VELOCITY'` and `'modFactor_MASTER_HANDLE'` don't contain the instant value of the inputs, but they are updated more slowly, as if a low-pass filter was applied to them.

```
  float modFactor_MASTER_VELOCITY = map(speedBar.getPosPrecise(), 0, 784,
                                                    0, 1000);
  float modFactor_MASTER_HANDLE = map(handleBar.getPosPrecise(), 0, 784,
                                                    0, 1000);
```

Code snippet 3.27: two main input parameters 'velocity' and 'handle inclination'

Now the evolution in time of the components used in the algorithm is discussed (the code that resides in the `'draw()'` function). *From here on, only the code that differs from the two previous implementations, or the code that is added, is showed.*
Also, when talking about velocity or handle "scrollbar value", what is meant is the value of input parameters with the low-pass filter in time already applied to them (with derivative already managed).

For what concerns the frequency shift of the waves, nothing changes from previous implementations: the shifts all depend on the same input parameter and have the same behavior.
The first main difference arises in the amplitude evolution of the 4 waves from the first implementation, that is the waves whose frequency shifts are a function of handle inclination and its derivative. Before in fact their amplitude would just increase linearly from 0.01 to 0.1, following handle inclination value. This is not acceptable anymore, because now also the waves from the motorcycle velocity are present, and in some points they could superimpose and generate a constructive interference: this could result in a much higher volume if they perfectly overlap, or in the so called "beating effect" (r) if they have slightly different frequencies.
The solution used to mitigate this phenomenon is to make the amplitude of these waves function of both the handle and the velocity scrollbars: when only handle inclination is applied, the behavior is the same as in the first implementation, but as the velocity contribution also starts increasing, the final amplitude is given by the difference between the two contributions.
Here is the implementation:

```
float wavesHandle_Ampl_HANDLE = map(modFactor_MASTER_HANDLE, 0, 1000, 0.01,
0.10);
  float wavesHandle_Ampl_VEL = map(modFactor_MASTER_VELOCITY, 0, 1000, 0.0,
0.08);
  float wavesHandle_Ampl_TOT = 0;
```

```
if(modFactor_MASTER_HANDLE - modFactor_MASTER_VELOCITY >= 0 ){
    wavesHandle_Ampl_TOT = wavesHandle_Ampl_HANDLE - wavesHandle_Ampl_VEL;
  } else if(modFactor_MASTER_HANDLE - modFactor_MASTER_VELOCITY < 0 ){
     wavesHandle_Ampl_TOT = abs(wavesHandle_Ampl_HANDLE -
wavesHandle_Ampl_VEL)/4;
  }
  }
  wave1_handle.setAmplitude(wavesHandle_Ampl_TOT);
  wave2_handle.setAmplitude(wavesHandle_Ampl_TOT);
  wave3_handle.setAmplitude(wavesHandle_Ampl_TOT);
  wave4_handle.setAmplitude(wavesHandle_Ampl_TOT);
```

Code snippet 3.28: Amplitude shift of waves whose frequency is a function of handle inclination. In this hybrid solution, the amplitude is function of both handle inclination and motorcycle velocity.

First the two contributions `'wavesHandle_Ampl_HANDLE'` and `'wavesHandle_Ampl_VEL'` are calculated, the first one being the same as in the first implementation and the second one having values tuned for the purpose.
A third parameter `'wavesHandle_Ampl_TOT'` is calculated as a weighted difference between the two. In particular, there are two cases:

- The handle scrollbar value is higher than the velocity one, so final amplitude is just handle contribution – velocity contribution, thus decreasing original amplitude as velocity increases.
- The handle scrollbar value is lower than the velocity one, so final amplitude is $\frac{|handle\,contrib. - velocity\,contrib|}{4}$

  In this case the final amplitude is divided by 4 to further decrease the amplitude: it is the scenario in which motorcycle is going very fast but accelerator is pressed very little, so the velocity component of sound should prevail over the engine's cycles one.

Concerning the low frequency wave, the sub-frequency wave, the high frequency wave and the 5 waves coming from second implementation (function of velocity), the algorithm for their amplitude evolution is unchanged.
Also the algorithm for the evolution of the amplitude modulation LFOs is unchanged for all waves having this LFO contribution (4 waves from first implementation, sub-frequency wave, low frequency wave and high frequency wave).

Then there is the `'BP_Noise_DYN'`: this is a band-bass filter applied to a normal white noise source. The bandwidth is always 2000Hz, but the central frequency moves between

4000Hz and 6000Hz, depending on the value of 'LFO_BPCutoff', to give a sense of a running engine. This is an LFO with fixed amplitude of 2000, offset of 5000Hz, and frequency that is now a function of both handle and velocity, differently from the first implementation where it was only a function of handle inclination.

```
float lfoFreq_BP_HANDLE = map(modFactor_MASTER_HANDLE, 0, 1000, 4, 8);
float lfoFreq_BP_SPEED = map(modFactor_MASTER_VELOCITY, 0, 1000, 0, 8);
float lfoFreq_BP_TOT = abs(lfoFreq_BP_HANDLE - lfoFreq_BP_SPEED);
LFO_BPCutoff.setFrequency(lfoFreq_BP_TOT);
```

Code snippet 3.29: Frequency modulation of the LFO acting on the central freqeuncy of band-pass filter 'BP_Noise_DYN'. In this hybrid solution it is now function of handle scrollbar and velocity scrollbar.

In this implementation, as the speed component approaches the handle inclination one, the LFO's frequency decreases, reaching 0 when the two contributions are equal.
Note that because of the 'abs', the final LFO frequency increases as the handle component increases respect to the velocity one, but also when the opposite happens, that is when speed is high but accelerator is not pressed. This effect is not too realistic because the LFO here is used to give the idea of running engine; in order to mitigate this effect, the volume of the noise on which this dynamic filter is applied is increased only as handle inclination increases: this way, when motorcycle velocity is high but accelerator is barely pressed, the noise is too low to be heard with respect to the rest of the sound. Instead, when the accelerator contribution starts increasing, the volume of this noise also increases and peaks out. This can be seen in Code snippet 3.30.

The last difference between this hybrid implementation and the previous two is the volume of low frequency noise: before this value was set to 0.5 and held static, now it starts at 0.5 but decreases as the handle inclination increases. This is because now also the overdrive noise is linked to handle inclination: it starts low and tends towards an engine "roar" as the handle inclination increases.
At full thrust, keeping the two types of noises unchanged would result in a dirty sound at low frequencies.
Here below the code lines regulating volumes of the two last discussed noises (medium-high frequency noise and low frequency noise) is shown:

```
float noiseLVL = map(modFactor_MASTER_HANDLE, 0, 1000, 0.05, 0.3);
HighNoiseLVL.setConstant(noiseLVL);
float noiselowFreq_LVL = map(modFactor_MASTER_HANDLE, 0, 1000, 0, 0.5);
noiseLowFreq.amplitude.setLastValue(0.5 - noiselowFreq_LVL);
```

Code snippet 3.30: Volume adjustment of medium-high frequency noise and low frequency noise in the hybrid Processing implementation

4

Phase 3: Implementation of the solution on embedded chip

In the previous chapter a base model for the motorcycle engine sound was built. The result that was obtained was satisfactory enough to move on and start concentrating on the last and most important step: the implementation of a synthesis algorithm on chip. On the one hand the sound that was synthesized in previous chapter has a lot of room for improvement and can be optimized, on the other though it was clear how working on the details already at that stage was useless and counterproductive. When working on a chip in fact, the tools that are available (such as libraries), the strategies (such as paying attention to system resources, which are much more limited than on PC) and the workflow can change a lot, especially when the chip choice still needs to be evaluated: adding too much detail in the Processing environment could lead to over-engineering a solution prematurely. Furthermore, the obtained synthesis model provides a solid base for the algorithm implementation: any algorithm can easily be adapted, modified or re-built on top of this model.

Here are the basic elements from the model built in Chapter 3 that are core to the algorithm, and therefor also used in the on-chip implementation:

- Some periodic components (sine waves) shifting up in frequency as the handle inclination (acceleration) increases, and other periodic components with frequencies following motorcycle speed.
- The implementation of a MyWave class, in which each oscillator has an amplitude modulation LFO associated to it. Therefor the need to control these LFOs at each time instant by changing their amplitude and frequency.
- Some stochastic component, given by noise properly filtered.
- A noise source on which an overdrive effect is applied, to render the “roar” of an engine at full thrust.

This chapter covers the third and last phase of this thesis work: the implementation of a synthesis algorithm on chip, a microcontroller de facto.
In the first subchapter, an evaluation of the possible embedded boards and audio synthesis/processing libraries is necessary. These two aspects are strongly related: the libraries that are available are usually different on boards from different manufacturers, and on some others they aren't even available.
Next, once the choice of the embedded board is done, all the preliminary preparation steps are treated: the connection of some of the pins on the board to external components (potentiometers), and the evaluation of different development environments to program the microprocessor. This is discussed in Subchapter 4.2.
At this point a quick overview of the chosen audio library is done (Subchapter 4.3), the classes that were implemented from scratch are presented (Subchapter 4.4), and finally the implementation is discussed (Subchapter 4.5).

4.1 Preliminary work: Evaluation of different embedded boards and relative Audio/DSP libraries

When evaluating a board for a specific task, many aspects must be analyzed: internal resources, power consumption, size, resistance to environmental changes.
It is important, in fact, to remember the final destination and purpose of this board: it will be mounted on a motorcycle, and therefore it will constantly be solicited to vibrations and temperature changes.
This is why it should be solid and possibly with everything embedded onto it: a micro SD reader (present in many modern microcontrollers) for example could be useful, but the functioning of the program shouldn't rely on it, because time, vibrations and humidity could corrupt the SD card or the electronic contacts between it and the reader. It is therefore a secondary feature, and the program should be loaded onto and started from an embedded flash memory, not from an SD.

