An Auto-Tune based plugin for digital audio workstations

Trabajo de Fin de Grado en Ingeniería Informática

Tania Romero Segura

Dirigido por

Jaime Sánchez Hernández

Universidad Complutense de Madrid

Madrid, Junio de 2022

Abstract

Today, music production is at anyone's reach. The DAWs (Digital Audio Workstations) include in one single tool most, if not all, of the elements needed in a professional studio. Thanks to their intuitive use, the wide variety of tools included and their affordable prices for regular users, a DAW can be found in an amateur's musician computer and in a professional producer's computer.

Another key to the success of DAWs is that if they do not include an utility, it can be added through external programs called plugins. These programs extend the functionality of DAWs and can be included into projects in an easy and straightforward manner. A very popular group of plugins are the pitch correction tools, like AutoTune. Essentially, they detect the mistakes in the pitch of a singer's voice and then correct them. With time, the characteristic sound that these tools can generate has come to be considered as an stylistic effect in on its own and many artists use it as part of their own personal touch.

This project aims to recreate how pitch correction tools work in a plugin format. Pitch correction has two main steps: pitch detection (finding the current pitch) and pitch shifting (modifying the audio to fix the current pitch). Both of these steps are complex and computationally expensive, specially pitch detection. Therefore, implementing a pitch correction effect as a plugin poses many challenges and would allow the end result to be used in any DAW.

Keywords

Music production, DAW, plugin, VST, AutoTune, pitch, pitch detection, pitch shifting, tuning

Resumen

Hoy en día, la producción musical está al alcance de cualquiera. Los DAWs (Digital Audio Workstations) son programas que permiten tener todos los elementos necesarios en un estudio de grabación integrados en un entorno de producción en una única herramienta. Gracias a su uso fácil e intutitivo, a la increible variedad de herramientas que incluye y al coste económico asumible por el usuario doméstico puede encontrarse un DAW tanto en el ordenador de un músico amateur como en el de un productor profesional.

Otra de las claves del éxito de los DAWs es que si no incluyen una utilidad, puede incluirse mediante el uso de programas externos llamados plugins. Estos programas amplian la funcionalidad de las DAWs y pueden inclurise en proyectos de manera fácil y rápida. Un grupo muy popular de plugins son las herramientas de corrección de afinación como AutoTune. En su uso más directo, permiten detectar y corregir errores de afinación en las notas producidas por los cantantes. Con el tiempo, el sonido característico que pueden generar estas herramientas ha llegado a considerarse como un efecto estilístico por sí mismo y muchos artistas lo utilizan como parte de su sello personal.

Este proyecto pretende recrear el funcionamiento de estas herramientas de corrección de afinación como un plugin. La corrección de afinación tiene dos pasos principales: detectar la afinación actual y corregir el audio para arreglar la afinación. Ambos pasos son complejos y tienen un alto coste computacional, especialmente la detección de pitch. Por lo tanto, implementar un efecto de corrección de afinación como un plugin presenta muchos retos y permitiría que el resultado final se uitlice en cualquier DAW.

Palabras clave

Producción musical, DAW, plugin, VST, AutoTune, tono, detección de tono, cambio de tono, afinación

Contents

Document structure

Before we start, we would like to give an explanation on how this document is structured to give the reader a guide to navigate its contents. The document is divided into specific chapters that have a distinct content:

- **Chapter 1:** The first chapter serves as an introduction to the project. The background section provides some context and essential information to understand the overall topic. We will give some information here about what music production is and the tools used nowadays that have an important role in this project. Also, since the main focus of the project is pitch correction we will also provide some information on how the human voice works and what is the pitch. Then we present the motivation for the project and the objective that we had when we started.

- **Chapter 2:** The second chapter aims to give a simple manual so the reader can try the plugin and application that were developed as a result of the project for themselves.

- **Chapter 3:** The third chapter contains the more theoretical part of the project. Pitch detection and pitch shifting are complex tasks. Different algorithms were extensively studied and implemented as small Python prototypes to better understand pitch detection and pitch shifting in an effort to improve the performance of the developed tool. Here, we explain all the algorithms that we studied providing the different mathematical equations involved and their purpose.

- **Chapter 4:** The fourth chapter provides an explanation on how the final chose pitch correction prototype was implemented. We implemented different small prototypes to test the pitch detection algorithms. Once we had chosen one we extended it to include the pitch shifting step. Afterwards this extended prototype was used as a guideline for the development of the plugin and application. The first section of the chapter provides some information on the technologies that were used for this step.

- **Chapter 5:** The fifth chapter explains how the plugin and standalone application were implemented. First we talk about the JUCE framework and the interpolation library used for the development. We then get into the details of the implementation of both the plugin and application in separate sections.

- **Chapter 6:** The sixth chapter contains the conclusions derived from the project and an overview of possible steps that could be taken in the future to improve the end result.

As can be seen, the document itself follows the same steps that were taken to achieve the final result, starting with the context, the different algorithms on a theoretical level, the implementation of the prototype and the implementation of the plugin and application. This way, the previous chapters provide all the information needed to understand each chapter. All the chapters follow these idea except the user manual. If the reader is not interested in the development of the pitch correction tools then they only need to read the first two chapters to understand what they do and how to use them.

Chapter 1

Introduction

1.1 Background

1.1.1 Music Production

Music production can refer to the entire lifecycle of a piece of music, from songwriting and composition to recording, mixing and mastering. Most people, when they think about a music production studio, imagine a hardware with many wires, sliders, and such behind a big window with a man moving all over the place to adjust the values. They imagine hundreds of recordings of the same piece (either whole or just fragments of it) and someone spending hours and hours choosing the best materials to put together the final product. While that was true a few years ago, the music producers of today use DAWs (Digital Audio Workstations) and can have all the functionalities of the classical audio workstation within their computer. This has tremendously lessened the workload for the production of full pieces.

DAWs

A DAW takes the essential components of a recording studio's control room: the mixing console, outboard gear... and puts them together into a single computer program. DAWs come in a wide variety of configurations from a single software program on a laptop, to an integrated stand-alone unit, all the way to a highly complex configuration of numerous components controlled by a central computer.

Once analog signals could be represented digitally using samples, several experiments in digital recording were carried out over the years. After a few years, the first digital recording

