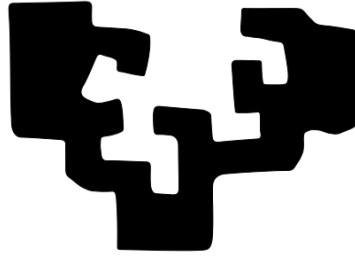


eman ta zabal zazu



Universidad
del País Vasco

Euskal Herriko
Unibertsitatea

▪ Degree's Final Project ▪

Computer Engineering

An Experiment on Melody Training
for Amateur Musical Instrument
Players

Author: Ainhize Goenaga

Director: Manuel Graña





Acknowledgements

I would like to thank my director, Manuel Graña, for lending me a hand whenever I had problems, for teaching me the steps and a proper way to develop a project with such dimensions and for keeping me motivated all the time.

I would also like to thank Leire Ozaeta, for helping me with everything she could, mainly the NAO robot part of my project (even if, in the end I had to leave the robot apart).

Another important person I would like to thank is Borja Fernández, who introduced me to Manuel Graña and for his help whenever my director wasn't available.

Last, but not least, I would like to thank my family and my friends, for supporting me and for believing in me.





Abstract

The long term goal of the research started in this project is to develop systems able to assist people training to sing or hum melodies, or play them on musical instruments. The current project will focus on single instrument training for short monophonic melodies.

The interaction between the user and a fully developed application would go as follows: The user selects a melody from the database and the application reproduces it. The human plays the melody with an instrument while the application records it, so, when finished; it processes the recording to give its approval or sings again the melody if it hasn't been well reproduced.

Materials: computational experiments are carried out on the recordings of a well known melody played on two instruments (piano, flute), performed by a custom laptop computer microphone, in order to test robustness against rich harmonics interferences when trying to extract the melody.

Methods: Programming has been carried out in Python, using available open source resources: NumPy, SciPy, Matplotlib, PyAudio. The computational process steps are the following: signal windowing, Fourier transform, transform spectre downsampling, fundamental melody frequency extraction, pattern comparison by correlation.



Resumen

El objetivo a largo plazo de este proyecto de investigación es crear sistemas capaces de entrenar personas para cantar o tocar melodías en instrumentos musicales. El proyecto actual se centrará en el entrenamiento de un único instrumento a la vez con melodías monofónicas cortas.

La interacción entre el usuario y la aplicación desarrollada por completo será la siguiente: El usuario elige una melodía de la base de datos, y la aplicación la reproduce. El usuario toca la melodía con un instrumento musical mientras que la aplicación lo graba. Cuando el usuario termina de reproducir la melodía, la aplicación procesará lo grabado para dar su aprobación o volverá a cantar la melodía si ha habido fallos.

Materiales: trabajaremos con una melodía conocida que será grabada con el micrófono de un ordenador portátil. Se probará con 2 instrumentos musicales diferentes (piano, flauta) para probar la solidez del algoritmo cuando aparecen armónicos indeseados en el momento de procesar la melodía.

Métodos: La programación de la aplicación se hará en Python, y se utilizarán recursos de código abierto tales como: NumPy, SciPy, Matplotlib, PyAudio. Los pasos del algoritmo son los siguientes: inventanado de la señal, transformada de Fourier, diezmado del espectro, extracción de la frecuencia fundamental de la melodía, comparación de patrones utilizando la correlación.



Laburpena

Ikerketa honen luzerako helburua pertsonak kantatu eta musika instrumentuak jotzen trebatzen laguntzeko sistemak garatzea da. Proiektu hau instrumentu bakarreko melodia monofonikotan zentratuko da.

Zeharo garatutako produktu eta erabiltzailearen arteko elkarrekintza ondorengo izango da: Erabiltzaileak melodia bat hautatuko du datu-basetik, eta aplikazioak erreproduzitu egingo du. Erabiltzaileak entzundakoa instrumentu batekin joko du aplikazioa grabatzen dagoen bitartean. Erabiltzaileak bukatzean, aplikazioak grabatutakoa prozesatuko du emaitzaren onespena emango du edo, gaizki eginez gero, berriro kantatuko du.

Materialak: Ordenagailu eramangarri bateko mikrofonotik grabatutako musika instrumentu batekin jotako melodia ezagun batekin arituko gara lanean, melodia bi instrumentu desberdinekin joko da (pianoa, txirula) sendotasuna frogatzeko, melodia prozesaketan ager daitezkeen harmoniko interferentziak direla eta.

Metodoak: Aplikazioaren programazioa Python lengoaiari egingo da, eskura dauden kode irekiko baliabideak erabiliz: NumPy, SciPy, Matplotlib, PyAudio. Prozesuaren pausoak ondorengoak dira: seinalearen leihoketa, Fourier-en transformatua, transformatutako espektroaren hamarrena hartu (downsampling), melodiatik oinarritzko frekuentzia lortu, patroien konparaketa korrelazioaren bitartez.





Index

Introduction.....	1
1. Motivation.....	1
2. Objectives	1
3. Structure of the Memory.....	1
Project Management.....	3
1. Work Breakdown Structure	3
2. Time Estimation.....	3
3. Gantt Chart.....	4
4. Risk Management	6
5. Monitoring and Control	6
5.1 Communication.....	6
5.2 Deviations	6
Introduction to Pitch Estimation.....	7
1. Definitions.....	7
1.1 Pitch	7
1.2 Musical Tone	7
1.3 Octave	7
1.4 Melody	8
1.5 Polyphony	8
1.6 Harmonics	9
1.7 MIR (Music Information Retrieval)	9
2. Mathematics.....	10
2.1 Fourier Transform.....	10
2.2 Downsample / Decimation	11
Pitch Estimation Algorithms.....	12
1. Frequency Domain Algorithms	12
1.1 HPS (Harmonic Product Spectrum)	12
1.2 Cepstrum Analysis.....	13
1.3 Parabolic Interpolation.....	14
2. Time Domain Algorithms	14
2.1 ZCR (Zero-crossing Rate)	14
2.2 Autocorrelation	14



3. Frequency Domain vs. Time Domain.....	15
Work Environment.....	17
1. Hardware.....	17
2. Software.....	17
2.1 NumPy vs. SciPy.....	18
3. Decisions.....	18
3.1 Python.....	18
3.2 Speech Recognition.....	19
3.3 Monophonic melody.....	19
3.4 Harmonic Product Spectrum (HPS).....	19
3.5 Hanning Window.....	19
Design.....	21
1. Interaction.....	21
2 Sound Process.....	22
3. Comparison.....	23
Implementation.....	24
1. Sound Processing.....	24
1.1 Read_File.....	24
1.2 HPS_Algorithm.....	25
1.3 Create_File.....	26
1.4 Correlation.....	27
2. Interaction.....	28
2.1 Song_Choice.....	28
2.2 Sing_Song.....	28
2.3 Listen_Song.....	29
2.4 Compare_Files.....	30
3. Synthesized Song.....	30
3.1 note.....	30
3.2 main.....	30
4. Tuner.....	31
4.1 Record.....	31
4.2 Convert_To_Note.....	32
Testing.....	33



1. Application.....	33
1.1 Song	33
1.2 Flute	33
1.3 Piano	36
2. Tuner.....	39
2.1 Flute	39
2.2 Piano	43
2.3 Conclusions.....	45
Improvements.....	47
1. Improve Application	47
1.1 PDA Precision	47
1.2 Feedback System	47
1.3 Interface	48
2. Possible Developments	48
2.1 Implement into NAO	48
2.2 Polyphonic Sound.....	48
2.3 Songs with Lyrics	48
Conclusions.....	50
1. Project's Conclusions	50
2. Personal Conclusions.....	50
2.1 Personal Experiences	50
2.2 Learnt Lessons	51
NAO's Involvement	52
1. Introduction.....	52
1.1 What is Nao?.....	52
1.2 Some properties	52
2. Software for Developers	53
2.1 Choregraphe.....	53
2.2 SDK	55
2.3 NAOqi.....	56
3. Project's Development.....	56
3.1 Starting.....	56
3.2 Design	57



3.3 First Version - Choregraphe	58
3.4 Second Version - Remote coding	58
4. Problems	59
4.1 Nao Singing	59
4.2 Library Installation	59
4.3 File Transfer.....	59
5. Written Code.....	60
5.1 main.py	60
5.2 MyClass.py	60
Full Code.....	63
1. Sound Processing	63
1.1 soundProcessing.py	63
1.2. MyClass.py	66
1.3 main.py	69
2. Tuner	69
2.1 findNote.py	69
2.2 tuner.py	71
3. Synthesize Sound.....	72
3.1 twinkle_S.py	72
References.....	73
1. Websites	73
2. Books & Documents	73



Image Index

Figure 2.1: Work Breakdown Structure	3
Figure 2.2: Gantt Chart	5
Figure 3.1: Do Scale	8
Figure 3.2: Equivalence of Music Sheet and Tablature	8
Figure 3.3: Harmonics of the Fundamental Frequency	9
Figure 3.4: MIR research areas	10
Figure 3.5: Decimation	11
Figure 4.1: Harmonic Product Spectrum Algorithm.....	12
Figure 4.2: Cepstrum Algorithm.....	13
Figure 4.3: Difference between Time domain and Frequency domain	15
Figure 5.1: Hanning Window application's graphical view	20
Figure 6.1: Interaction diagram between User and Application	21
Figure 6.2: Sound Processing block diagram	22
Figure 7.1: Read_File function's block diagram	24
Figure 7.2: HPS_Algorithm function's block diagram	25
Figure 7.3: Create_File function's block diagram.....	26
Figure 7.4: Correlation function's block diagram	27
Figure 8.1: Twinkle, twinkle, little star song's music sheet.....	33
Figure 8.2: Signal data obtained from performing the song with the flute.....	34
Figure 8.3: Signal data's zoom in the first note.....	34
Figure 8.4: Fourier Transform of the signal data from the flute.....	35
Figure 8.5: Comparison between synthesized audio's fundamental frequency sequence and recorded audio's fundamental frequency sequence (FLUTE)	35
Figure 8.6: Comparison between synthesized audio's auto-correlation and recorded audio's correlation (FLUTE).....	36
Figure 8.7: data obtained from performing the song with the piano	37
Figure 8.8: Fourier Transform of the signal data from the piano	37
Figure 8.9: Comparison between synthesized audio's fundamental frequency sequence and recorded audio's fundamental frequency sequence (PIANO).....	38
Figure 8.10: Comparison between synthesized audio's auto-correlation and recorded audio's correlation (PIANO)	38
Figure 8.11: Do's Scale with silence	39



Figure 8.12: Fundamental frequency sequence of octave with silence	40
Figure 8.13: Do's Scale without silence	40
Figure 8.14: Fundamental frequency sequence of octave without silence	41
Figure 8.15: Twinkle, twinkle, little star song's first part (FLUTE)	41
Figure 8.16: Song's fundamental frequency sequence (TUNER - FLUTE)	42
Figure 8.17: Do's Scale	43
Figure 8.18: Do's Scale fundamental frequency sequence (PIANO).....	44
Figure 8.19: Twinkle, twinkle, little star song's first part (PIANO)	44
Figure 8.20: Song's fundamental frequency sequence (TUNER - PIANO)	45
Figure A1.1: Nao	52
Figure A1.2: Nao's sensors and actuators	53
Figure A1.3: Choregraphe's graphical interface	54
Figure A1.4: Say box in the Flow Diagram.....	54
Figure A1.5: Script of a random box (Choregraphe).....	55
Figure A1.6: Function ALProxy for Choregraphe script writer	56
Figure A1.7: Interaction block diagram for Nao	57



Table Index

Table 2.1: Time Estimation.....	4
Table 4.1: Pros and Cons of Time domain and Frequency domain algorithms.....	16
Table 5.1: Difference between Python and C/C++	18



CHAPTER 1

Introduction

1. Motivation

Music has taken an important part in our lives for thousands of years, and that makes it a highly interesting topic of study from a computational point of view.

There are two main lines of research regarding intelligent systems applied to music. One of these lines is the realization of music synthesis systems, which create musical compositions applying aesthetic rules and generative systems. The other one is recognition of music pieces from audio signals, based on signal processing.

The problem of music content recognition can be posed in several degrees, ranging from the very literal signal matching to the most complex problem of recognizing the musical composition despite variations of the interpretation.

This project is a simplification of the second line. I want to recognize easy monophonic melodies interpreted by a single voice (human or instrument) and detect deviations from the melody.

2. Objectives

The main objective of the experiments carried out in this project is to develop an application for people who are interested in music or who are learning music to practice melodies with musical instruments. The experiments have been carried out on recordings of a song, "Twinkle, twinkle little star", created synthetically for a perfect sequence of note frequencies in order to have some gold standard to compare with.

3. Structure of the Memory

This report is divided into different chapters. Each chapter will explain a certain aspect of the project, and will be divided into different sections.

- Chapter one is the Introduction chapter. Here, the reader can find the motivation for the project and the objectives
- Chapter two explains the project management. It is divided into 5 sections. The first 4 sections explain the planning made before starting the project substantiated in the Work Breakdown Structure, Time Estimation, Gantt Chart, and Risk Management. The last section explains the changes in this planning that took place along the project development.
- Chapter three explains some interesting concepts about Pitch Detection, which is the main topic of the project. This chapter contains definitions and mathematical concepts regarding to music retrieval.



- Chapter four lists different Pitch Detection Algorithms (PDA) running either in time domain or in frequency domain.
- Chapter five explains decisions made and technical specifications assumed as preliminary steps before start proper project development.
- Chapter six contains the design of the application. It is divided into two parts; the first is about the interaction between the application and the user, and the second about sound process.
- Chapter seven details the actual implementation of the application.
- Chapter eight explains the results of the test that have been made on experimental data gathered by the current implementation.
- Chapter nine proposes possible future improvements. These improvements are divided into two sections. On the one hand, improvements based on the results of the tests. On the other hand, commercial applications that can be developed from the experience obtained in this degree project.
- Chapter ten explains the conclusions made out of this project and the personal experience.

The memory also includes two Appendices:

- The first one explains the long term goal of the research started in this project, which is developing this application on the Nao robot. In the Appendix there is an introduction to the Nao, which explains its features and the different software tools to develop a project with the robot. It also contains the design for the implementation and the implementation. It also explains the problems that led to leave the robot aside.
- The second Appendix contains the full code of the application, divided into different files. It also explains different aspects of the files.

Finally, the memory offers a full list of all the references used to develop this project. These links will be numbered and will be used through the whole memory to indicate where the relevant information has been found.

CHAPTER 2

Project Management

1. Work Breakdown Structure

The whole project will be divided into 5 branches, which are Management, Research, Development, Documentation and Finalization. All these tasks include some work packages that correspond to the tasks mentioned above.

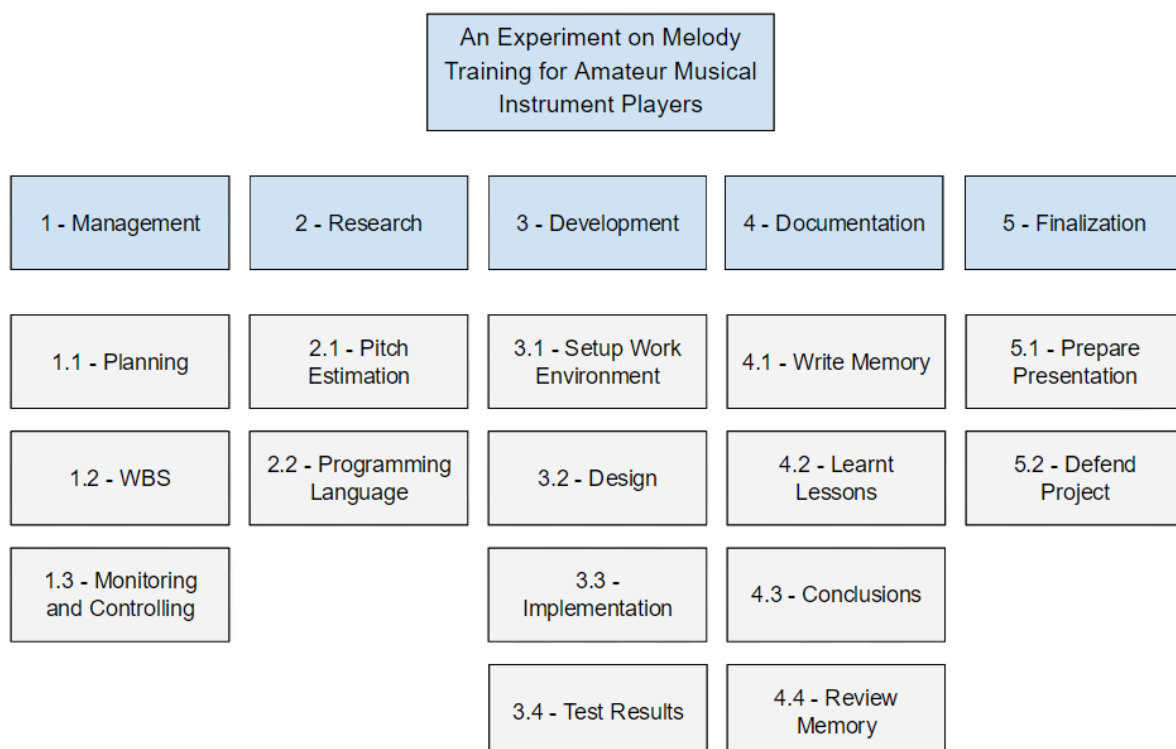


Figure 2.1: Work Breakdown Structure

2. Time Estimation

The following table (see table 2.1) shows the time estimations made for each work package mentioned in the Work Breakdown Structure (see figure 2.1).



Tasks	Estimated time (hours)
1 - Management	39
1.1 - Planning	25
1.2 - WBS	4
1.3 - Monitoring and Controlling	10
2 - Research	42
2.1 - Pitch Estimation	20
2.2 - Nao	20
2.3 - Programming language	2
3 - Development	105
3.1 - Setup Work Environment	7
3.2 - Design	15
3.3 - Implementation	80
3.4 - Test Results	3
4 - Documentation	81
4.1 - Write Memory	65
4.2 - Learnt Lessons	2
4.3 - Conclusions	2
4.4 - Review Memory	12
5 - Finalization	22
5.1 - Prepare Presentation	20
5.2 - Defend Project	2
Total	289
Cushion	11

Table 2.1: Time Estimation

3. Gantt Chart

The Gantt Chart (see figure 2.2) illustrates the project schedule. It includes the starting date and the finishing date of each work package.

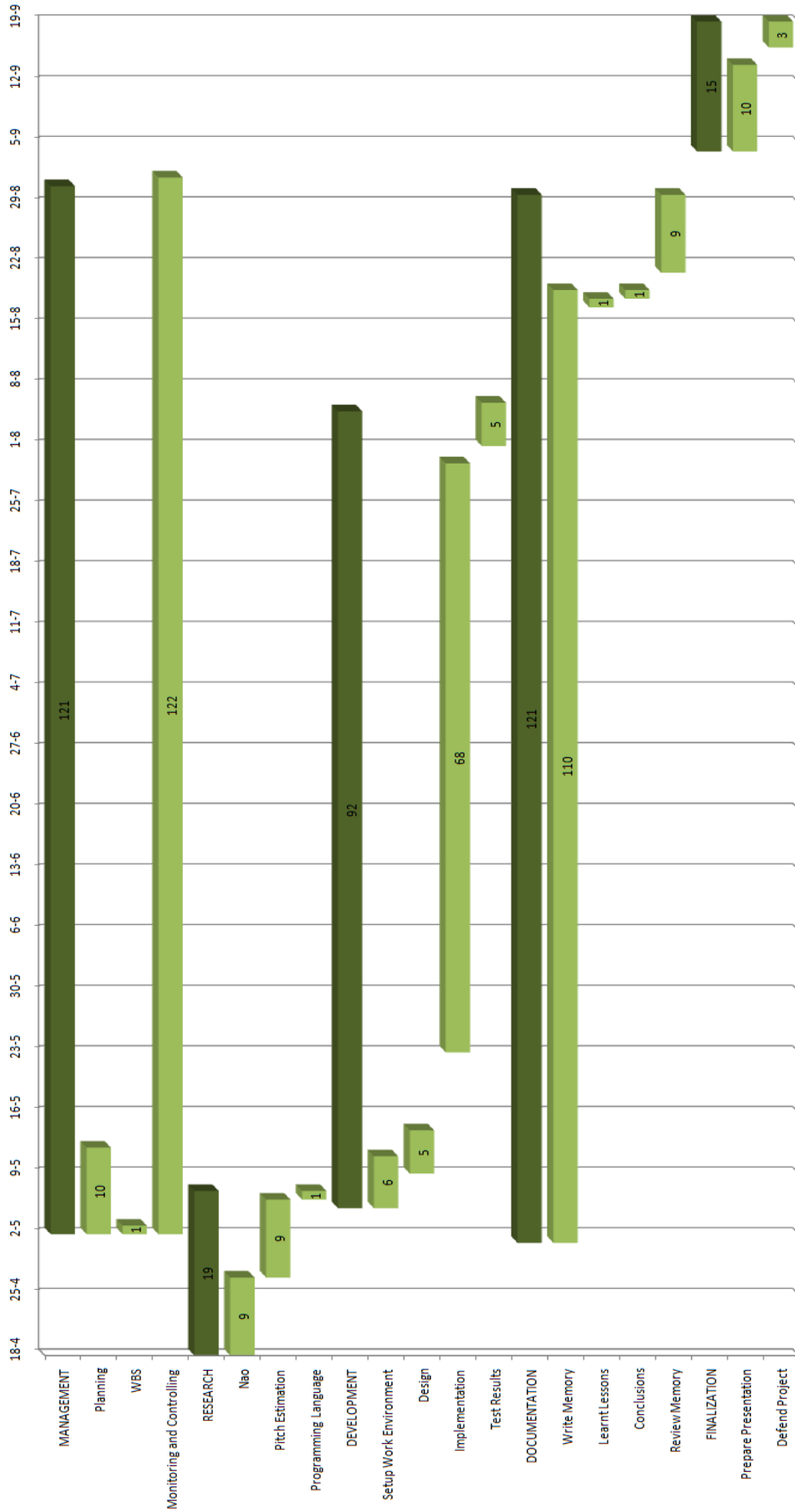


Figure 2.2: Gantt Chart



4. Risk Management

For this type of project, the following risks have to be taken into account:

- **Information loss:** As the project's aim is to create an application, there is a high chance that the program or the data needed to develop it will get lost. To solve this problem, each update of the application and the needed extra files (such as the music files) will be saved into a well-known cloud platform called "Drive" as well as in the laptop's hard disk.
- **Lack of experience:** The development of a fully fledged project is something never done before by the developer (me), hence the risk of having difficulties understanding the subject and fulfilling different tasks is a risk that may happen many times during the time of the implementation. This will lead to delays. I will rely on stackoverflow or similar websites that provide solutions. The cushion mentioned in the Time Estimation table (see table 2.1) is also a good solution.
- **Software/Hardware related problems:** The use of technological devices involves lots of risks, such as: hardware failure, software version incompatibilities, hardware (NAO robot) and software incompatibilities and so on. These problems may lead to different adversities, such as updating the systems that need an update, changing the programming language used, finding another way to solve the problem and, in the worst of the cases, changing the aim of the project because of incompatibilities.

