SCHOOL OF SCIENCE ENGINEERING

EGR 4402 CAPSTONE DESIGN

April, 22, 2019

Nassim El Abed

Supervised by: Dr. Assem Nasser

Student Statement

I, the designer, state that I have applied ethics to the design process and in the selection of the final proposed design. I also state to have held the safety of the public to be paramount and have addressed this in the presented design wherever may be applicable.

Nassim El Abed

Approved by the Supervisor(s)

SUPERVISOR Dr. N. Assem

Table Of Contents

I.	Acknowledgments	4
II.	Abstract	5
III.	Introduction	7
IV.	STEEPLE Analysis	9
V.	Initial Research	11
VI.	Main Research	15
VII.	Requirements Specification	20
VIII.	Technology Of Choice, Part 1/2	22
IX.	Architecture & Implementation, Part 1/2	25
X.	Project Evolution: A Creative Test Case	32
XI.	Technologies Of Choice, Part 2/2	34
XII.	Architecture & Implementation, Part 2/2	37
XIII.	Future of the Project & Conclusions	42

// I. Acknowledgments

I would like to acknowledge *Dr. Assem Nasser* for accepting my project proposal and for the project help and follow up. I would to thank the online communities that wrote and provided extended documentation for all the software I used. Finally, I would to thank my parents, brother and family.

II. Abstract

The aim of this project is to develop a distributed approach to the emulation of specialized computer systems, such as game consoles and sound synthesizers. The project is to make use of parallelism in processing data in order to reduce the latency during the emulation.

The analysis phase consists of three parts. The first part is concerned the analysis of the emulated system(s), and is to be done by studying both the hardware and software specifications of the original system(s). The second part deals with the distribution and parallelization of the emulation tasks in the host system, and aims at determining the most optimal choices of programming languages and paradigms for the live processing of the input data. The third part consists of analyzing the possibilities for automation in the emulation system's architecture. The degree up to which the task distribution in the host system can be automated depends on the complexity of the emulated system, and is thus to be determined depending on the results of the two first parts of the analysis.

The design phase is directly dependent on the results of the analysis of the emulated system's hardware, and is thus determined after the completion of the analysis phase.

The implementation phase is intertwined with the testing phase, as every step in the implementation process needs to be tested extensively, using different input data, before moving on to the next. The success of a test case can be determined by comparing the similarity between the data (such as audio/ visual signals) outputted from the original system and that of the host’s emulation system, at a given point in time, using the same input

settings. Considering the project's focus on parallelism, it is also important to measure the latency numbers throughout this phase and test for the most optimal configurations.

The software is to make use of recent development concepts in computer parallelism and is to be extensively documented. The societal and ethical implications of the project can be summed up as an advancement towards the reverse engineering and development of complex systems, a process which can virtually benefit all complex systems, which are known to require heavy and lengthy computations. Improving these processes, the technologies of which are directly transferable to sciences such as physics, chemistry, and multidimensional ethical problems in game theory for instance would thus be of use to multiple disciplines.

III. Introduction

There are many fields that have been revolutionized with the advent of high performance processors and shifts in computing paradigms. However, technologies from older generations have to undergo adaptation in order to run on new hardware and software. This process is called emulation.

The process of software emulation consists of developing software solutions for host computer to be able to simulate another computer, whose hardware architecture and operating system are different from the host.

The reasons for emulation differ from one field to another. In the case of video games for instance, emulation enthusiasts aim for the enjoyment of entertainment software that are either discontinued or hard to find because of limited releases.

In the case of audio emulation, which is the focus for my projects, the aim is to simulate the sound processing enabled by specific sound system consoles, for example delay units and synthesizers, so as to either generate or shape sound in specific ways, by using a personal computer only.

The physical weight and typically expensive costs of audio generators and processors are amongst the biggest motivations for music artists to use emulation.
The problems faced when emulating systems are mainly of two sorts. First, the reverse engineering efforts required on the developers' end are numerous and uncertain, as a lack of public documentation for either hardware architecture or software components is not

uncommon when trying to write an emulator for a given system. Second, there is the problem of performance, which this project tries to deal with.

The problems of performance arise from the guest architecture being typically specific in its makeup, differing in instructions from the host's (i.e. MIPS versus x86). Thus, every instruction from each chip needs to undergo live translation, all of which results in performance lags. Another reason for performance lags in the specific case of audio processor emulation is the sample buffer limit dictated by a given guest system design, which become much more taxing on the CPU.

The project attempts to solve problems of performance by identifying and restructuring the processes that can be distributed and data that can be cached and retrieved, and, ideally, to do so automatically.

IV. STEEPLE Analysis

Social

The democratization of musical technologies with the advent of laptops and open source software like Audacity has been a large factor in the growth of individual artistic developments throughout the whole world. Platforms like Soundcloud allowed people to share their content with their peers online, and are mainly fueled by content outputted from Digital Audio Workstations that rely on audio plugins, the like of which whose performance I try to improve in this project. Also, developments in parallel audio processing means increments of individual artist capabilities thanks to cloud services. All of this ultimately contributes to society through the arts.

Technological

The technological improvements that may be contributed with this project can be applied to different disciplines, whose impacts can range from health (i.e. Distributed simulation in Pharmacology) to crime prevention (i.e. Multidimensional societal simulation problems, as in Network Analysis)

Economical

If offered as a service, distributed audio console system emulation through cloud computing can generate massive revenue from music artists and companies. The music software market is, according to Business Wire, expected to grow to USD 6.26 billion in the 2018-2022. The economical impact can prove to be very important is high performance upgrades are achieved.

Environmental

The impact of the computing power required by cloud services, if these are eventually used, can be a negative environmental factor. The project does not, however, have other impact, be

it positive or negative, on the environment.

Political

No impact on a direct level. However, the applications of parallel processing and signal processing, due to their broad nature, might branch out into fields of political nature. The subject is however not the focus here.

Legal

The emulated systems contain copyrighted software and patented architecture, and should not be distributed unless we are authorized to do so.

Ethical

The ethical implications are that digital emulation does not maintain the “integrity’ of the original hardware sound, according to many musicians. This extends a decades-old debate about digital versus analog music.

V. Initial Research

The process of system emulation consists of imitating the behavior of a system, which results in the making of a virtual system. The process has been used since the early days of computing.

The process requires a thorough study of the original hardware, as well as the embedded software, so as to produce results that are very similar to the original's.

In the specific case of my project, I was to study the architecture of specialized computer systems ("*Specialized computer systems handle specific tasks,…such as point of sale systems, ATMs, and GPS."*) in order to find possible room for improvement in performance.

The potential cycle of improvement I focused on was the following:

1. Identify sequential processes which can be parallelized. That a data flow was sequential while it could have been parallelized could have been due to, for instance, the availability of inferior technology (in comparison to now) at the time during which the system was designed.
2. Determine alternative designs which would take advantage of computer parallelization. Considering the complexity of system architectures, a direct change in the architecture is likely to break the data flow. Instead, the approach I opt for is a middle ground between *Emulation* and *Simulation.*

During the research phase of my project, I established a list of possible hardware systems to emulate, classified by category. The list is presented in the following table.