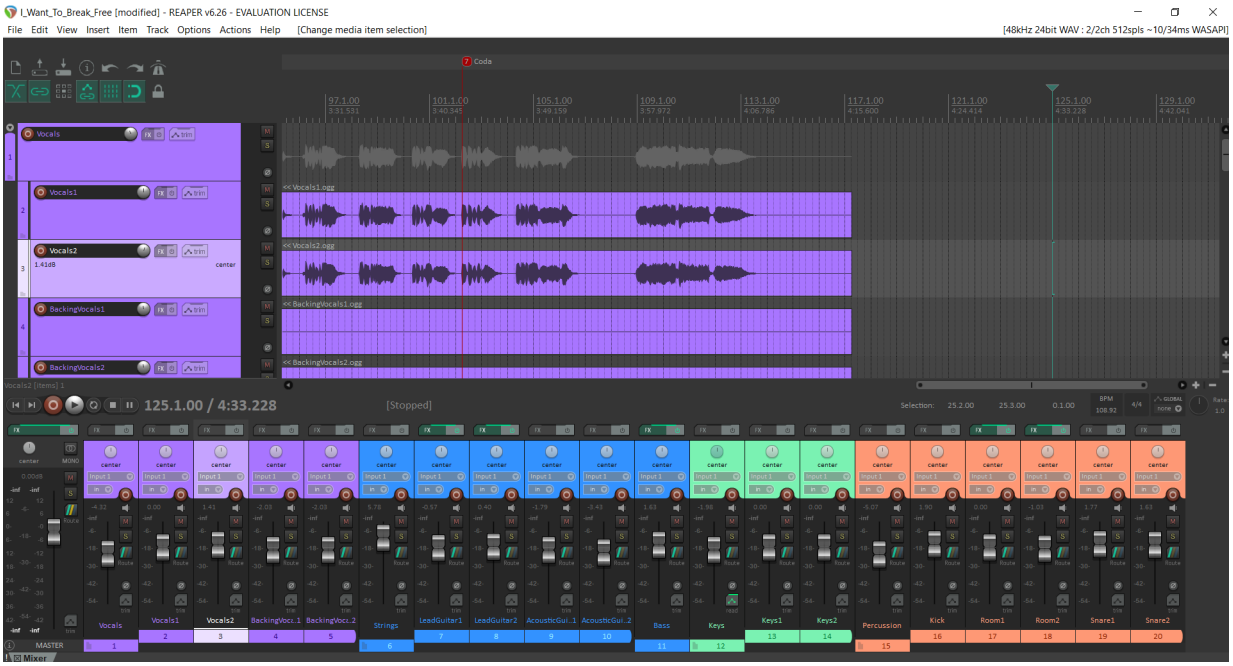


Figure 1.1: REAPER DAW with multiple tracks

software was released by the company Soundstream [13]. Thanks to this development, and the evolution of computers that provided the computational power necessary for audio editing, we now have Digital Audio Workstations.

Since then DAWs have only been growing and growing. With their evolution came better user interfaces along with many more features packed into them. Today, DAWs allow high quality multi-track digital recording with a track count that has grown rapidly and a user-friendly interface.

Audio Plugins

With the apparition of DAWs, audio plugins came along. This programs can be imported into the DAWs and applied to different tracks. Therefore, if a DAW does not provide a specific tool, an external plugin can be used to extend the DAW to cover this functionality. Each plugin will receive the samples from the track (either the original samples or the resulting samples from the execution of other plugins placed previously in the execution order) and operate on said samples. Plugins can receive either float point samples representing the audio signal and/or MIDI messages [1].

[1]MIDI is a protocol that allows for the communication between electronic instruments, computers and other devices using the exchange of messages. For example, when someone plays a key in a keyboard, a message is created specifying the tone, the speed/intensity and other musical parameters.

Some of the most commonly known plugins are virtual synths, equalizers, compressors, reverbs, delays, etc. There are 3 main types of plugins:

- **Instrument plugins:** We can distinguish between two sub-types depending on the method utilized to create the sound.
  - Virtual synthesizers: These plugins use digital signal processing to create sounds.
  - Samplers: These plugins use prerecorded samples of sound.
- **Effect plugins:** These plugins have the objective of modifying or completely reshaping a given signal. Here are a few examples of the most used effect plugins:
  - Filters: Their function is to attenuate or highlight certain frequencies selected by the user.
  - Equalizers: These plugins are a combination of a number of different filters allowing the user to specify different filters for different ranges of frequencies with one single tool.
  - Compressors: They affect the dynamic range of a signal. The idea is to make the loudest parts lower in volume and vice-versa.
- **Analyser plugins:** The primary purpose of this type of plugin is to analyze some characteristics of the signal being processed. For example, a spectrum analyzer normally presents the producer with a graph that shows frequency against loudness in real time. A lot of plugins incorporate analyser sections that give information relevant to understanding the result of modifying the user specified values. For example, some equalizer plugins include a spectrum analyzer to visualize how the filtering parameters affect the output signal.

It is worth mentioning that a plugin often fall into more than one category. There are different possible formats, standard or not. Nonetheless, we have found the VST format to be the most widely used standard format. VST is supported by all DAWs which is the reason why audio software developers normally use this format to create, test and distribute their plugins.

1.1.2 The Human Voice

When a human speaks or sings, a sound is produced. This sound comes from the vibration of the vocal folds (see figure 1.2), creating a group of frequencies that our ears identify as one single sound. The lowest frequency among this group of frequencies is what is called the fundamental frequency.

The fundamental frequency establishes the pitch and determines what sound our ears will identify. The other frequencies intertwined with the fundamental are what we call harmonics. These harmonics are waves with a frequency that is a positive multiple of the fundamental frequency. The set of harmonics forms a harmonic series. For example, if the fundamental frequency is 50 Hz, the following harmonics will be at 100 Hz, 150 Hz, 200 Hz, etc. In music, the harmonics follow a very particular pattern in terms of the intervals separating the different "notes" that each frequency represents. If the first harmonic is the fundamental frequency, the second harmonic has double the frequency, the third has three times the frequency, etc. This translates to specific musical intervals. For example, if the fundamental tone is C, then its second harmonic is C in the next octave up, and the third will be G an octave and a perfect fifth higher, and the fourth will be C two octaves up, the next one will be E which is a third higher creating a full major triad chord (C, E, G), etc.

The reason for the difference in pitch for different people is the differences in the vocal folds themselves (as well as other physiological aspects) like the size, mass, and tension. With age, the length of the male vocal folds grows more than the female vocal folds creating noticeable differences between male and female voices, since these changes are correlated with a decrease in fundamental frequency. Therefore, females have a higher frequency range due to having a smaller larynx as well as shorter vocal folds while males have a lower frequency range for the opposite reasons.

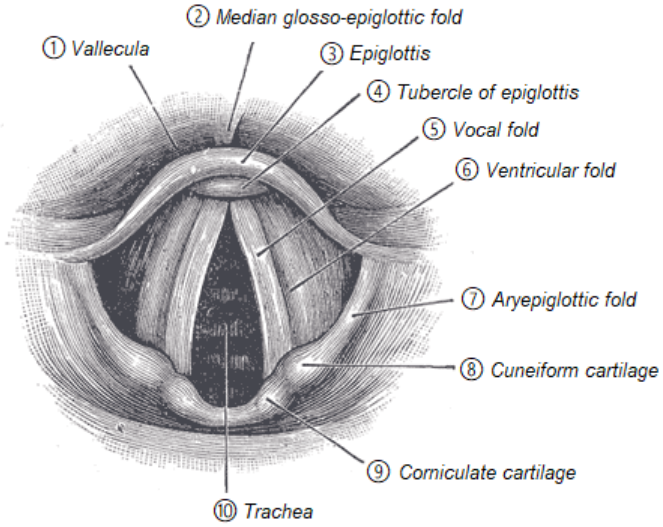


Figure 1.2: Human vocal folds. Image by Sandrarossi - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=81403708

1.1.3 AutoTune

Pitch Correction

Among the many different effects achievable with audio processing, pitch correction effects are some of the most popular. The first pitch correction apparatus, called AutoTune, was invented by Hildebrand and still is one of the best, if not the best, pitch correction effect nowadays. The original pitch correction effects aimed to find the fundamental frequency of the voice of a singer, see whether or not this sound is in tune and correct it if it isn't. Today, these effects normally come in a plugin format to use them in DAWs.

However, now these effects also provide ways to decide which note we want and then modify the recording to match the note, even if it is not close to the original pitch. Furthermore, different improvements have been done to these effects, allowing for the customization of the produced sound to make it more metallic, add vibrato, modify the smoothness of the transitions, etc. Taking all of these features into consideration, it is clear why these effects are so popular and can be an essential part of an artist's style.

History of AutoTune

In the early 90s, there was no way to automatically correct the tune of a singer's voice. Then, Harold Hildebrand came to the rescue.

At that time, Hildebrand worked for Exxon Mobil (an American multinational oil and gas corporation) as an electrical engineer applying digital signal processing to geology to use ground-penetrating signals to find oil. Hildebrand was an accomplished musician. One day, while having lunch with a group of colleagues, he queried the group's opinions about what the music industry could need that an electrical engineer might provide. As a joke, one of his friends expressed a desire to have a tool that would make her sing in tune all the time. While the group laughed it off, the idea was implanted in Hildebrand's mind.