5. Monitoring and Control

5.1 Communication

The advisor had access to the entire project's information through the storage system called drive. Apart from that, every two weeks, the control meetings were held with the project advisor to decide the following steps and to discuss the development of the project and make decisions.

5.2 Deviations

Through the development of this project, there have been a lot of deviations, most of them regarding to the fact that the main idea had to be changed due to incompatibility factors. However, once the current project's development was started, things went smoothly.



CHAPTER 3

Introduction to Pitch Estimation

In this chapter there will be some definitions and explanations about the most important part of the project, which is pitch estimation or pitch detection.

1. Definitions

1.1 Pitch

Pitch is a perceptual auditory phenomenon which is closely related to frequency. After being processed in the human brain, the listener associates the pitch with a musical tone, which is determined by a musical scale based on the perception of the *frequency* of vibration.

Even if pitch and frequency are closely related, they are not the equivalent. Frequency is an objective, scientific attribute that can be measured, pitch, on the other hand, is the subjective perception of the *sound wave* that depends on the listener and cannot be directly measured.

Sounds can be ordered on a scale from low to high. The relative position in that scale depends on how fast the sound wave is oscillating, rapid oscillations corresponding to “higher” pitch, and slow oscillations corresponding to “lower” pitch.

In physical terms, pitch can be considered as fundamental frequency, which can be measured. While the perceived tone may differ from the perceived fundamental frequency, in a periodic signal, they correspond to each other.

1.2 Musical Tone

A musical tone is a steady periodic signal. It has 4 attributes, which are duration, pitch, intensity (loudness) and timbre (quality). However, in music, musical tones may include other aspects such as vibrato, transients and modulation.

A pure tone corresponds to a sinusoidal wave, this means that it contains a single frequency. A complex tone is not sinusoidal, but it is periodic and can be described as a sum of sinusoid wave fronts.

1.3 Octave

An octave is the interval between one musical pitch and another with half or double its frequency. The musical scale is composed of 7 notes, represented in different ways depending on the country. The most common ways to represent it are the following: with letters (C, D, E, F, G, A and B), with syllables (Do, Re, Mi, Fa, Sol, La, Si) which is called

Solfège, or with numbers (1, 2, 3, 4, 5, 6 and 7). Two notes separated by an octave have the same representation.

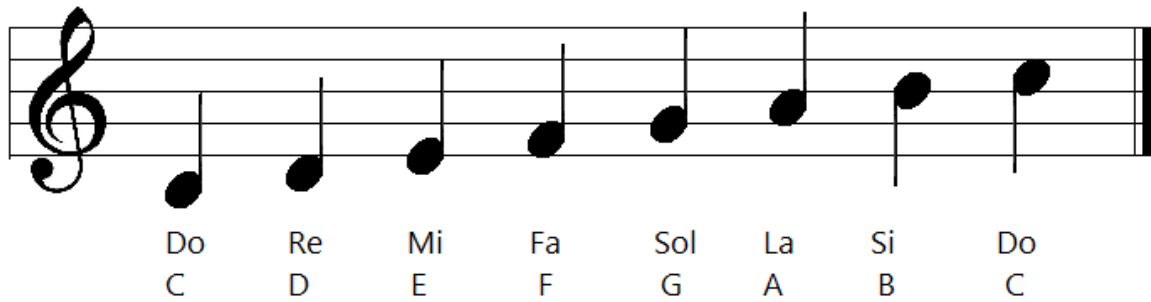


Figure 3.1: Do Scale

There are notes in between the well-known 7 notes, called semitones, which can be represented as the note followed by the # symbol, which is called “sharp” or by a \flat symbol, which is called “flat”.

1.4 Melody

Melody is a concept based on the judgment of human listeners, so it has lots of different definitions depending on the context. However, it can be simplified as “a linear succession of musical notes that the listener perceives as a single entity”.

To represent a melody, there are two different ways: (a) the Tablature, which is unique for each musical instrument, (b) the Music Sheet, which is a common one for all the instruments. The Tablature is much easier to understand because it is based upon a diagrammatic representation of the position of the fingers. The Music Sheet, on the other hand, is more abstract and complex, but it is written in a common musical language.

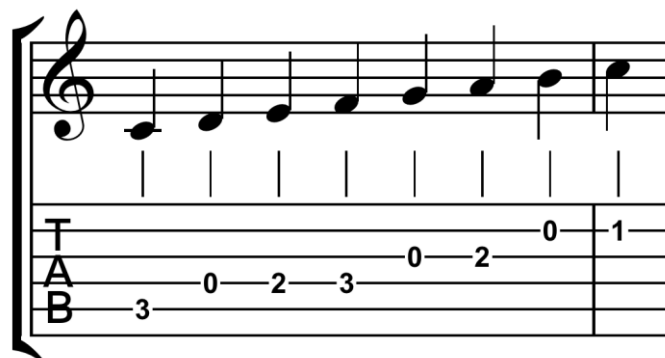


Figure 3.2: Equivalence of Music Sheet and Tablature

The image above (see figure 3.2) shows part of a music sheet and part of a tablature for a guitar, and the connection between both methods.

1.5 Polyphony

Polyphony is a property that has some musical instruments to play more than one note at a time, these instruments are called polyphonic. The instruments that do not have this property are called monophonic.

Despite human brains are capable of separating different sources of sound, automated computer analysis is an extremely complex process, therefore, this project will focus on monophonic sounds.

1.6 Harmonics

It is applied to repeating signals, such as sinusoidal waves. An harmonic is a positive integer multiplication of the fundamental frequency, aka first harmonic. The positive integer multiplication makes the harmonics to have the same period as the fundamental frequency, so that, sometimes it is difficult to decide which is the original frequency, as some harmonics can have a higher amplitude.

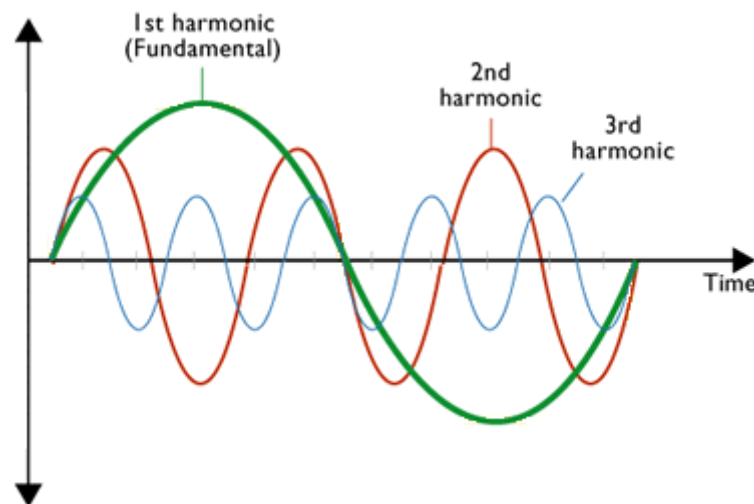


Figure 3.3: Harmonics of the Fundamental Frequency

As shown in the image above (see figure 3.3), all the sinusoids meet at the same points. From the green sinusoid only one period appears in the picture. From the red one, there are 3 periods, and from the blue one 5 periods, but the main concept is that all of them meet at half of the period of the fundamental frequency and the point where the period of the fundamental frequency ends.

1.7 MIR (Music Information Retrieval)

Music Information Retrieval is the interdisciplinary science of retrieving information from music. It is used by academics and businesses to categorize, manipulate and even create music.

There are different types of research areas that can be included in the field of MIR:

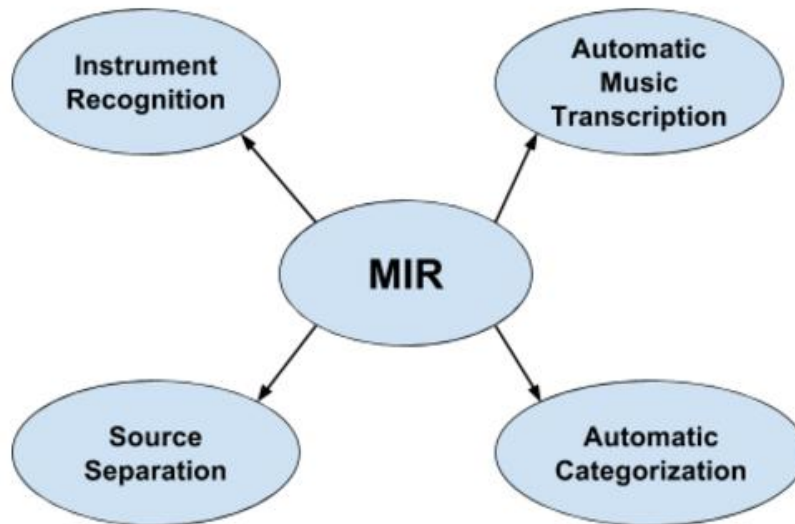


Figure 3.4: MIR research areas

2. Mathematics

2.1 Fourier Transform

For this project, the Fourier Transform is a very important mathematical concept. It decomposes a time based function (a signal) into the frequencies that the signal possesses. For audio signals, the analysis in frequency domain provides information closer to human perception than the analysis in time domain.

Fourier Transform can be applied either in continuous time, $x(t)$ with period T (time interval), or in discrete time, $x(n)$ with period N (number of samples). The absolute value of the amplitude of the result of applying the Fourier Transform represents the amount of the frequency of the original signal, and the complex part represents the phase of the sinusoid of that frequency. Therefore, Fourier analysis decomposes the signal in a combination of harmonics.

The formula of this mathematical concept is the following:

$$X(F) = \sum_{n=-\infty}^{+\infty} x(n) e^{j2\pi Fn}$$

where:

F : denotes the frequency

As it is impossible to store and process ∞ numbers, the Discrete Fourier Transform (DFT) is the one used. Its formal definition is as follows:

$$X(k) = X\left(\frac{k}{N}\right) = \sum_{n=0}^{N-1} x(n) e^{j\frac{2\pi k}{N}n}$$

Apart from the Fourier Transform and Discrete Fourier Transform, there also exist Short-Term Discrete Fourier Transform and it is used to combine the frequential analysis with time domain analysis.

The Fast Fourier Transform (FFT) is an improved Discrete Fourier Transform. The best known version of this algorithm is the Fast Fourier Transform in base 2. This algorithm uses the idea that the DFT can be shown to be composed of two signals containing the even and odd numbers of $x(n)$ respectively.

$$X(K) = F_1(k) + w_N^k F_2(k)$$

This algorithm reduces the number of complex multiplications of N^2 to $\frac{N}{2} \log_2 N$ and the complex additions of $N(N-1)$ to $N \log_2 N$. It is the basis for a multitude of practical applications of the Fourier transform in signal processing and communications.

2.2 Downsample / Decimation

Downsampling (aka decimation) consists in reducing the samples of the digital signal to a lower rate. This expands the frequency spectra and compresses the amplitude as shown in the following picture (see figure 3.5):

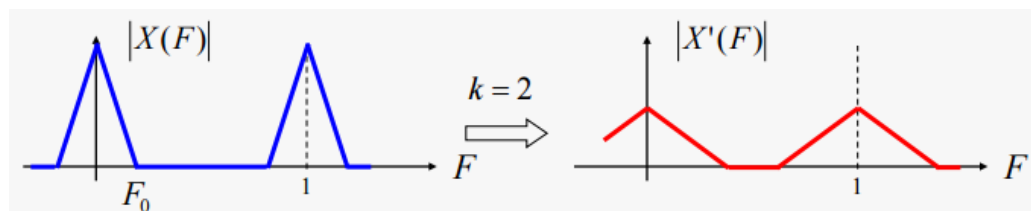


Figure 3.5: Decimation

The process of decimation consists in: first, chose a decimation factor, M , which is an integer value, and, second, keep every M th value in the discrete signal, discarding all the other values.

Even if downsampling and decimation aren't the same, for this project the main concept is that the signal is compressed, so the difference doesn't matter. In reality, downsampling means to only take into account the M th values and throw the others away, on the other hand, when applying decimation, a low pass filter has to be applied, as, otherwise, it could appear aliasing if the Nyquist bound frequency is violated.

CHAPTER 4

Pitch Estimation Algorithms

There are lots of algorithms that can be used to determine the pitch of each note, either played in an instrument or sang. These algorithms can be implemented in the time domain or the frequency domain. Here, some of the best known will be explained briefly.

1. Frequency Domain Algorithms

Frequency domain algorithms are based on the signal's appearance number within each given frequency band over a range of frequencies. Frequency domain algorithms rely on the Fourier Transform.

1.1 HPS (Harmonic Product Spectrum)

Harmonic Product Spectrum measures the maximum coincidence for harmonics for each spectral frame. The algorithm used is the following:

$$Y(\omega) = \prod_{r=1}^R |X(\omega r)|$$

where:

R: number of harmonics being considered

X: magnitude spectrum of the frequency

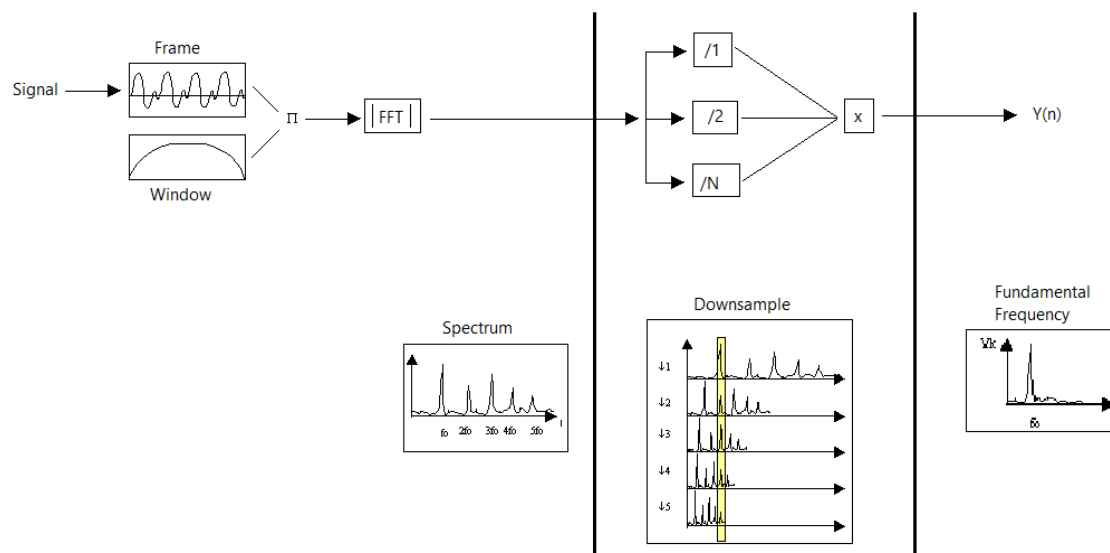


Figure 4.1: Harmonic Product Spectrum Algorithm

Being the input a musical note, the audio has to be windowed to apply the Fourier Transform. As shown in the picture above, (figure 4.1) the spectrum of the signal window consists of a series of peaks that correspond to the fundamental frequency of the note and its harmonics. After applying the Fourier Transform, the result has to be downsampled n times, this makes the peaks stand out and, after downsampling, the n results are multiplied by each other to obtain a single peak, which should be the fundamental frequency.

This method has its pros and its cons. The pros are that it is computationally inexpensive and reasonably resistant to noise. However, the low pitches are tracked less accurately. Another disadvantage is that the resolution depends on the fft length, if the Fourier Transform is short and fast, the result is limited in the number of discrete frequencies considered. To make the algorithm more precise, the fft should be longer, which will take more time.

Octave error is also very common in these kinds of algorithms. This consists on estimating the musical note one octave above or below as they share the same periodicity.

1.2 Cepstrum Analysis

This algorithm applies the inverse Fourier Transform to the logarithm of the signal's spectrum. The name derives from changing the order of the first 4 letters of "spectrum".

The formula for this algorithm is the following:

$$|F^{-1}\{\log(|F\{f(t)\}|^2)\}|^2$$

where:

F : the Fourier Transform

F^{-1} : inverse Fourier Transform

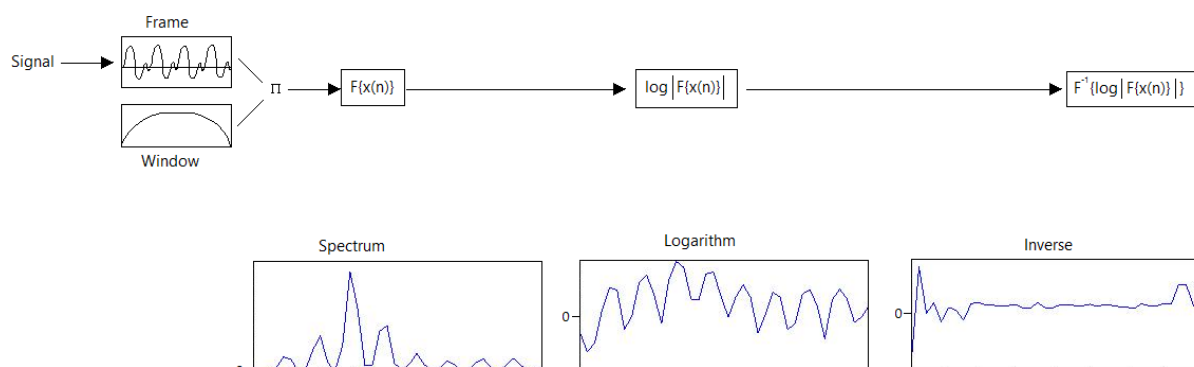


Figure 4.2: Cepstrum Algorithm

As seen in the image above (figure 4.2), the Cepstrum Analysis first frames the original signal and then applies a window to each frame. After windowing the frame, the Fourier Transform is applied. The spectrum obtained by doing the fft is represented by a series of peaks. The following step is to convert into a logarithmic representation, which accentuates the sinusoidal appearance of waves. Finally the algorithm uses the Inverse of Fourier Transform to obtain the fundamental frequency peak.



Cepstrum Analysis uses Homomorphic Filtering, which is a nonlinear transform applied image or speech processing and it is used to convert the signal obtained from the convolution of two signals into the sum of two signals.

Cepstrum Analysis has many applications, such as analysis of human speech and pitch detection. However, it was first developed to detect seismic echoes.

1.3 Parabolic Interpolation

As any frequency domain algorithm, Parabolic Interpolation uses Fourier Transform to get the frequencies from the time domain.

The goal of this algorithm is to obtain the abscissa of the peak of the sample analyzed by successively fitting parabolas. It uses 3 points of the sample to create the parabola and find the peak (either maximum or minimum).

2. Time Domain Algorithms

Time domain algorithms are based on how the signal changes over time. In this case, the variable is always measured against time.

2.1 ZCR (Zero-crossing Rate)

Zero-crossing rate counts the number of times a signal passes through the value 0. In time domain, the distance between two adjacent 0s is half the frequency of the signal (which is represented by the amplitude), as the sinusoid has a positive and negative amplitude, so, if that is taken into account, zero-crossing can be used to obtain the frequency of a signal.

The formula is the following:

$$ZCR = \frac{1}{T-1} \sum_{t=1}^{T-1} 1_{R<0}(s_t s_{t-1})$$

or

$$ZCR = \frac{1}{T} \sum_{t=1}^T |s(t) - s(t-1)|$$

where

s: signal of length T

$1_{R<0}$: indicator function

2.2 Autocorrelation

Autocorrelation is used to find similarities between the original signal and the signal shifted. This is used to find the fundamental frequency by detecting the highest point of the autocorrelation.

This method is the most used and most precise when attempting to estimate pitch in time domain.

To apply autocorrelation, the theoretical formula is the following:

$$R(m) = \lim_{N \rightarrow \infty} \frac{1}{2N+1} \sum_{n=-N}^N x(n) * x(n+m)$$

where

N: length of analyzed sequence

M_0 = number of autocorrelation points to be computed

However, in pitch detection, as the samples are finite, the following formula is used:

$$R(m) = \sum_{n=0}^{N-1-m} x(n) * x(n-m)$$

3. Frequency Domain vs. Time Domain

The following image (figure 4.3) shows the representation of a 1 kHz signal in time domain and in frequency domain. It can be seen that they both are very different, as the first one shows a periodic signal over time, the unmistakable figure of a sinusoid, and the second one shows a peak at 1 kHz, as the x axis represents the frequency.