Another very important aspect is boot-time: when the motorcycle is turned on, the sound should start immediately, and not wait for the chip to boot.
Many embedded boards have an operating system on it, usually a Linux distribution. While adding an additional layer of abstraction over the bare-metal programming can be useful and potentially very powerful, it does increase significantly the boot-time in the order of tens of seconds, and it makes access to low-level communication protocols less direct. Even if a bootstrap could be done in order to boot these boards directly with the implemented program instead of the OS, it was decided to opt on other boards that can be

programmed bare-metal directly. This is why for example the RaspberryPi boards or the ones from the NanoPi series (http://www.nanopi.org/index.html) were discarded.
Also size is a crucial point: the smaller the boards the better because they should fit somewhere inside the motorcycle, which doesn't have much space available. All the fancy peripherals such as USB or Ethernet ports are not necessary: the inputs here will be connected to normal GPIOs, either digital or analog. This can decrease a lot the required board size.

From a resources point of view, most of the modern embedded microcontrollers have limitations that are well above what is needed in this case.
Synthesizing a sound is in fact a relatively simple task for modern microprocessors: it just means continuously processing buffers of samples and sending them to an integrated DAC. Also, the sound that is to be synthesized here shouldn't be of extreme quality: working with 12 or 16 bits per sample and a sampling frequency of 44100 Hz is largely sufficient.

Finally, as already mentioned, one of the most important aspects in the evaluation of an embedded board is the existence of good libraries to support the programmer. The alternative would in fact be to code everything from scratch: from the oscillator objects, to filters and effects, to the connections between every component.
Every manufacturer has its own packets of libraries: some are more aimed at simple audio processing, while others are designed more for audio synthesis. In some cases the two can coincide.

All of this research finally led to reducing the number of possible solutions to two, and these are discussed in the next two sections: the STM32 series from STMicroElectronics and the Teensy boards. Note that also

4.1.1 STM32 series from STMicroElectronics

The first set of embedded boards that are analyzed are the STM32 F4 series [14], where 'F4' means they contain a 32 bit Arm Cortex-M4 microprocessor. This is probably one of the most popular low-cost microprocessors in the embedded world: it can reach a frequency of 180 MHz and it is low power.

One of the strong points of this F4 series of microcontrollers is the fact that they support DSP instructions. This means that, even if there is no physical DSP processor on the board, if some basic signal processing s needed, it can be directly done by a "dedicated

section" of the Cortex M4: there are some Assembly instructions crafted to optimize typical parallel digital processing operations. This allows saving money (a dedicated DSP is an expensive solution) and power consumption (everything is done within the board, without the need of an external dedicated DSP) [15].
In order to support these signal processing instructions there is a dedicated library: it is part of the CMSIS package (Cortex Microcontroller Software Interface Standard) and its name is CMSIS-DSP.
This library is included in "STM32Cube" [16], the development environment designed by ST to speed up the developers' work on STM32 boards, with an intuitive graphical interface and a set of useful tools. Also, there is a large number of support material, examples and an active community, all elements that add up to create a great work environment.

Another interesting series from STMicroElectronics is the NUCLEO series [17]. These have more of an educational purpose. They are generally pretty small, cheap and low-power: in this regard, there is also an "ultra low-power" family of boards, the NUCLEO-L series.
The advantage of these boards is that, since they have the STM32 microprocessors embedded onto it, they support previously discussed libraries such as CMSIS-DSP, but they also support the "mbed" [18] IDE and APIs from ARM.

The problem with the CMSIS-DSP library is that it only does audio processing. The whole synthesis part is missing, and with that also an intuitive connection/patching system. This library was in fact designed for general purpose signal processing, not audio specifically. It is a great analysis tool, but requires some additional work to use it in a more musical manner.
One possibility is to use "ST-Audio Weaver" [19], an audio-processing engine with a nice graphical programming tool called "Audio Weaver Designer". These two together constitute a nice development environment and make the process of programming digital audio synthesis and processing patches easy on embedded boards.
Through this set of tools, elements such as sine waves and filters can easily be programmed and connected together.

Another possibility to overcome the absence of a synthesis module in CMSIS-DSP is the OOPS library [20], developed by two researchers at Princeton University: Michael Mulshine and Jeff Snyder. The goal of OOPS is to have a unified audio synthesis library written in C for all kinds of 32-bit microprocessors, thus not being limited to a particular manufacturer. It contains many objects such as sine wave oscillators, n-pole filters, effects and some typical sound samples.

This, in conjunction with CMSIS-DSP could have been a valid option to fully work on the motorcycle sound. The only problem with OOPS is that there is not much documentation yet and a major high-level functionality is missing: the possibility to connect different components together.

All of these reasons put together led to at least try to look for alternative solutions, and see if there exists an embedded board specifically aimed at audio synthesis and processing.
The next section discusses the Teensy board, which in the end was the final solution for this thesis work.

4.1.2 Teensy board

The Teensy boards [21] are a family of 32-bit USB-based microcontrollers, very small in size but extremely powerful, developed by Electrical Engineer Paul Stoffregen.
They can be used to implement many types of projects, and they dispose of a wide array of libraries, each focused for a specific topic.

Almost all of the models mount a Cortex-M4 microprocessor, reaching clock frequencies of up to 180 MHz, which can be overclocked to 240 MHz. The most basic models have an ATMEGA AVR microprocessor running at 16MHz.

There are two major strong points of the Teensy boards:

- The development environment is Arduino, basically a revised C++. This allows for high-level and object oriented programming, which is a great plus for the scope of this thesis work. Also, all of the useful Arduino libraries are available for analog and digital read, communication through serial protocols and many more.
  Furthermore, all of the boards are programmed through a simple micro USB port.
- There is a great audio library built for Teensy boards, the "Teensy Audio Library" [22]. It contains many useful objects, such as oscillators, filters, effects and analysis blocks (FFT).
  On top of these objects there is a structure for connections between them. The great thing about this patching system is that it can be implemented directly in the code, or through a nice and easy to use HTML graphical interface, the "Audio System Design Tool", accessible online through the main website [23]. This works both as a sort of graphical programming tool (only for the connections of the blocks, which then still have to be programmed in the code) and a documentation, because by

selecting each component, its description and functionalities are described on the panel.

Given these interesting strong points of working in the Teensy environment, the choice of a development board was directed towards these. In particular, the Teensy 3.2 board has excellent specifics for the purpose of this thesis, and these are briefly described now.
First of all it has a Cortex-M4 microprocessor which arrives up to 72 MHz in normal state and 96MHz if overclocked. This is highly sufficient for audio applications, where a samplig frequency of 44.1 kHz is usually used.
It then has a 256 kByte embedded flash memory: this is where the compiled program is loaded every time and ready to run. This space is big enough for medium projects, and when actually programming on a Teensy board, the first portion of memory that is used is for libraries (about 10-15% of the memory).
It then has 34 digital I/O pins, 24 of which are usable on a breadboard. The input analog pins instead are 21, and most of them coincide with the digital ones: they can be configured as analog or digital from the code.
There is only 1 analog output pin, and it's the DAC (Digital to Analog Converter). Its resolution is of 12 bits (2^{12}=4096 values), bit this can be extended to 16 bits (2^{16}=65536 values) through an external adapter, the "Audio Adaptor for Teensy 3.0-3.6" [24].
Finally, there are many usable timers (clocks) and some pins dedicated to external communication protocols such as SPI, I2C and CAN.
The board is powered either directly by USB or through a standard 3.3V supply, 5V tolerant.
It is a very small chip (about the size of two 10 cents € coins) and cheap, around 20$, which double if also the audio shield is used.
All of this information can be found here: [25]
The pin schematics of the Teensy 3.2 can be found here [26] and here [27].

4.2 Preparation of the Teensy 3.2 board and evaluation of the development environment: Sloeber Eclipse

Now that the choice of using the Teensy 3.2 board is done, it's time to discuss the preparation of everything that is necessary to actively work on the chip.
The first step was in fact to install and solder through regular short pins the Teensy 3.2 on the Audio Adaptor. This is an external piece of hardware, and two main reasons led to make use of it: it extends the bit resolution of the DAC to 16 bits, and it incorporates a

3.5 mm female jack connector, which makes it easy to immediately hear what is programmed through a pair of headphones or some loudspeakers.
It also supports stereo or mono microphone input, and has a built in SD card reader, which is not used in this thesis work, but can be useful.
This audio shield communicates with the Teensy through different communication protocols.
I2C is used to synchronize and adjust some parameters on the Teensy board; this is done through the two signals: SCL (clock) and SDA (data).
I2S is instead used to transmit audio data. It is a protocol specifically designed for audio applications, and it has a transmitter signal (TX), a receiver one (RX) and one or more clocks. In this configuration, the audio shield works as a slave, meaning that all of the clock signals are inputs.
Finally, the SPI protocol is used for the SD reader.

The second main work that had to be done is the connection of two analog pins to external potentiometers, in order to simulate velocity and handle inclination input parameters. This was previously done in Processing (as discussed in Section 3.4.1) through two MIDI faders.
A very small circuit was designed, connecting the potentiometers pins to some GPIO on the Teensy.
The chosen potentiometers are sliders (not knobs), they have a linear response and a 10 kOhm resistance.
The circuit schematic is shown in Figure 4.1, with the two input analog pins named as in the Teensy schematics.