Within a few years, Hildebrand found a way to use the same techniques he was applying at Exxon to make a singer's voice, in real time, to be perfectly in tune. He did this by deriving a new way to utilize the autocorrelation function to track the fundamental frequency of a singer's voice. Armed with these equations, he designed an integrated system that consisted of a microcontroller loaded with a program capable of executing said mathematics. This system was dubbed Auto-Tune.

The patent [21] was filed in 1998 to the company of Hildebrand's founding, Antares Audio Technologies. The first song that hit the airwaves with Auto-Tune was Cher's "Believe" single, and since then the recording industry has never been the same. Now just about any

popular track you listen to has been auto-tuned to some effect in the studio. Sometimes it's used to fix a missed note, and other times to correct the pitch of an entire song.

Figure 1.3: Original AutoTune interface

1.2 Motivation and Objectives of the work

AutoTune was, and still is, an incredibly successful pitch correction tool. Nowadays, most applications working with pitch use frequency-domain techniques involving some kind of Fourier Transform. Having seen the many algorithms that exist to achieve pitch detection, it seemed most curious that the original AutoTune patent used a time-domain method.

Pitch correction is a very complex task in on its own. Its main step, pitch detection, is an active field of research today. Many different algorithms have been proposed with varying degrees of accuracy and speed. Furthermore, pitch detection, when applied to singing voices, becomes a far more costly task given the increase in harmonic complexity and the wider range of possible fundamental frequencies. Like AutoTune's method, other algorithms use autocorrelation as part of its calculations. However, most of them use some sort of frequency-domain processing (normally Fourier Transform [4]) to speed up the computation time. AutoTune's patent, on the other hand, reduces the computational cost by deriving simplified

equations from an un-normalized autocorrelation function (we will introduce these equations in section 3.1.1).

To summarize, we have the following objectives wrapped within the overall objective of implementing a pitch correction tool:

- Study AutoTune's patent pitch detection and pitch shifting methods.
- Implement a Python prototype to test the algorithm.
- Implement the algorithm as a VST plugin (preferably using the JUCE framework).

Chapter 2

User Manual

In the end, two different tools have been developed: an audio plugin and an application. This chapter provides the necessary instructions to use both of them.

2.1 Plugin

To use the plugin, there are three main steps to follow overall: include the plugin in a DAW, add the plugin to a track and configure the plugin to obtain different results. In this manual the DAW chosen is REAPER [11] but the steps can be translated to any other DAW.

2.1.1 Including the plugin in the DAW

To include the plugin in REAPER first download the plugin from [17]. Once the VST has been downloaded, open REAPER to include it in the plugin list. There are two different options to make sure REAPER will find the plugin:

- For the first option place the plugin inside the C:\Program Files\Common Files\VST3 directory since this is one of the VST search paths where REAPER will look for the plugin.
- Another option is opening the *Preferences* dialog under the *Options* menu. In the side menu select the VST option in the section Plug-ins. Then, edit the path list to include the directory where the plugin currently is. Afterwards apply changes and, to ensure the plugin has been fully added and can be used straight away, use the Re-scan option before closing the *Preferences* dialog (see 2.1).

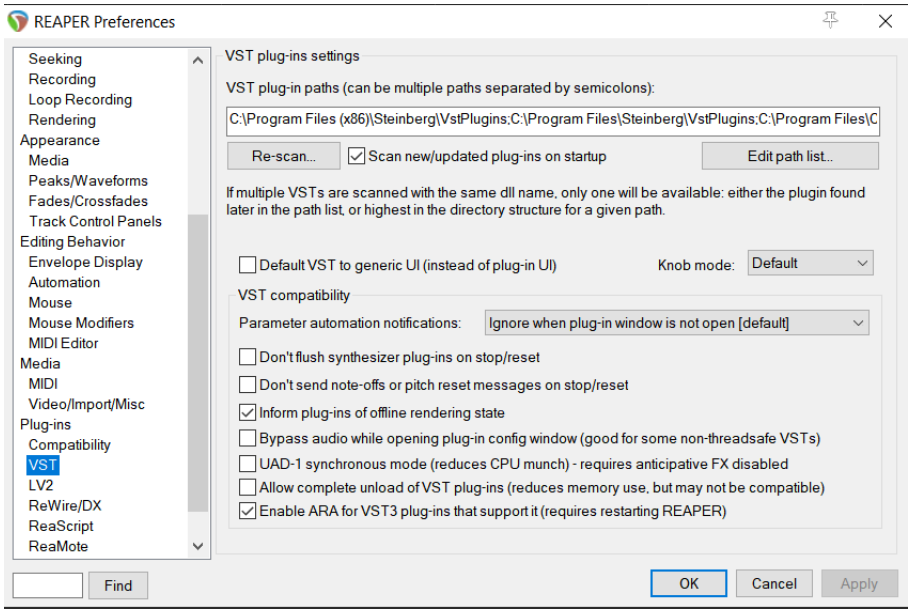


Figure 2.1: Preferences dialog, VST section

2.1.2 Adding the plugin to a track

Once the plugin has been added to the DAW it can be used for any project from now on. In this tutorial, we will begin with an empty project and create a track with a prerecorded audio and add the plugin to it.

There are different ways of adding a track using a prerecorded audio. The most straightforward way is by simply dragging the file into the REAPER interface. This will automatically add a new track containing the given audio at the end of the track list.

After creating the track with the audio we want to correct then we can add the plugin to the track. To do that click the *FX* button in the corresponding track to manage the track's plugins.Since the track does not have any plugins added, two dialog will open. One of them is used to search through the list of plugins to add one and the other is used to manage the added plugins. In the first dialog, instead of looking through the whole list use the filter to write the word pitch to reduce the list and select the plugin *PitchCorrectionPlugin*. Then, click the *Add* button to add the plugin to the track (see figure 2.2). As a result the pitch correction plugin should now appear in the other dialog (see figure 2.3).

We now have a REAPER project with a track containing our prerecorded wavfile with the plugin added to it a fully operational.