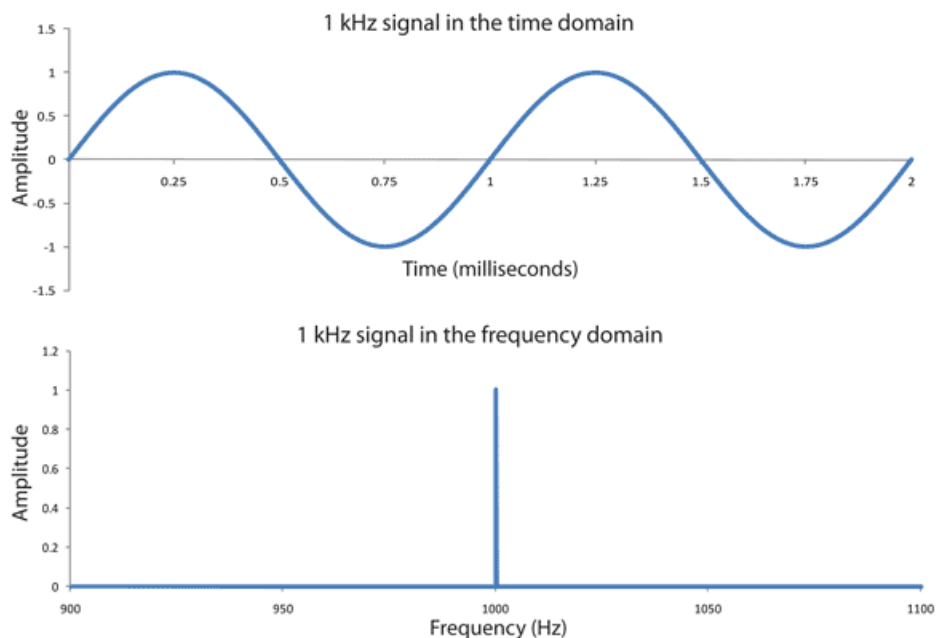


Figure 4.3: Difference between Time domain and Frequency domain

The table below contains the pros and the cons of each method.



	Pros	Cons
Frequency Domain	<ul style="list-style-type: none"> - Resistant to noise - Able to process polyphonic sounds - Performs well with large shift factors 	<ul style="list-style-type: none"> - Computationally expensive - Difficult to implement
Time Domain	<ul style="list-style-type: none"> - Fast - Easier to implement 	<ul style="list-style-type: none"> - Unable to process polyphonic sounds - Less accurate with noise - Performs poorly with large shift factors

Table 4.1: Pros and Cons of Time domain and Frequency domain algorithms



CHAPTER 5

Work Environment

This chapter will explain the steps previous to start the work of implementing the application. It will include the decisions made before designing the application. It also gives the description of the technical elements that have been used to develop the project.

1. Hardware

For this project, the main device used has been a Toshiba laptop, version Satellite L870-10Z, which includes a microphone to record the sounds. It is run in a 64-bit system. Another device used is a tuner, to compare some results regarding to note frequencies.

2. Software

The laptop OS runs in Ubuntu 14.04 and has installed different packages to implement the application.

The packages used are the following:

- **SciPy**: SciPy is a python-based ecosystem of open source software for mathematics, science and engineering.
- **NumPy**: NumPy is the fundamental package for scientific computing with python. It offers the following uses: a powerful N-dimensional array object, sophisticated functions, tools for integrating C/C++ and Fortran code and useful linear algebra, Fourier transform and random number capabilities. (belongs to SciPy)
- **MatPlotLib**: MatPlotLib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. (belongs to SciPy)
- **PyAudio**: PyAudio is a library that provides Python bindings for PortAudio v19, the cross-platform audio I/O library. PyAudio can be used to play and record audio streams on a variety of platforms (e.g., GNU/Linux, Microsoft Windows, and Mac OS X).
- **Wave**: The wave module provides a convenient interface to the WAV sound format. It does not support compression/decompression, but it does support mono/stereo.

All the previously mentioned packages are open source and are used in python, which is the programming language that has been used to implement the application. They can be obtained by installing them via *pip*, which is the recommended tool to install Python packages, or by downloading them from their main website.



2.1 NumPy vs. SciPy

Even if the packages NumPy and SciPy are installed separately, they both belong to the same module, which is SciPy.

If the goal is to develop a big scientific project in python, it is recommended to install both of them, considering that they each focus on different aspects.

The NumPy package contains the array data type and some basic operations such as indexing, sorting, reshaping and so on. But the reality is that NumPy focuses on compatibility, so it tries to retain features supported by its predecessors. Even if NumPy contains some linear algebra functions, Scipy is more complete in this area, so that's why it is recommended to install both of them.

3. Decisions

3.1 Python

The decision was between four programming languages, which are Python, Matlab, Java and C/C++.

After some research, Java was put aside, due to the fact that it doesn't have much support for these type of projects.

Matlab was the wisest choice as it was the one used in the subject "Procesado Digital de Sonido e Imagen", where some similar aspects were taught. However, one of the main ideas behind this project was to learn something new, so Matlab was also put aside.

The final choice was between C/C++ and python, each one with good open source resources and pretty complete for this project. Below, a table with the differences between both programming languages (see table 5.1):

	Python	C/C++
Length of code	Short	Long
Declaration	No need	Necessary
Indentation	Necessary	No need (brackets)
Execution speed	Slower	Fast
Learning	Easy	Hard

Table 5.1: Difference between Python and C/C++

Observing the table above (see table 5.1) it can be seen that python has a faster development and is more user friendly, and taking into account that the first version of the project, where the robot was used to make the interaction, was written in python, the final decision was to implement the application in this programming language.



3.2 Speech Recognition

Making an application that recognizes speech, and detects the notes of the song would be a really interesting project, as the song chosen for the project has lyrics. Nevertheless, speech recognition is a very complex discipline, and, for that reason, it was left aside in this project.

On top of that, to make them both, speech recognition and pitch detection, work together would be extremely laborious, as both are represented by frequencies, and there would be mixed concepts.

3.3 Monophonic melody

An example of polyphonic sound would be playing a polyphonic musical instrument, as guitar, piano, violin, and playing more than one note at the same time. That would mean the additional task of separating different sources of audio, which is quite difficult. For that reason, the idea of processing polyphonic sounds was put aside.

3.4 Harmonic Product Spectrum (HPS)

The first step for choosing the correct PDA (Pitch Detection Algorithm) was to decide between time domain and frequency domain algorithms. After some research it was decided that the most suitable one was frequency domain, as musical notes have a fundamental frequency and harmonics, and frequency domain algorithms focuses on these concepts.

Next step was to decide the algorithm. Even if any algorithm would work for this project, HPS focuses more on harmonics, as downsampling the frame and then multiplying the results highlights the fundamental frequency based on the harmonics.

3.5 Hanning Window

Windowing a signal before applying the Fourier Transform is an essential step. There is a wide range of window functions to choose from. There are a lot of criteria to choose the most suitable window for each particular signal. However, the Hanning Window is generally the one that works well in most of the cases, having a 95% of satisfactory results.

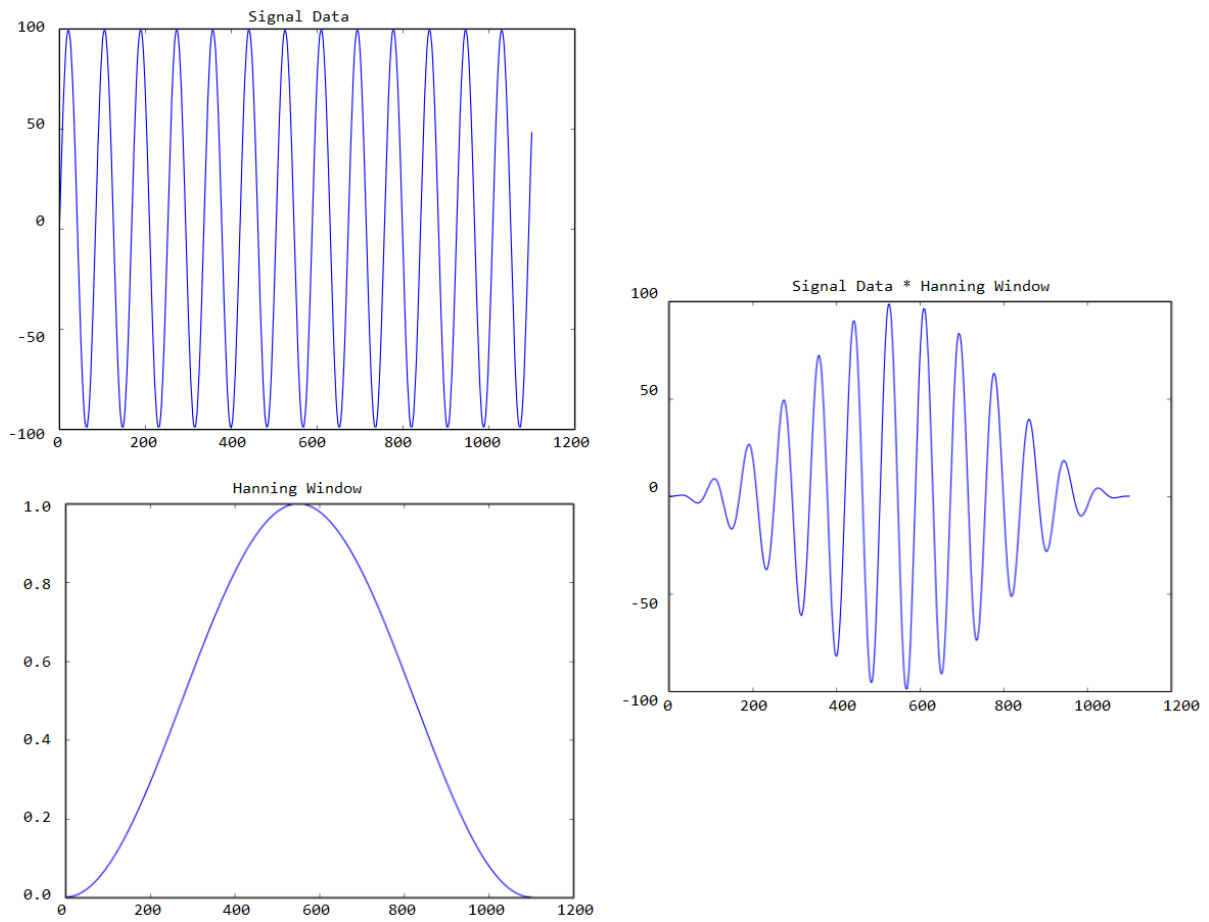


Figure 5.1: Hanning Window application's graphical view

CHAPTER 6

Design

Before starting the implementation, a design is needed, to know the steps to follow. It can be said that the application has two main parts, one regarding to the interaction between user and application, and the other the sound process, the most important part of the application.

1. Interaction

The interaction is very basic and it is explained in the following diagram:

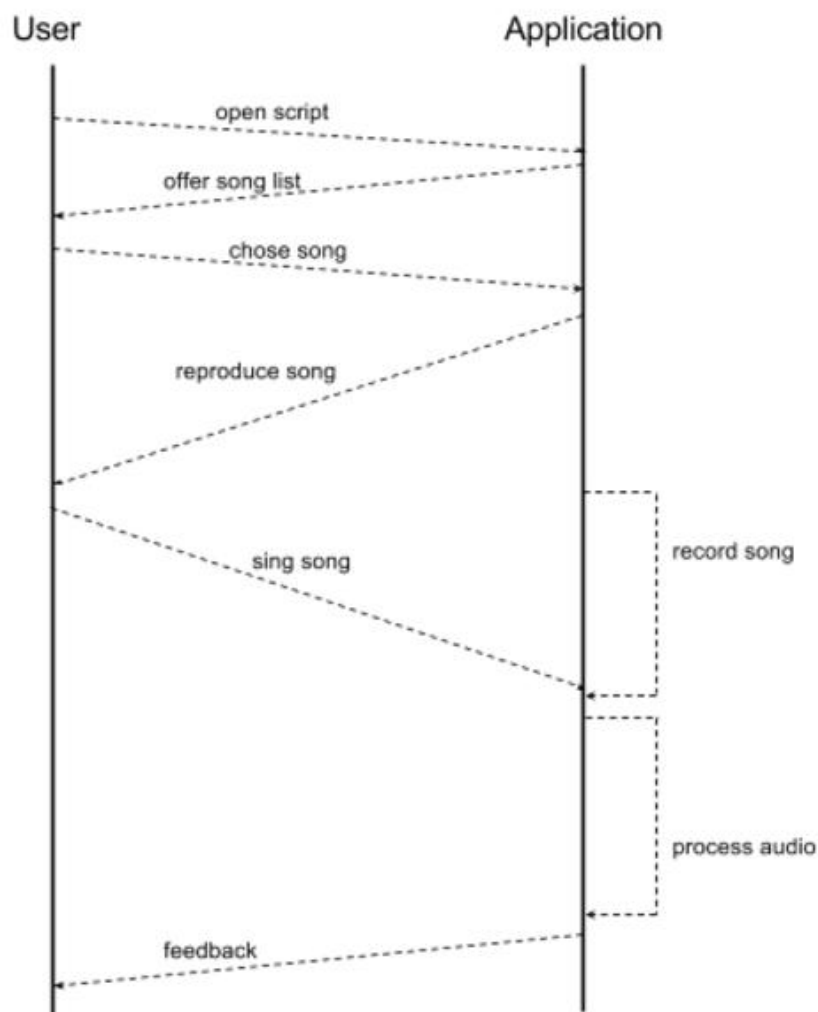


Figure 6.1: Interaction diagram between User and Application

This diagram shows both the user and the application perspective.

From the top, the user opens the application and the application shows on screen the songs available (in this case there is only one song, but the choice is there for future versions of the application). After the user chooses the song, the application reproduces the synthesized version of the song. When it finishes reproducing, the user plays back or sings back the song and, when finished, presses <Enter> on the keyboard. While doing this, the application is recording the song. After that, the application must process the audio recorded to give its feedback.

2 Sound Process

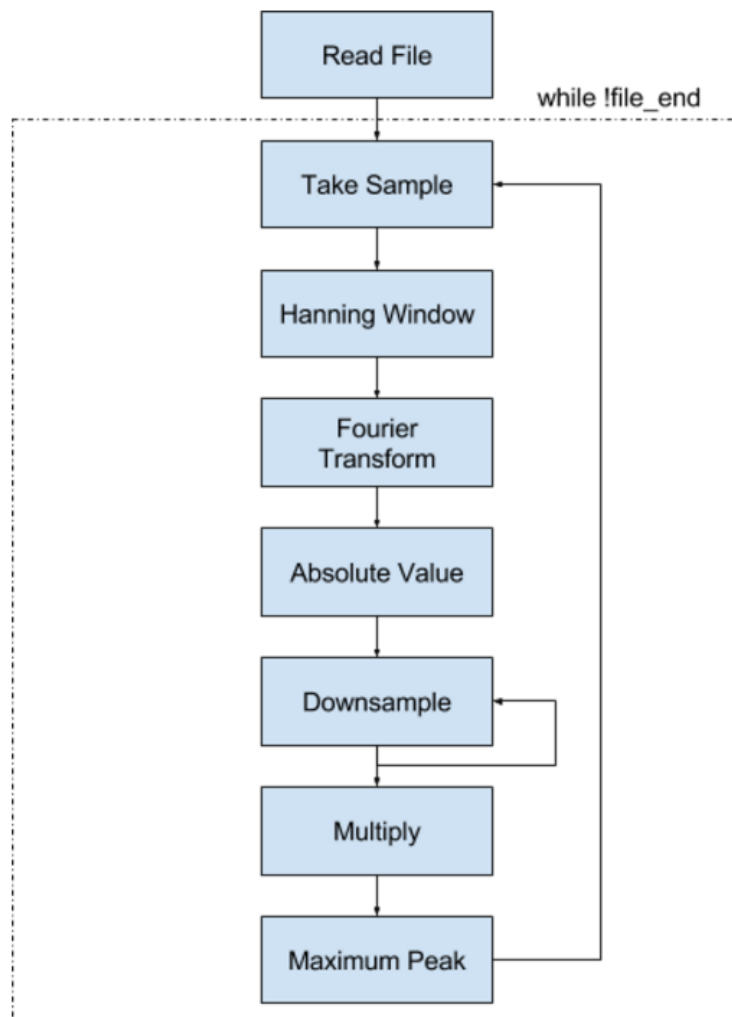


Figure 6.2: Sound Processing block diagram

The diagram above (see figure 6.2) shows the algorithm of the Harmonic Product Spectrum applied to the recorded audio file.

- First, the application reads the file recorded and takes from it the interesting information, such as the sample rate and the audio data.
- The information to be processed is the audio data. In a loop, which ends when the audio file finishes, the steps are the following:



- First a frame is taken from the audio data. The size of the frame should not be too big, nor too small. Big frames leave fewer frequencies to be analyzed, so the end result is less accurate. Small frames make the performance of the algorithm more expensive, so more time is spent.
- After having the frame, a Hanning Window is applied. This improves the characteristics of the sample.
- Next step is to Apply Fourier Transform to the windowed frame. This will make the data of the signal become a series of frequencies so that fundamental frequency extraction is easier.
- From the frame, take the absolute values.
- Once the frame is processed, apply downsampling to make the peak more visible. The number of downsamples has been chosen after different attempts and taking the one that shows the best result.
- Once the frame is downsampled N times, those results are multiplied by each other to have a unique result.
- The result of the downsampled frame multiplication, the maximum peak is chosen, and, from that peak, it is taken the x-axis value, which is the frequency. This value is turned into frequency by doing some operations that take into account the sample size and the sample rate.

3. Comparison

The recorded file, after being processed, has to be compared with the synthesized song to observe the difference between each other to determine if the song was played properly.

For this, the steps are the following:

- Auto-correlate the synthesized song. This will look like a perfect parabola.
- Correlate the synthesized audio file with the recorded one. The result will also look like a parabola, but it will have some parts different from the previous one, as the sound isn't exactly the same.
- As the recorded audio can have a little bit of delay (the user starts playing the song seconds after the recording starts), this delay is calculated, and the start of the song that corresponds to the delay is deleted.
- The correlation between the synthesized audio and the recorded audio is done a second time, this time without delay. The result will deduce if the song was played properly or not.

CHAPTER 7

Implementation

This chapter will explain the implementation of the application. It will be separated into different parts, as the application is composed by more than one script, each one focused on different aspects.

1. Sound Processing

Everything that has to do with sound processing is located in this script, called `soundProcessing.py`. This script is separated into different functions, to make the code clearer, and the code has comments to make it more understandable.

From top to bottom, the functions that appear in the script will be explained:

1.1 Read_File

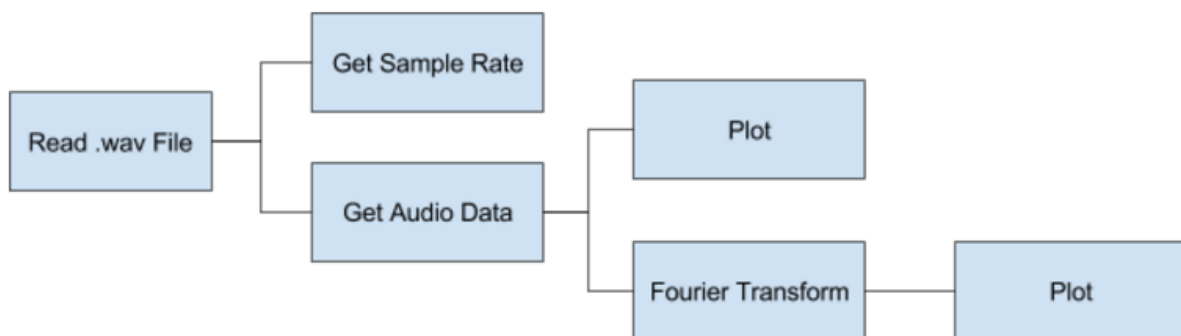


Figure 7.1: Read_File function's block diagram

This function focuses on reading an audio file and taking from it the information needed to do sound processing. The interesting information is the sample rate and the audio data. The audio data is plotted to see the graphical view of it. In addition, Fourier Transform is applied to plot the audio data in frequencies. Last but not least, the function returns the sample rate and the audio data.

```

# read audio file depending on mono or stereo
if synth == 0:
    self.fs, self.data = wavfile.read(file_name)
else:
    self.fs, self.data2 = wavfile.read(file_name)
    self.data = self.data2.sum(axis=1) / 2

return [self.fs, self.data]
  
```

To obtain the important information from the audio file it has been used the function `wavfile.read`, which is part of the SciPy's input/output package. To apply Fourier Transform, the function used has been `fft.rfft`, this function belongs to NumPy package. Finally, to plot the results it has been used the package Matplotlib, and from it the functions `pyplot.plot` and `pyplot.show`.