Specialized Computer Type	**Processed Data**	**Notable Examples**
Video Game Console	Audio/Visual	Sony Playstation One Sega Dreamcast Nintendo Famicom
Sound Generator	Audio	Yamaha DX7 Mellotron Mk II Moog Prodigy
Sound Processing	Audio	Fender Bassman (Amplifier) Lexicon 480L (Reverb) Sontec 432 (EQ)

After establishing the list, the following phase of the research consisted in reading as much documentation, technical details and emulation history concerning each type of system. The choice I have made after this phase was emulate a *Sound Processing* unit, as it would be an interesting case of distributed emulation, which processed data that is sent in realtime -or almost, in this case, audio data,, something that the other types of specialized systems were not concerned with doing. It would open the doors for system distribution using a server-like system which takes sound signals as input, and thus proved to be the most exciting alternative in application.

To emulate sound processors and sound generators has been a major subculture since the dawn of computer music, and has eventually grown into a multibillion dollar industry of audio software and audio software plugins, whose aim is to emulate expensive and large hardware consoles in the best way possible for computer devices such as laptops, which now dominate the music industry as the main tool of production

. Notable companies include *Waves Technology, iZotope Inc.,* and *Steinberg,* which developed the industry standard, *VST Technology.*

"Virtual Studio Technology (VST) is an audio plug-in software interface that integrates software synthesizer and effects in digital audio workstations. VST and similar technologies use digital signal processing to simulate traditional recording studio hardware in software. Thousands of plugins exist, both commercial and freeware, and a large number of audio applications support VST under license from its creator, Steinberg." (*https://en.wikipedia.org/wiki/Virtual_Studio_Technology*)

As for the parallel processing distribution, the aim is to achieve the same results as in sequential processes, but with less audio lag, since the computation involved in high quality audio hardware emulation are typically intensive and result in glitches and delays. The problem of audio latency is renown amongst musicians who use computers to record music (which is pretty much all musician at this point), and was experienced by anyone using music software.

There are four approaches of possible performance improvements that I determined:

1. Horizontal CPU task partitioning. i.e: CPU multithreading.
2. Vertical CPU task partitioning. i.e: Divide & conquer algorithms.
3. Horizontal GPU partitioning. i.e: GPU multithreading
4. Vertical GPU partitioning. i.e: Complexity reduction in matrices.

The methods are to be discussed in details later on during the implementation & architecture phase, since the architecture of choice was determined from series of trial and error, by measuring performances, instead of preemptive design.

With all this data, the project decision was finalized, and the *Sound Processing Unit* I am to emulate is a *Pitch Shifter / Harmonizer.*

The Eventide H910 Harmonizer Rack Unit

A *Pitch Shifter* is the following: *"...a sound recording technique in which the original pitch of a sound is raised or lowered. Effects units that raise or lower pitch by a pre-designated musical interval (transposition) are called pitch shifters or pitch benders."* (https://en.wikipedia.org/wiki/Pitch_shift). A *Harmonizer* is a *Pitch Shifter* which works in the same way, but outputs a combination of multiple signals (instead of one), which are shifter using different values. It is used to generate vocal harmonies (such as in choirs) using a single sound signal, which is typically sung vocals, but can be, for example, a guitar sound signal.

The work of parallelization accomplished in this project can be easily transferred to the emulation of other hardware processing systems such as *delay units* or *impulse-based reverb units*, as will become clear in the following section.

VI. Main Research

The software emulation of *Pitch Shifters* and *Harmonizers* aims at changing the pitch (read: tone) of a sound, without changing its duration, which would simply consist of interpolating the signal along a different range.Since frequencies of the waves determine the pitch of the signal, and the rate and frequency of a signal are related by the formula:

Rate = Frequency * Wavelength

it is a challenging, multi-step process to approximate a different frequency value and leaving the rate unchanged, so that the duration of the signal remains the same while its pitch changes.

The process works as follows:

1. Signal A is obtained from an input source, such as a microphone.
2. Signal A is split to and processed by chunks of fixed size (i.e. a buffer of 2048 integers).
3. Signal A is stretched by a factor F, determined from the following formula:

   F= 2**(1.0 * *S* / 12.0)

   with: S = +/- semitones.

   A semitone is the "musical distance" between a note and the one which follows it (assuming the standard *Western tuning of twelve equal intervals per octave*), where 12*S represents a full octave.

   Non-integers can be used as *S* values for *Microtonal Music* ("*a microtone may be thought of as a note that falls between the keys of a piano tuned in equal temperament*" https://en.wikipedia.org/wiki/Microtonal_music)

The stretching of a signal can be accomplished in different ways. The one I opt for consists of:

a. Splitting the signal chunk into smaller, overlapping parts - "*windows*"

b. Rearranging the spacing between the *windows* (smaller space for shorter, bigger inner space for a longer)