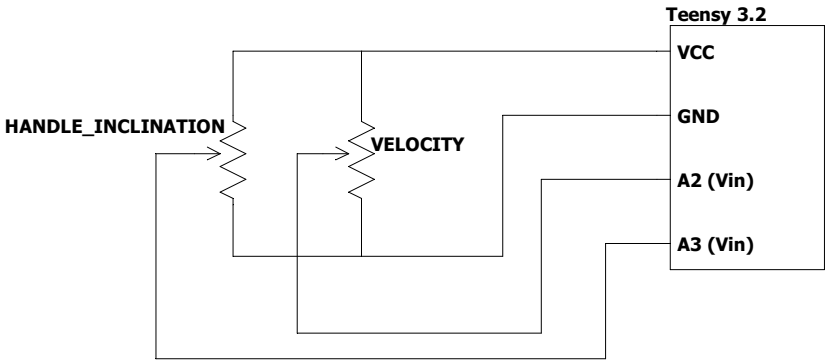


Figure 4.1: Circuit schematic of the two potentiometers connected to the Teensy 3.2 board. Vcc = 3.3V. A2 and A3 are the two analog voltage inputs.

A real picture of the final configuration of Teensy 3.2 with the audio adapter and the connections to the two potentiometers is shown in Figure 4.2.

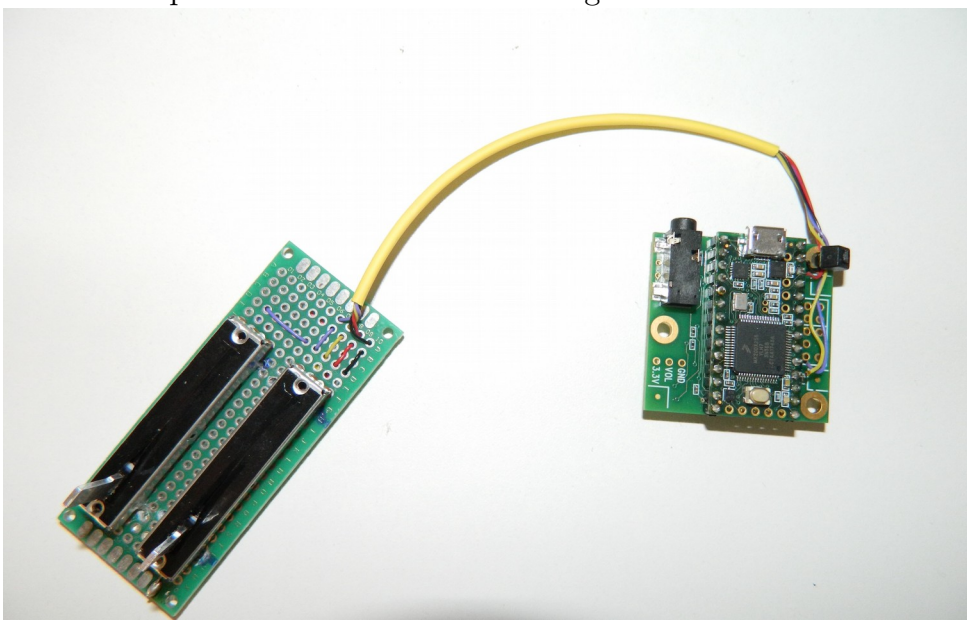


Figure 4.2: Final configuration of Teensy 3.2 board: it lies on the Audio Adaptor; four pins are connected to the two potentiometers (Vcc, GND, A2, A3).

Now that the board is ready for use, the next step is to evaluate the development environment on PC through which the Teensy 3.2 is programmed.
First of all, as already stated, Teensy boards are programmed through Arduino sketches. These are very similar to the sketches used in Processing (discussed in Subchapter 3.1): there is a `'setup()'` function where all the variables used in the program are initialized and the memory is managed, and then there is a main `'loop()'` function, where all the pin writes and reads are done and where the logic occurs: in this case the audio processing.
In order to communicate with the Teensy board, a small tool is used: the "Teensy Loader" [28]. This compiles the program written in Arduino, it generates an HEX file (the binary executable) and loads it into the board's flash memory, ready to boot.
For what concerns the actual development environment, the first one that comes to mind is the Arduino IDE: it's completely free, very easy to install and to use (with a minimalist GUI), but it has some major flaws.
There is no syntax highlighting nor auto-completion: this doesn't seem like a big problem but, unless it's a simple project, it can significantly slow down work, especially when using external libraries. It is indeed very important to be able to quickly explore the methods or the public fields of a class. It's also crucial to directly access the implementation of a library class to see how it works internally and eventually modify it, without going every

time to the GitHub page and exploring manually. Most modern IDEs in fact provide these functionalities.
For example, at one point in the project, the Overdrive class will be implemented from scratch, and for this reason it is crucial to read the implementation of similar processing objects (such as the Gain class), but also to explore all of the included '.h' header files and to see how these objects communicate with the hardware.
If the board was a simple Arduino board, many easy solutions could have been used: the Programino IDE, Arduino for Visual Studio, Arduino for Atmel Studio etc. (all of these solutions can be found at ref r).
The problem here is that in most of these environments, the Teensy boards are not yet supported by default; these boards are in fact just programmed in Arduino style, but they are not real Arduino boards. It would be necessary to link the IDEs to the "Teensy Loader" every time the compile&load button is pressed, and this is possible in some environments, but it's really not that immediate.
Instead, a full solution that already integrates all of these functionalities was found with "Sloeber: The Arduino Eclipse IDE" [29]. It's an open source project made by programmer Jan Baeyens, and it's basically an adapted version of Eclipse, which is a widely used IDE for embedded applications.
It works with all the boards that are compatible with Arduino, thus also with Teensy boards, and it automatically launches and links the "Teensy Loader". The only thing that needs to be done is to provide the path for the external libraries (such as the "Teensy Audio Library") and for the core libraries (the ones that are needed for Arduino projects to work).
Sloeber Eclipse is a fully functioning application and it supports syntax highlighting and auto-completion. A class or a field definition can be accessed by simply holding Ctrl and clicking on the desired object: in case for example it is a class it will open the corresponding '.h' header file, in case instead it's a method it will open the implementation of it.

4.3 "Teensy Audio Library" and "Teensyduino" library overview

Now that all the preliminary setup is done, the actual code will be discussed. Before though, an overview of the libraries that are used for audio synthesis/processing and resources management is discussed in this subchapter.

The "Teensy Audio Library" is the one that manages from a high-level both the communication with the Teensy hardware and the software implementation of some processes, such as wavetable synthesis and filtering.
The "Teensyduino" library instead, is basically a revised version of the Arduino core libraries, which manage basic I/O features, such as reading a value from a GPIO pin.

The base processing unit of the audio library is a buffer of samples, not the single samples as in Processing. This means that, when creating chains of effects, the whole buffer is passed from a stage to another.
First of all, the communication between the Teensy board and the audio shield and the memory management is discussed.
The "Audio Adaptor" can be accessed from the code through the class `'AudioControlSGTL5000'`. Once an instance of this class is created, it just needs to be activated through its method `'enable()'` and the output volume should be regulated by the method `'volume(float)'`, which accepts a value between 0.0 and 1.0.
In order to pass some audio data from the Teensy to this audio shield, the I2S protocol is used. However, the whole process is managed from a high-level perspective through the class `'AudioOutputI2S'`: just sending audio buffers to this object results in actually sending sound data to the audio shield, and so directly to the jack output.
Both of these classes are contained in the "Teensy Audio Library" and an example of their use can be seen in Code snippet 4.1.

Next, connections between different components should be managed. This means for example connecting a sine wave to a filter, but also the output of a filter to this `'AudioOutputI2S'` object: the last connection, in every program that needs to output sound on the teensy audio shield, is always to the 'i2s' class.
Connections are done through the object `'AudioConnection'`, contained in the "Teensyduino" library. It is basically an extension of the classic Arduino `'AudioStream'` class, readapted specifically for the Teensy board.
The way it is instantiated is by taking four arguments, listed below:

1. The first object, the one that sends data
2. The output number of the first object, starting from 0 (some components could in fact have more than one output)
3. The second object, the one that receives data
4. The input number of the second object, starting from 0 (some components could in fact have more than one input)

When the outputs from the first object and the inputs from the second one are both one, than only the two objects can be passed as arguments. An example of `'AudioConnection'` is seen in Code snippet 4.1.