1.2 HPS_Algorithm

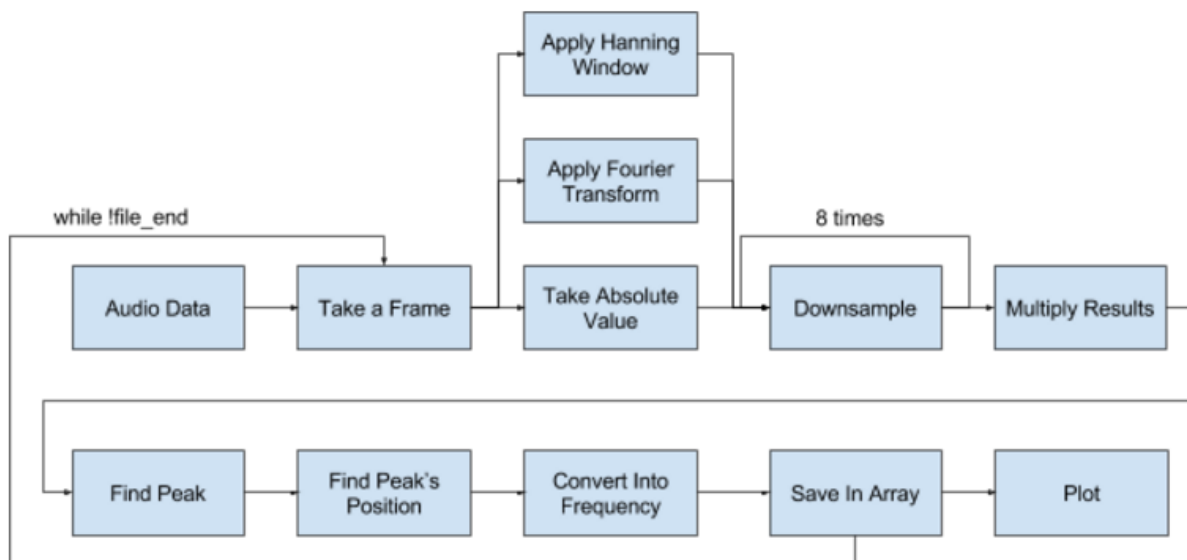


Figure 7.2: HPS_Algorithm function's block diagram

This function takes as parameters the sample rate and the audio data from `Read_File`. As explained in the previous Chapter, which explains the design of the application, the audio process has to be done in frames, as the whole file is too big to be processed in once. The size of the frame is 2048, as it isn't too small nor too big. The frame is modified by applying a Hanning Window, applying the Fourier Transform and taking the absolute values. After that, it is downsampled 8 times and these results multiplied to get the Maximum Peak. From the Maximum Peak, the application obtains its position in the x-axis, as that corresponds to the frequency. This position is converted into frequency by multiplying with the sample rate and dividing with the frame size and, after that, it is saved in an array so that the whole data can be plotted. The function returns the array and the number of frames.

```

for i in range(n_frag):
    frag = data[f_start:f_end]           # take a fragment of the file
    frag_win = frag * hann(SAMP)        # apply a hanning window to the fragment
    X = abs(fft.rfft(frag_win))         # do the fourier transform

    hps = copy(X)                       # copy to do the downsample

    # downsampling
    for h in range(2, 6):
        downsample = decimate(X, h)
        hps[:len(downsample)] += downsample

```




```
# find the position of the max peak and convert to freq
peak_pos = argmax(hps[:len(downsample)])
freq = fs * peak_pos / SAMP
```

The Hanning Window is part of the SciPy's signal package, and it is called `hann`. Fourier Transform is the same as in the previous script, which is `fft.rfft` from NumPy. To downsample the frame, it has been used the function `decimate`, which is part of the SciPy's signal package. After that, to find the maximum peak's position, `argmax` is the function that has been used, which is part of the NumPy package. Finally, the plotting of the frequencies is done with `pyplot`, explained earlier. All the other operations don't need any additional package.

1.3 Create_File

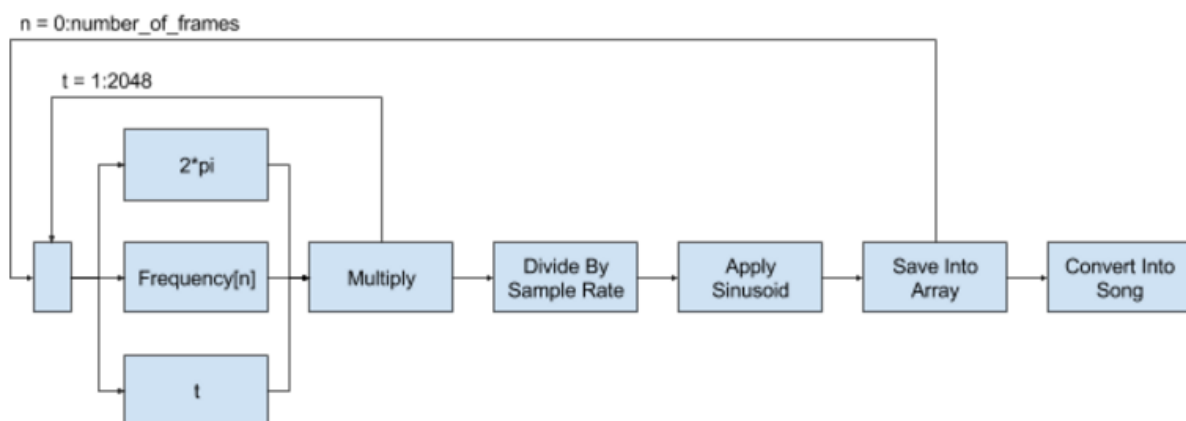


Figure 7.3: Create_File function's block diagram

This function creates sinusoids based on the frequency array obtained from the previous function. To do this, there are two loops, one inside the other. To convert the frequency into sinusoid, the function multiplies each frequency of the array with 2π and another variable which goes from 1 to 2048 (the frame size). Without this last variable, the file obtained would be 2048 times smaller than the original file, as, when framing the audio data, each frequency is obtained from a 2048 sized frame. After that, the result is divided by the sample rate and converted into a sinusoid, to save into an array that will be used to obtain the file.

```
for n in range(n_frag):
    for t in range(1, 2048):
        phase = sin(2 * pi * f0[n] * t / fs)
        song.append(phase)

# create final audio file
npsong = array(song)
wavfile.write('f0.wav', fs, npsong)
```



The sinusoid function used is called `sin` and it is part of the NumPy package. To generate the audio file, it has been used `wavfile.write`, which is included in the input/output package of SciPy.

1.4 Correlation

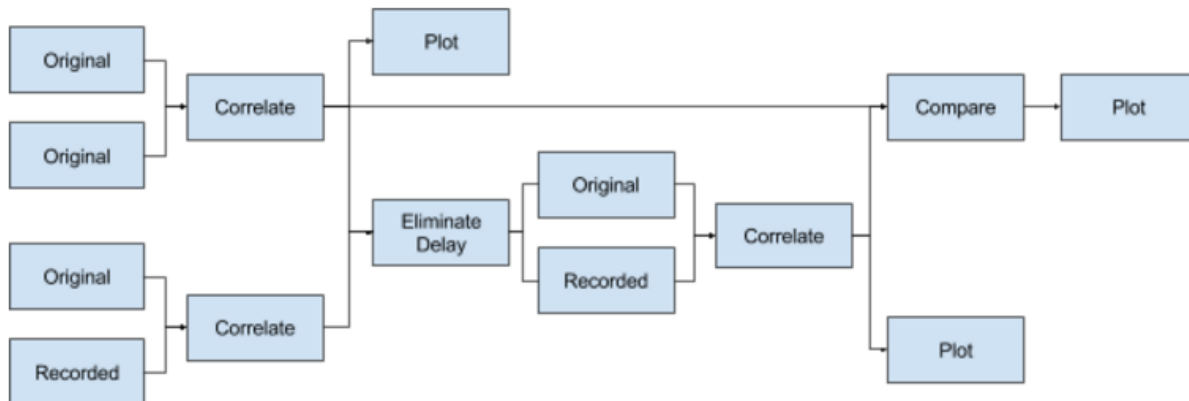


Figure 7.4: Correlation function's block diagram

This function first auto-correlates the original song and correlates the original song with the recorded one. After that, it calculates the delay between the recorded song and the original song and it eliminates it. It correlates again the original song with the recorded song without any delay and, after plotting both correlations, it compares the original song's auto-correlation with the recorded song's correlation and plots the result.

```

# auto-correlate the synthetised audio
corr_orig = corr_s(orig, orig, mode='same')          # correlation

# correlate the synthetized audio with the recorded one
corr_reco = corr_s(orig, reco, mode='same')         # correlation

# calculate the delay between the two signals
delay_reco = int(len(corr_reco)/2) - argmax(corr_reco)
print delay_reco
print len(reco)

# delete the delay of the recorded audio signal
delay_reco *= 2
while delay_reco > 0:
    reco.pop(0)
    delay_reco -= 1

# correlate again without delay
corr_reco = corr_s(orig, reco, mode='same')

delay_reco = int(len(corr_reco)/2) - argmax(corr_reco)
  
```

The correlation function is part of the SciPy's signal package and is called `correlate`. To plot the results it is used the package Matplotlib, and from this package, it



uses `pyplot.plot`, `pyplot.subplot` (to plot more than one graphic in the same image) and `pyplot.show`.

2. Interaction

For the interaction between the user and the application, another script was created, which is called `MyClass.py`. This script contains the visual aspects and the interactive diagram shown previously (see figure 6.1). The first function is `Introduction`, which don't need any explanation as the only thing it does is print some lines on the console. This script is also separated in different functions that are listed below:

2.1 Song_Choice

`Song_Choice` opens the file which contains the song list, reads each line one at a time and saves them on a vector variable to show them on screen. After this, the user selects a song, if the choice is wrong, the application lists again the available list of songs.

```
choice = 1;
# opens the file which contains the songs available
f = open("songs", 'r')

# reads the file to get the songs
for line in f:
    print line
    words = line.split('.')
    for word in words:
        self.vocabulary.append(word)

# while the number entered is wrong, keeps asking the correct number
while choice == 1:
    self.song = raw_input("\nWrite the number of the song and press enter: ")

    if self.song in self.vocabulary:
        choice = 0

    else:
        print "\nThat song doesn't exist.\n"
        choice = 1
```

The first variable shown in the code is `choice`, which is used as a flag variable, to know if the number introduced by the user is correct or incorrect. After that, the function opens the file which contains the list of the songs available and reads the file line by line to, then, print each line, which will be the identification number for the song, and the name of the song. The song's identifier is also saved in an array, called `self.vocabulary`. After that, the code starts a loop until the user chooses a song that is in the array.

2.2 Sing_Song

This function reproduces the song that the user has chosen in the previous function.



```
# open the audio file selected
if self.song == "1":
    print "\nYou chose Twinkle twinkle little star."
    wf = wave.open("twinkle_synth.wav", 'rb')

# create a pyaudio type variable to reproduce the sound
streamp = p.open(format=p.get_format_from_width(wf.getsampwidth()),
channels=wf.getnchannels(), rate=wf.getframerate(), output=True)
datap = wf.readframes(1024)

# while the file doesn't reach the end, keep reproducing the audio file
while datap != '':
    streamp.write(datap)
    datap = wf.readframes(1024)
```

First, the function uses the PyAudio package's `PyAudio()` function to initialize a variable, called `p`, that will be used to reproduce the audio file. After that, the code saves the audio file information using the package `Wave`, which contains the function `open()`. Using the variable `p`, the function creates a stream with the function `open()` included in the PyAudio package that will be used to reproduce the song. This variable, called `streamp`, will be initialized with the necessary information for reproducing the file, such as the sample rate, number of channels and the format. All this information will be obtained from `wf`, the variable which contains the audio information. After that, the script takes information from `wf` frame by frame, with a size of 1024, and reproduces it until the audio data is finished.

2.3 Listen_Song

`Listen_Song` is the function that records the song played by the user.

```
# create a pyaudio variable to record the sound
streamc = pc.open(format=pyaudio.paInt16, channels=2, rate=44100, input=True,
frames_per_buffer=1024)
frames = []

# record the sound until <Enter> is pressed
while True:
    datac = streamc.read(1024)
    frames.append(datac)
    if L:
        break
```

This function initializes a variable called `pc`, which is the same as `p` from the previous function. For that, it is used the function `PyAudio()` from the package `PyAudio`. After that, a stream variable is initialized, called `streamc`. This variable uses the function `open()` from the package `PyAudio`, and some additional information is included, such as the sample rate, the number of channels and the format. In a loop, that will end when the user presses the key `<Enter>`, the function records the audio frame by frame, which has a size of 1024. For



recording audio it uses the function `read`, applied to the variable `streamc`, included in the PyAudio package. When the user presses <Enter>, the function closes all the variables and terminates the recording.

2.4 Compare_Files

`Compare_Files` uses the functions from the script `soundProcessing.py` to apply the HPS algorithm and to apply the correlation for the comparison.

First, two variables are initialized, which are going to be used to save the frequency sequence of the synthesized song and the recorded audio. These variables are two arrays called `f0_orig` and `f0_reco`. After that, the function `Read_File` from the script `soundProcessing.py` is used to open both audio files, the synthesized song and the recorded audio. As this function returns the sample rate and the audio data, those variables are used in the `HPS_Algorithm` function from `soundProcessing.py`. Finally, the `Correlation` function from `soundProcessing.py` is used, to compare both audio files.

3. Synthesized Song

To compare the recorded audio with the perfect frequency sequence of the song, a synthesized song was created. This application creates a sequence that consists of the frequencies of the song's notes and it converts it into an audio file.

3.1 note

This function takes as arguments the frequency, the amplitude and the sample rate and it converts into a sinusoid, so that an audio file can be created out of those frequencies.

```
def note(freq, len, amp=10000, rate=44100):  
    t = linspace(0, len, len*rate)  
    data = amp*sin(2*pi*freq*t)  
    return data.astype(int16)
```

To create a variable with a specific length for each note `linspace` from NumPy is the function used. For example, if the played note is a quarter note (also called crotchet), the length is 1, if it is an eighth note (also called quaver) the length is 0.5, and so on. Then, to convert it into sinusoid, it is used `sin` from NumPy, and the frequency is multiplied by 2π and the time obtained with `linspace`.

3.2 main

Apart from the function mentioned, this script contains all the frequencies that the song is made of, and using a concatenation function, all the frequencies are concatenated and converted into an audio file.



The concatenation function used is called `concatenate`, which is also part of NumPy. And, finally, to create the audio file, it is used `wavfile.write`, from SciPy's input/output package.

4. Tuner

Apart from the application that is focused on the project's purpose, another application was made to visualize on screen the notes played by the user. For this application Parabolic Interpolation was used, as the results between HPS and this algorithm were quite different, and the second one had the option of processing in real-time. The script that contains the main functions for the tuner is called `findNote.py`, and its functions are used in the script `Tuner.py`.

4.1 Record

This function processes the audio that is recording in real-time. For this, it uses the Parabolic Interpolation algorithm. As the function processes audio in real-time, the audio that is being recorded has to be processed at the same time. To do that, the chunk that is recorded, which has a length of 1024, is saved and converted into an array, as the recorded piece is in binary format. After that, the algorithm applies the Hanning Window, performs Fourier Transform and finds the maximum peak.

```
# Information for the recording
stream = p.open(format=pyaudio.paInt16, channels=1, rate=44100, input=True,
frames_per_buffer=1024)
frames = []

# Record audio frame by frame
while True:
    data = stream.read(1024)
    dec = numpy.fromstring(data, numpy.int16); # convert binary data into
array
    indata = dec*hann(1024) # apply hanning window
    fft = abs(numpy.fft.rfft(indata))**2 # do Fourier transform

    peak = fft[1:].argmax() + 1 # find max peak

    # Parabolic Interpolation
    if peak != len(fft) - 1:
        y0, y1, y2 = numpy.log(fft[peak - 1:peak + 2:])
        x1 = (y2 - y0) * .5 / (2 * y1 - y2 - y0)
        freq = (peak + x1)*44100 / 1024 # find the frequency and
output it

    else:
        freq = peak * 44100 / 1024

    a.Convert_To_Note(freq) # obtained frequency's corresponding
musical note
    f0.append(freq)
```



```
if L:
    break
```

First of all, the function initializes the variable that will be used to record the audio. This variable is called `p`, and it uses `PyAudio()` from the package `PyAudio`. After that, the streaming variable is initialized using the variable `p` and the function `open()` from `PyAudio`. It also needs additional information such as, sample rate, number of channels and the format. Then, the function enters in a loop that will end when the user presses the key `<Enter>`. In this loop, the audio will be recorded in frames of the size of 1024. This frame, first, has to be converted into a normal format, which is done by using the function `fromstring()` from the package `NumPy`. When the format is the correct one, the frame is modified by applying a Hanning Window, function `hann()` from `SciPy`'s signal package, the Fourier Transform, function `fft.rfft()` from `NumPy`, and taking the absolute value. First it is used the function `argmax()` to obtain the maximum peak. To this peak, the function applies the Parabolic Interpolation method, which will obtain a single maximum peak depending on the surroundings of the peak obtained earlier. Finally, the maximum peak will be converted into its corresponding frequency by multiplying it with the sample rate and dividing it with the frame size, and the function will print the frequency and its corresponding musical note.

4.2 Convert_To_Note

`Convert_To_Note` focuses on converting the frequencies obtained when applying the Parabolic Interpolation to the audio into its equivalents for musical notes. For this, it is used a file which contains all the musical notes and its frequencies on a list. The audio recorded might not be the exact frequency of the note, to solve this problem, it was used an approximation which consists on dividing the note by $2^{10/1200}$.

```
a = 1.005792941          # 2**(10/1200)

# find corresponding note
for n in range(len(notes)):
    if float(notes[n][1])/a <= f0 <= float(notes[n][1])*a:
        print "Frequency: ", f0, "\tMusical Note: ", notes[n][0]
```

First thing is to open the file which contains the musical notes and their corresponding frequencies. After that, each line from the file is saved in an array to separate the frequency and the name of each musical note. The variable `a` is used to have a margin in the process of finding the corresponding musical note, as the notes played with the instrument aren't always the exact frequency of the note. After that, the function enters in a loop that will last until all the frequencies from the file are compared with the recorded note and the frequency and the corresponding musical note's name will be printed. If, by any chance, the frequency recorded doesn't correspond to any frequency from the file, despite of the margin, then, the frequency won't be printed.

CHAPTER 8

Testing

This chapter includes all the tests that have been carried out. The chapter is divided into two sections, the first one being the main application for the project, and the second one being the tuner explained previously.

This chapter also includes the notes that form the song chosen.

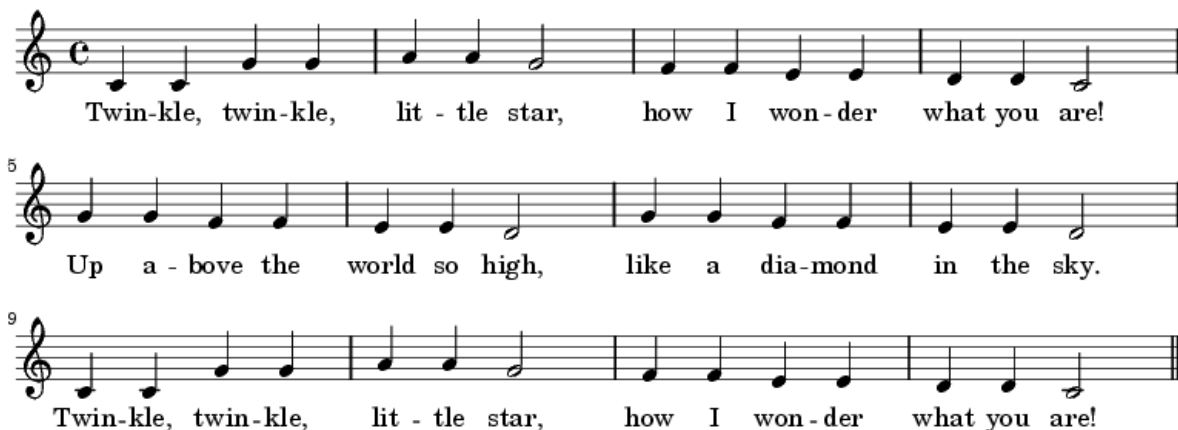
1. Application

1.1 Song

As mentioned through the memory, the song chosen to perform the tests for the application is “Twinkle, twinkle, little star”, a well-known English lullaby translated into several languages.

The decision to choose this song was because it is a very easy song to play, and that gives a better clarity when testing the application.

The following score or music sheet is the one used, as it can have variations depending on the octave and the musical notation.



Twin-kle, twin-kle, lit - tle star, how I won - der what you are!

Up a - bove the world so high, like a dia-mond in the sky.

Twin-kle, twin-kle, lit - tle star, how I won - der what you are!

Figure 8.1: Twinkle, twinkle, little star song's music sheet

The musical notes used for this song are: C (Do), D (Re), E (Mi), F (Fa), G (Sol) and A (La), and the corresponding frequencies are the following: 523.25, 587.33, 659.26, 698.46, 783.99, 880 and 987.77.

For this project, it was only taken the first part of the song.

1.2 Flute

This is the experiment that was made performing the song on a flute.