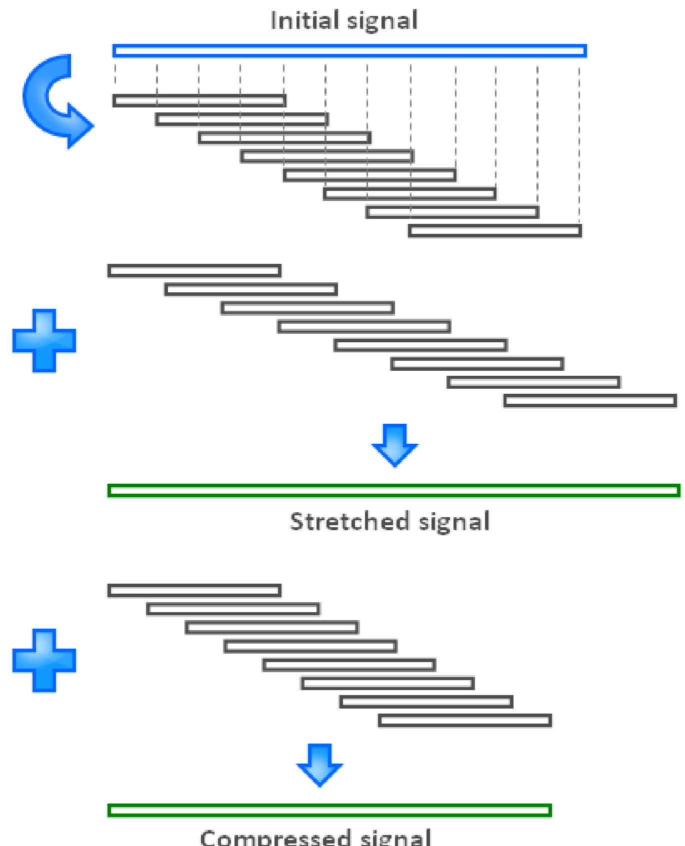
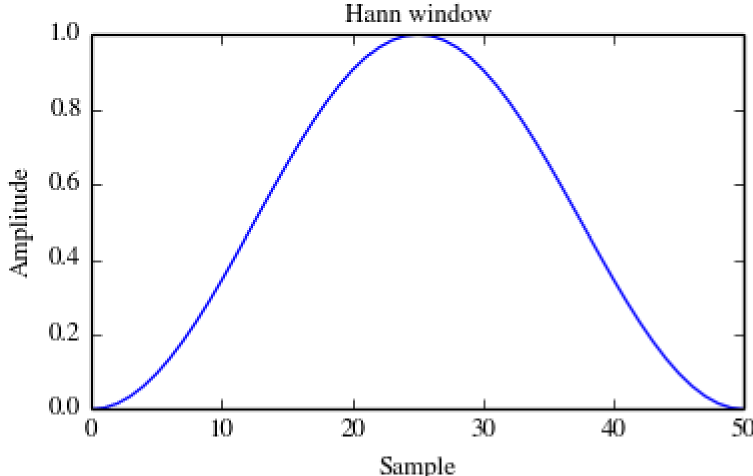
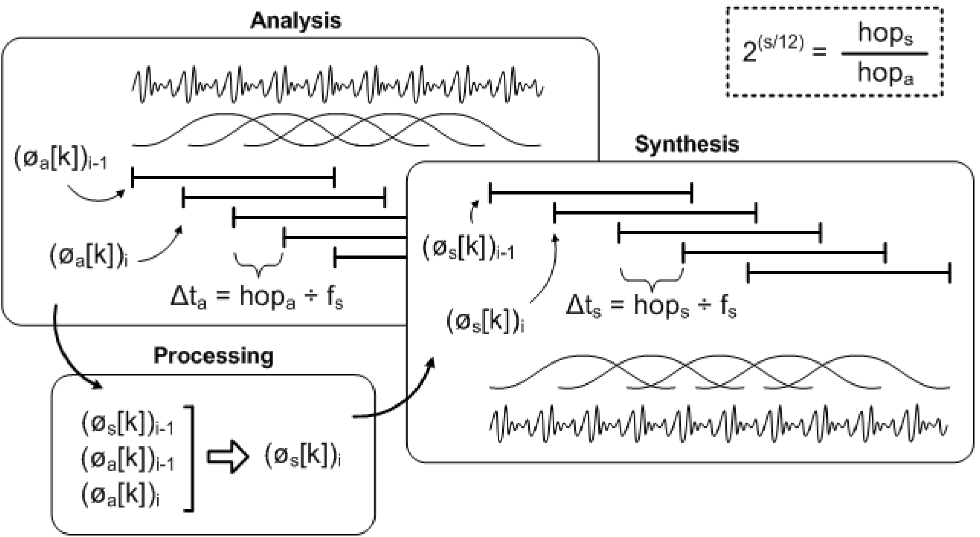


Illustration via *http://www.guitarpitchshifter.com/algorithm.html*

c. Element-wise multiplication of the *windows* by *Hanning windows* of the same size. The aim is to have the overlapping windows *fade* into each other, in order to obtain a smooth signal.



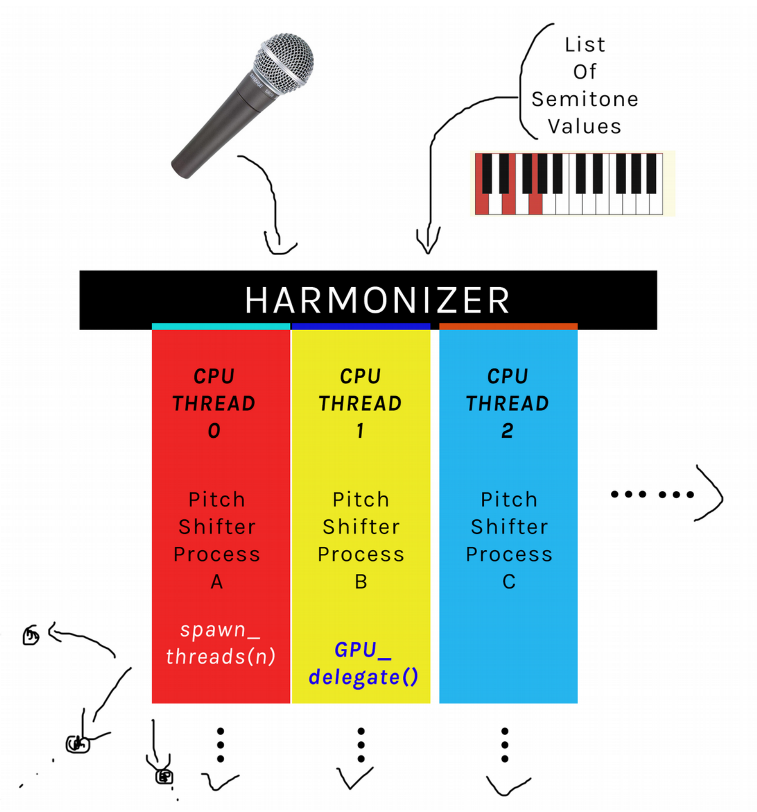
d. Fix the signal discontinuities obtained by rearranging the chunk windows (which result in audio glitches) by rephasing them. This phase is computationally the heaviest, as it requires an *FFT* ("*The Fourier transform decomposes a function of tim -a signal into its constituent frequencies*." - Wikipedia), from which we can rephrase the nonaligned *FFT* sinuses according to each frequency bin, and then proceeding with an *Inverse FFT* to reconstitute a signal - Signal B - from the rephased sinus waves that represent the sound frequencies the original signal contains. This phase can be parallelized by doing the matrix multiplications of the *FFT* and *Inverse FFT* on the GPU, as it specializes on such operations.



via http://www.guitarpitchshifter.com/algorithm.html

4. A 1-Dimensional linear interpolation of the signal B, which can be parallelized by splitting the signal chunk further into smaller subparts and interpolating them. The interpolation process is called resampling, and aims at recovering the initial signal's rate after the stretching operations, resulting thus in a signal with the same length but of a different pitch.

A *Harmonizer* would distribute the *Pitch Shifting* process described above along different threads, with each subprocess using a different semitone *S* value (read: to obtain a different musical pitch), then *mux ("select between several digital input signals and forwards it to a single output line"* - Wikipedia) the resulting signals into one final signal output - the harmonized sound.



The distributed Harmonizer in theory

VII. Requirements Specification

Functional Requirements

- The application should

Non-Functional Requirements

- Distribution Requirements

    The software is written with distribution of resources as a primary goal, and should be capable of parallel processing through the use of both network nodes and local ressources.

- Security Requirements

    The software is an experiment in processing speed optimization, and is thus not concerned with security at this stage.

- Confidentiality Requirements

    Same as above.

- Integrity Requirements

    Same as above.