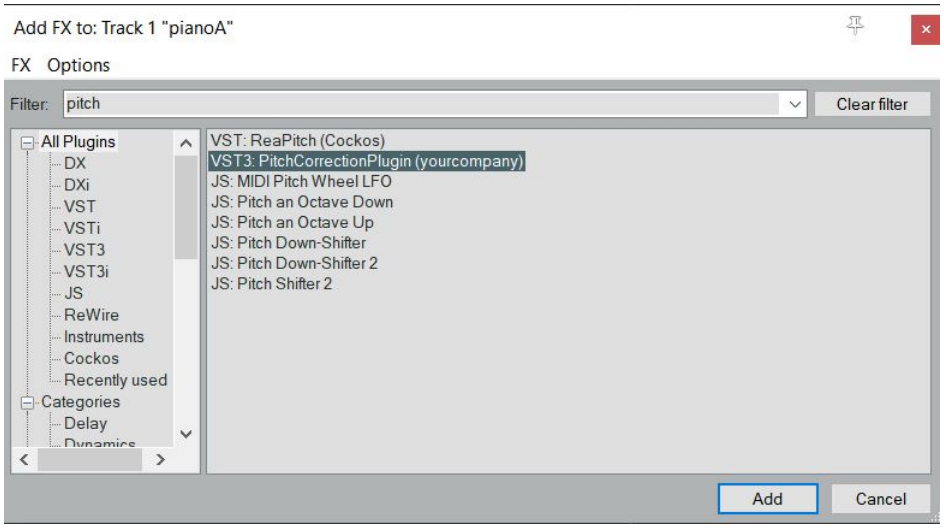


Figure 2.2: Tracks add plugin dialog

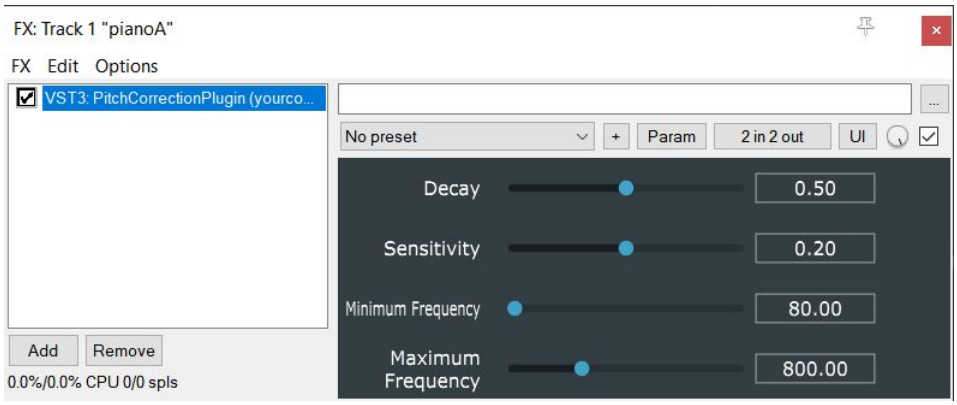


Figure 2.3: Tracks plugin manager

2.1.3 Configure the plugin

Now, the only thing left to do is adjust the parameters of the pitch correction. This is done by modifying the sliders shown in the interface (see figure 2.3). To access the plugins parameters open the tracks plugin manager again by clicking on the *FX* button of the track. The different parameters offered are:

- Decay: this parameter allows to specify how smooth we want the transition between samples to be. A value of 0 is equivalent to not using crossfading. Crossfading makes the audio change gradually from the original sound to the modified sound reducing artifacts and creating a smoother result. The usage of low values would create more

abrupt changes when correcting the pitch adding a more electronic kind of feel to the voice.

- Sensitivity: this represents how sensitive the pitch detection step is. A low value means that the plugin will not be as picky when lookig for the pitch. This means that if the fundamental frequency is 400Hz the plugin could find it to be 387 Hz with a low sensitivity. However, very high values for sensitivity could result in the plugin not finding the correct pitch by either finding a harmonic or not finding anything at all.
- Minimum and maximum frequency: these values represent the range of frequencies that we want the plugin to detect. This should be lowered or raised accordingly depending on the content of the audio to reduce the amount of mistakes and the time necessary to modify the audio.

2.2 Application

To use the application, the user has to first run the application and then follow three steps to get a playable modified audio file: load a file, correct the pitch of the file and save the file for playing later. The application can be downloaded in [10].

2.2.1 Load a File

The first step is loading a file. To do that click the *Load File* button (see figure 2.4). A file explorer will then open. Then, all that's left to do is look for the file and select it. It is important to mention that the application only supports the wav format and will not find any extension other than wav.

2.2.2 Correct the Pitch

Once a file has been loaded, the next step is telling the application to correct the pitch of the audio. To do that, click on the button *Correct Pitch* (see figure 2.5). But first, it may be desirable to adjust the parameters of the pitch correction to the current needs. The parameters are almost identical to the parameters offered in the plugin. For more details on what each of them represents see 2.1. The only difference is that the application requires an additional parameter. The user has to specify audio's sampling rate since the application cannot find it on its own.

Figure 2.4: Applications load button

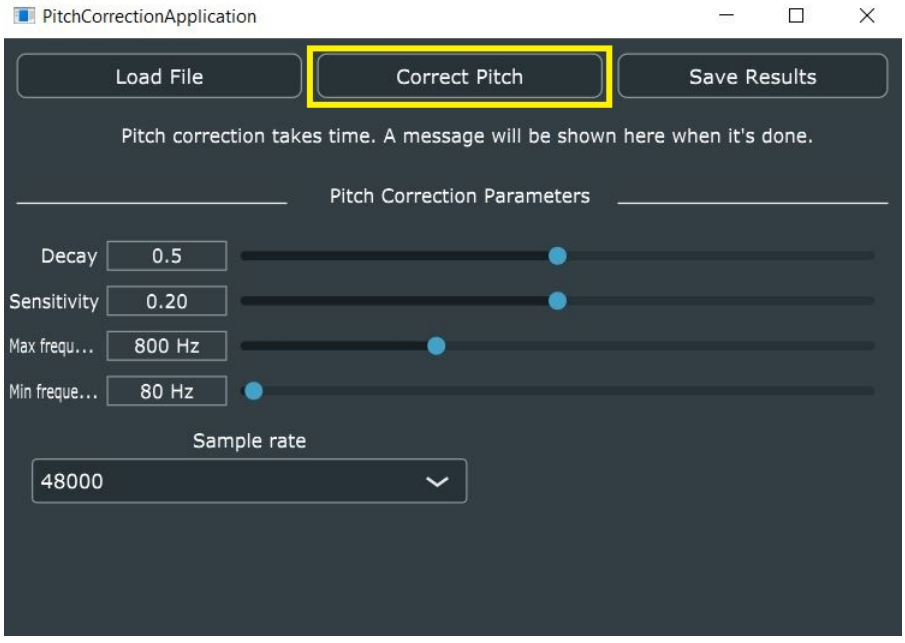


Figure 2.5: Applications correct pitch button

2.2.3 Save the results

Finally, save the results in a wavfile to later listen to the results. To do that click on the button *Save Results* (see figure 2.6). Like with the loading of a file, this button opens a file explorer that allows the user to browse through the folders to choose a destination and the manually input the name of the file that is being created. The only format supported is the wav format. Once the destination file has been specified the result of the pitch correction (or the original audio if no pitch correction was done) will be saved into a file at the location given with the chosen name.

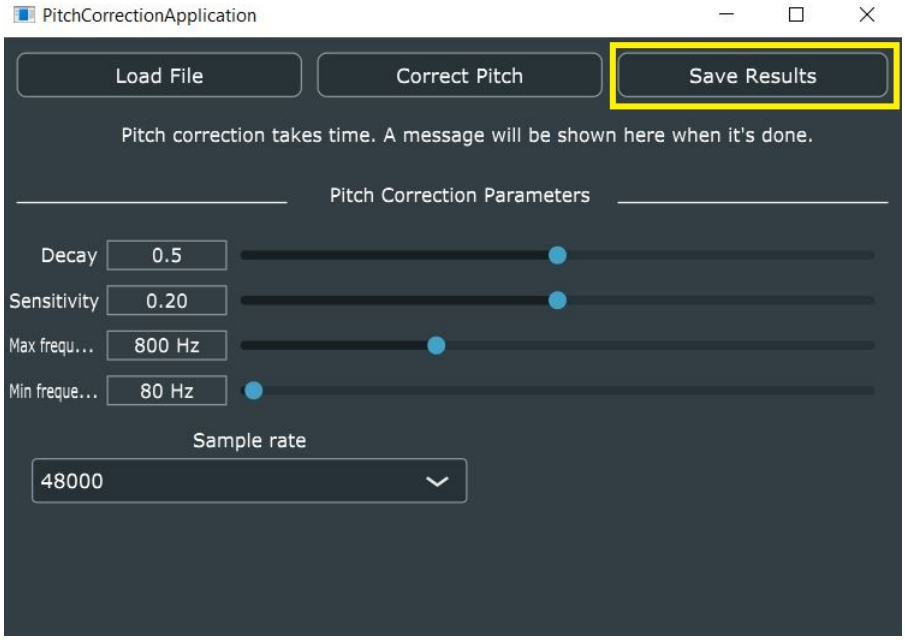


Figure 2.6: Applications save button

Chapter 3

Pitch Correction

In this chapter we will discuss the two main parts of any pitch correction algorithm: pitch detection and pitch shifting. Any program that aims to correct the pitch must first find it using a pitch detection method and then modify the pitch of the audio using pitch shifting to make the end result have the desired pitch. Both these steps are very complex. However, the more complicated part about pitch correction is by far pitch detection. Furthermore, pitch detection is still an active field of research nowadays.