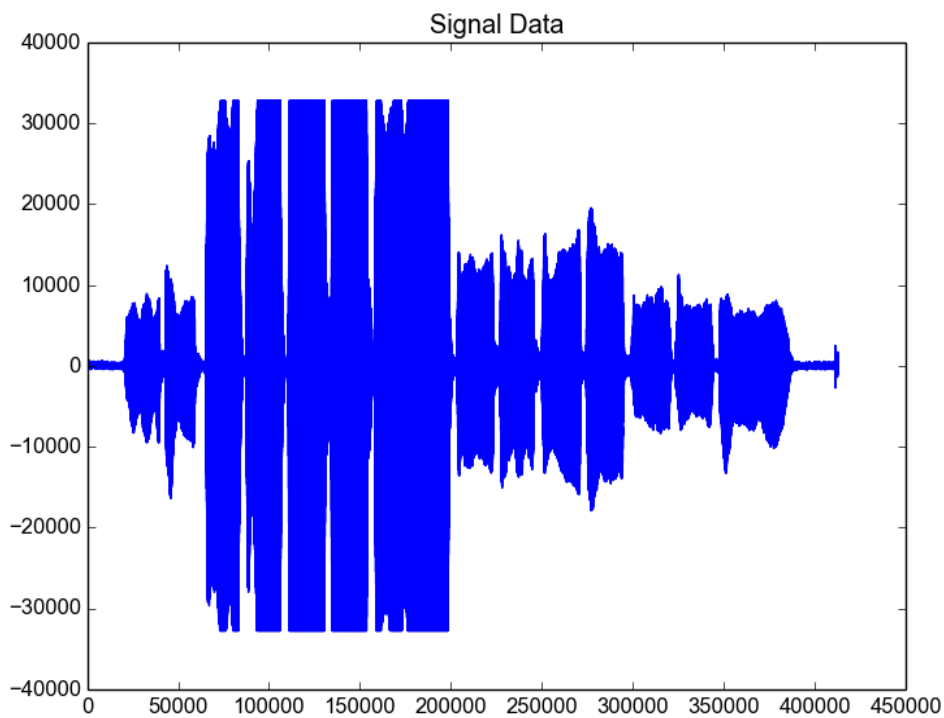


Figure 8.2: Signal data obtained from performing the song with the flute

This picture is the audio data recorded with the microphone without any modification. We can observe clearly that the notes are separated by gaps, which mean that there is a silence in between. The next picture is a zoom to the first note:

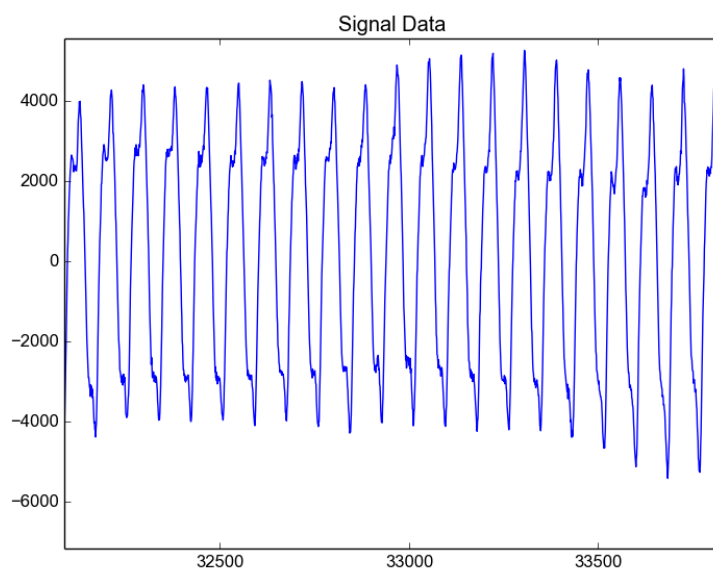


Figure 8.3: Signal data's zoom in the first note

In this picture we can see that the graph is made of sinusoid waves. To determine the frequency from this note, we can calculate the distance between two contiguous peaks and compute some operations. The operations are the following:



$$Freq = \frac{1}{dist * (1/f_s)}$$

In this case, it would be: $1 / (84 * (1 / 44100))$, and the result is 525 Hz. That frequency corresponds nearly to the note C3 (Do).

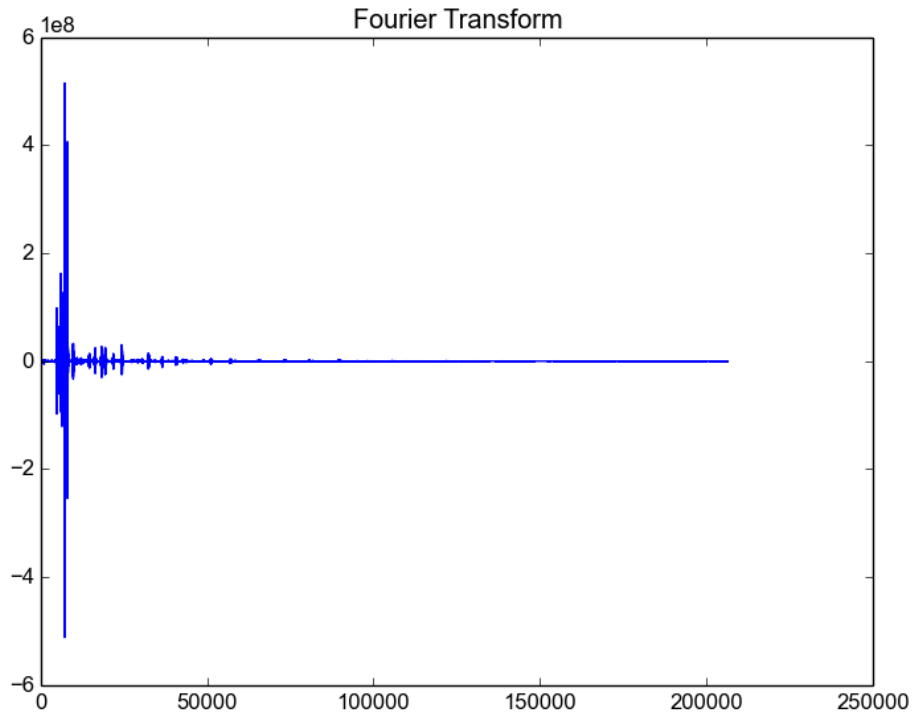


Figure 8.4: Fourier Transform of the signal data from the flute

The picture above represents the frequencies that appear in the audio recorded. To obtain this graph, the Fourier Transform was applied to the audio data.

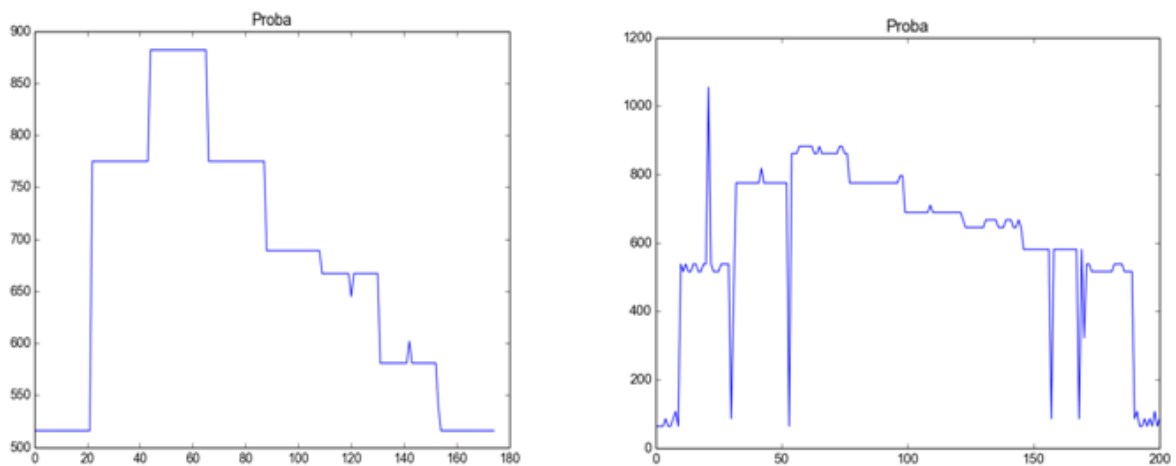


Figure 8.5: Comparison between synthesized audio's fundamental frequency sequence and recorded audio's fundamental frequency sequence (FLUTE)

The graph on the left shows the synthesized song's result after processing the audio. On the other hand, the graph on the right shows the results of the song performed with the flute. The results aren't exactly the same comparing with the synthesized audio, as, the recorded file could contain noise that can create distortions. However, the shape of the wave is very similar to the original one, which means that the algorithm used is appropriate for flute sounds.

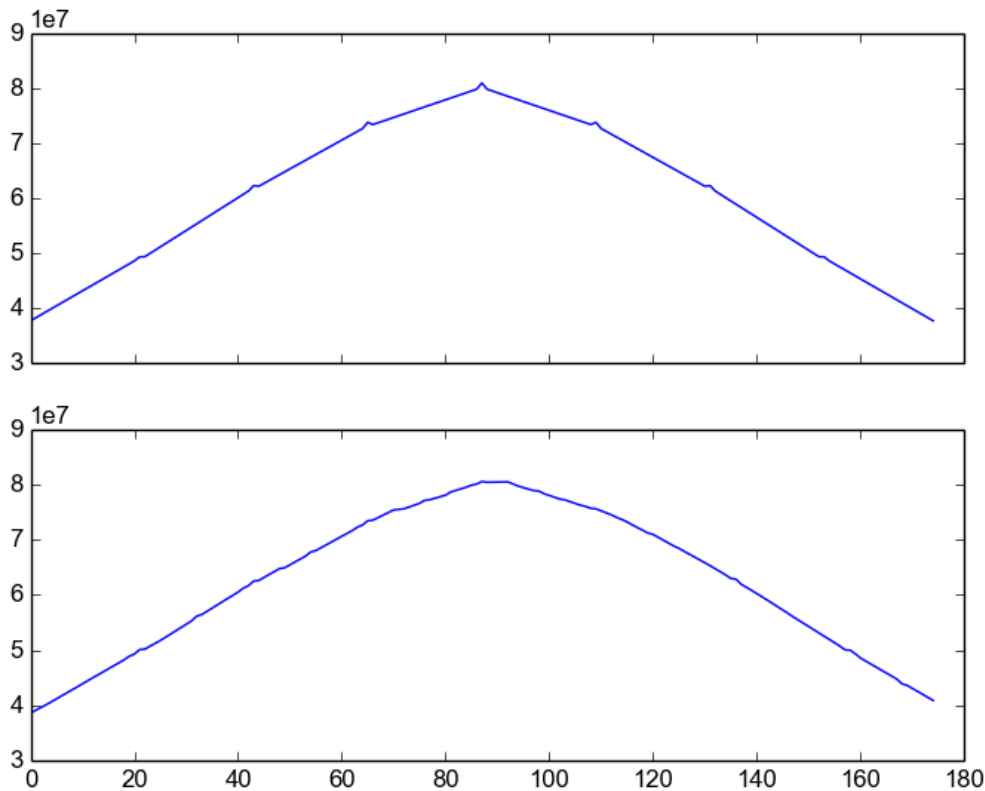


Figure 8.6: Comparison between synthesized audio's auto-correlation and recorded audio's correlation (FLUTE)

The graph above shows the difference between the synthesized audio's auto-correlation and the correlation between the synthesized audio and the recorded one. We can say that, the main shape of the correlation is similar to the auto-correlation, as, at the beginning, the graph goes up and then down. However, in the auto-correlation graph we can see small peaks that in the other graph don't appear. Additionally, the second graph's wave is irregular in some places.

1.3 Piano

This experiment shows the results obtained when performing the song with the piano:

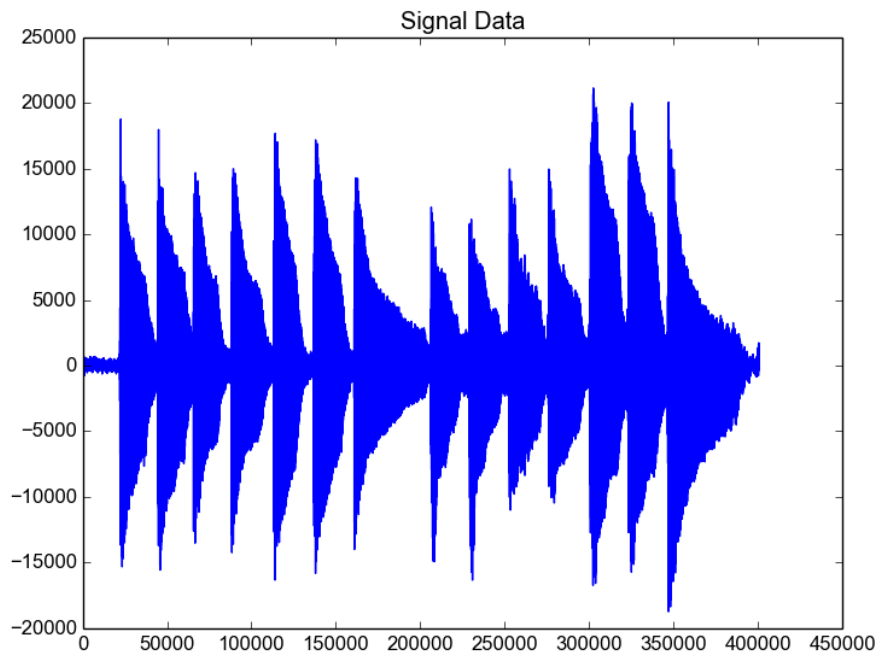


Figure 8.7: data obtained from performing the song with the piano

To begin with, we can compare this audio data to the one in the previous experiment, and we can observe the difference between them. An aerophone musical instrument depends on the air that the user throws through an opening. That means that the amplitude depends on the force of the air. On the other hand, with string instruments, the amplitude depends on the vibrations of the strings; at the moment of playing the key, the string moves with more power than at the end. That can be seen in this picture, where the beginning of each note has higher amplitude, and it decreases gradually until the next note is played.

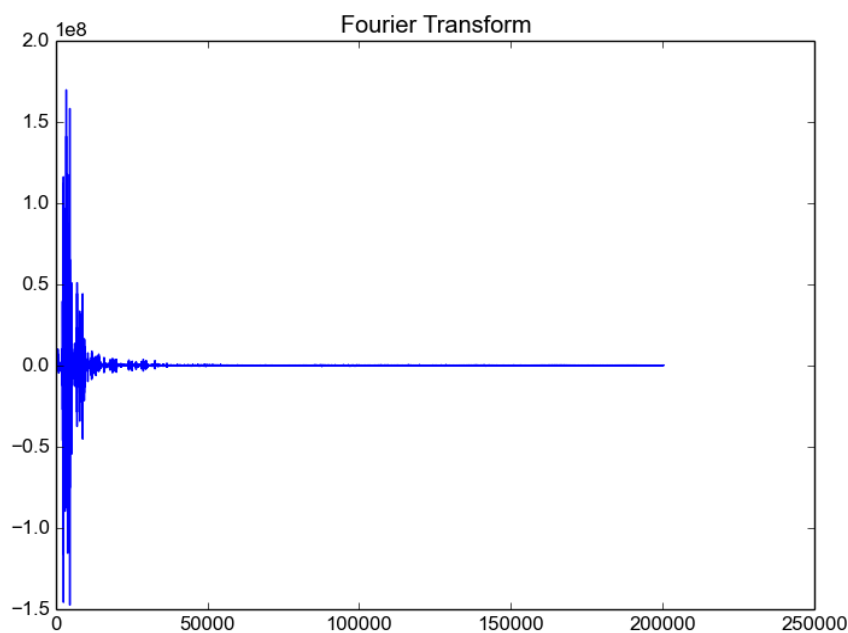


Figure 8.8: Fourier Transform of the signal data from the piano



The picture above shows the Fourier Transform of the recorded audio. This is used to process the audio and obtain the fundamental frequencies.

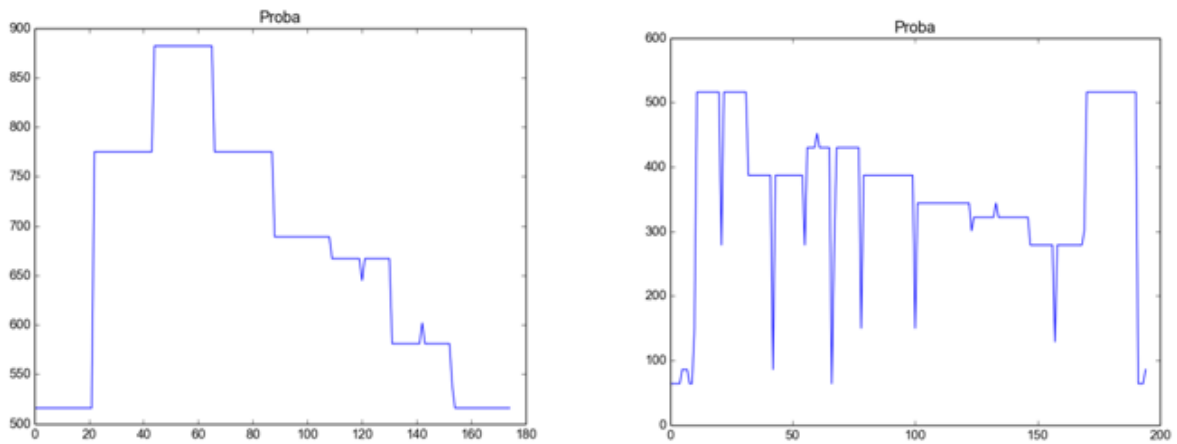


Figure 8.9: Comparison between synthesized audio's fundamental frequency sequence and recorded audio's fundamental frequency sequence (PIANO)

The same way as in the previous experiment, the first picture represents the fundamental frequencies of the synthesized audio, and the second picture shows the fundamental frequencies obtained when processing the recorded audio data. We can observe a difference between these two pictures, as well as between previous experiment's and this experiment's recorded audio's results. The problem starts in the second note of the song, which, in reality, it should be higher than the first one, but, when processing the audio, the algorithm detects a lower note. If we compare the frequency obtained and the table of frequencies, we notice that the note was detected correctly, but the algorithm detected the note in a lower octave than the actual one. That is a common problem for inharmonic musical instruments, and it is called "octave error" or "octave problem".

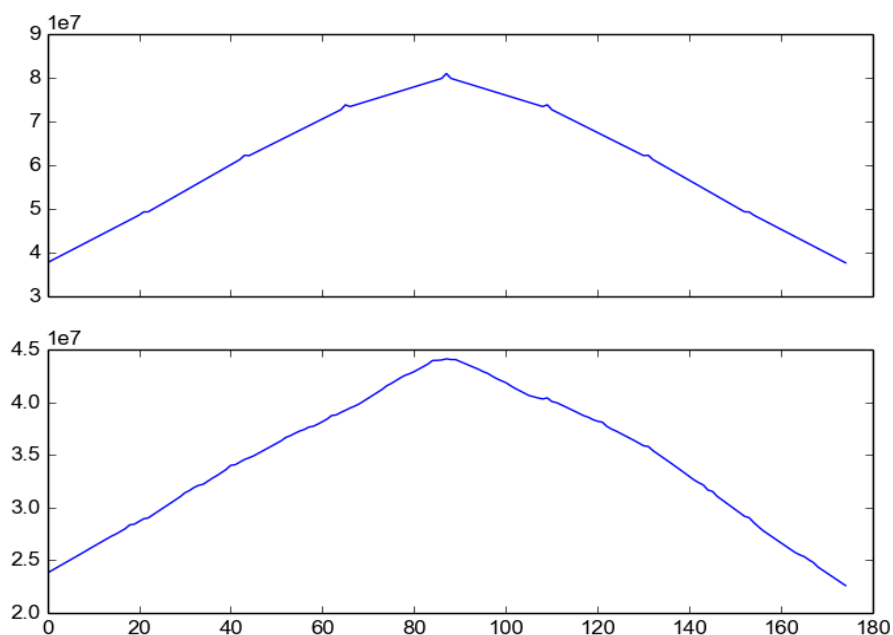


Figure 8.10: Comparison between synthesized audio's auto-correlation and recorded audio's correlation (PIANO)

This picture shows the difference between the auto-correlation of the synthesized audio and the correlation between the synthesized audio and the recorded one. The shape of the graph below is similar to the one on top, however, comparing to the result obtained with the flute, we can see that this one differs more than the previous one, that is because of the octave error.

2. Tuner

As explained earlier, the tuner is only a tool to know the musical note that it is being played. To perform these tests, apart from the song mentioned, random musical notes were used as well as continuous notes of a specific octave.

Something to take into account is that the PDA used in the tuner is Parabolic Interpolation, so the results might be different from the HPS one.

2.1 Flute

- **Octave with silence:**

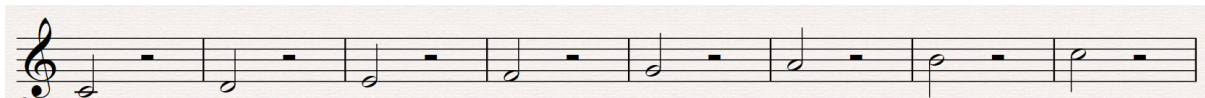


Figure 8.11: Do's Scale with silence

The first try with the flute was playing an entire octave from C5 (Do) to C6 (Do) making stops between two contiguous notes. The following test is what the application prints on screen. However, some lines were deleted to take a sample of 3 frequencies from each note.