There are then the actual synthesis or processing components, all belonging to the "Teensy Audio Library".
These include waves of different shapes, filters and effects. For example, an oscillator is defined as an instance of the class `'AudioSynthWaveform'`. When using one, first the object must be created, and then its basic parameters should be set:

- Frequency: through the `'frequency(float)'` method
- Amplitude: though the `'amplitude(float)'` method (it accepts values between 0.0 and 1.0)
- Waveform: through the `'begin(int)'` method, which accepts a global variable describing the different waveforms

Below, a fully working example of a 440 Hz sine wave outputted to the jack on the audio shield:

```
#include <Audio.h>

AudioSynthWaveform       waveform;
AudioOutputI2S           i2s;
AudioControlSGTL5000     sgtl5000_1;
int current_waveform=0;

AudioConnection          patchCord1(waveform, 0, i2s, 0);
AudioConnection          patchCord2(waveform, 0, i2s, 1);

void setup() {
  Serial.begin(9600);
  AudioMemory(10);

  // Audio Shield initialization
  sgtl5000_1.enable();
  sgtl5000_1.volume(0.4);

  // Configuration of the wave:
  // 440 Hz, maximum amplitude and waveform definition
  waveform.frequency(440);
  waveform.amplitude(1.0);
  current_waveform = WAVEFORM_SINE;
  waveform.begin(current_waveform);
}
```

Code snippet 4.1: Small program that outputs a 440Hz sine wave to the "Teensy Audio Adaptor"

Before the `'setup()'` function, all of the components that are necessary are declared: the synth wave, the object for I2S communication, the audio shield and the current waveform (which is an int).
After this two connections are done: two instances of the class 'AudioConnection' are declared and initialized. They are connecting the output of the waveform to the inputs of the `'i2s'` object; note how the inputs are two, one for the left channel and one for the right one.
This will automatically transfer sound to the audio shield and then to the jack connector.
Inside the `'setup()'` function, the Serial Monitor is initialized with a baud rate of 9600 (this means 9600 bit/sec): this is for debugging purposes.
Then, the audio memory is set through the `'AudioMemory(int)'` function. This value indicates the amount of RAM memory that should be allocated for audio connections. It is a value that is hard to predict, and depends a lot on the program; as stated by Paul Stoffregen himself [30], each block (the number that is passed to this function) is 260 bytes. At least one for every audio connection and one for each audio source is needed, but then other factors may come into play. In any case, the `'AudioMemoryUsageMax()'` returns the number of maximum used blocks when application is running, so that the value can be tweaked accordingly.
Also note that, as stated by Paul Stoffregen, the space occupied by this memory allocation is seen as "global variable" space. Some space should always be left for local variables used in the code and in the libraries that are used.
At this point, the audio shield is activated and its volume is initialized with a value between 0.0 and 1.0. This enables communication of audio data between the Teensy board and the audio shield.
The last thing that remains is to initialize the wave: frequency, amplitude and waveform, which is set through a global variable defined in the corresponding header file "synth_waveform.h".
Once the waveform is set through the `'begin()'` function, the wave starts playing and is audible.

This ends the overview of the "Teensy Audio Library". More classes and components will be necessary in the actual implementation, but they will be discussed when used.

4.4 Algorithm implementation on Teensy 3.2 board

This subchapter discusses the final version of the audio synthesis algorithm on the Teensy 3.2 board.
The "Teensy Audio Library" is used and, in order to implement this solution, many elements are taken from the model discussed in the previous Chapter 3.
Unlike what was done in the Processing environment, while implementing this solution no visual debug could be done (no wave or spectrum drawing), but only the Serial Monitor debug, which means outputting some values at different points in the code.

Before starting with the actual logic, a preliminary discussion needs to be done on three classes that were implemented from scratch: the `'MyWave'` class, the `'AudioEffectOverdrive'` class and the `'SignalSampleReader'` class. These are all used in the code and will be discussed in the next 3 Sections (4.4.1, 4.4.2 and 4.4.3).

4.4.1 MyWave class in Teensy implementation

As in the previous implementation in the Processing environment, also here most of the oscillators that are used undergo an amplitude modulation, so a class is needed to incorporate both the actual wave, the LFO and all the methods to control these: it's the `'MyWave'` class. Being that a different library is used, its implementation is slightly different from the one in Processing; furthermore, some additional details are added.

First, the header file "MyWave.h" is discussed, to see which elements are used.

```
class MyWave {
public:
      MyWave(float f, float a, int wf, float f_LFO, float a_LFO);
      virtual ~MyWave();

      AudioSynthWaveform wave;
      AudioEffectMultiply mult;
      AudioFilterBiquad lpAntiAlias;

      void setFrequency(float f);
      void setAmplitude(float a);
      void setLFO_Frequency(float f);
      void setLFO_Amplitude(float a);
      void setLFO_Ampl_Freq(float a, float f);
```

```
private:
      AudioSynthWaveform LFO;
      AudioConnection patchCord1;
      AudioConnection patchCord2;
      AudioConnection patchCord3;
};
```

Code snippet 4.2: 'MyWave.h' header, with all the fields that are used

The constructor takes as arguments the frequency, amplitude and waveform of the main wave (the carrier) and then the frequency and amplitude of the LFO (the modulator).
There is a destructor, whose implementation remains empty because no pointers are used in this class.
Next, three fields are declared:

- The main wave
- An `'AudioEffectMultiply'` instance used for amplitude modulation. This object just multiplies two signals together and the reason why a multiplicative stage was used (and not for example a add) is explained when discussing the '.cpp' implementation.
- An `'AudioFilterBiquad'` instance (a low-pass filter) is used to filter each sine wave. In fact, even if ideally the spectrum of a sine wave is a single delta function, in reality it's a very narrow Gaussian curve, and it has some harmonics. Even if this is not a big problem when using only one sine wave, because these harmonics are very small, when using many sine waves together (as in this thesis work) they could add up and create unwanted audible frequencies. This low-pass filter avoids this effect.

The the five methods for setting frequency and amplitudes of the wave and the LFO are declared.
Finally, as private fields, there is the LFO wave, and three connections are needed: one from the wave to the low-pass filter, one from this filter to the multiplicative stage and a third one from the LFO to the multiplicative stage.
Now the "MyWave.cpp" is discussed:

```
MyWave::MyWave(float f, float a, int wf, float f_LFO, float a_LFO):
                  patchCord3(wave, 0, lpAntiAlias, 0),
                  patchCord1(lpAntiAlias, 0, mult, 0),
                  patchCord2(LFO, 0, mult, 1){
```

```
      LFO.frequency(f_LFO);
      // Changing the amplitude of 'wave' actually means changing the
      // 'LFO.offset'
      LFO.amplitude(a_LFO);
      LFO.offset(a);

      wave.frequency(f);
      wave.amplitude(1.0);
      wave.begin(wf);

      lpAntiAlias.setLowpass(0, 15000);
}
void MyWave::setAmplitude(float a){
      LFO.offset(a);
}
void MyWave::setFrequency(float f){
      wave.frequency(f);
}
void MyWave::setLFO_Amplitude(float a){
      LFO.amplitude(a);
}
void MyWave::setLFO_Frequency(float f){
      LFO.frequency(f);
}
void MyWave::setLFO_Ampl_Freq(float a, float f){
      LFO.amplitude(a);
      LFO.frequency(f);
}
```

Code snippet 4.3: "MyWave.cpp" implementation

First, the three `'AudioConnection'` instances are initialized in the initialization list of the constructor, making the appropriate connections.
In the "Teensy Audio Library" there is no predefined object that perform amplitude modulation, and it isn't possible to connect the LFO directly to the wave amplitude (as done in Processing). Therefor, a different approach is needed.
The main wave is defined as an oscillator always having amplitude of 1.0 (the maximum); the LFO instead is defined as a sine wave having a certain amplitude and frequency, but offset equal to the desired amplitude for the main wave.
These are then multiplied, and this way, considering for example the peaks of the main wave, which are always equal to 1.0, the result after multiplication is always the LFO sample value at that time instant. This can be seen in Figure 4.3.

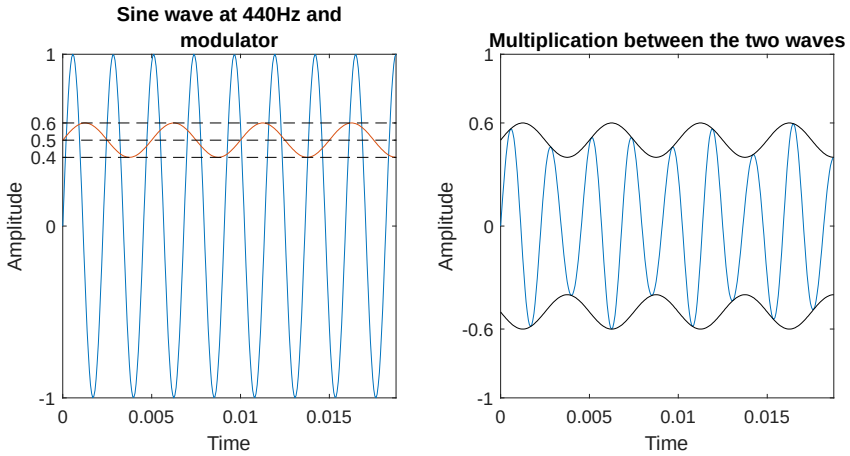


Figure 4.3: Shows an amplitude modulation done by multiplication. The carrier has amplitude 1.0, the modulator has amplitude 0.1 and offset 0.5.

This is why `'wave'` is initialized with amplitude 1.0, and the actual amplitude that is passed goes to the LFO offset; the other elements are initialized as expected.
After the constructor, the five methods used to set amplitude and frequency of the two waves are implemented.

4.4.2 AudioEffectOverdrive class in Teensy implementation

Also in the "Teensy Audio Library" there is no component or class for the overdrive effect. This is why it's necessary to implement this class from scratch. It will be used not only to generate the burst effect of the engine, but also on some sinusoids to add some color, so it is a central component of the final algorithm.
As already stated, this library processes sound samples in buffers, and not by taking the single sample each time as in Processing. Furthermore, also the patching system is different and has to be managed when creating a new class that processes these buffers.