3.1 Pitch Detection

Estimating the pitch of a sound is a very complex process. There are many challenges involved like distinguishing between voiced sounds and "silence" since true silence does not exist in a recording due to the presence of environmental sounds like the act of breathing. Also, it is important to determine whether the frequency that was found is actually the fundamental frequency or pitch rather than one of its harmonics. These problems along with many others have resulted multiple different methods for detection the pitch throughout the years. These methods can be divided into 3 main types:

- Time-domain methods.
- Frequency-domain methods.
- Hybrid methods use a combination of both time-domain and frequency-domain methods.

All of them have their advantages and disadvantages so choosing one particular method can depend a lot on the particular application and can be a grueling process. It is very

important to take into account that for each type different methods have been proposed for different objectives. Therefore, an algorithm used to detect the pitch of a human speaking may not be well suited for finding the pitch of the voice of a singer and vice versa. Many papers have been written discussing the differences between algorithms and their use [26, 28, 29].

3.1.1 Time-Domain Methods

Time-domain methods are performed directly with the signals information directly on the time domain without ever using frequency domain techniques. For this type of pitch detectors the peaks and valleys of waveforms and other information like zero-crossing points (points where the signals samples go from positive values to negative values or vice versa) are crucial to find the pitch. In this section, we explain how some algorithms widely used to detect the pitch of a singers voice work.

Autocorrelation

Autcorrelation is the most common resource for time-domain pitch detection algorithms and many of them use it as a starting point and then try to improve its computational cost and/or accuracy. Essentially, autocorrelation consists of comparing the signal with a shifted copy of itself. To find the fundamental frequency of a perfectly periodic signal, one can progressively shift the signal by a time value (lag) until such a value is found where the shifted signal perfectly matches the original.

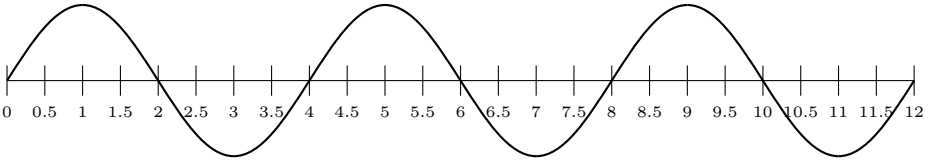


Figure 3.1: Original waveform

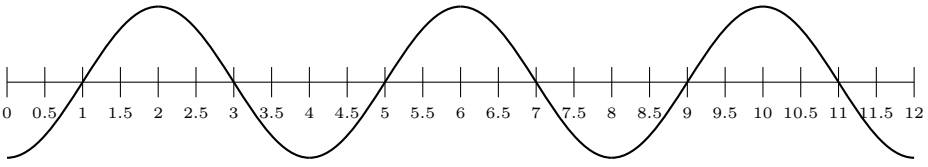


Figure 3.2: Shifted waveform not matching the original (lag = 1)

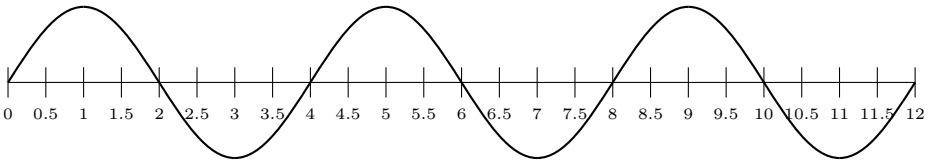


Figure 3.3: Shifted waveform perfectly matching orginal (lag = 4 = period)

AutoTune Patent

The original AutoTune Patent [21] proposed a time-domain method for pitch detection using a modified version of the autocorrelation function as its base. The original system had to first input an audio signal into an A/D converter to get the samples for the computational processing. Afterwards the data could be processed using the modified autocorrelation. The data processing had two modes of operation: detection mode, which occurs when the fundamental frequency is not known and correction mode which occurs when the fundamental frequency is known.

In the pitch detection mode, the pitch detection claimed to be instantaneous since the device was supposed to be able to detect the pitch in a periodic sound within a few cycles. The patent also includes some other methods to make the accuracy better by applying what is called pitch tracking. The pitch tracking occurs in between executions of the pitch detection to detect small variations in the pitch. If the variation found in the pitch tracking is higher than expected, then pitch detection is performed again.

To detect the pitch the patent uses an un-normalized version of the auto-correlation function Φ. This function is described in the patent [21].

$$\Phi_L(n) = \sum_{j=0}^{L} x_j x_{j_{-n}} \tag{3.1}$$

We assume x to be periodic with a fundamental period L. The auto-correlation function is also periodic. x_j denotes the current time step audio sample and x_{j-n} denotes the audio sample n time steps before (with n representing the lag being currently evaluated).

At a time i given a sequence of a sampled periodic waveform with period L, the auto-correlation of the function (x_j) can be expressed as

$$\Phi_{i,L}(n) = \sum_{j=i-L-1}^{i} x_j x_{j_{-n}} \tag{3.2}$$

Now, the key of the success of Auto-Tune was speeding up this process to apply it in real time. To do that, the following functions were derived: $E_i(L)$ y $H_i(L)$.

$E_i(L)$ is the auto-correlation of two cycles of a periodic waveform evaluated at 0 lag defined as dependent on the guess of the period of the cycle L. The function $E_i(L)$ represents the accumulated energy of the waveform over two periods ($2L$).

$$E_i(L) = \Phi_{i,2L}(0) = \sum_{j=0}^{2L} x_j^2 \tag{3.3}$$

$H_i(L)$ is the auto-correlation of a single cycle of the periodic waveform evaluated and the guess of the period L. Again, this function is defined with the guess of the period L as the dependent variable.

$$H_i(L) = \Phi_{i,L}(L) = \sum_{j=0}^{L} x_j x_{j-L} \tag{3.4}$$

Equations 3.3 and 3.4 can be expressed as follows to speed up the calculations:

$$E_i(L) = E_{i-1}(L) + x_i^2 - x_{i-2L}^2 \tag{3.5}$$

$$H_i(L) = H_{i-1}(L) + x_i x_{i-L} - x_{i-L} x_{i-2L} \tag{3.6}$$

At any given time point i these two functions allow us to determine the period of a periodic waveform by evaluating the following equation for L.

$$E_i(L) - 2H_i(L) = 0 \tag{3.7}$$

Equation 3.7 states that when L is set to the true period of the periodic waveform x_j it becomes true that equation 3.3 is equal to two times equation 3.4. Therefore, if L is the true period then equation 3.7 becomes true. To simplify, since $E_i(L)$ represents the accumulated energy over two periods, if a perfect autocorrelation has been found the value should be equal to two times the $H_i(L)$ value since the $H_i(L)$ operates on one single period.

However, the signal being analyzed at a given time point is very unlikely to have a period that matches perfectly with the integer sample rate. Furthermore, it is highly unlikely to have a truly periodic signal in the first place. To manage this situation, when analyzing the

real sampled data, L is said to be the fundamental period of the waveform if the following equation is satisfied:

$$E_i(L) - 2H_i(L) \leq \epsilon E_i(L) \tag{3.8}$$

Where ϵ can range from 0.00 to 0.40 (values suggested in the patent [21] and represents the sensitivity of the calculations. Once a value that satisfies equation 3.8 has been found, a range of L values surrounding the satisfying L value are used to perform a polynomial approximation of $E_i(L) - 2H_i(L)$ in order to find the local minimum. This floating point minimum represents the true period of the waveform.

In the pitch correction mode, the system takes the period found and the desired period and finds the ratio. Then, it uses that ratio to do pitch shifting modifying the pitch of the signal to match the desired pitch. Methods for finding the desired period included a MIDI interface to allow the user to input the notes manually and a method to determine the desired period by finding the period of the closest note from a musical scale.