Frequency: 526.247236642	Musical Note: C5
Frequency: 526.224334769	Musical Note: C5
Frequency: 526.159616679	Musical Note: C5
Frequency: 588.956842449	Musical Note: D5
Frequency: 590.643585642	Musical Note: D5
Frequency: 590.517785397	Musical Note: D5
Frequency: 658.406786313	Musical Note: E5
Frequency: 662.375983472	Musical Note: E5
Frequency: 662.203928416	Musical Note: E5
Frequency: 701.555209472	Musical Note: F5
Frequency: 701.042208649	Musical Note: F5
Frequency: 700.902345274	Musical Note: F5
Frequency: 781.118648009	Musical Note: G5
Frequency: 785.453010747	Musical Note: G5
Frequency: 784.960631155	Musical Note: G5
Frequency: 875.955704832	Musical Note: A6
Frequency: 875.135642098	Musical Note: A6
Frequency: 876.762218065	Musical Note: A6



Frequency: 983.297576904	Musical Note: B6
Frequency: 986.625169048	Musical Note: B6
Frequency: 987.200393186	Musical Note: B6
Frequency: 1041.42400976	Musical Note: C6
Frequency: 1042.52368264	Musical Note: C6
Frequency: 1046.24383968	Musical Note: C6

The graphical plot of this recorded piece is the following:

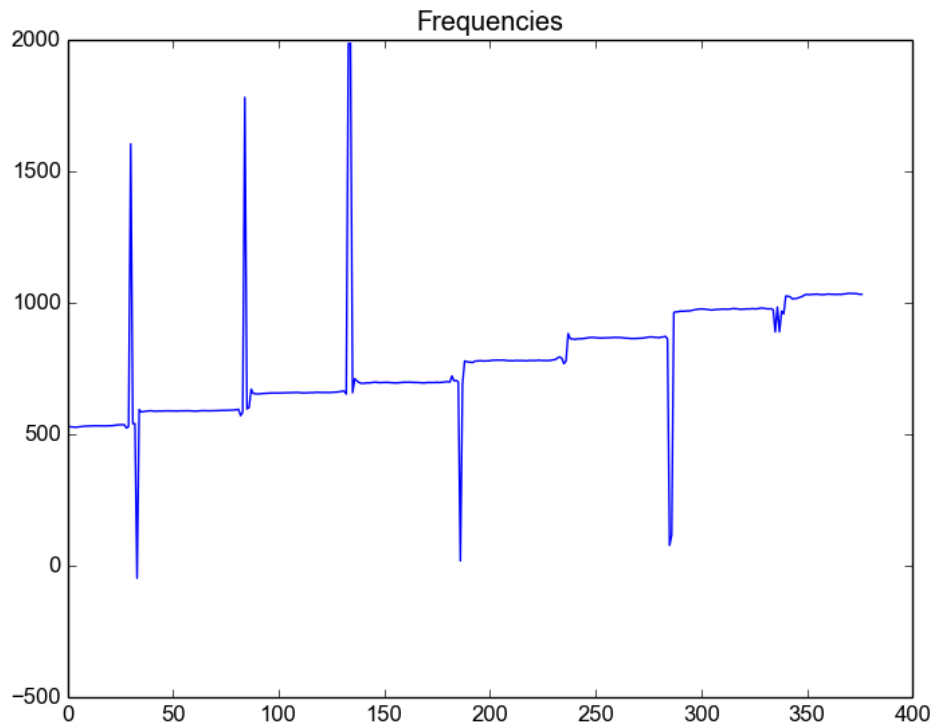


Figure 8.12: Fundamental frequency sequence of octave with silence

This graphic shows that, between two contiguous notes, there is a gap where the frequency goes up or down. The frequencies that go down mean that, at that moment, there wasn't any sound, which corresponds to the stops made between the notes. The frequencies that go up mean that the air was thrown in the flute when changing from one note to another, which makes a sound that doesn't correspond to any note.

- **Octave without silence:**



Figure 8.13: Do's Scale without silence

Continuing with the octave tests, the following try with the flute was playing the same octave from C5 (Do) to C6 (Do) without stops between notes. For this try, the frequencies and its corresponding names are the same as above, so it won't be shown the result that prints the application.

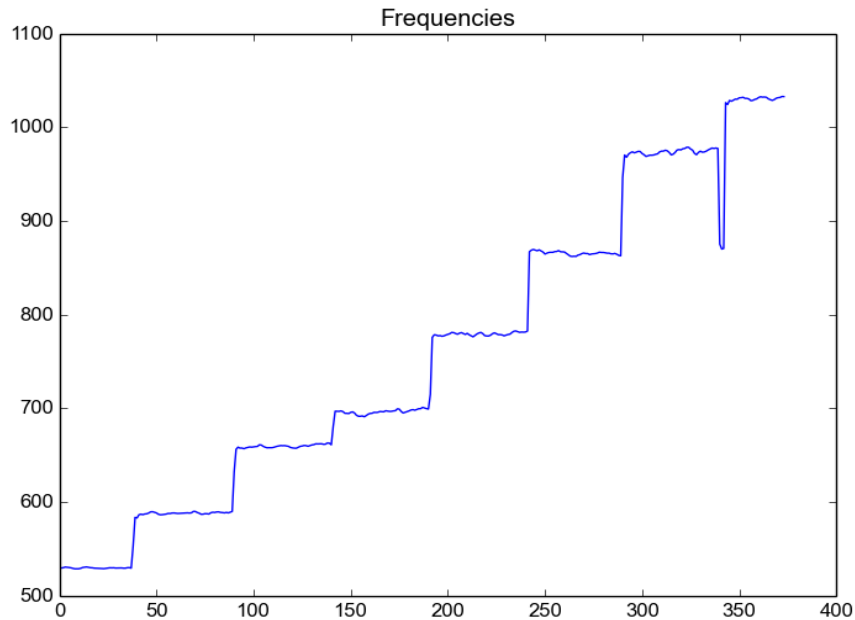


Figure 8.14: Fundamental frequency sequence of octave without silence

Comparing this graphic to the one made with silences between notes, it can be seen that there isn't any outstanding peak between two contiguous notes. In addition, it shows in a better way the ladder form of the graphic which corresponds to the idea of the octave.

- **Song:**



Figure 8.15: Twinkle, twinkle, little star song's first part (FLUTE)

The next test was made with the song of the project, which is “twinkle, twinkle, little star”. The following is the result printed by the application with some lines deleted due to the fact that the result is too long to put it here:

Frequency: 525.631523081	Musical Note: C5
Frequency: 525.572564781	Musical Note: C5
Frequency: 525.401649625	Musical Note: C5
Frequency: 525.616385601	Musical Note: C5
Frequency: 788.318749984	Musical Note: G5
Frequency: 780.096245818	Musical Note: G5
Frequency: 781.276024409	Musical Note: G5
Frequency: 780.993046956	Musical Note: G5
Frequency: 882.838535146	Musical Note: A6
Frequency: 875.577822102	Musical Note: A6
Frequency: 882.852953386	Musical Note: A6
Frequency: 883.699923059	Musical Note: A6
Frequency: 779.710306131	Musical Note: G5



Frequency: 781.132983449	Musical Note: G5
Frequency: 781.628988041	Musical Note: G5
Frequency: 782.379080532	Musical Note: G5
Frequency: 700.584601515	Musical Note: F5
Frequency: 698.254403266	Musical Note: F5
Frequency: 700.819653946	Musical Note: F5
Frequency: 698.557013487	Musical Note: F5
Frequency: 657.567920286	Musical Note: E5
Frequency: 661.379031402	Musical Note: E5
Frequency: 657.736365692	Musical Note: E5
Frequency: 656.321432355	Musical Note: E5
Frequency: 586.801040294	Musical Note: D5
Frequency: 586.937245306	Musical Note: D5
Frequency: 588.231736403	Musical Note: D5
Frequency: 588.756656659	Musical Note: D5
Frequency: 522.769037898	Musical Note: C5
Frequency: 522.792949687	Musical Note: C5
Frequency: 523.483139718	Musical Note: C5
Frequency: 524.096733887	Musical Note: C5

The following image (see picture 8.16) is the graphical representation of the performance:

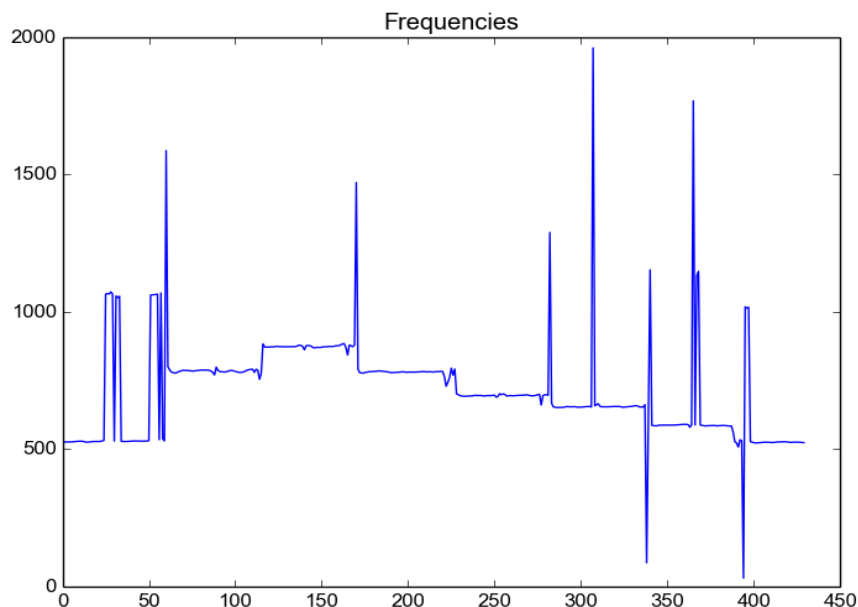


Figure 8.16: Song's fundamental frequency sequence (TUNER - FLUTE)

Even if the graphic isn't very precise, it can be seen that it has a ladder form with some jumps bigger than others. Between the first and the second notes the jump is big

because between the notes C5 (Do) and G5 (Sol) are 5 notes of distance. The other jumps are small because there only is one note of difference. As explained before, between two contiguous notes, they appear some peaks that could mean silences or air thrown between two notes.

2.2 Piano

- **Octave:**



Figure 8.17: Do's Scale

The first test with the piano was playing an octave from C5 (Do) to C6 (Do). This is the result that the application prints on screen. As in the other tests, some lines were deleted to make the result shorter:

Frequency: 523.635714521	Musical Note: C5
Frequency: 523.744862777	Musical Note: C5
Frequency: 522.737147255	Musical Note: C5
Frequency: 293.010867189	Musical Note: D4
Frequency: 292.965463528	Musical Note: D4
Frequency: 293.824974524	Musical Note: D4
Frequency: 329.101377619	Musical Note: E4
Frequency: 329.197428718	Musical Note: E4
Frequency: 329.997675646	Musical Note: E4
Frequency: 1048.55117457	Musical Note: C6
Frequency: 1047.52998599	Musical Note: C6
Frequency: 1045.28746697	Musical Note: C6
Frequency: 392.174443322	Musical Note: G4
Frequency: 391.744272351	Musical Note: G4
Frequency: 391.364691171	Musical Note: G4
Frequency: 440.548147057	Musical Note: A4
Frequency: 440.084143551	Musical Note: A4
Frequency: 441.004700839	Musical Note: A4
Frequency: 493.743881637	Musical Note: B4
Frequency: 494.253102287	Musical Note: B4
Frequency: 494.939419253	Musical Note: B4
Frequency: 523.849572947	Musical Note: C5
Frequency: 524.682059343	Musical Note: C5
Frequency: 522.781395046	Musical Note: C5

Observing the frequencies and the corresponding names on the result, we can conclude that something went wrong, as, being an octave, the notes and the frequencies should follow an increasing pattern. However, the note itself was well detected, the problem appears when detecting the octave.

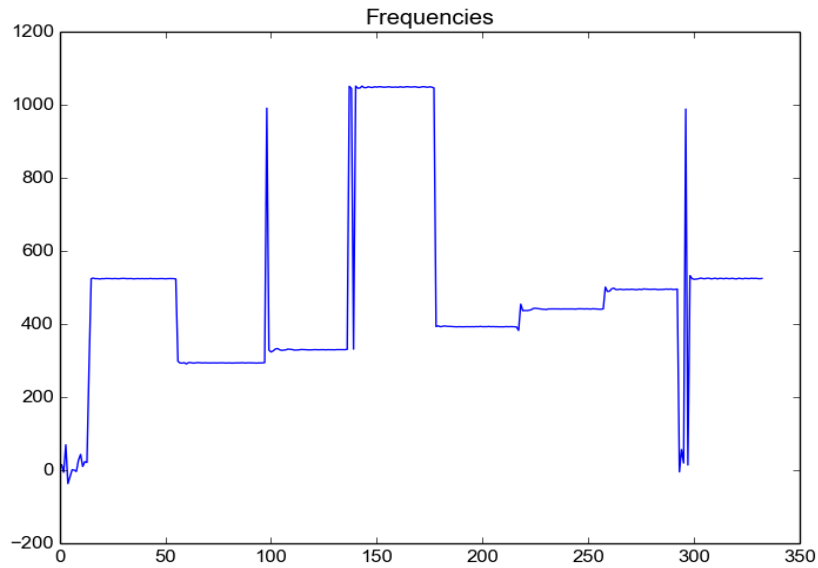


Figure 8.18: Do's Scale fundamental frequency sequence (PIANO)

The graphical representation shows better the errors that occurred in this test.

• **Song:**



Figure 8.19: Twinkle, twinkle, little star song's first part (PIANO)

The second test with the piano was playing the song “twinkle, twinkle, little star”.

Frequency: 522.079064831	Musical Note: C5
Frequency: 524.312346383	Musical Note: C5
Frequency: 523.961795776	Musical Note: C5
Frequency: 523.406730141	Musical Note: C5
Frequency: 392.617493509	Musical Note: G4
Frequency: 392.230856632	Musical Note: G4
Frequency: 391.983795191	Musical Note: G4
Frequency: 393.020316425	Musical Note: G4
Frequency: 441.234334883	Musical Note: A4
Frequency: 440.953248788	Musical Note: A4
Frequency: 439.861105811	Musical Note: A4
Frequency: 440.717828713	Musical Note: A4
Frequency: 391.568809265	Musical Note: G4
Frequency: 391.730325363	Musical Note: G4
Frequency: 391.789909422	Musical Note: G4
Frequency: 391.442072355	Musical Note: G4



Frequency: 348.811604952	Musical Note: F4
Frequency: 350.634331014	Musical Note: F4
Frequency: 349.064900064	Musical Note: F4
Frequency: 351.165899308	Musical Note: F4
Frequency: 329.570107036	Musical Note: E4
Frequency: 328.702621348	Musical Note: E4
Frequency: 329.152449987	Musical Note: E4
Frequency: 328.690307473	Musical Note: E4
Frequency: 292.828883515	Musical Note: D4
Frequency: 293.345225334	Musical Note: D4
Frequency: 292.924093785	Musical Note: D4
Frequency: 293.677714315	Musical Note: D4
Frequency: 523.237991751	Musical Note: C5
Frequency: 523.839721143	Musical Note: C5
Frequency: 523.173968454	Musical Note: C5
Frequency: 523.061814656	Musical Note: C5

As in the previous test, this one also shows that there has been a problem when detecting the octave, despite of the fact that the note is detected properly.

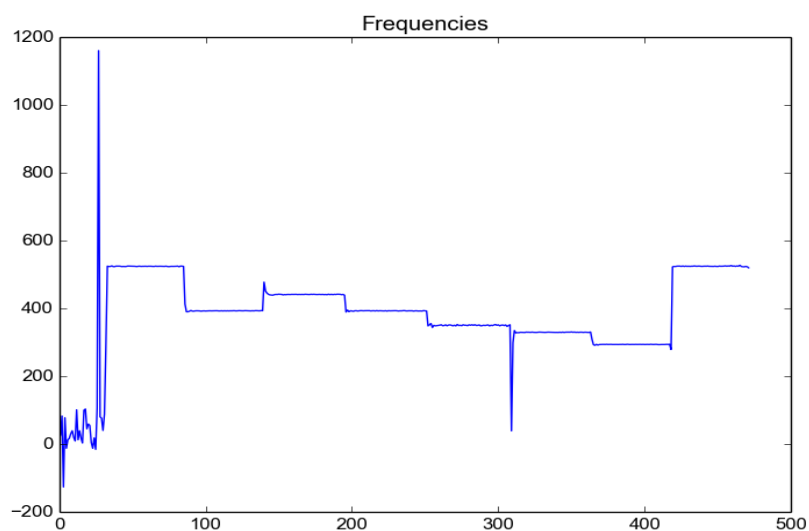


Figure 8.20: Song's fundamental frequency sequence (TUNER - PIANO)

The graphical representation also shows the problem that appears when playing the song on the piano comparing to the results obtained with the flute.

3. Conclusions

The comparison between the results obtained when performing with the flute and the piano are quite different.



In the first case, the flute, being a harmonic musical instrument, the results are the ones expected. Even if there can appear some variations that the algorithm doesn't detect as a note, overall, it is satisfactory.

However, the piano, being an inharmonic musical instrument, the results obtained aren't as good as the flute's ones. This happens when the harmonics of the fundamental frequency aren't exactly its multiples. An example could be that the first harmonic of the frequency 440 Hz, instead of being 880 Hz, it appears as 880.4 Hz. The inharmonicity in stringed instruments depends on the string's elasticity, length and stiffness.



CHAPTER 9

Improvements

This chapter will include possible future improvements that can be made in the application to have a better accuracy and robustness. It also will include some applications that could use this experiment to make a bigger project.

1. Improve Application

There are some areas where the application can be improved. These areas are the precision of the PDA (Pitch Detection Algorithm), the feedback system and the interface.

1.1 PDA Precision

At the moment of testing the application, it can be seen that it doesn't have much precision when processing the recorded audio.

When comparing the Pitch Detection Algorithms, HPS was thought to be the best one for this project, as it was oriented more to music than the other ones. Even if it is quite precise, it doesn't have the precision required to make the application outstanding. To improve this aspect, there are two ways:

- **Test HPS with more settings:** HPS can be implemented with different settings, such as a different frame size and different number of downsamples. This option has been tested in this project briefly, but hasn't been given much time, as the first time that the results were passable, the parameters were left like that.
- **Use another PDA:** There are a lot of Pitch Detection Algorithms that can be used instead of HPS. When implementing the Tuner, it was used Parabolic Interpolation, which gave better results than HPS and in real-time. When applying this algorithm for the main application of the project, the results were worse than using HPS. That doesn't mean that it can't be used. In addition, there are more algorithms apart from these two that haven't been tested in this project.

1.2 Feedback System

The feedback system couldn't be finished because of bad time management. The initial idea was to show the points where the performance was worse and remark the percentage that had been well performed.

To improve this, apart from the mentioned aspects, some other things could be included. The application could process sound in real-time and give a real-time feedback, such as make a beep sound when the user fails in a note.



1.3 Interface

As an experimental application, the interface shown is a very basic one, which only depends on the console. This can be improved by using a GUI package from Python. One GUI package that can be used would be Tkinter, but there are lots of options.

The application would have a welcoming page that explains the main idea of the application. Then, it would have a ComboBox from where the user could choose the song. After reproducing the song chosen, the application would show the whole process while recording the audio, such as a graphical view of the sound. It would also contain an option that would be Advanced Options. This CheckBox would open a menu where the user could choose different settings for the sound processing part, such as the frame size, the PDA algorithm used (this choice would involve implementing different PDA algorithms instead of using only HPS), the way that the feedback is given and so on. Finally, apart from the main application, the graphical interface would also include the tuner, which has been implemented in this project.

2. Possible Developments

In this section there will be a list of different applications that can be made based on this experiment. These applications are only some ideas that came to my mind, however, anyone reading this document can think of other possible developments.

2.1 Implement into NAO

As the original idea of the project, having more time, this experiment could be used to develop a system dependent of a humanoid robot, such as NAO, to have a more comfortable and easy interaction between user and application.

The initial work can be seen in the Appendix I, as, for this project, the implementation was started using NAO.

2.2 Polyphonic Sound

The experiment uses only monophonic sound, which means it can only be played one musical instrument at a time to process the sound. However, this project can be used to make an application that processes polyphonic sound. To do this, it is necessary having a more deep knowledge about sound processing, as separating the sources of the audio is really hard.

2.3 Songs with Lyrics

A possible development based on this experiment could have the additional aspect of processing the lyrics of the recorded audio, as well as the musical instrument sounds. This development can be separated into three parts:

- **Only lyrics:** An application that processes the lyrics of a song could be used to memorize the song itself, without taking into account the melody or the musical instruments that accompany the song. For this, the only method that would be used is speech recognition.



- **Lyrics and Melody:** This application would process the melody sang by the user and would do speech recognition to process the lyrics. This is a more complex implementation that the previous one. On the one hand, the melody is mixed with the lyrics, so separating both things is needed. On the other hand, speech recognition is based on formants, which are the representations of the vowels and consonants that are based on frequencies.
- **Lyrics and Musical Instruments:** The last option would be to have a musical instrument accompanying the lyrics. This would be a mix of the second point of this section and the experiment of this project. This application would need to separate the musical instrument's sound from the lyrics, and, after that, the lyrics and the melody.



CHAPTER 10

Conclusions

This chapter will explain personal experiences lived through the period of the development of the project and the learnt lessons that could be useful for future projects.

1. Project's Conclusions

The project's main idea has been successfully accomplished, which is developing an application capable of processing an audio recorded and comparing it with a synthesized version to see how similar they are. However, the application can be improved to achieve more accuracy and efficiency.

At the beginning, deciding to use HPS as the algorithm for the sound processing part was a good idea, as, comparing to other algorithms was the best for this project. However, the fact that it can't process sound in real-time was a huge disadvantage. I also tried using the Parabolic Interpolation algorithm, which has the ability to process sound in real-time, but, when the testing came, I realized that HPS has more precision, so I decided to make a tuner as an extra task.

The sound processing was the task that took me the most time to accomplish, and it is one of the reasons that I couldn't build a good feedback system. The initial idea was to compare the synthesized and the recorded audios and to show a percentage of how well it was performed and to point out the notes that hadn't been well played. However, in the end the application shows some graphics that represent the difference between both audios, which is not a good feedback system.

2. Personal Conclusions

2.1 Personal Experiences

Developing such a big project for the first time was an experience that couldn't be compared with any other projects that I had to make during my university period. Even if teachers taught us well, this project was in a completely different level. On the one hand, to come up with the idea without knowing certainly that it could be made was a very scary situation. On the other hand, having to develop everything on my own was also scary and, sometimes, discouraging. Nevertheless, with the help of my tutor, friends and family, I can say that, at the end, this has been a really fascinating experience that helped me grow and see things with another point of view.



The fact that the initial idea for the project had to be put aside caught me by surprise, and it was discouraging knowing that the hours spent with the robot were going to be useless. However, at this moment, after finishing the project, I can say that, despite the fact that it was quite depressing, it made me work harder and see things from another perspective.

The choice of developing something related to music was really attractive for me, as I always loved and will love music. Before starting with the project, the only things that I knew about music were playing an instrument and singing. Being able to implement this application gave me the chance of deepening my knowledge about music by learning the physics of it, and I found out that it is something that I like a lot.