First, the header "AudioEffectOverdrive.h" is discussed:

```
class AudioEffectOverdrive: public AudioStream {
public:
      AudioEffectOverdrive(void) : AudioStream(1, inputQueueArray){}
      virtual ~AudioEffectOverdrive();

      virtual void update(void);
```

```
      void setThreshold(float th){
                  if(th > 1.0){
                        th = 1.0;
                  }
                  if(th < 0.0){
                        th = 0.0;
                  }
                  threshold = th * 32767;
      }

      void setMultiplyFactor(int m){
                  multiplyFactor = m;
      }

private:
      audio_block_t *inputQueueArray[1];
      int16_t threshold;
      int16_t multiplyFactor;
};
```

Code snippet 4.4: 'AudioEffectOverdrive.h' header, with all the fields that are used

Starting from the bottom, the private fields that are used are three:

- The `'threshold'`: it's a 16 bits int that represents the maximum absolute value that a sample can assume. If a sample is greater than this threshold in absolute value, than it is reset to the threshold itself (maintaining the sign). This creates discontinuities in the waveform and therefor introduces some harmonics. This technique is called "clipping" and is discussed in Section 3.5.2 and showed in Figure 3.6.
- The `'multiplyFactor'`: it's a 16 bits number that holds the value to which each sample should be multiplied before clipping. If the sound is to be distorted a lot (introduction of many harmonics) then this value is usually high, so that the probability that a sample after multiplication is greater than the threshold is big. In some other cases this value can be 1, meaning that no multiplication should occur, but just clipping.
- The `'inputQueueArray'`: it's a pointer to an array of 'n' elements, where n corresponds to the number of inputs that the class can have. In this case, the overdrive element should have one input (the incoming signal to be modified), therefor n=1.
  The pointer is of type `'audio_block_t*'`. This is basically a struct that holds different useful parameters, and most importantly the buffer of samples.

Going back to the constructor, the important thing to analyze is the initialization list: since this class inherits from the class `'AudioStream'`, it sets this `'AudioStream'` by passing as arguments the number of inputs of the class and the `'inputQueueArray'`.
Without going into to much detail, which would result in deeply analyzing the library, the logic is that once the `'inputQueueArray'` size is defined (the number of inputs), when an audio connection will be made in future, this will result in connecting one of the outputs of source object to one of these `'audio_block_t*'` pointers contained in `'inputQueueArray'` of the destination object. In other words, samples blocks will be transmitted from the source object to this one, arriving exactly at `'inputQueueArray[n]'`, where 'n' is the input index defined when connecting.
Next, the `'update()'` virtual function is declared: it is the core of the processing, where the sample buffers are read and modified.
Finally, two small public methods are implemented: they set the threshold value and the multiplication factor, after controlling boundedness.
Note how the threshold is converted from a number in the interval [0.0, 1.0] to a integer in the interval [0, 32767]. Each sample is in fact represented with 16 bits, which, considering also negative values, corresponds to $2^{16}/2$=32768.

Now the "AudioEffectOverdrive.cpp" is discussed:

```
void AudioEffectOverdrive::update(void){
      uint32_t i;
      audio_block_t *block;
      block = receiveWritable();
      if (!block) {
          return;
      }

      for(i = 0; i < AUDIO_BLOCK_SAMPLES; i++){
            int32_t overdriveSample = block->data[i] * multiplyFactor;
            if(overdriveSample<=threshold && overdriveSample >= -threshold){
                  block->data[i] = overdriveSample;
            }
            else if(overdriveSample > threshold){
                  block->data[i] = threshold;
            }
            else if(overdriveSample < -threshold){
                  block->data[i] = -threshold;
            }
      }

      transmit(block);
      release(block);
}
```

Code snippet 4.5: "AudioEffectOverdrive.cpp" implementation

As showed, only the `'update()'` virtual method is implemented in the ".cpp" file. This is where the actual processing occurs.
First the object `'block'` of type `'audio_block_t*'` is declared: this will hold the sample buffer. The values of the buffer are filled through `'receiveWritable()'`, a method belonging to the `'AudioStream'` class. The "writable" part means the values of this block can be modified; there is also the `'receiveReadOnly()'` method.
After a check to see if block is not empty, a for loop over the samples of the buffer contained in `'block'` starts. The local variable `'overdriveSample'` holds the value of the sample already multiplied by the multiplication factor.
At this point a series of checks is done to see if this `'overdriveSample'` is smaller than the threshold and should be left unchanged, or if it's greater and it should thus be reset to the threshold value.
Finally, the block is transmitted to outgoing connections and, once the transfer of the sample buffer is done, it is released from memory in the local instance.

4.4.3 SignalSampleReader class in Teensy implementation

When doing amplitude modulation in Section 4.4.1, the component that was used was the `'AudioEffectMultily'`, a simple class that multiplies two signals.
The limitation though is that the two signals that enter are limited in amplitude in the interval [0.0, 1.0], and as a consequence also the output is bounded to that interval.
If a modulating signal with huge amplitude is needed (for example in frequency modulation, where amplitude corresponds to the frequency shift in Hz), there is no way of using it, because every signal in "Teensy Audio Library" will be eventually clipped to the range [0.0, 1.0].
For frequency modulation of a sine wave, there is a special class, the `'AudioSynthWaveformSineModulated'`, which takes as input a modulating signal with amplitude in the interval [0.0, 1.0] and outputs the modulated sine wave: when the value of modulating signal is +1, the frequency of the outputted sine wave doubles, when it is -1 it becomes 0.
This is a good compromise when FM Modulation is needed, but it's restricted only to sine waves and only for frequency modulation.
In this thesis work, at some point, other types of modulations are needed: for example modulating the center frequency of a band-pass filter with an LFO. It is always "frequency modulation", but not of a sine wave, and there is no component specifically built for that.

This is the reason why a new class was implemented, the 'SignalSampleReader': it takes as input a sine wave with amplitude 1.0 and it reads at each time instant the current sample value, a float between -1.0 and 1.0. Once this value is returned in the main program, it can be used as desired just by multiplying by the correct factor. For example if a frequency shift of 200Hz is wanted, the value that this class returns will be multiplied by 200.
It is important to note that this approach doesn't work in all cases, because the measurement unit of "Teensy Audio Library" is a buffer of 128 samples, not the sample itself. This means that at each time instant, when the sample value is needed, only one of the currently available 128 samples can be returned: it doesn't matter which one, it could be the first or the last.
The problem is that, with this approach, the actual sampling frequency of this modulating wave reduces drastically: only one sample every 128 is read, resulting in a sampling frequency of 44100/128=344.53Hz. This means that waves with frequency higher than 344.53/2=172.27Hz used as modulators would produce aliasing.
In this thesis work though, the LFOs that are used stay in the order of 1-20Hz, so this approach works relatively well.

Now that these clarifications are done, here below the "signalSampleReader.h" is presented:

```
class SignalSampleReader : public AudioStream {
public:
      SignalSampleReader() : AudioStream(1, inputQueueArray){}
      virtual void update(void);
      float read();
      virtual ~SignalSampleReader();

private:
      audio_block_t *inputQueueArray[1];
      int16_t sample;
};
```

Code snippet 4.6: "SignalSampleReader.h" header, with all the fields that are used

As in the previously discussed class 'AudioEffectOverdrive', also this inherits from the class 'AudioStream'. It has one input, so 'inputQueueArray' has size 1, it initializes this array in the initialization list of the constructor, and it has the virtual method 'update()', in which calculations are done. For more details on these fields coming from 'AudioStream', refer to previous Section 4.4.2.
The variable 'sample' stores the current value, and is constantly updated; the method 'read()' just returns this 'sample'.

Here below, the actual "signalSampleReader.cpp" implementation:

```
void SignalSampleReader::update(void){
      uint32_t i;
      audio_block_t *block;

      block = receiveReadOnly();

      if (!block) {
                  return;
      }

      sample = block->data[AUDIO_BLOCK_SAMPLES - 1];

      release(block);
}

float SignalSampleReader::read(){
      return sample/32767.0;
}
```

Code snippet 4.7: "SignalSampleReader.cpp" implementation

In the `'update()'` method, the buffer of samples is obtained in read-only mode, the last sample of this `'block'` is stored into the variable `'sample'`.
At this point, every time the programmer wants to access this variable from the main program, the method `'read()'` must be called, which returns a value between -1.0 and 1.0.

4.5 Final implementation of the audio synthesis algorithm on the Teensy 3.2 board

This is the final implementation: the core objective here was to refine the synthesized sound as much as possible, based on the model built in the previous chapters.
In this implementation some components were discarded to clean up the sound, especially the stochastic components such as noise. Other components were instead added to the code to make a richer and more elaborate sound, provided that there was room for these components.
It was more a work of tweaking the parameters, constantly focusing on what sounds better, even if this meant throwing away some logical algorithmic blocks, or re-thinking new ways to obtain a better result.

Before starting to present the actual code, the general schema in Figure 4.4 shows all the components that will be used and their connections. This graph was made through the online "Audio System Design Tool", accessible from the official Teensy site.
It is generated automatically by importing the source code section of the patches in the program, and so the names of the blocks are the same as the variable names.
This schema gives a visual representation of the different processing stages, and it's a good point of reference when the single components are presented in the code.