- Availability Requirements

    The software should make use of all possible available resources while processing data.

- Access Requirements

    The software should be accessible whenever requested with sent data.

- Extensibility Requirements

    The software is to provide the core functions necessary to build future distributed audio processing tools.

- Performance Requirements

    The software should aim for real time processing whenever possible, and, if not, process and return the processed data with the least amount of delay possible.

- Scalability Requirements

  The software is built with scalability as the main aim, and should thus distribute tasks amongst computer clusters and nodes whenever gain in processing time is to be had.

IIX. Technologies of Choice, Part 1/2

Name: Python

Version: 3.6 / x64

Documentation Link: https://docs.python.org/3/

Project Role: The core of the software and all of the processing was written using Python.

Reasons of Choice: Speed of prototyping, availability of libraries and books, very strong community.

Name: Numpy (Python module)

Version: 1.16

Documentation Link: https://www.numpy.org/devdocs/

Project Role: List manipulation, data conversion, matrices & math operations.

Reason of choice: The primary choice for most projects written in python which involve complex list operations. Renown for its clarity and speed.

Name: Scipy (Python module)

Version: 1.2.1

Documentation Link: https://docs.scipy.org/doc/scipy/reference/

Project Role: Importing, decoding and writing audio, interpolation functions.

Reason of choice: Same as Numpy. The two modules are typically used in combination with each other, as they are part of the same ecosystem.

Name: NVIDIA Cuda Toolkit

Version: 9.1

Documentation Link: https://docs.nvidia.com/cuda/

Project Role: Enabling GPU operations through the GPU SDK so as, to speed up matrix operations which are numerous in audio processing.

Reason of choice: The use of NVIDIA GPU units requires this SDK.

Name: CuPy(Python module)

Version: 5.4

Documentation Link: https://docs-cupy.chainer.org/en/stable/

Project Role: Augments the Numpy library, used as a layer between the software and the GPU, used in order to do FFT transforms on the GPU, which, in theory, would provide faster results (depending on the computer's hardware). Also used to transfer data between the CPU and the GPU when needed..

Reason of choice: Preferred choice after using alternatives. The library serves the exact purpose I was looking for, and thus removed the need to port Numpy functions at a lower

level to parallelize the operations.

Name: Parallel Python / UQ Foundation Fork for Python 3 (Python module)

Version: 1.6.4.9

Documentation Link: https://github.com/uqfoundation/ppft

Project Role: Distribute function execution amongst CPU worker threads, in both the local system and by using network nodes.

Reason of choice: Preferred choice after using alternatives, namely MPI. Low overhead, concise syntax, network capabilities.

Name: SocketIO (Python module, ported from JS)

Version: 4.0.0

Documentation Link: https://python-socketio.readthedocs.io/en/latest/

Project Role: Audio data chunks back and forth transfers between client and master cluster node.

Reason of choice: Familiarity with the module, as I have used it exensively with NodeJS.

IX. Architecture & Implementation, Part ½

The implementation consists of three main parts:

1. Core logic and processing
2. Parallel task distribution and collection, in a master-slave system
3. Server-like logic to receive and return audio signal through websockets

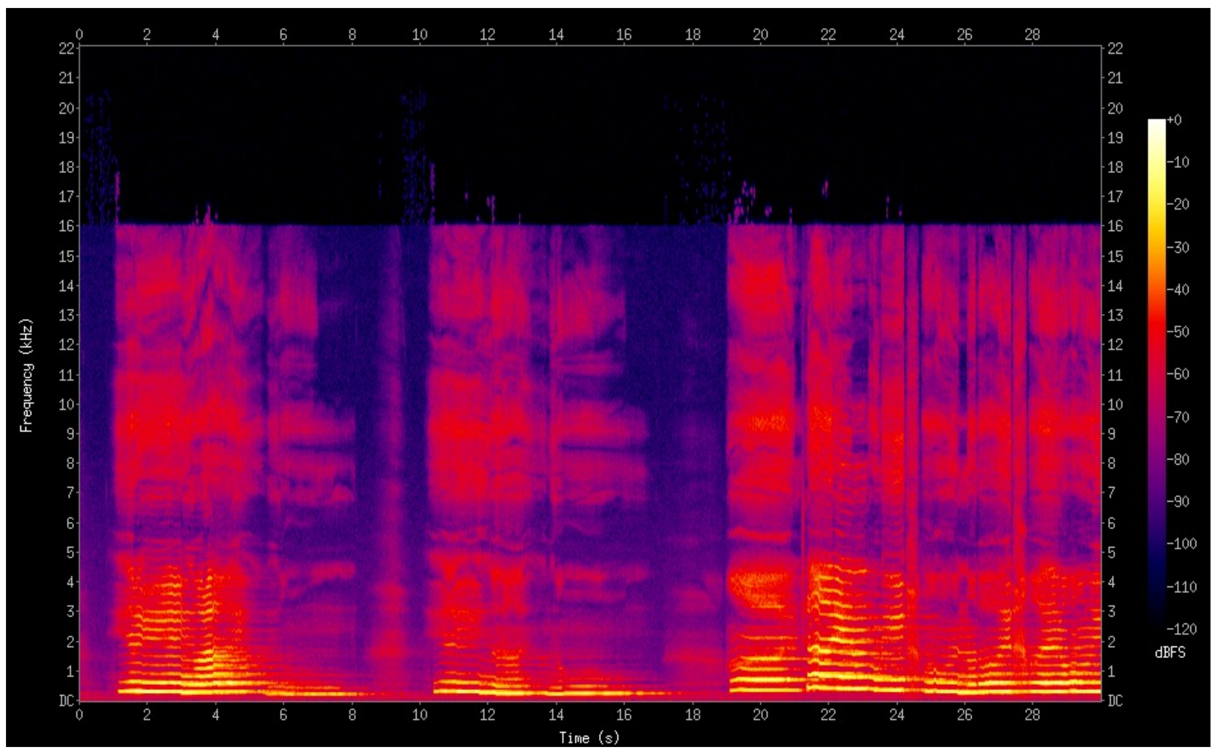
Concerning the core logic, it can be broken down series of, at first, sequential function, which are then parallelized in the next phase. These functions can be illustrated by plotting the signal (power and spectrogram) as a signal goes through them. For this purpose, I am importing an audio file obtained from a recording then running it through the functions to demonstrate their work. As follows:

```
95  #import and parse soundwave file using library
96  rate,data = scipy.io.wavfile.read('recording.wav')
97  #convert from stereo to mono, to process one channel at a time
98  data = data[:, 0]
99  lower = pitchshift_0(data,-5)
100 wave.write(f'outputs/lower.wav',rate,low)
101
```