YIN

The YIN method [20] proposes another time-domain pitch detection algorithm that produces less errors compared to other pitch detection methods. YIN is also created with autocorrelation as its foundation but other mathematical mechanisms have been applied in order to achieve a higher accuracy in its estimations. This algorithm is widely used in real applications.

The autocorrelation function (ACF) of a discrete signal x may be defined, as seen before, as

$$r_t(\tau) = \sum_{j=t+1}^{t+W} x_j x_{j_{+\tau}} \tag{3.9}$$

where $r_t(\tau)$ is the autocorrelatin function of lag τ calculated at time index t and W is the window size.

The autocorrelation method makes too many mistakes for many applications. Said mistakes are more pronounced when the signal has higher harmonic complexity. The following steps have the objective of minimizing said mistakes. The first modification involves modeling the signal x_t as a periodic function with period T

$$x_t - x_{t+T} = 0, \forall t \tag{3.10}$$

If we take the square and pick the average over a window

$$\sum_{j=t+1}^{t+W} (x_j - x_{j+T})^2 = 0 \tag{3.11}$$

Therefore, an unknown period can be found by using the difference function

$$d_t(\tau) = \sum_{j=1}^{W} (x_j - x_{j+\tau})^2 \tag{3.12}$$

and searching for the values of τ for which the function is zero. However, the amount of possible values that are multiples of the period is infinite. The squared sum can be expanded and expressed in terms of the autocorrelation function.

$$d_t(\tau) = r_t(0) + r_{t+\tau}(0) - 2r_t(\tau) \tag{3.13}$$

As can be seen, equation 3.13 is very similar to equation 3.7 proposed in Auto-Tune's patent. The first two terms are energy terms. While the first term is the same than in the Auto-Tune patent the key lies in the second term. That term varies with τ which means that the maximum of $r_i(\tau)$ and the minimum of $d_i(\tau)$ may not coincide. Indeed, the error rate falls to 1.95% from 10% which is a pretty significant change.

Another problem to solve is that the difference function is zero at zero lag and often nonzero at the period because of the imperfect periodicity. The proposed solution is replacing the difference function by the cumulative mean normalized function. When testing the implementation of YIN applying each modification gradually, it was found that this step has a very high computational cost and lowers the performance significantly. Modifications to the YIN algorithm found change this step for other alternatives with better performance. Many of those other algorithms use the Fast Fourier Transform (FFT) [4, 30] and therefore mix both time domain and frequency domain techniques.

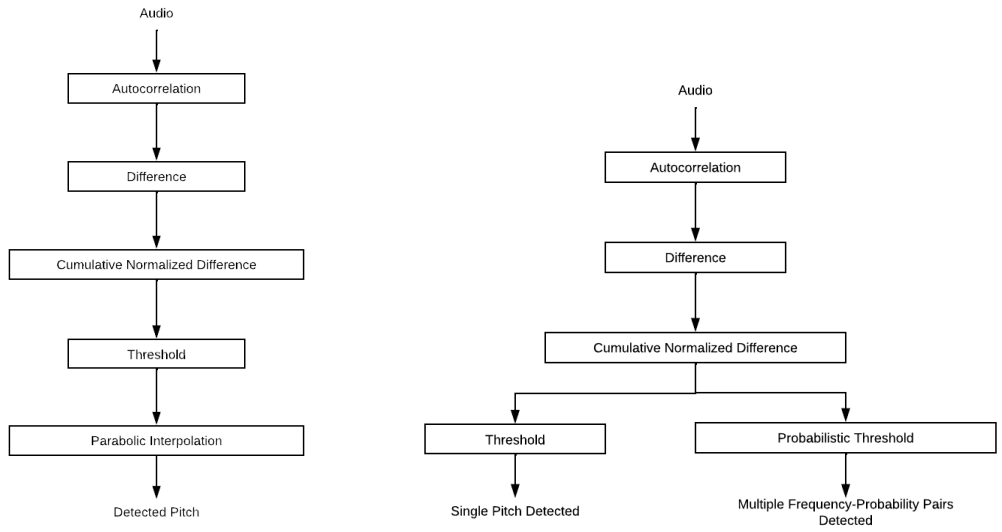
$$d'_t = \begin{cases} 1, & \text{if } \tau = 0 \\ d_t(\tau) / [(1/\tau) \sum_{j=1}^{\tau} d_t(j)] & \text{otherwise} \end{cases} \tag{3.14}$$

The original idea behind equation 3.14 is to divide the values of the old by its average over shorter lag values. It starts at 1 rather than 0, tends to remain large at low lags and drops bellow one only where $d(\tau)$ falls bellow average. The paper claims that this addition makes the error rate drop to 1.69% where before it was 1.95%.

Now, sometimes we can detect the subharmonics instead of the correct period. This is a very common error mentioned in many pitch detection algorithms and is referred to as the "octave error" although this is not correct since it doesn't necessarily mean the error is in a power of 2 ratio with the correct value. The same happened with the autocorrelation method.

The YIN algorithm utilizes an aboslute threshold and chooses the smallest value of τ that gives a minimum of d' lower than said threshold. If none is found the global minimum is chosen instead. For example, a threshold of 0.1 lowers the error rate to 0.78%. To finish, parabolic interpolation is used to find a better estimation of the period.

To sum it all up, to estimate the pitch we use autocorrelation to find the difference function. Then the result is normalized using a cumulative approach. Afterwards a threshold is used to find an estimate of the pitch. Finally, a better estimation is found using parabolic interpolation. The probabilistic YIN algorithm is essentially the same but instead of using a single value for the threshold a distribution is used which leaves us with multiple pitch-probability pairs. This second method is widely used as well. However, it is slower than the regular YIN and normally requires the use of the FFT (Fast Fourier Transform) to make it a viable choice for real time pitch detection.



(a) YIN diagram (b) Probabilistic YIN diagram

Figure 3.4: YIN and PYIN flow diagrams

MPM

The MPM method [24] has many similarities with the YIN method. The first difference lies in the type of autocorrelation function used as a starting point. There are two main ways used for defining the autocorrelation function (type I and type II). While YIN uses normally uses type I, MPM uses type II. The type II autocorrelation function has the peculiarity of reducing the window with higher values of τ. Here is the definition of type II autocorrelation

$$r'_t(\tau) = \sum_{j=t}^{t+W-\tau-1} x_j x_{j_{+\tau}} \tag{3.15}$$

Next, following the same steps, the square difference function is defined. However, since we use the type II autcorrelation function the resulting function is slightly different

$$d'_t(\tau) = \sum_{j=t}^{t+W-\tau-1} (x_j + x_{j+\tau})^2 \tag{3.16}$$

If equation 3.20 is expanded, the autocorrelation function is found within, by defining the equation

$$m'_t(\tau) = \sum_{j=t}^{t+W-\tau-1} (x_j^2 + x_{j+\tau}^2) \tag{3.17}$$

then the following result can be obtained

$$d'_t(\tau) = m'_t(\tau) - 2r'_t(\tau) \tag{3.18}$$

Here is where the innovation of this method starts. Once the square difference function at time t has been found the choice has to be made to decide which τ value corresponds to the pitch. This value is not necessarily the overall minimum but it usually is among the local minima. To simplify the problem, the authors propose using noramlization via the Normalised Square Difference Function (NSDF)

$$n'_t(\tau) = 1 - \frac{m'_t(\tau) - 2r'_t(\tau)}{m'_t(\tau)} = \frac{2r'_t(\tau)}{m'_t(\tau)} \tag{3.19}$$

The result is in the range of -1 to 1 where 1 means perfect correlation, 0 means no correlation and -1 means perfect negative correlation. With this normalised values, the problem of choosing the pitch period is simplified since the range is well defined. The process of choosing the best maximum is called peak picking.