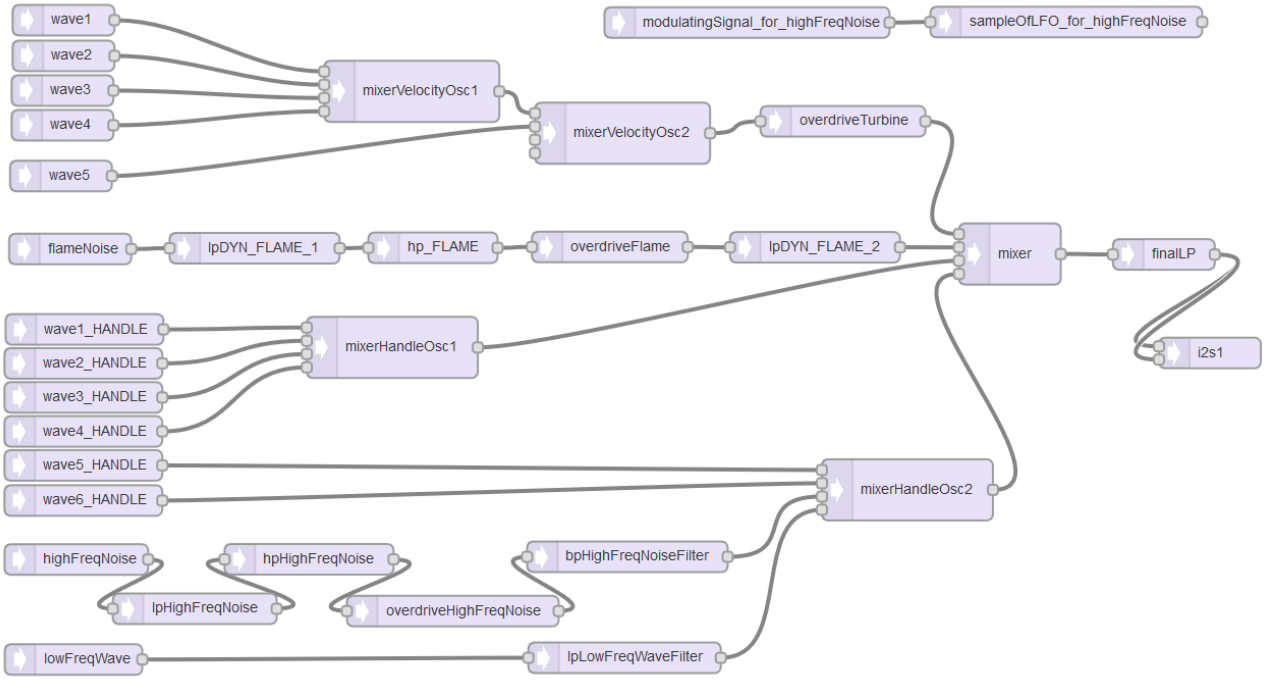


Figure 4.4: Connection schema of the Teensy algorithm implementation

All of the sound sources (sine waves or noise sources) are listed on the left side; everything else is either an effect (filter, overdrive) or a mixer that collects different signal in one. Note how in "Teensy Audio Library" mixers have a maximum of 4 inputs, so it wasn't possible to put one final big mixer collecting everything.
In the top-right corner there are two blocks that are separated from everything else: this is an LFO connected to an instance of `'SignalSampleReader'` (Class discussed in Section 4.4.3). This class just reads the current sample value of the incoming LFO, and this value is then used in the code to move the central frequency of a band-pass filter, but no connection is needed.

Now the implementation details of these components and the audio processing algorithm will be discussed.

As in the hybrid implementation of Subchapter 3.6, also here the sound synthesis in divided in two main blocks: the turbine, which has higher contribution given by sine waves components, and the flame, with a noise-like sound that reproduces the engine roar when accelerating.

4.5.1 Declaration of components and preliminary work:

First, the declaration of all the turbine components is presented here below:

```
AudioOutputI2S           i2s1;
AudioControlSGTL5000     sgtl5000_1;

// ============= TURBINE ========================
MyWave wave1(309.7, 0.0, WAVEFORM_SINE, 0.0, 1.0);
MyWave wave2(449.5, 0.0, WAVEFORM_SINE, 0.0, 1.0);
MyWave wave3(558.8, 0.0, WAVEFORM_SINE, 0.0, 1.0);
MyWave wave4(747.1, 0.0, WAVEFORM_SINE, 0.0, 1.0);
MyWave wave5(1100.0, 0.0, WAVEFORM_SINE, 0.0, 1.0);

AudioEffectOverdrive overdriveTurbine;

MyWave wave1_HANDLE(150.0, 0.05, WAVEFORM_SINE, 5.0, 1.0);
MyWave wave2_HANDLE(250.0, 0.05, WAVEFORM_SINE, 5.0, 1.0);
MyWave wave3_HANDLE(350.0, 0.05, WAVEFORM_SINE, 5.0, 1.0);
MyWave wave4_HANDLE(450.0, 0.05, WAVEFORM_SINE, 5.0, 1.0);
MyWave wave5_HANDLE(600.0, 0.05, WAVEFORM_SINE, 5.0, 1.0);
MyWave wave6_HANDLE(520.0, 0.05, WAVEFORM_SINE, 5.0, 1.0);

MyWave lowFreqWave(80.0, 0.01, WAVEFORM_SQUARE, 2.5, 1.0);
AudioFilterBiquad lpLowFreqWaveFilter;

AudioSynthNoiseWhite highFreqNoise;
AudioFilterBiquad bpHighFreqNoiseFilter;
AudioEffectOverdrive overdriveHighFreqNoise;
AudioFilterBiquad lpHighFreqNoise;
AudioFilterBiquad hpHighFreqNoise;
AudioSynthWaveform modulatingSignal_for_highFreqNoise;
SignalSampleReader sampleOfLFO_for_highFreqNoise;

AudioFilterBiquad finalLP;
```

Code snippet 4.8: Declaration of turbine components in the Teensy implementation

As it can immediately be seen, the first five waves start exactly at the same frequencies used in previous implementation: these are the partial components taken from Andy

Farnell's book typical of a propulsion engine sound. In this implementation they are associated with the velocity of the motorcycle and they have no LFO modulating their amplitude: LFO is in fact initialized with frequency 0.0 and amplitude 1.0, a constant signal at 1 that is then multiplied internally (see `'MyWave'` discussion in Section 4.4.1) to the actual carrier wave; note that if the amplitude of LFO was set to 0.0, this would result in generating a 0.0 signal after the multiplication stage.
`'overdriveTurbine'` is an instance of class `'AudioEffectOverdrive'`: it applies a small overdrive to these 5 sine waves. The details of operation are discussed later on, but it is important to note that this component wasn't present in the previous implementations: it is here used to introduce some harmonics, thus making these sine waves a little more dirty and less squeaky.
This is a first example of a component added not to change the synthesis logic, but to perfect the sound that is heard.
Next, 6 more sine waves are created. These are the ones coming from the first implementation discussed in Subchapter 3.4, and more precisely from the model built through analysis in Chapter 2.
In this implementation it was decided to use 5 harmonics (equispaced of a Δf=150Hz) and 1 partial in a medium-low band of frequencies: the fundamental starts at 150Hz and they all double in frequency as handle inclination increases. The partial at 520Hz was introduced to give some inharmonicity to the sound, which would otherwise sound too musical.
There is one low frequency square wave starting at 80Hz: it is square to get a richer sound out of one single oscillator. The partials after 600Hz are filtered out through `'lpLowFreqWaveFilter'`.
The `'highFrequencyNoise'` is the noise that in previous implementations was subject to a band-pass filter with undulating central frequency: it was just normal filtered white noise. The sound that was generated before though was too flat and unrealistic/artificial, and so now the approach is the same as the one used for the "flame" sound: a white noise source goes through a chain of filters and an overdrive stage (all the remaining instances) to introduce some harmonic components and create a "ripped air effect", a more engine-like sound (for more details about this approach, refer to Subchapter 2.2, for the implementation details refer to Section 3.5.2).
The chain of processing stages through which it passes can be seen in Figure 4.4; the important thing to note is that the last filter it goes through, `'bpHighFreqNoiseFilter'`, has its central frequency regulated by the `'modulatingSignal_for_highFreqNoise'` LFO, and `'sampleOfLFO_for_highFreqNoise'` is used to read the sample values of this LFO.
The `'finalLP'` component instead is a filter used before sending the final synthesized sound to the audio shield, in order to remove any unwanted high frequency components.

The declaration of all the components in the "flame" sound are instead presented below:

```
// ============= FLAME ============================
AudioSynthNoiseWhite flameNoise;
AudioFilterBiquad lpDYN_FLAME_1, lpDYN_FLAME_2;
AudioFilterBiquad hp_FLAME;
AudioFilterBiquad bpTRY;

AudioEffectOverdrive overdriveFlame;
```

Code snippet 4.9: Declaration of flame components in the Teensy implementation

The logic here is the same one used in previous implementation (as discussed in Section 3.5.2): a white noise source (`'flameNoise'`) first goes through a low-pass filter (`'lpDYN_FLAME_1'`) to create some space in the frequency domain; then a high pass filter (`'hp_FLAME'`) is applied to it to avoid audio fragmentation; at this point, a very high overdrive (`'overdriveFlame'`) is applied to it, meaning the samples are multiplied by a very large number and then clipped back to a threshold value, thus generating a sort of non-periodic square wave. Finally the resulting sound goes through a dynamic low-pass filter (`'lpDYN_FLAME_2'`), whose cutoff frequency increases with handle inclination, thus letting through more of this engine roaring sound.

All the patching system is done by creating `'AudioConnection'` instances, and by passing the correct components to connect them appropriately.
Since it is better to visualize these connections through Figure 4.4 rather than reading the full list of the patching system, only a few of these connection declarations are showed as an example.

```
AudioMixer4 mixerVelocityOsc1, mixerVelocityOsc2, mixer, mixerHandleOsc1,
mixerHandleOsc2;
AudioConnection patchCord1(wave1.mult, 0, mixerVelocityOsc1, 0);
AudioConnection patchCord6(flameNoise, lpDYN_FLAME_1);

AudioConnection patchCordToFinalLP(mixer, finalLP);
AudioConnection patchCordOutputL(finalLP, 0, i2s1, 0);
AudioConnection patchCordOutputR(finalLP, 0, i2s1, 1);
```

Code snippet 4.10: Examples of 'AudioConnection' instantiation in the final Teensy implementation

Note how when connecting a `'MyWave'` instance (such as `'wave1'`) to any other component, in reality the output of the internal `'AudioEffectMultiply'` (the `'mult'` instance) must be connected; this output is in fact the result of the amplitude modulation, as discussed in Section 4.4.1.