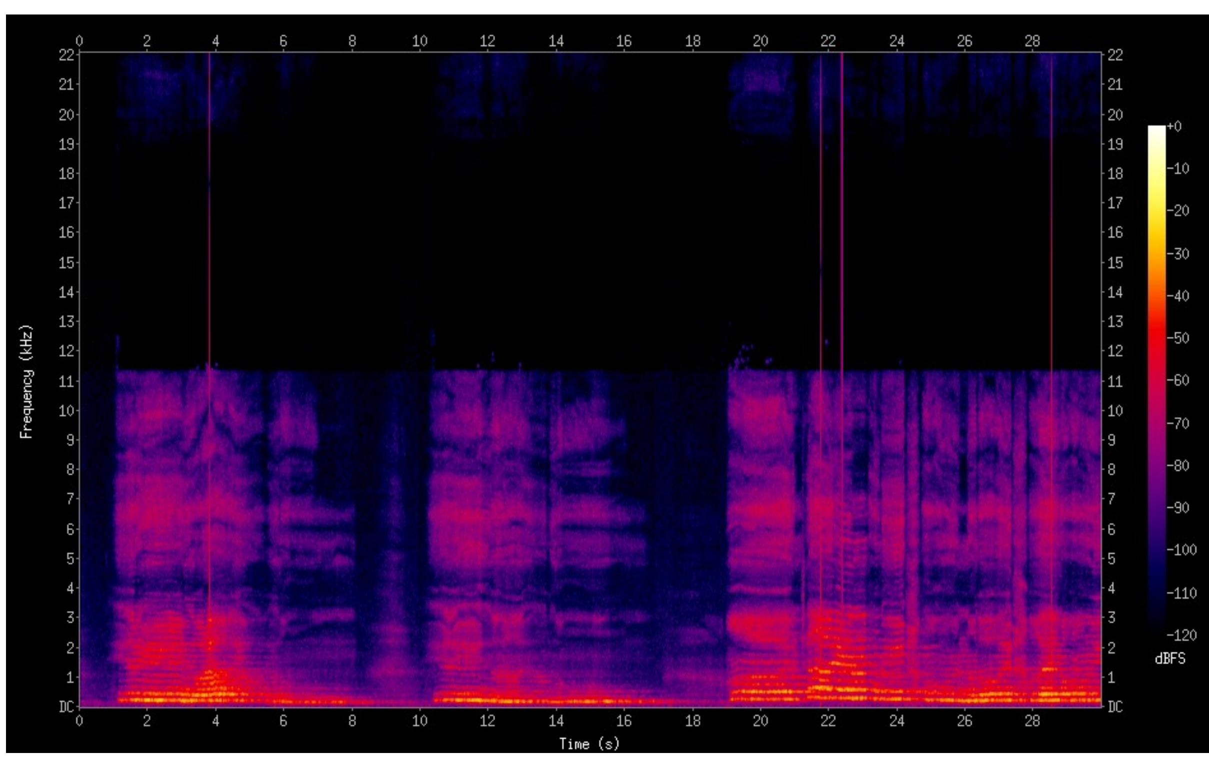
The audio then goes through the following function, which I annotated using comments.

```python
28 def pitchshift_0(audio, s, winsize=DEFAULT_WINSIZE, h=DEFAULT_H):
29     #determine the stretching factors from the semitones value
30     factor=2**(1.0*s/12.0)  ;    stf = 1.0/factor
31     #   step 1 / stretch
32     #declare empty array, used to rephase
33     pw = np.zeros(winsize)
34     #declare hanning window of same size
35     hanningwin = np.hanning(winsize)
36     #declare array to store result
37     stretched_signal = np.zeros(  int(len(audio)/stf + winsize))
38     #loop through overlapping chunk windows
39     for i in np.arange(0,len(audio)-(winsize+h),h*stf  ):
40         i = int(i)
41         #FFT analysis of a window
42         #using the variable 'h' to determine the dephasing value
43         a1 = audio[i:i+winsize] :   a2 = audio[i+h:i+winsize+h]
44         s1 = np.fft.fft(hanningwin*a1)  ;   s2 = np.fft.fft(hanningwin*a2)
45         #calculating the phase offset within this frequency bin
46         #then rephasing in the same variable (pw == 'phasing window')
47         pw = (pw + np.angle(s2/s1)) % 2*np.pi
48         #reconstructing a signal (B) from the rephased FFT signals
49         a2_repwd = np.fft.ifft(np.abs(s2)*np.exp(1j*pw))
50         #result indexing
51         i2 = int(i/stf)
52         #append to result
53         stretched_signal[i2:i2+winsize] += hanningwin*a2_repwd.real
54     #   step 2 / 1D interpolation
55     #generating the interpolation function
56     #i use cubic interpolation because it sounds better after i tried all the alternatives
57     n = np.ceil(stretched_signal.size / factor)
58     f = scipy.interpolate.interp1d(np.linspace(0, 1, stretched_signal.size), stretched_signal, 'cubic')
59     #return signal, interpolated and in the proper format
60     return f(np.linspace(0, 1, n)).astype('int16')
61
```

Now, let’s plot the spectrogram of the before and after signals, to show that indeed the pitch has been shifted and the duration remained the same.



The original signal



The exported signal after being shifted down by 6 semitones, hence the concentration around lower frequencies

Concerning the parallelization of tasks, there were 4 main contenders in order to

distribute and synchronize CPU threads (which would then spawn GPU threads). I tried each one in order to select the best choice, after testing for distributed excution timing as well.

Module Name	**Used?**	**Why?**
MPI	No	Fast execution, but in this case, threads spawning and synchronizing needs to be managed dynamically from the master node, something in which it performed poorly as it required too complex work for this purpose.
Redis	No	Data overhead to spawn threads across network nodes results in latency.
ParallelPython	No	Not available for Python 3
PPFT	Yes	Is a port of ParallelPython with just the exact amout of complexity needed for the task (set up number of worker nodes, set up server that is connected to those nodes, distribute function with 4 arguments, oneline synchronization, clear syntax)

.