The paper also states that a property had been found (Symmetry property) that states that there are the same number of evenly spaced samples being used from either side of time t for a given τ and that said samples are symmetric in their distance form t. This creates a frequency averaging effect. Equation 3.20 can be made to hold this property by shifting the center to time t

$$d'_t(\tau) = \sum_{j=t-(W-\tau)/2}^{t+(W-\tau)/2-1} (x_j + x_{j+\tau})^2 \tag{3.20}$$

The paper also includes a section where a method for speeding up the computation of the square difference function. Essentially, by obtaining the equation 3.17 the calculations have already been improved. To speedup the calculation of the second term of equation 3.18 a way to calculate the ACF with the Fast Fourier Transform is proposed:

1. Zero pad the window by the number of normalised values required (W/2).
2. Obtain the Fast Fourier Transform of the real signal.
3. For each complex coefficient multiply it by its conjugate.
4. Do the Inverse Fast Fourier Transform.

Furthermore, the two terms of m in equation 3.18 can be calculated incrementally using the result from $\tau - 1$ and subtracting the appropriate x^2.

Peak picking

The first step is finding all the local maxima for which τ potentially represents the period associated with the pitch. Taking only the highest maxima between every positively and negatively sloped zero crossing works well for this. It must be noted that the maximum at delay 0 is ignored and the starting point is the first positively sloped zero crossing. Where there a positively sloped zero crossing towards the end without a negatively sloped one, the highest maximum found so far (if one has been found) is accepted.

To make the accuracy of the finding of the positions of the maxima parabolic interpolation is used using the highest local value and its two neighbours.

From the maxima found a threshold is defined which is equal to the value of the highest maximum multiplied by a constant k ranging from 0.8 to 1 to avoid the aforementioned "octave error". The first maximum above this threshold is taken and the period is estimated as its delay τ.

3.1.2 Frequency-Domain Methods

Frequency-domain methods do spectral analysis (normally using some variation of the Fourier Transform and other Fourier analysis methods) to find the fundamental frequency. In this section we will overview some of the most used methods for frequency-domain pitch detection when working with singer's voices. However, it is worth mentioning that the Phase Vocoder method is one of if not the most successful method. More modern implementations of the most popular pitch corrector (AutoTune) use a Phase Vocoder.

Phase Vocoder

The phaser vocoder (PV) analyses a digital sound using frequecy-domain methdos and then resynthesizes it. The uses of phase vocoder vary from application to application. It can be used to change the pitch of a sound without altering the playback speed (a sound can be made to have a higher pitch by speeding up the playback like with the chipmunk effect). PV can also be used to instead alter the speed without altering the pitch. It is common for phase vocoders to combine both possibilities allowing the user to choose either.

Phase vocoders, whichever their use may be, follow a defined sequence of steps:

1. Windowing: the first part of the process divides the input signal into segments or blocks of samples. Those blocks are multiplied by a an envelope called a windowing function. This blocks of information will allow the system to analyze the pitch separately for different sections of the signal. By applying the windowing function the results will be less affected by the information contained in close blocks. Common windowing shapes (all of them bell-shaped) for phase vocoding are: Hamming, Kaiser, Blackman and Hann. We have found that the Hann windowing is the most popular since its particular bell shape creates very smooth results compared with the other windowing functions.

2. Fourier Transform: the resulting windows are subjected to the STFT (Short-Time Fourier Transform) giving the time and spectrum for each window. The result for each FFT window is called a frame and consists of two sets of values: amplitudes of frequency bands and the initial phase of each band.

3. Resynthesis: the analyzed data is used to resynthesize the signal (which essentially means reconstructing the signal with the windows) using the inverse STFT. The frames may be resynthesized by several methods like the OverLapp Add method (see 3.2.2). It is at this step where modifications are made to achieve the different results.

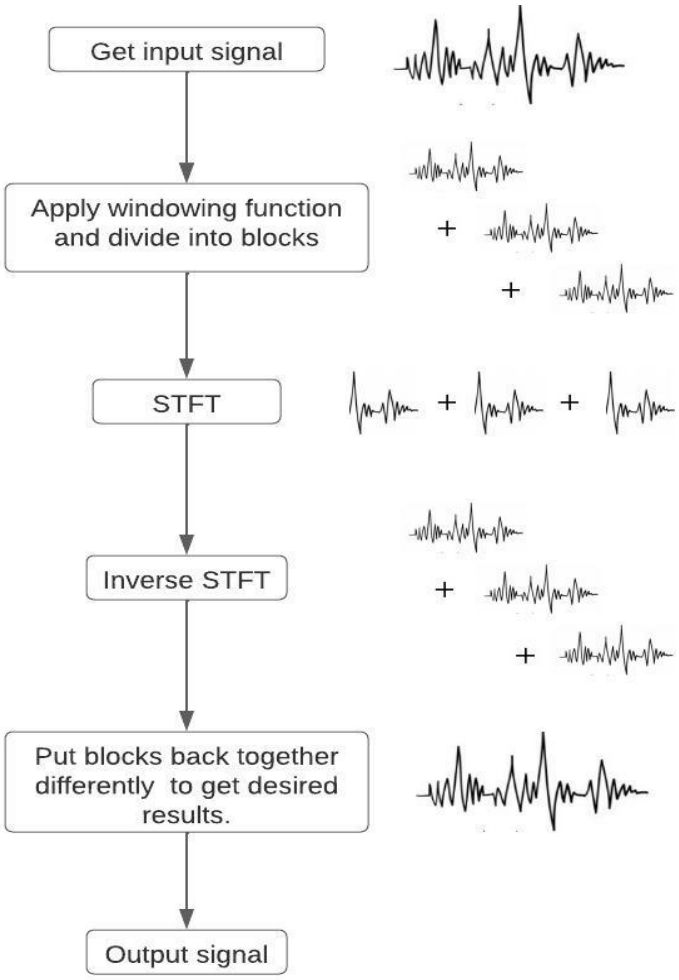


Figure 3.5: Phase Vocoder Diagram

Harmonic Product Spectrum

The first step in the algorithm [23] is dividing the input signal into windows using the Hann windowing method. Then, we apply the Short Time Fourier Transform to convert the windows from the time domain to the frequency domain.

If the input signal is a musical note, the spectrum should consist of a series of peaks corresponding to the fundamental frequency. Harmonics should be found at integer multiples of the fundamental frequency. Therefore, when we compress the spectrum a few times via downsampling and compare it with the original the strongest peaks would line up. The first peak in the original spectrum coincides with the second peak if the compression factor was two for example. When multiple spectrums are multiplied together, the result should form a clear peak marking the fundamental frequency.

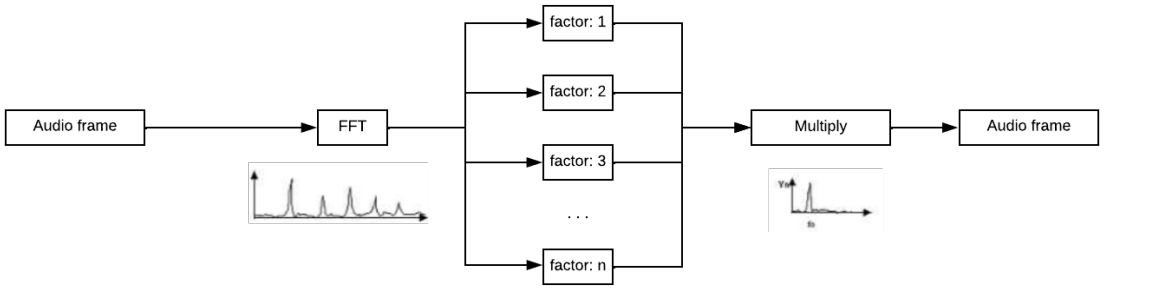


Figure 3.6: Harmonic Product Spectrum Diagram

3.2 Pitch Shifting

Pitch shifting is a technique that modifies the pitch of a recorded audio signal by either raising it or lowering it. Modifying the pitch of a sound can have many uses. One common use is transposing a song by changing its pitch using a fixed amount of tones or semitones.