2.2 Learnt Lessons

I would say that, the main learnt lesson during this project is the programming language that I used, which is Python. At the beginning I thought that, implementing the application in a language that I haven't used before would be a disadvantage, but, after writing small testing programs, I realized that it wasn't going to be a big problem, as the language itself is easy to learn and easy to code. However, as every programming language is different from the other ones, I had to get used to programming with this one, as it is quite different from the others that I have used.

Sound processing was also a pretty challenging task. Even if in the subject "Procesado Digital de Sonido e Imagen" we have learnt the basics, the level of the skills needed for this project was in a completely different level and the majority of the references are taken from the Internet and other similar projects.

This project also taught me that planning everything in advance is almost impossible and that leaving some space for unexpected events is necessary for this type of projects. The obstacles that I had to overcome weren't planned and make me waste precious time that, thankfully, I had, as I started pretty early with the project and during a period that I didn't have anything else to do.

Patience and perseverance are another two features that I gained throughout this period. If things are going wrong and you are getting nervous, furious or depressed, the last thing you should do is quitting. Sometimes, taking some time away from the problem and continuing with another task is a good solution to clear your mind and to start with regained forces.

APPENDIX I

NAO's Involvement

This Appendix will include a brief introduction about the NAO robot and its main features. It will also include the software needed to develop any project using the robot. It will also contain the first version of the project and it will explain the problems that happened during the first period that led to put the NAO aside and continue without it.

1. Introduction

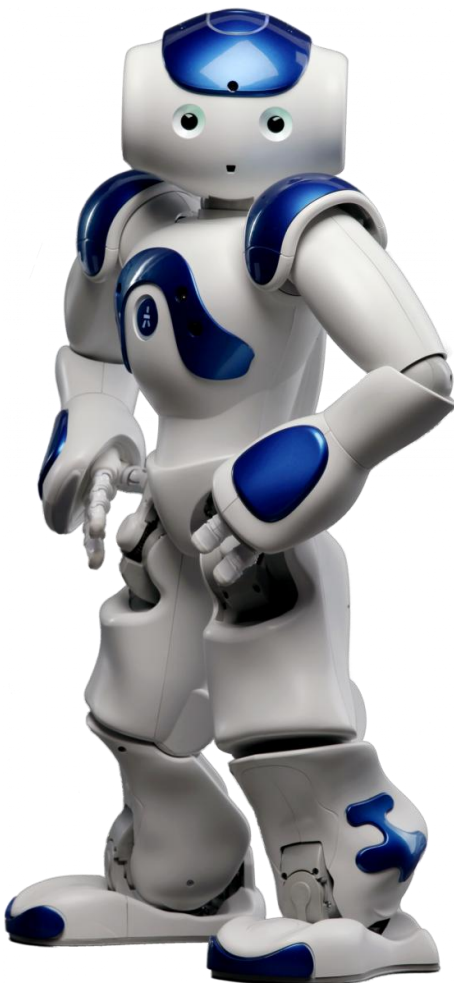


Figure A1.1: Nao

1.1 What is Nao?

Nao (see figure A1.1) is an autonomous, programmable humanoid robot developed by Aldebaran Robotics. The first version of the robot was released in 2006 and has been evolving until now.

Nao is used for many different purposes, such as investigation and research, academic purposes or medical purposes, this last one to teach autistic children in schools, as these children find the robot more relatable than the human beings.

1.2 Some properties

Nao has the appearance of a 5 year-old child. He is 58 cm tall and he weighs 4.3 kg.

He has 25 degrees of freedom, and the humanoid shape allows it to move and adapt to the world around him. It has an inertial unit which allows him to keep balance. It also has tactile sensors so that he can perceive if someone is touching him and he can respond to that touch. He can talk and hear through two loudspeakers and four microphones that are located in his head. The HD cameras located in his forehead and the mouth enables him to see the world and interact with the environment. Internet connectivity is mainly used to pass and compile the programs needed. The

Ethernet port is located in Nao's back and it is used to set the Wi-Fi connection. Once the connection is set, the Ethernet is no longer needed.

The picture below (see figure A1.2) shows the location of these sensors and actuators.

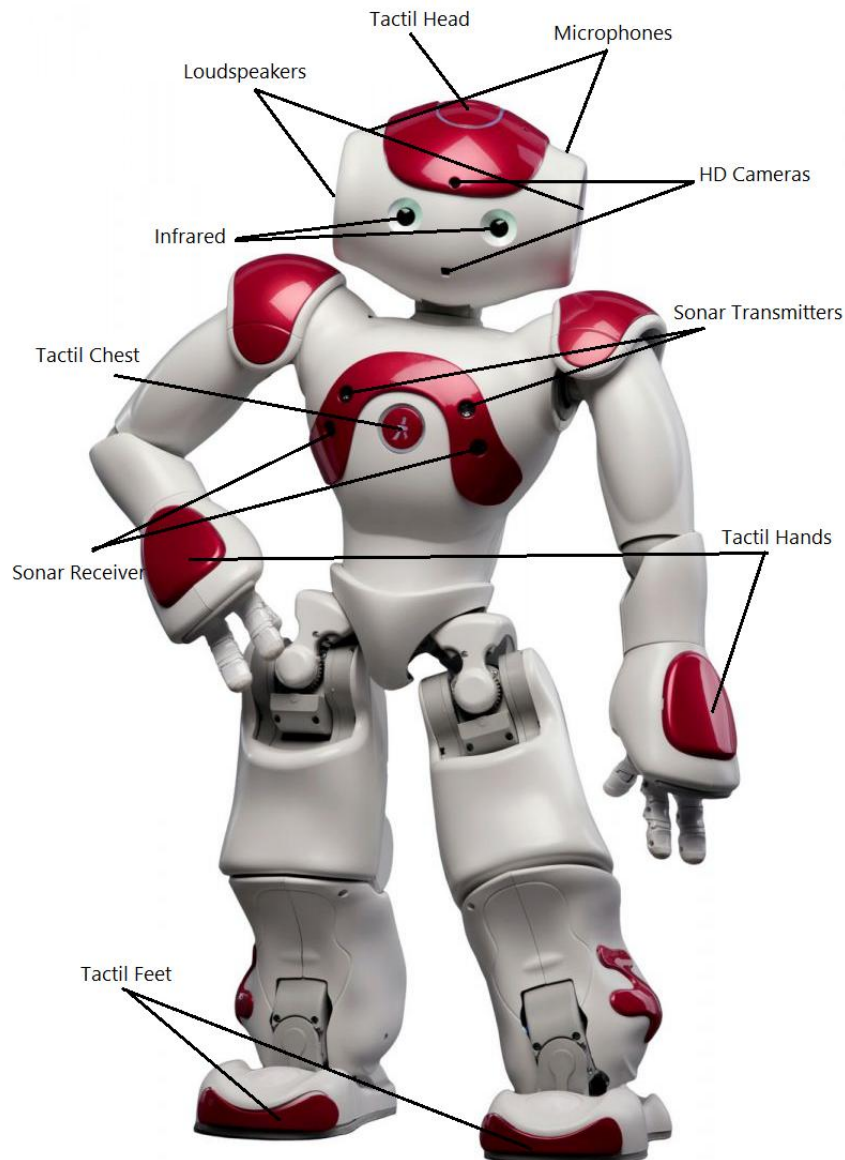


Figure A1.2: Nao's sensors and actuators

2. Software for Developers

There are different ways to develop a project using NAO. First of all, the user should have an account in Aldebaran Robotics (now called Softbank Robotics). This is due to the fact that all the programs needed to develop projects with NAO are in the Aldebaran website, and they cannot be downloaded without an account.

Once the user owns an account these are different programs that can be downloaded:

2.1 Choregraphe

Choregraphe is the main software created to develop Nao programs. It was specifically created to make the programming easier as it doesn't need many skills to create a simple animation or application. For bigger projects it can also be pretty useful, as veterans can modify the code as they wish.

The image below (see figure A1.3) shows the interface of the program.

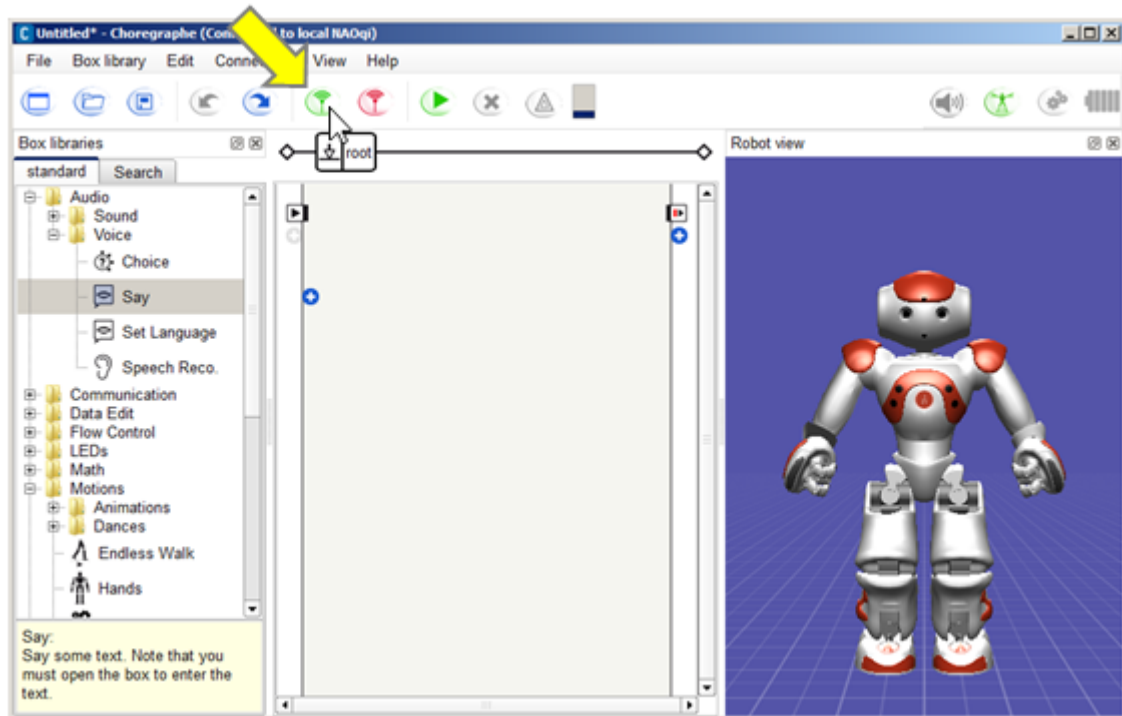


Figure A1.3: Choregraphe's graphical interface

Choregraphe works with connecting boxes. It contains many different types of boxes, such as for speaking, for walking, for speech recognition and much more. These boxes are located on the left side. To use a box, it has to be dragged to the white surface (the Flow Diagram) of the interface and link it to the “beginning”, as shown in the next picture (see figure).

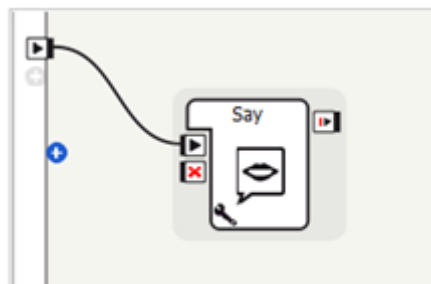


Figure A1.4: Say box in the Flow Diagram

To change the code of each box, double click on the box. Some boxes may be set up with more than one box, so, when double click, other boxes may appear. To go into the code, double click again the box that wants to be modified.

The following picture (see figure A1.5) shows the script of a random box.

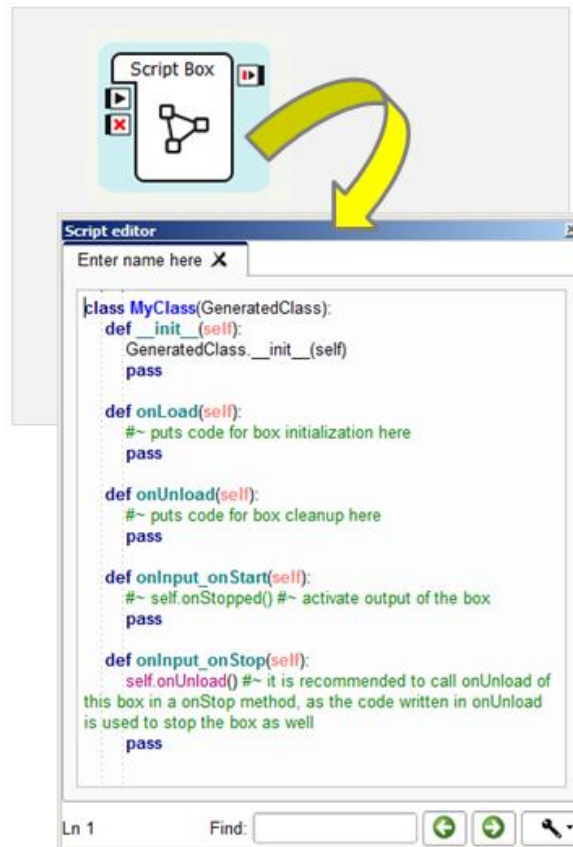


Figure A1.5: Script of a random box (Choregraphe)

Apart from using the default boxes that come with Choregraphe, anyone can create a customized box and write any code. To do so, right click on the “Flow Diagram” and click on “Add a new box”.

For this project, the first tool used to implement the program was Choregraphe, but, as it is kind of limited in some aspects, it had to be left aside and implement in the computer using the SDK that can be installed from the official website.

2.2 SDK

SDK is used to develop the projects from the PC, without using Choregraphe. This is more appropriate for big projects, as Choregraphe is a bit limited.

The programming languages that support Nao are Python, C/C++, Java and Javascript, so any program written in one of those programming languages can be compiled in the robot.

One of the disadvantages of the Nao is that it has very little storage capacity, so if the project needs a big library, such as SciPy, it is recommended to run it from the computer instead of installing it in the robot. This is something that happened during the development of this project, and some things had to be changed.



2.3 NAOqi

NAOqi is the main software that runs on the robot and controls it. The NAOqi Framework is the programming framework used to program Aldebaran robots.

With NAOqi, modules can be run independently from the robot or from the computer. Each module can be called from other modules. These modules are managed by a Broker that is integrated in the NAOqi. The modules that had been used in the first version of the project are the following:

- **ALAudioPlayer**: This module allows playing .mp3 or .wav files.
- **ALAudioDevice**: This module allows managing audio inputs and outputs.
- **ALMemory**: This allows storing data that can be accessed subscribing to it.
- **ALTextToSpeech**: This module makes the robot speak written phrases.
- **ALSpeechRecognition**: It makes the robot understand what humans say.

To create a remote project, proxies are used to create the modules. These are made by using NaoQi's ALProxy module, where the module's name, Nao's IP address and the port has to be added, as shown below (see figure A1.6):

```

module's name          Nao's port (default)
      |                 |
      v                 v
self.tts = ALProxy("ALTextToSpeech", "192.168.1.96", 9559)
                        |
                        v
                    Nao's IP address
  
```

Figure A1.6: Function ALProxy for Choregraphe script writer

The IP address can be obtained by pressing for some seconds the button on the chest of the robot. The port 9559 is the default one.

Each module has its functions that are called using the proxy variable, in the previous case (see figure A1.6) using “self.tts”.

3. Project's Development

3.1 Starting

To develop the first version of this project, “Computational Intelligence Group” provided the Nao robot called Endor and the username and password to download the necessary software to develop the project.

Before starting with the implementation, some introductory and exercise books [] were used to learn how to program the robot.

After getting to know the basics, Choregraphe was used to create the first prototype. At first, the advance was really slow, as it was a new way to do things and the environment was new, and, even if some introductory books were used, there were things that weren't explained in them.

3.2 Design

To develop an application with such dimensions, the first step is to have everything settled. For this, it is very important to design it.

As the project is based on the interaction between the human and the robot, this interaction has to be designed. For this project, the following interaction scheme was designed (see figure A1.7):

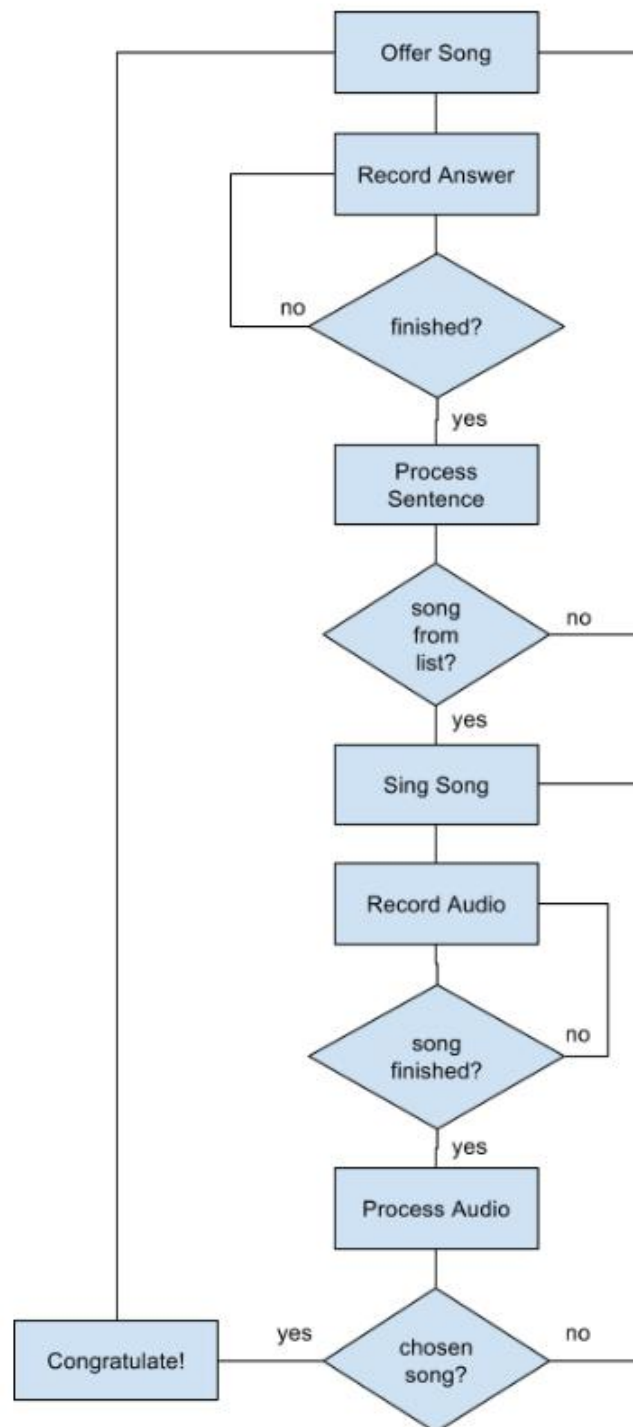


Figure A1.7: Interaction block diagram for Nao



This diagram is from the point of view of the robot. These are the steps explained more detailed:

1. **Greet the user and offer the list of songs:** When Endor detects that someone is near, he will greet the user and offer the songs from his database, so that the user can chose the one he wants to practice.
2. **Record the answer of the user:** After offering the list of the songs, the robot will stay recording the answer until the user makes a signal saying that the song was chosen.
3. **Decide if the song is in the list:** The next step is to determine if the recorded audio is a song from the list or any other thing. For this, Nao offers a function that does Speech Recognition.
4. **Sing the song or greet again:** If the recorded audio is part of the list, Endor will sing the selected song. On the other hand, if the recorded audio is something else, the algorithm will start from the beginning, and Endor will offer the songs again (go to step 1)
5. **Record the user's song:** Once the song is sung by the robot, it is the turn of the user. Endor will stay recording audio until the user makes a signal that represents the end of the song.
6. **Process recorded audio:** When the song is finished and recorded, the sound processing will begin.
7. **Congratulate or record again:** After processing the audio recorded, Nao will decide if the song was sung correctly or if there were mistakes. If it was sang correctly, Endor will congratulate the user, if not, he will sing again so that the user can listen again (go to step 4).

3.3 First Version - Choregraphe

As explained before, for the first version Choregraphe was used. The first task was to make Endor ask the user which song he/she wanted to practice, and provide him/her a list of available songs. After this, the user had to select one of the songs, and Endor had to be able to tell apart if the song the user selected was on the list or not. Then, Endor had to sing the song selected, and, finally, he would listen to the song sang or played by the user.

For this, there were used some default boxes, other default boxes with modified code and some others that were created from scratch.

The problem in the first version was that installing the SciPy in Endor was a completely difficult task, if not impossible. So this led to start working with Endor remotely, without using Choregraphe.

3.4 Second Version - Remote coding

With the necessary libraries installed on the computer and accessing Endor remotely, the development could continue. For this version, there were taken some codes from the boxes of the Choregraphe version to advance faster in the development. Nevertheless, the fact



that Endor was accessed remotely changed a lot of pieces of code. Fixing this took some extra time that wasn't planned.

The first task was to make the Choregraphe's version work. For this, it was important to know how NaoQi Framework worked, as a lot of modules had to be used to compensate the Choregraphe's automatic connection with the robot.

After advancing a little bit with the project another problem was found: The audio recorded by Endor was saved in the robot and it had to be copied or moved to the computer to process it. As too much time was spent with this issue without a solution, the decision to start implementing the application putting aside the robot was taken.

4. Problems

The problems that appeared during this period of time are explained briefly above, here they will be explained with more detail.

4.1 Nao Singing

Even if the first problem isn't exactly a problem, it is interesting to be mentioned as it took some time to solve it. At first, the idea was that the Nao could sing the melody by itself, without using an audio file. After searching on the Internet and watching some YouTube videos of Nao singing an e-mail was sent to the developer of one of those videos about how it was done, and the answer was that Nao couldn't sing by itself.