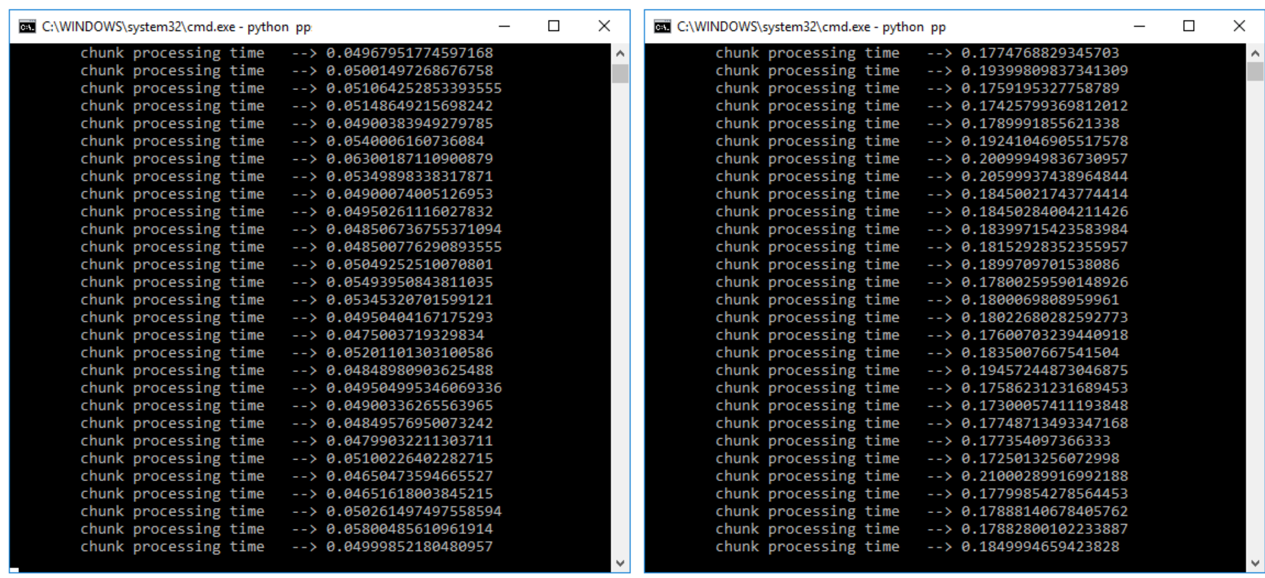
Thus, for horizontal CPU distribution, I used PPFT. There were many possible configurations for horizontal CPU distribution. For example:

1. Should it only distribute global pitch shift threads, with one thread spawned for every *Harmonize* musical note?
2. Should it distribute 1D array interpolation by splitting arrays?

The way to determine answers was to test for timing. Here are screenshots of these unit tests, by using the following parameters:

- Audio format rate: 44100 (signal points per second)

- Chunk size : 44100/10 = 4100
- *Harmonizer* concurrent pitch shifting processes : 3



single-process 1D linerp　　　　multi-process (4 threads) 1D linerp

Test timings show that in some cases, multiprocessing is not the best recourse, a single process with less overhead scored better times than 4 threads in unison for linear interpolation (linerp)

Thus, through trial and error, the main function architecture of the distributed functions was finalized as follows:

```
121  def pitchshift_w_GPU(data,n,WINSIZE=WNSDEF,h=HDEF):
122      if n == 0: return data.astype('int16')
123      f = 2**(1.0 * n / 12.0)
124      stretch_f = 1.0/f
125      pw = cp.zeros(WINSIZE,dtype=data.dtype)
126      hanning_window = cp.hanning(WINSIZE)
127      result = cp.zeros(  int(len(data)/stretch_f + WINSIZE))
128      step_size = h*stretch_f
129      end_loop = len(data)-(WINSIZE+h)
130      for i in np.arange( 0,  end_loop, step_size):
131          i = int(i)
132          a1 = cp.array(data[i:i+WINSIZE]);a2 = cp.array(data[i+h:i+WINSIZE+h])
133          s1 = cp.fft.fft(hanning_window*a1);s2 = cp.fft.fft(hanning_window*a2)
134          pw = (pw + cp.angle(s2/s1)) % 2*cp.pi
135          a2_repwd = cp.fft.ifft(np.abs(s2)*cp.exp(1j*pw))
136          i2 = int(i/stretch_f)
137          result[i2:i2+WINSIZE] += hanning_window*a2_repwd.real
138      result_cpu = cp.asnumpy(result)
139      n = np.ceil(result_cpu.size / f)
140      f = scipy.interpolate.interp1d(np.linspace(0, 1, result_cpu.size), result_cpu, 'cubic')
141      return f(np.linspace(0, 1, n)).astype('int16')
142
143  def distributed_harmony(data,list_of_semitones):
144      jobs = []
145      for n in list_of_semitones:
146          jobs.append(job_server.submit(
147              pitchshift_w_GPU,
148              (data,n),
149              ))
150      return [job() for job in jobs]
```

Note: The GPU functions required back and forth data transfers between the CPU and the GPU, hence the 'cloning' of data at certain points.

Concerning the server-capabilities part, the data is sent back and forth using the *SocketIO For Python* module. This will enable us to host it the cloud if we want to, by *dockerizing* a single python script per cluster node. The client application would simply require connecting to the *SocketIO* server and sending audio data in chunks. This part is implemented as follows:

```
179  from aiohttp import web
180  import socketio
181  sio = socketio.AsyncServer()
182  app = web.Application()
183  sio.attach(app)
184  @sio.on('connect', namespace='/')
185  def connect(sid, environ):
186      print("connect ", sid)
187  @sio.on('audio', namespace='/')
188  async def message(sid, data):
189      #<------------------- process and reply with result---------------->
190      await sio.emit(distributed_harmony(data.audio,data.semitones_list))
191  @sio.on('disconnect', namespace='/')
192  def disconnect(sid):
193      print('disconnect ', sid)
194  if __name__ == '__main__':
195      web.run_app(app)
196
```

A final note concerning the core implementation: In order to speed up threads spawned by *PPFT*, i injected code which imports all the libraries used by the software core in the worker node instantiation code, so as to avoid extra overhead.

```
32  http://www.parallelpython.com - updates, documentation, examples and support
33  forums
34  """
35  from __future__ import with_statement
36
37  import atexit
38  import logging
39  import errno
40  import getopt
41  import sys
42  import socket
43  import threading
44  import random
45  import string
46  import signal
47  import time
48  import os
49  import six
50  #<---------- injected-----
51  import numpy as np
52  import cupy as cp
53  import scipy.interpolate
54  #--------------------------->
55  import pp
56  import pp.auto as ppauto
```

The *PPFT Module* files in which i injected importation code are:

- *{python36}/lib/site-packages/ppft/__main__.py*
- *{python36}/lib/site-packages/ppft/server/__main__.py*

X. Project Evolution: A Creative Test Case

After drafting the *Harmonizer* emulator, I wanted to take the project further by demonstrating new ways to utilize the emulated gear. The demonstration would be an opportunity to field test the parallelized aspects of the emulated hardware by testing it on real workloads, as would be the case in a music studio setting.

I designed an experiment the aim of which is to automatically generate semitone lists, which represent music chords, that are send along with voice signals to the *Harmonizer*.

We can consider this as a creative test unit, to test the parallelization distribution of the system. It is called the ***N0.***

The *N0 Distributed Approach To System Emulation Test Unit* is inspired by musician *Bon Iver* and his engineer *Chris Messina*, as the two made use of creative *Harmonizers* in the past few years.