Also note how it's not always necessary to pass output and input number of each component: when the source component has 1 output and the receiver component has 1 input, then these numbers can be omitted.

The last components that are declared before the actual initialization inside the `'setup()'` function are two float numbers. These are a filtered version of the instantaneous value of Velocity and Handle Inclination input parameters, meaning they are values that change more slowly with respect to the raw input signals.

```
float filteredVelocity;
float filteredHandleAngle;
```

Code snippet 4.11: The two float numbers which will represent a filtered version of the instant Velocity and Handle inclination input parameters. Final Teensy implementation

Their value is updated at the beginning of the `'loop()'` function, when reading from GPIO pins the value of the two potentiometers:

```
float knob_A3 = (float)analogRead(A3) / 1023.0;
float knob_A2 = (float)analogRead(A2) / 1023.0;

filteredVelocity = filteredVelocity + (knob_A2 - filteredVelocity)/512;
filteredHandleAngle = filteredHandleAngle + (knob_A3 - filteredHandleAngle)/256;
```

Code snippet 4.12: Reading and filtering of the two potentiometers value inside the 'loop()' function

The function `'analogRead()'` is a default function from the Arduino library: it receives a pin ID and reads the value, mapping it to the interval [0, 1023.0].
The two variables `'filteredVelocity'` and `'filteredHandleAngle'` are then set, by applying a classic filtering formula.
Note how the coefficient is 1/512 for the velocity (transitions go slower) and 1/256 for the handle inclination (transitions go faster).
These two variables are the ones used to control everything in the processing stages.

The first components that are initialized inside the `'setup()'` function are the ones that deal with memory management and communication between the Teensy board and the audio shield:

```
Serial.begin(9600);
AudioMemory(64);
sgtl5000_1.enable();
sgtl5000_1.volume(0.4);
```

Code snippet 4.13: Debug, audio memory and communication between Teensy board and audio shield initialization

The memory is set to 64, which corresponds roughly to 16,25 kByte of RAM space dedicated to the audio components and the communication between them (for details about why this number and the need of using this function, refer to Subchapter 4.3).
The audio shield is enabled and the output volume is set to 0.4, which seems a good compromise.

4.5.2 Turbine sound: Initialization and evolution in the 'loop()' function

The main components of the turbine are two sets of periodic sine waves:

- Six sine waves with frequencies that are function of handle inclination and with an LFO applied to them
- Five sine waves with frequencies changing with velocity and a light overdrive applied to them.

Considering the first set, nothing should be initialized in the `'setup()'` function, since they only have an LFO modulating their amplitude, and this is managed internally in the `'MyWave'` class.
Their evolution inside the `'loop()'` function is presented below:

```
float freqShiftFactor_2 = map(filteredHandleAngle, 0.0, 1.0, 1.0, 2.0);
      wave1_HANDLE.setFrequency(150.0*freqShiftFactor_2);
      wave2_HANDLE.setFrequency(250*freqShiftFactor_2);
      wave3_HANDLE.setFrequency(350*freqShiftFactor_2);
      wave4_HANDLE.setFrequency(450*freqShiftFactor_2);
      wave5_HANDLE.setFrequency(600*freqShiftFactor_2);
      wave6_HANDLE.setFrequency(520*freqShiftFactor_2);

// Main oscillator's LFOs
float LFO_amplitude_handleOsc = map(filteredHandleAngle, 0.0, 1.0, 0.025,
                                                    0.05);
float LFO_frequency_handleOsc = map(filteredHandleAngle, 0.0, 1.0, 5.0,
                                                    8.0);
wave1_HANDLE.setLFO_Ampl_Freq(LFO_amplitude_handleOsc,
                                          LFO_frequency_handleOsc);
wave2_HANDLE.setLFO_Ampl_Freq(LFO_amplitude_handleOsc,
                                          LFO_frequency_handleOsc);
wave3_HANDLE.setLFO_Ampl_Freq(LFO_amplitude_handleOsc,
                                          LFO_frequency_handleOsc);
```

```
wave4_HANDLE.setLFO_Ampl_Freq(LFO_amplitude_handleOsc,
                                        LFO_frequency_handleOsc);
wave5_HANDLE.setLFO_Ampl_Freq(LFO_amplitude_handleOsc,
                                        LFO_frequency_handleOsc);
wave6_HANDLE.setLFO_Ampl_Freq(LFO_amplitude_handleOsc,
                                        LFO_frequency_handleOsc);
```

Code snippet 4.14: Evolution of waves function of handle inclination in Teensy implementation

The frequencies double as the handle inclination reaches the maximum value. Note how `'wave6_HANDLE'` starts at 520Hz and ends at 1040Hz: this sine wave is a partial component that gives a nuance of dissonance and makes the final sound less harmonic.
The amplitude of these waves doesn't change, and remains at 0.05, as declared when initializing them.
The LFO modulating the amplitude of these waves evolve in this way:

- Their amplitude starts at 0.025 (half of the carrier's amplitude) and ends at 0.05, which coincides with the carrier's amplitude, bringing it to 0 in the negative peaks
- Their frequency starts at 5 Hz and reaches 8 Hz at full thrust

Considering now the second set of five sine waves (the ones whose frequency is function of velocity), they don't have an LFO modulating their amplitude, but they do have a light overdrive applied to them. This is why the components that take part in the overdrive effect have to be initialized in the `'setup()'` function:

```
overdriveTurbine.setMultiplyFactor(2);
overdriveTurbine.setThreshold(1.0);
```

Code snippet 4.15: Initialization of overdrive components applied to the waves in Teensy implementation

The multiplication factor is set to 2, basically doubling the amplitude: this still preserves most of the shape of the signal and doesn't distort it as a multiplication factor of 100, as the one used for the flame.
The threshold is initialized at 1.0, but will later be updated constantly in the `'loop()'` function.
The evolution of these waves inside the `'loop()'` function is presented below:

```
float freqShiftFactor = map(filteredVelocity*filteredVelocity, 0.0, 1.0,
1.0, 7.0);
wave1.setFrequency(309.7*freqShiftFactor);
wave2.setFrequency(449.5*freqShiftFactor);
wave3.setFrequency(558.8*freqShiftFactor);
wave4.setFrequency(747.1*freqShiftFactor);
wave5.setFrequency(1100.0*freqShiftFactor);
```

```
float tot = 0.0;
float wavesAvgAmplitude = map(filteredVelocity*filteredVelocity, 0.0, 1.0,
0.0, 0.2);
wave1.setAmplitude(tot += wavesAvgAmplitude * 0.5);  // * 0.5
wave2.setAmplitude(tot += wavesAvgAmplitude * 0.25); // * 0.25
wave3.setAmplitude(tot += wavesAvgAmplitude * 0.5);   // * 0.5
wave4.setAmplitude(tot += wavesAvgAmplitude * 0.4);  // * 0.4
wave5.setAmplitude(tot += wavesAvgAmplitude * 0.4);  // * 0.4
float th = tot * 0.3; // 30%

overdriveTurbine.setThreshold(th);
```

Code snippet 4.16: Evolution of waves function of velocity in Teensy implementation

The first difference from previous implementations discussed in Chapter 3 is that now the shift in frequency of these waves increases of a factor x7 instead of x10. A factor x10 was in fact too big and it was making the sound too high-pitched, which can be annoying and disturbing if listened for long periods. A more comfortable sound was thus chosen in exchange for a little less frequency dynamics.
For what concerns the amplitude of these waves, the common multiplication factor (`'wavesAvgAmplitude'`) goes from 0.0 to 0.2 but with a parabolic behavior (note the *"filteredVelocity*filteredVelocity"* inside the `'map()'` function).
Some individual weights are then applied when assigning the amplitude, and the total is calculated and stored in `'tot'`. This is done because when multiple waves enter in a mixer, what actually happens is that the resulting output signal is a sum of all the input ones.
As it can be seen in Figure 4.4, the overdrive is then applied on this output signal, not on each one of these individual waves. This is why the threshold is set to 30% of the amplitude of output signal, which is a sum of the 5 sine waves.

The next crucial component of the turbine sound is the mid-high noise going through a band-pass filter whose central frequency undulates following an LFO.
This noise is not raw white noise, but it is processed before: it goes through a chain of filters and an overdrive stage, as explained in Section 4.5.1.
This is why many components must be initialized inside the `'setup()'` function:

```
highFreqNoise.amplitude(0.1);
lpHighFreqNoise.setLowpass(0, 150);
hpHighFreqNoise.setHighpass(0, 120);
overdriveHighFreqNoise.setMultiplyFactor(50);
overdriveHighFreqNoise.setThreshold(0.2);

bpHighFreqNoiseFilter.setBandpass(0, 1500, 1);
modulatingSignal_for_highFreqNoise.frequency(4);
modulatingSignal_for_highFreqNoise.amplitude(1.0);
```

Code snippet 4.17 Intialization of components for mid-high noise processing in Teensy implementation

First the white noise source's amplitude is set to 0.1. The first stage it goes through is a low-pass filter with cutoff frequency set at 150Hz.
Then it goes through a high-pass filter with cutoff frequency at 120Hz: this is done to avoid sound fragmentation at low frequencies, a disturbing effect that occurs when clipping low frequencies.
At this point the overdrive component `'overdriveHighFreqNoise'` is initialized: the multiplication factor is set to 50 and the clipping threshold is set to 0.2.
The last stage it goes through is a band-pass filter centered at 1500Hz and with Q=1, meaning the BW=1500Hz.
The LFO setting the central frequency of this band-pass filter is then initialized with a frequency of 4Hz and amplitude of 1.0.