4.2 Library Installation

As said before, the first problem was that the library needed to start with the audio process couldn't be installed in the robot. After digging deep in this aspect, and with some external help, it was decided that it wasn't possible to continue using Choregraphe to develop the project.

Choregraphe only works with libraries that are installed directly in the Nao, and, as Nao doesn't possess a lot of storage capacity, the library needed, SciPy, couldn't be installed.

4.3 File Transfer

The last issue was that the file recorded by Endor couldn't be transferred to the computer to process it. For this there were used different libraries that allow scp from a python script, an example of this is "paramiko".

At first, the attempts of transferring with this library were made only using the robot. After spending some time trying and failing, the idea that maybe the library didn't work appeared, and the library was used to transfer a file from two different university computers. This last one worked, so it was kind of obvious that the problem was with the robot and not the library.



5. Written Code

5.1 main.py

```
#!/usr/bin/env python

from MyClass import MyClass

robot = MyClass()
robot.Introduction()
robot.Song_Choice()
robot.Sing_Song()
robot.Listen_Song()
#robot.Move_Files()
```

`main.py` is the script that calls all the functions in the order needed. It needs the class where the functions are implemented and an object called “robot” is used to do the calling.

5.2 MyClass.py

```
#!/usr/bin/env python

import time
import scipy
import paramiko
from naoqi import ALProxy

class MyClass():

    # Initialization of Proxy
    def __init__(self):

        self.tts = ALProxy("ALTextToSpeech", "192.168.1.96", 9559)
        self.asr = ALProxy("ALSpeechRecognition", "192.168.1.96", 9559)
        self.mem = ALProxy("ALMemory", "192.168.1.96", 9559)
        self.aup = ALProxy("ALAudioPlayer", "192.168.1.96", 9559)
        self.aud = ALProxy("ALAudioDevice", "192.168.1.96", 9559)
        self.aam = ALProxy("ALAutonomousMoves", "192.168.1.96", 9559)
        self.amp = ALProxy("ALMotion", "192.168.1.96", 9559)

        self.tts.setLanguage("English")
        self.asr.setLanguage("English")
        self.vocabulary = []

    def Introduction(self):

        self.amp.setBreathEnabled("Legs", False)
        self.amp.setBreathEnabled("Arms", False)

        #print self.aam.getExpressiveListeningEnabled()
        #print self.aam.getBackgroundStrategy()
        #self.aam.setBackgroundStrategy("none")
        #self.aal.setState("disabled")
```



```
self.tts.say("Hello!")
self.tts.say("Which song would you like to play?")
self.tts.say("Please, say the number of the song.")
#self.aal.setState("solitary")
#self.aam.setBackgroundStrategy("backToNeutral")

def Song_Choice(self):

    # opens the file which contains the songs available
    f = open("songs", 'r')
    songs = []

    # reads the file to get the songs
    for line in f:
        self.tts.say(line)
        words = line.split('.')
        for word in words:
            self.vocabulary.append(word)

    # adds the songs to the word recognition vocabulary
    self.asr.setVocabulary(self.vocabulary, False)

    # sunscribes to the event of recognizing the words
    self.asr.subscribe("Test_ASR")
    self.mem.subscribeToEvent("WordRecognized", "asrModule",
"onWordRecognized")
    time.sleep(10)

    # recolects the recognized words
    self.recolist = self.mem.getData("WordRecognized")

    # unsubscribes from the events of word recognition
    self.asr.unsubscribe("Test_ASR")
    self.mem.unsubscribeToEvent("WordRecognized", "asrModule")

def Sing_Song(self):

    self.aal.setState("disabled")
    print self.recolist[0]
    if self.recolist[0] == "1":
        self.aup.playFile("/home/nao/songs/twinkle_synth.wav")
    if self.recolist[0] == "2":
        self.aup.playFile("/home/nao/songs/happy_synth.wav")

def Listen_Song(self):

    self.tts.say("Now it's your turn.")
    self.aud.startMicrophonesRecording("/home/nao/proba.wav")
    time.sleep(5)
    self.aud.stopMicrophonesRecording()
    self.aup.playFile("/home/nao/proba.wav")

def Move_Files(self):

    ssh = paramiko.SSHClient()
    ssh.load_system_host_keys()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy)
```



```
print "Starting connection"
ssh.connect('192.168.1.96', 9559, 'nao', 'nao24')
print "Connected"
#sftp = ssh.open_sftp()
#print "aaa"
#sftp.get("/home/nao/proba.wav", "/home/NAO/proba.wav")
#sftp.close()
ssh.close()
```

`MyClass.py` is the script where all the functions are implemented. At the beginning of the script there are the libraries that are used in the script. `scipy` is the mathematical library, that would be used to process the audio recorded. `paramiko` is used to make a ssh connection to transfer files using `scp`. `NAOqi` is the library that includes all the modules from Nao.

To start with, whenever an object of this class is created, the `__init__` function is called. This function initializes all the proxies that are used during the program. The language is also set in the `__init__` function as well as a vector to save the list of the songs that Nao will play, called `vocabulary`.

With the `Introduction` function Nao greets the user and asks him/her which song he/she wants to play. The other lines are used as an attempt of leaving the Nao still, but they don't work so they don't need to be taken into account. Everything that has to do with Nao talking uses `ALTextToSpeech` module.

`Song_Choice` is where Nao reads the song list from a file, saves them on the vector variable initialized in `__init__` and lists them so that the user can choose one. After this, the user selects a song and Nao decides if it belongs to the list or it doesn't exist. If it doesn't exist, he, again, lists the songs available until the user selects a song of the list. The user has 10 seconds to select the song (the idea was to let the user say the song and, with a signal such as a clap that Nao was capable of knowing that the song was chosen, but, as finally Nao wasn't used, this idea wasn't developed). To recognize the song, Nao uses the `ALSpeechRecognition` module.

After `Song_Choice`, Nao sings the song using `Sing_Song` function. This function uses the `ALAudioPlayer` module to reproduce the audio file located in the Nao (as transferring the file wasn't working). The list of songs has 2 songs "twinkle, twinkle, little star" and "happy birthday", using the `vocabulary` vector.

After singing the song, Nao waits until the user sings it back using the function `Listen_Song`. In this case, the user has 5 seconds to sing or play the song (the same as the song choice happens here). For recording the audio `ALAudioDevice` is used.

Finally, the code includes the attempt of making the file transfer. This function is called `Move_Files` and it uses the `paramiko` library. First there are some initializations, after that it is made the connection with the Nao by adding the IP address, the port, the login user and the password, but, as said before, it doesn't work.



APPENDIX II

Full Code

1. Sound Processing

1.1 soundProcessing.py

```
#!/usr/bin/python

import time
from numpy import fft, log, copy, sin, pi, array, argmax
from scipy.io import wavfile
from scipy.signal import decimate, hann
from matplotlib import pyplot
from scipy.signal import correlate as corr_s
from scipy.signal import convolve as conv_s

class soundProcessing():

    # This function reads the audio file and plots results
    def Read_File(self, file_name, synth):

        # read audio file depending on mono or stereo
        if synth == 0:
            self.fs, self.data = wavfile.read(file_name)
        else:
            self.fs, self.data2 = wavfile.read(file_name)
            self.data = self.data2.sum(axis=1) / 2

        # plot original audio file
        pyplot.figure(1)
        pyplot.title("Signal Data")
        pyplot.plot(self.data)
        pyplot.show()

        # plot fourier transform of the original audio file
        pyplot.figure(1)
        pyplot.title("Fourier Transform")
        pyplot.plot(fft.rfft(self.data))
        pyplot.show()

        return [self.fs, self.data]

    # It processes the audio data using HPS algorithm
    def HPS_Algorithm(self, data, fs):
```



```
# Initialize values
SAMP = 2048                                # sample length
f_start = 0                                # start of fragment
f_end = SAMP                                # end of fragment
N = float(len(data))                        # length of signal
n_frag = len(data) / SAMP # number of fragments
f0 = []                                     # list to save the
fundamental frequencies

# Harmonic Product Spectrum
for i in range(n_frag):
    frag = data[f_start:f_end]              # take a fragment of the
file                                         # take a fragment of the
    frag_win = frag * hann(SAMP)            # apply a hanning window to
the fragment                                # apply a hanning window to
    X = abs(fft.rfft(frag_win))              # do the fourier
transform                                   # do the fourier
    hps = copy(X)                            # copy to do the
downsample                                  # copy to do the

    # downsampling
    for h in range(2, 6):
        downsample = decimate(X, h)
        hps[:len(downsample)] += downsample

    # find the position of the max peak and convert to freq
    peak_pos = argmax(hps[:len(downsample)])
    freq = fs * peak_pos / SAMP

    f0.append(freq)                          # save the frequencies on a list

    f_start += SAMP
    f_end += SAMP

# visualize the frequency list
pyplot.figure(1)
pyplot.title("Fundamental Frequency Sequence")
pyplot.plot(f0)
pyplot.show()

return [f0, n_frag]

# It processes the audio data using PI algorithm
def Parabolic_Interpolation(self, data, fs):

    # Initialize values
    SAMP = 2048                                # sample length
    f_start = 0                                # start of fragment
    f_end = SAMP                                # end of fragment
    N = float(len(data))                        # length of signal
    n_frag = len(data) / SAMP # number of fragments
    f0 = []                                     # list to save the fundamental frequencies

    # Fourier Transform with Parabolic Interpolation
    for i in range(n_frag):
```



```

    frag = data[f_start:f_end]                # take a fragment of
the file                                     #
    frag_win = frag * hann(SAMP)             # apply a hanning
window to the fragment                       #
    X = abs(fft.rfft(frag_win))**2           # do the fourier
transform                                     #
    peak = X[1:].argmax() + 1                # find peak position
with argmax function                          #
    # do parabolic interpolation
    if peak != len(X) - 1:
        y0, y1, y2 = log(X[peak - 1:peak + 2:])
        x1 = (y2 - y0) * .5 / (2 * y1 - y2 - y0)
        freq = (peak + x1) * 44100 / 1024    #
find the frequency and output it
    else:
        freq = peak * 44100 / 1024

    f0.append(freq)                          # save the frequencies on a list

    f_start += SAMP
    f_end += SAMP

# visualize the frequency list
pyplot.figure(1)
pyplot.title("Fundamental Frequency Sequence")
pyplot.plot(f0)
pyplot.show()

return [f0, n_frag]

# Creates a synthesized audio file from fundamental frequencies
def Create_File(self, f0, n_frag, fs):

    song = []

    # convert the frequencies into audio
    for n in range(n_frag):
        for t in range(1, 2048):
            phase = sin(2 * pi * f0[n] * t / fs)
            song.append(phase)

    # create final audio file
    npsong = array(song)
    wavfile.write('f0.wav', fs, npsong)

# Applies correlation to compare two fundamental frequency sequences
def Correlation(self, orig, reco):

    # auto-correlate the synthetised audio
    corr_orig = corr_s(orig, orig, mode='same')    # correlation

    # correlate the synthetized audio with the recorded one
```




```

corr_reco = corr_s(orig, reco, mode='same')           # correlation

#conv_orig = conv_s(orig, orig, mode='same') # convolution
#conv_reco = conv_s(orig, reco, mode='same') # convolution

# visualize the results
fig1, (ax_corr_o, ax_corr_r) = pyplot.subplots(2, 1, sharex=True) #
correlation
ax_corr_o.plot(corr_orig)
ax_corr_r.plot(corr_reco)
fig1.show()

#fig2, (ax_conv_o, ax_conv_r) = pyplot.subplots(2, 1, sharex=True)
# convolution
#ax_conv_o.plot(conv_orig)
#ax_conv_r.plot(conv_reco)
#fig2.show()

# calculate the delay between the two signals
delay_reco = int(len(corr_reco)/2) - argmax(corr_reco)
print delay_reco
print len(reco)

# delete the delay of the recorded audio signal
delay_reco *= 2
while delay_reco > 0:
    reco.pop(0)
    delay_reco -= 1

# correlate again without delay
corr_reco = corr_s(orig, reco, mode='same')

#conv_reco = conv_s(orig, reco, mode='same')

delay_reco = int(len(corr_reco)/2) - argmax(corr_reco)

# visualize the results without delay
fig1, (ax_corr_o, ax_corr_r) = pyplot.subplots(2, 1, sharex=True) #
correlation
ax_corr_o.plot(corr_orig)
ax_corr_r.plot(corr_reco)
fig1.show()

#fig2, (ax_conv_o, ax_conv_r) = pyplot.subplots(2, 1, sharex=True)
# convolution
#ax_conv_o.plot(conv_orig)
#ax_conv_r.plot(conv_reco)
#fig2.show()

raw_input('Press <ENTER> to continue')
```

1.2. MyClass.py

```
#!/usr/bin/env python
```



```

import time
import scipy
import pyaudio
import wave
import thread
from soundProcessing import soundProcessing

class MyClass():

    # Initialization of variables
    def __init__(self):

        self.vocabulary = []
        self.s = soundProcessing()

    # Let user select a song from the list
    def Song_Choice(self):

        print "\n\n\t\tMelody Training"
        print "\n\n ***** \n\n"
        print "Choose a song\n\n"

        choice = 1;

        # opens the file which contains the songs available
        f = open("songs", 'r')

        # reads the file to get the songs
        for line in f:
            print line
            words = line.split('.')
            for word in words:
                self.vocabulary.append(word)

        # while the number entered is wrong, keeps asking the correct
        number
        while choice == 1:
            self.song = raw_input("\nWrite the number of the song and
            press enter: ")

            if self.song in self.vocabulary:
                choice = 0

            else:
                print "\nThat song doesn't exist.\n"
                choice = 1

        # Application reproduces the chosen song
        def Sing_Song(self):

            p = pyaudio.PyAudio()

            # open the audio file selected
            if self.song == "1":

```



```

        print "\nYou chose Twinkle twinkle little star."
        wf = wave.open("twinkle_synth.wav", 'rb')

        # create a pyaudio type variable to reproduce the sound
        streamp = p.open(format=p.get_format_from_width(wf.getsampwidth()),
channels=wf.getnchannels(), rate=wf.getframerate(), output=True)
        datap = wf.readframes(1024)

        # while the file doesn't reach the end, keep reproducing the audio
file
        while datap != '':
            streamp.write(datap)
            datap = wf.readframes(1024)

        streamp.stop_stream()
        streamp.close()
        wf.close()

        p.terminate()

    def input_thread(self, L):

        raw_input()
        L.append(None)

    # Record the user's song
    def Listen_Song(self):

        a = MyClass()
        pc = pyaudio.PyAudio()

        print "\n\nNow it's your turn.\n\n"

        # create a pyaudio variable to record the sound
        streamc = pc.open(format=pyaudio.paInt16, channels=2, rate=44100,
input=True, frames_per_buffer=1024)
        frames = []

        print "\t* Start recording *"

        print "\n Press <Enter> to finish the recording."
        L = []
        thread.start_new_thread(a.input_thread, (L,))

        # record the sound until <Enter> is pressed
        while True:
            datac = streamc.read(1024)
            frames.append(datac)
            if L:
                break

        print "\n\t* Done recording *"

        streamc.stop_stream()

```



```

        streamc.close()

        pc.terminate()

        # save the recorded data in an audio file
        wfc = wave.open("recorded.wav", 'wb')
        wfc.setnchannels(2)
        wfc.setsampwidth(pc.get_sample_size(pyaudio.paInt16))
        wfc.setframerate(44100)
        wfc.writeframes(b''.join(frames))
        wfc.close()

    def Compare_Files(self):

        f0_orig = []
        f0_reco = []

        # Read the synthesized and the recorded audio files
        fs_orig, data_orig = self.s.Read_File("twinkle_synth.wav", 0)
        fs_reco, data_reco = self.s.Read_File("piano_recorded.wav", 1)

        # Process the synthesized and recorded audio data
        f0_orig, n_frag_orig = self.s.HPS_Algorithm(data_orig, fs_orig)
        f0_reco, n_frag_reco = self.s.HPS_Algorithm(data_reco, fs_reco)

        self.s.Create_File(f0_reco, n_frag_reco, 44100)

        # Compare synthesized and recorded results
        self.s.Correlation(f0_orig, f0_reco)

```

1.3 main.py

```

#!/usr/bin/env python

from MyClass import MyClass

app = MyClass()
app.Song_Choice()
app.Sing_Song()
app.Listen_Song()
app.Compare_Files()

```

2. Tuner

2.1 findNote.py

```

#!/usr/bin/python

import struct
import pyaudio
from matplotlib import pyplot
import wave

```



```

import thread
import numpy
from scipy.signal import hann, decimate
from decimal import Decimal

class findNote():

    def input_thread(self, L):

        raw_input()
        L.append(None)

    # Records user's audio and prints the frequency and musical note
    def Record(self):

        a = findNote()
        p = pyaudio.PyAudio()
        f0 = []

        # Information for the recording
        stream = p.open(format=pyaudio.paInt16, channels=1, rate=44100,
input=True, frames_per_buffer=1024)
        frames = []

        print "\t* Start recording *"

        print "\n Press <Enter> to finish the recording."
        L = []
        thread.start_new_thread(a.input_thread, (L,))

        # Record audio frame by frame
        while True:
            data = stream.read(1024)
            dec = numpy.fromstring(data, numpy.int16); # convert
binary data into array
            indata = dec*hann(1024) # apply hanning window
            fft = abs(numpy.fft.rfft(indata))**2 # do Fourier
transform

            peak = fft[1:].argmax() + 1 # find max peak

            # Parabolic Interpolation
            if peak != len(fft) - 1:
                y0, y1, y2 = numpy.log(fft[peak - 1:peak + 2:])
                x1 = (y2 - y0) * .5 / (2 * y1 - y2 - y0)
                freq = (peak + x1)*44100 / 1024 # find the
frequency and output it

            else:
                freq = peak * 44100 / 1024

            a.Convert_To_Note(freq) # obtained frequency's
corresponding musical note
            f0.append(freq)

```



```

        if L:
            break

        frames.append(data)

    stream.stop_stream()
    stream.close()

    p.terminate()

    # save recorded audio into an audio file
    wf = wave.open("recorded.wav", 'wb')
    wf.setnchannels(1)
    wf.setsampwidth(p.get_sample_size(pyaudio.paInt16))
    wf.setframerate(44100)
    wf.writeframes(b''.join(frames))
    wf.close()

    pyplot.figure(1)
    pyplot.title("Frequencies")
    pyplot.plot(f0)
    pyplot.show()

    return f0

# Read note's file and find frequency's corresponding note
def Convert_To_Note(self, f0):

    lines = []
    notes = []
    f = open("notes", "r")

    # save file's lines in an array
    for line in f:
        lines.append(line.split("\n"))

    # separate notes and frequencies from file
    for words in lines:
        l = words[0].split("\t")
        notes.append(l)

    a = 1.005792941          # 2**(10/1200)

    # find corresponding note
    for n in range(len(notes)):
        if float(notes[n][1])/a <= f0 <= float(notes[n][1])*a:
            print "Frequency: ", f0, "\tMusical Note: ",
notes[n][0]

```

2.2 tuner.py

```

from findNote import findNote

f0 = []
t = findNote()

```



```
f0 = t.Record()
```

3. Synthesize Sound

3.1 twinkle_S.py

```
#!/usr/bin/python

from numpy import linspace, sin, pi, int16, concatenate, array
from scipy.io import wavfile

def note(freq, len, amp=10000, rate=44100):
    t = linspace(0, len, len*rate)
    data = amp*sin(2*pi*freq*t)
    return data.astype(int16)

tC = note(523.25, 0.5)
tC2 = note(523.25, 1)
tD = note(587.33, 0.5)
tE = note(659.26, 0.5)
tF = note(698.46, 0.5)
tG = note(783.99, 0.5)
tG2 = note(783.99, 1)
tA = note(880, 0.5)
t = note(0, 0.01)

twinkle = concatenate((tC, t, tC, t, tG, t, tG, t, tA, t, tA, t, tG2, t, tF, t,
tF, t, tE, t, tE, t, tD, t, tD, t, tC2), axis=1)

nptwinkle = array(twinkle)
wavfile.write('twinkle_synth.wav', 44100, nptwinkle)
```



References

1. Websites

- [1] Wikipedia: https://en.wikipedia.org/wiki/Main_Page
- [2] Nao's Official Website: <https://www.ald.softbankrobotics.com/en>
- [3] Python's Official Website Documentation: <https://www.python.org/doc/>
- [4] SciPy's Official Website: <http://www.scipy.org/>
- [5][6] Pitch Detection Algorithms:
- <https://ccrma.stanford.edu/~pdelac/154/m154paper.htm>
 - <http://obogason.com/fundamental-frequency-estimation-and-machine-learning/>
- [7] Digital Signal Processing Website: <http://dsp.stackexchange.com/>
- [8] Procesado Digital de Sonido e Imagen:
<https://egela1516.ehu.eus/course/view.php?id=1713>

2. Books & Documents

- [9] Naotoshi Seo sonots@umd.edu “*ENEE632 Project4 Part I: Pitch Detection*”:
<http://note.sonots.com/?plugin=attach&refer=SciSoftware%2FPitch&openfile=pitch.pdf>
- [10] Mike Beiter, Brian Coltin, Somchaya Liemhetcharat, “*An Introduction to Robotics with Nao*”, Aldebaran Softbank Group
- [11] David Gerhard, “*Pitch Extraction and Fundamental Frequency: History and Current Techniques*”, Technical Report, University of Regina, CANADA
- [12] Justin J. Salamon, “*Melody Extraction form Polyphonic Music Signals*”, Tesis Doctoral, Barcelona, SPAIN
- [13] Silvia Maria Alessio, “*Digital Signal Processing and Spectral Analysis for Scientists*”, Springer, ISBN 978-3-319-25466-1