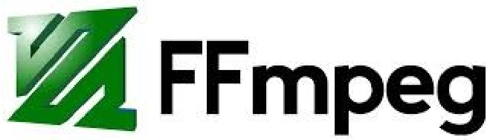
*Music artist **Bon Iver** using a vocal hamonizer along with a MIDI keyboard to send pitch shifting values.*

The concept of the *N0 Distributed Approach To System Emulation Test Unit* is the following:

1. Gather a random set of music tracks from various sources.
2. Split the tracks into ~20 seconds samples.
3. Analyze the samples using internal software I developed in order to extract musical notation, from which are determined notes then semitones to be used as input for the *Harmonizer* later on.
4. Make a data collection effort in the form of a web page, in which a user listens to a music track and is asked to evaluated the music track using 5 different *emotional* criteria: {Happy, Sad, Angry, Excited, Bored} and 3 possible scales {Low, Middle, High}
5. Generate a *Markov model* after interjoining the user-generated data and the musical notation data.
6. Automatically generate musical chord progressions based on the selected *emotional* values, which are then sent to the *Harmonizer* in the form of semitone values.

The technologies used to develop the *N0 Distributed Approach To System Emulation Test Unit* are detailed in the following section.

XI. Technologies of Choice, Part 2/2



Name: FFMPEG

Version: 4.1

Documentation Link: https://ffmpeg.org/documentation.html

Project Role: Audio format conversions and slicing for the data collection phase.

Reason of choice: One of my favorite tools, the best in its category. Clear and concise CLI syntax, ability to process all audiovisual formats.

Name: Falcon Analysis (Internal)

Version: 0.0.1

Documentation Link: Available on request.

Project Role: Music features extraction from sound, used to generate MIDI data through FFT-based analysis.

Reason of choice: Part of a software toolkit i developed over the past 2 years for audio processing and generation.

Name: NodeJs

Version: 9

Documentation Link: https://nodejs.org/en/docs/

Project Role: Server core during the data collection phase.

Reason of choice: Best experience so far in server creation, very strong community, extensive choice of libraries.

Name: VueJs (+HTML5+CSS3+JS)

Version: 2

Documentation Link: https://vuejs.org/v2/guide/

Project Role: Frontend development during the data collection phase.

Reason of choice: Personal favorite in comparison to every framework i tried in the past few years.

Name: SQLite (NodeJs module)

Version: 3

Documentation Link: https://www.npmjs.com/package/sqlite3

Project Role: Database management during the data collection phase.

Reason of choice: Fast and very light.

Name: Express (NodeJs module)

Version: 4

Documentation Link: https://expressjs.com/en/4x/api.html

Project Role: Manage HTTP queries during the data collection phase.

Reason of choice: Fast deployment.

XII. Architecture & Implementation, Part 2/2

The implementation of steps 1 & 2 (sample random music tracks) is the following. It calls ffmpeg commands from python:

```python
import os, subprocess, random
def proc(f):
    n_slices = 4
    dur = subprocess.check_output('cd tracks && ffprobe -v error -show_entries format=duration -of default=noprint_wrappers=1:nokey=1 "'+f+'"', shell=True)
    pdur = float(dur.decode('utf-8'))/n_slices
    for n in range(n_slices):
        os.system('cd tracks && ffmpeg -y -ss '+str(n*pdur)+' -t '+str(pdur)+' -i "'+f+'" -ab 320k "outputs\\'+f+'-p'+str(n)+'.mp3"')
    os.system('cd tracks && del "'+f+'"')
files = []
for f in os.listdir('./tracks'): files.append(f)
while len(files)>0:
    try:
        files = []
        for f in os.listdir('./tracks'): files.append(f)
        f = random.choice(files)
        if f!='outputs':
            print(f)
            proc(f)
    except:
        print('error')
        continue
```

The batch processing code for the two steps detailed above

Then, I analyze the samples using the internal software I developed in order to extract musical notation of notes contained in the analyzed sample. The task is repeated for each tracks obtained from the previous step and stored in tables.

	A	B
1	0	47
2	0	71
3	0	63
4	1	66
5	1	48
6	2	48
7	2	71
8	2	72
9	2	64
10	2	67
11	3	45
12	3	69
13	3	73
14	3	64

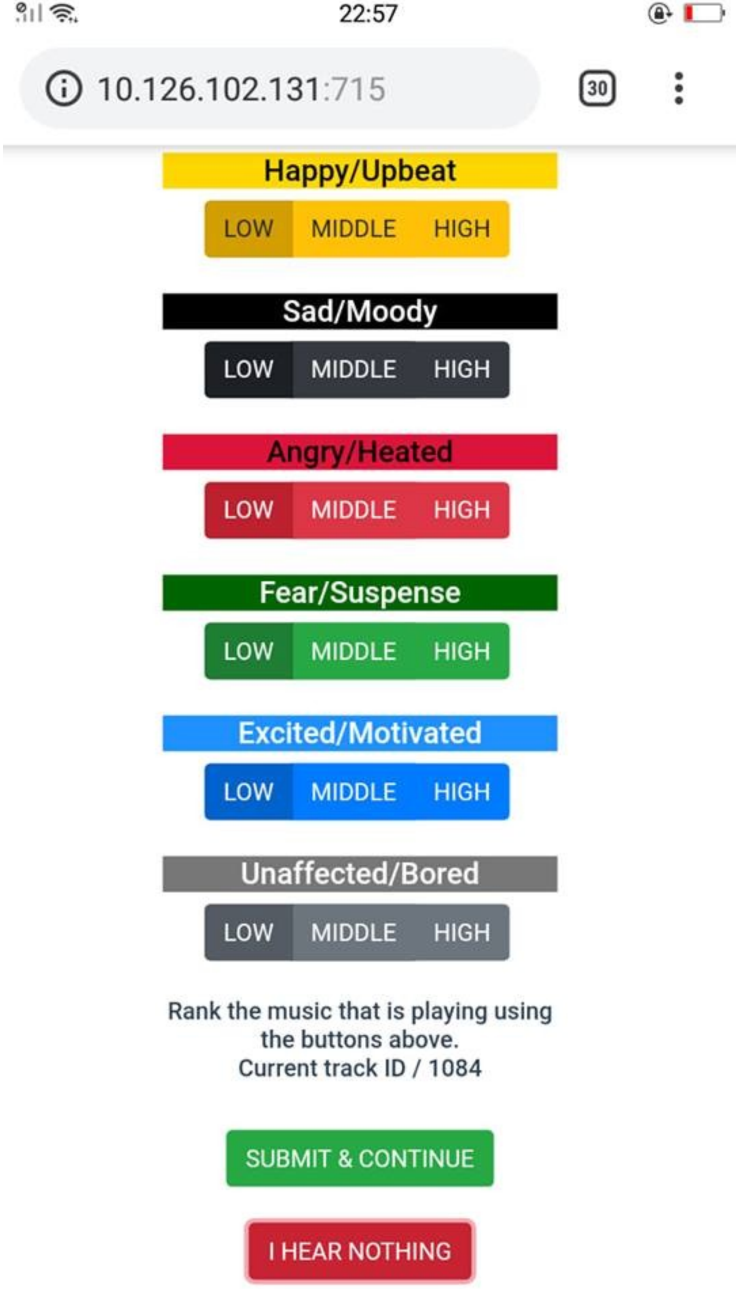
02 Svatba.fla

An example output, where each music note (from which Pitch Shifter semitones value are derived) in column B corresponds to a chord note. The column A contains the indices of the chords made from the notes in B.