The evolution of these components in the `'loop()'` function are presented below:

```
float fc = sampleOfLFO_for_highFreqNoise.read() * 500;
bpHighFreqNoiseFilter.setBandpass(0, 1500 + fc, (1500.0 + fc)/1000.0);

float freqHighFreqNoiseLFO = map(filteredHandleAngle, 0.0, 1.0, 4.0, 7.0);
modulatingSignal_for_highFreqNoise.frequency(freqHighFreqNoiseLFO);
```

Code snippet 4.18: Evolution of components for mid-high noise processing in Teensy implementation

First, the central frequency must be updated. The `'read()'` is a method of the class `'SignalSampleReader'`: the instance `'sampleOfLFO_for_highFreqNoise'` receives the LFO `'modulatingSignal_for_highFreqNoise'` and it returns a value between 0.0 and 1.0 (for more details refer to Section 4.4.3).
This value is multiplied by 500 and stored in `'fc'`, which represents and offset from the central frequency of 1500Hz. This is why the new central frequency is =1500+fc.

Also note how Q is dynamic: it is given by the new central frequency divided by 1000, the desired bandwidth. This guarantees that the bandwidth of this filter is always equal to 1000 Hz.
Finally the LFO regulating the central frequency of this band-pass filter is modified: it goes from 4 Hz to 7 Hz.

The last component of the turbine sound is a square low-frequency (80Hz) wave going through a low-pass filter at 600Hz:

```
void setup()
{
      …
      lpLowFreqWaveFilter.setLowpass(0, 600.0);
      finalLP.setLowpass(0, 12000);
}
```

Code snippet 4.19: Low-pass filter initialization for low frequency wave in Teensy implementation. Also the 'finalLP' initialization is showed.

In this last code snippet, also the `'finalLP'` low-pass filter is initialized with cutoff at 12000Hz. This is the filter through which the final synthesized sound goes, thus eliminating unwanted high frequency components.
The evolution of this low frequency wave is showed below:

```
float freqShiftFactor_3 = map(filteredHandleAngle, 0.0, 1.0, 0.0, 44.0);
lowFreqWave.setFrequency(80.0 + freqShiftFactor_3);
float lowFreqWaveAmplitude = map(filteredHandleAngle, 0.0, 1.0, 0.2, 0.5);
lowFreqWave.setAmplitude(lowFreqWaveAmplitude);
float LFO_amplitude_lowFreqOsc = map(filteredHandleAngle, 0.0, 1.0, 0.025,
                                                    0.2);
float LFO_frequency_lowFreqOsc = map(filteredHandleAngle, 0.0, 1.0, 2.5,
                                                    10.0);
lowFreqWave.setLFO_Ampl_Freq(LFO_amplitude_lowFreqOsc,
                                            LFO_frequency_lowFreqOsc);
```

Code snippet 4.20: Evolution of low frequency wave in Teensy implementation

The frequency of this wave goes from 80Hz to 124Hz following the handle inclination.
The amplitude shifts from 0.2 to 0.5. These are much higher values respect to the other waves in the algorithm, because human sensitivity is much less at low frequencies, as showed in Figure 3.4.
The LFO that is used for amplitude modulation has amplitude going from 0.025 to 0.2 (almost half of the final carrier amplitude), while the frequency goes from 2.5Hz to 10Hz.

4.5.3 Flame sound: Initialization and evolution inside the 'loop()' function

The flame sound only has one component: it is a white noise that goes through a chain of filters and an overdrive stage to produce the typical "roar" sound of a propulsion engine at full thrust.
Below the initialization of the blocks composing this flame sound:

```
flameNoise.amplitude(0.2);
lpDYN_FLAME_1.setLowpass(0, 50);
hp_FLAME.setHighpass(0, 120);

overdriveFlame.setMultiplyFactor(100);
overdriveFlame.setThreshold(0.3);

lpDYN_FLAME_2.setLowpass(0, 50);
```

Code snippet 4.21: Initialization of components for the "Flame" sound in Teensy implementation

The white noise source is initialized with amplitude of 0.2.
Next, the cutoff frequency of a dynamic low-pass filter is set to 50Hz. This value will increase following the handle inclination input parameter. This first low-pass filter is the one that creates some space in the frequency domain so that the harmonics introduced by following clipping peak out distinctively.
A high-pass filter with cutoff frequency at 120Hz is initialized. This is to avoid audio fragmentation, a disturbing effect that occurs when clipping low frequencies.
The sound then goes through the overdrive stage, which is initialized with a very high multiplication factor of 100 and a clipping threshold of 0.3. This basically creates a non-periodic square wave (for more details about this approach and its effects, refer to Subchapter 2.2).
Finally the sound goes through a second dynamic low-pass filter, which basically acts as a knob establishing how much this whole effect should be heard: when fully closed, no harmonics are heard and therefor the "roar" of the engine is inaudible; when fully open, this "roar" is at maximum audibility.

Below, the evolution of the two dynamic filters inside the 'loop()' function is presented:

```
float cutoff_FLAME_1 = map(filteredHandleAngle, 0.0, 1.0, 50.0, 150.0);
lpDYN_FLAME_1.setLowpass(0, cutoff_FLAME_1);

float cutoff_FLAME_2 = map(filteredHandleAngle, 0.0, 1.0, 50.0, 12000.0);
lpDYN_FLAME_2.setLowpass(0, cutoff_FLAME_2);
```

Code snippet 4.22: Evolution of components for the "Flame" sound in Teensy implementation

The first low-pass filter has a cutoff frequency shifting from 50Hz to 150Hz. The second low-pass filter instead goes from 50Hz to 12000Hz, to let through less or more of this flame sound as the handle inclination changes.

5
Conclusion and future work

Sound implementation for electric motorcycles (or electric vehicles more in general) is a novel concept, and is therefor open to an infinity of possible solutions.
The implementation that was developed in this thesis produces a satisfactory sound, and it follows the general guidelines that were imposed at the beginning of the work:

- It doesn't fall into a category of sound: it is neither an imitation of a real sound nor a completely new and unsettling sound.
  It is more of a fusion coming from different sources of inspiration: some samples in the real world, some sample coming from the movie industry (sci-fi) and a completely creative part not coming from anywhere.
- The algorithm is not too heavy, and it doesn't need too many resources. It can easily run on a Teensy 3.2 board, and even a lower model works great.
  The boot time of this board is within a few milliseconds, and therefor the constraint of having the sound starting as soon as the motorcycle turns on is satisfied
- The quality of the generated sound is pretty high, considering this will be played in an outside scenario and not in a perfect studio. It has a 44100Hz sampling frequency and 16 bit depth.

Although the sound that is produced is not disturbing per se, it would need a lot of testing on a large scale to see how people react to long exposure to this sound, and also to study the scenario in which multiple sources emit this sound (as in a real world scenario, e.g. a city).
It is certainly a work that is open to many future improvements and tweaking of the parameters.

REFERENCES:

[1] https://linksfoundation.com/
[2] https://www.nitobikes.it/
[3] https://hal.archives-ouvertes.fr/hal-01708888/document
[4] https://www.ted.com/talks/renzo_vitale_what_should_electric_cars_sound_like#t-3014
[5] https://www.nytimes.com/2017/07/08/climate/electric-cars-batteries.html
[6] https://www.sonicvisualiser.org/
[7] https://en.wikipedia.org/wiki/Piano_key_frequencies
[8] Designing Sound by Andy Farnell – The MIT Press
[9] https://en.wikipedia.org/wiki/Sound_design
[10] https://www.studiarapido.it/teatro-greco-struttura-architettura/#.XGmGmqDSKpo
[11] https://en.wikipedia.org/wiki/Processing_(programming_language)
[12] http://code.compartmental.net/tools/minim/
[13] https://processing.org/examples/scrollbar.html
[14] https://www.st.com/content/st_com/en/products/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus/stm32-high-performance-mcus/stm32f4-series.html?querycriteria=productId=SS1577
[15] https://www.st.com/content/ccc/resource/technical/document/application_note/group0/c1/ee/18/7a/f9/45/45/3b/DM00273990/files/DM00273990.pdf/jcr:content/translations/en.DM00273990.pdf
[16] https://www.st.com/en/ecosystems/stm32cube.html?querycriteria=productId=SC2004
[17] https://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-mpu-eval-tools/stm32-mcu-mpu-eval-tools/stm32-nucleo-boards.html?querycriteria=productId=LN1847
[18] https://www.mbed.com/en/
[19] https://dspconcepts.com/st
[20] http://www.nime.org/proceedings/2017/nime2017_paper0087.pdf
[21] https://www.pjrc.com/teensy/
[22] https://www.pjrc.com/teensy/td_libs_Audio.html
[23] https://www.pjrc.com/teensy/gui/index.html
[24] https://www.pjrc.com/store/teensy3_audio.html
[25] https://www.pjrc.com/teensy/techspecs.html
[26] https://www.pjrc.com/teensy/card7a_rev1.pdf
[27] https://www.pjrc.com/teensy/card7b_rev1.pdf
[28] https://www.pjrc.com/teensy/loader.html
[29] http://eclipse.baeyens.it/
[30] https://forum.pjrc.com/archive/index.php/t-29008.html