Then, for the data collection effort, the server side simply consists of storing the values submitted by the user through an HTTP request into a sqlite database file, then sending a new mp3 file for the user to evaluate..

```
59      server.route({
60          method: 'POST',
61          path: '/',
62          handler: function (request, h) {
63              const payload = request.payload
64              //console.dir(payload,{depth:true,colors:true})
65              return new Promise((resolve, reject)=>{
66                  db.run(`INSERT INTO scores(scoreid,id,happy,sad,angry,excited,fear,bored) VALUES(NULL,${payload.id},
67                              ${payload.happy},
68                              ${payload.sad},
69                              ${payload.angry},
70                              ${payload.excited},
71                              ${payload.fear},
72                              ${payload.bored})`, (err) => {
73                      if (err) return console.log(err.message)
74                      else {
75                          console.dir(payload,{depth:true,colors:true})
76                          bodycount++
77                          console.log(`\t\t\tcount: ${count} / ${new Date()}`)
78                          resolve(true)
79                      }
80                  })
81              })
82          }
83      })
84
85      await server.start()
86      console.log('Server running on %s', server.info.uri)
87  }
```

As for the front end, I used Vue to make a simple interface that sends POST and GET http queries to the server, then plays music whenever it is received. The end result is the following:



The visual interface of the data collection effort, as seen from my phone

I hosted everything on my laptop, let it run for 2 consecutive days while connected to the AUI wifi network, and asked other students in the students Facebook group to contribute with their evaluations. It registered 150 entries at the end.

Table : audio

	id	filename
	Filtre	Filtre
657	657	10-Areia De Salamansa.mp3-p0.mp3
658	658	10-Areia De Salamansa.mp3-p1.mp3
659	659	10-Areia De Salamansa.mp3-p2.mp3
660	660	10-Areia De Salamansa.mp3-p3.mp3
661	661	10. More Blues.mp3-p0.mp3
662	662	10. More Blues.mp3-p1.mp3
663	663	10. More Blues.mp3-p2.mp3
664	664	10. More Blues.mp3-p3.mp3
665	665	10. Rios.mp3-p0.mp3
666	666	10. Rios.mp3-p1.mp3
667	667	10. Rios.mp3-p2.mp3
668	668	10. Rios.mp3-p3.mp3
669	669	10.No Shade In The Shadow Of The Cross.flac-p0.mp3
670	670	10.No Shade In The Shadow Of The Cross.flac-p1.mp3
671	671	10.No Shade In The Shadow Of The Cross.flac-p2.mp3

The file referencing table

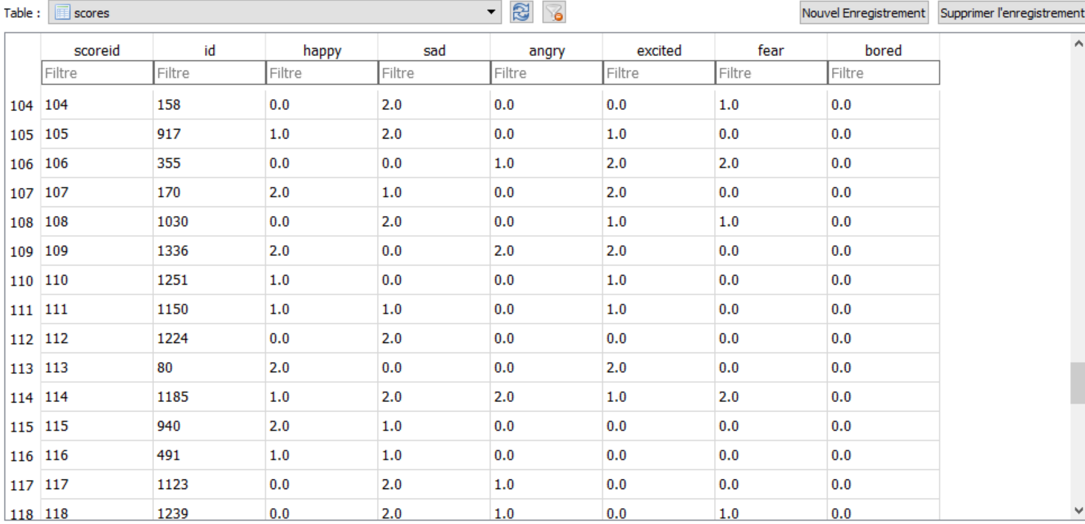


Table : scores

Nouvel Enregistrement | Supprimer l'enregistrement

	scoreid	id	happy	sad	angry	excited	fear	bored
	Filtre	Filtre	Filtre	Filtre	Filtre	Filtre	Filtre	Filtre
104	104	158	0.0	2.0	0.0	0.0	1.0	0.0
105	105	917	1.0	2.0	0.0	1.0	0.0	0.0
106	106	355	0.0	0.0	1.0	2.0	2.0	0.0
107	107	170	2.0	1.0	0.0	2.0	0.0	0.0
108	108	1030	0.0	2.0	0.0	1.0	1.0	0.0
109	109	1336	2.0	0.0	2.0	2.0	0.0	0.0
110	110	1251	1.0	0.0	0.0	1.0	0.0	0.0
111	111	1150	1.0	1.0	0.0	1.0	0.0	0.0
112	112	1224	0.0	2.0	0.0	0.0	0.0	0.0
113	113	80	2.0	0.0	0.0	2.0	0.0	0.0
114	114	1185	1.0	2.0	2.0	1.0	2.0	0.0
115	115	940	2.0	1.0	0.0	0.0	0.0	0.0
116	116	491	1.0	1.0	0.0	0.0	0.0	0.0
117	117	1123	0.0	2.0	1.0	0.0	0.0	0.0
118	118	1239	0.0	2.0	1.0	0.0	1.0	0.0

The user-submitted score evaluation table

Using both data, I built a simple *Markov* model which sends an automatically generated list of semitone values to the *Harmonizer's socketIO* server along with audio to harmonize it. The *socketIO* server can be queried from both pyhon or a web client.

XIII. Future of the Project & Conclusions

The project can be extended into a distributed framework for the emulation of audio hardware. The current standard framework, *VST Technology*, developed and maintained by the *Steinberg* company, consists on local processing which is fully dependent on local computer power. The alternative I suggest is to delegate heavy audio processing to cloud services nodes in order to remediate far too frequent latency problems. With high-speed connections (especially *Fiber Optic Internet*) and GPU-powered cloud node clusters, it is highly imaginable to have audio processing as a service. What is now audio plugin cost would be replaced by a pay-for-consumption cloud model, or membership fees.

The performances of the distributed emulation can be improved by:

1. Rewriting the prototype in C/C++, or Cython.
2. Further mathematical optimizations (matrix operations) for faster results.
3. Optimize performance by lessenning data conversions of signals.
4. Caching results, especially in GPUs.

And this concludes my project.