The simplest method of pitch shifting alters the pitch by changing the duration. If the pitch is to be raised the duration is lowered proportionally and if the pitch is to be lowered then the duration is raised. As an example, figure 3.7 poses as the original waveform. To make the pitch higher by an octave the frequency has to be doubled and therefore the duration becomes half the original. Figure 3.8 would be the result of doing said modifications.

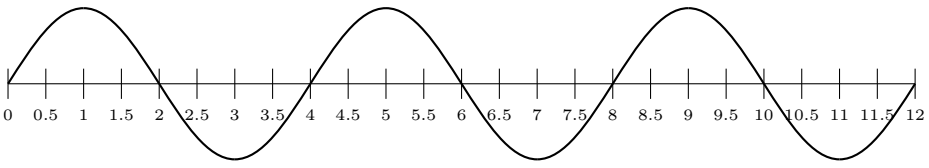


Figure 3.7: Original waveform

A more complex approach to pitch shifting aims to change the pitch without altering the playback speed. This is quite complex by itself since it involves more sophisticated resam-

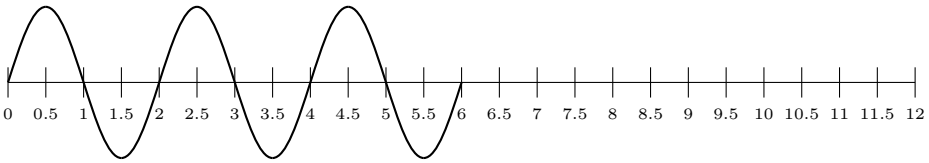


Figure 3.8: Waveform with modified pitch and duration

pling techniques to achieve this effect without producing any unwanted artifacts. Retaking the previous example, if the original pitch of the audio were to be shifted to half its original frequency without altering the speed the ideal result would look like the waveform depicted in figure 3.9.

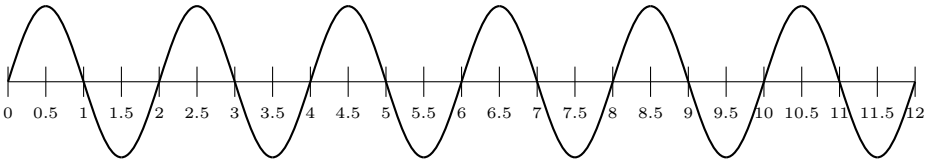


Figure 3.9: Waveform with modified pitch without altering duration

Transposing effects shift the pitch of a song by a fixed amount of tones and semitones. For example, a piece in C major could be transposed to E major by shifting the piece 3 tones up (see figures 3.10 and 3.11). However, if the piece was shifted 4 and a half tones up the new music scale would be G major (see figures 3.10 and 3.12).

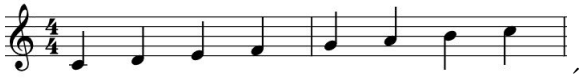


Figure 3.10: C major

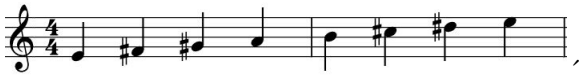


Figure 3.11: C major shifted by 3 tones becomes E major

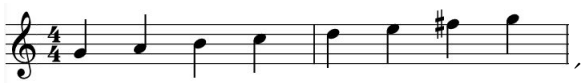


Figure 3.12: C major shifted by 4 tones and a half becomes G major

Pitch correction systems on the other hand, use the techniques to shift the pitch without altering the speed but with an added layer of difficulty. A song performed by a human

singer has many different pitches throughout its duration. Therefore, each of those pitches will have to be shifted accordingly.

Like with pitch detection, there are many different algorithms for pitch shifting. Each of them have their own purpose and utility. In this section we will discuss the three methods that where used at some point in this project.

3.2.1 Interpolation of the modified samples

Pitch shifting by tones or semitones, as seen before, uses very specific ratios. However, in a real song the differences between the correct pitch and the one obtained by the singer won't be as perfect since a pitch can be mistaken by just a few Hz. As a result, the first thing we need to do is find the ratio between the pitch found by the pitch detection algorithm and the correct pitch (see equation 3.21).

$$ratio = \frac{\text{detected period}}{\text{desired period}} \tag{3.21}$$

Once the ratio has been found, instead of simply using it, a cross-fading technique can be applied to make the transitions between samples smoother. To do that a decay parameter chosen by the user is applied to both the new ratio and the ratio previously found to smooth the modifications proportionally.

$$\text{final ratio} = decay \cdot \text{previous ratio} + (1 - decay) \cdot \text{found ratio} \tag{3.22}$$

After these calculations, the final ratio used for pitch shifting is obtained. With that, interpolation can be used to find the corresponding shifted value that needs to be written in the output. To better explain this step, say we take the audio represented by the first waveform shown in figure 3.13 and we want to shift it's pitch to double the original resulting in the second waveform shown in figure 3.13. Since these waveforms are represented as data in the computer the original waveform can be represented like shown figure 3.13. For example, to shift the pitch to half it's original value, the corresponding value for each data point has to be found. The ratio (without applying decay) would be 0.5 in this case. That means that the value for the data point found at 1 is the corresponding value that would be found at the point 0.5 in the original waveform. Since the data in the computer provided doesn't give us that value, it has to be interpolated using the data that is provided. After using interpolation to find all of the data points the resulting data for the modified waveform would be the one shown at the end in figure 3.13.

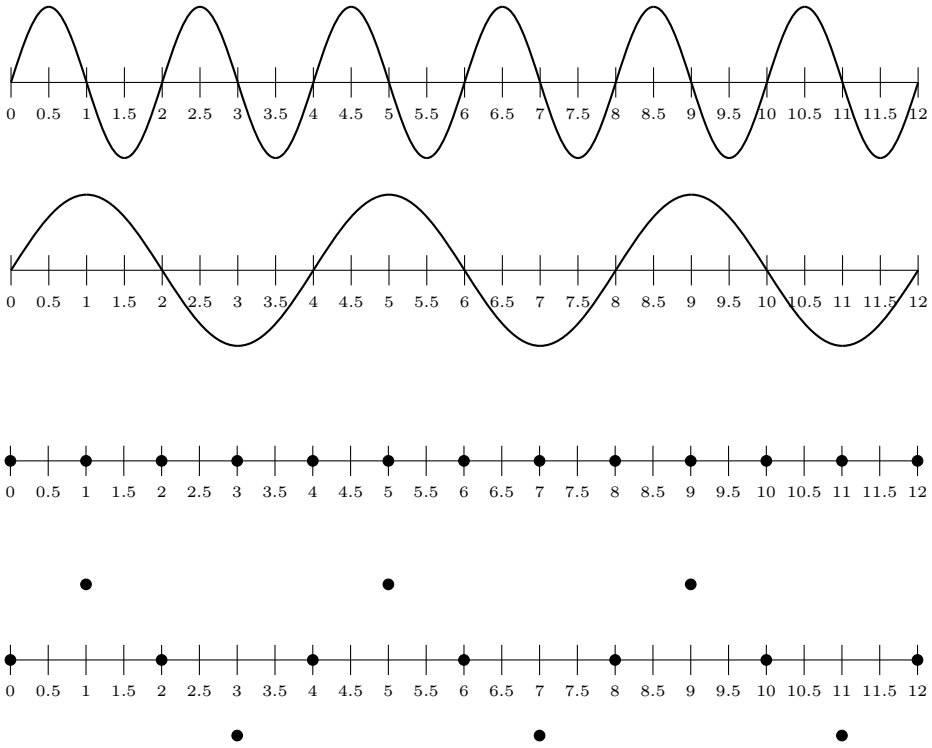


Figure 3.13: Waveforms and their corresponding data

3.2.2 Pitch-synchronous Overlap Add (PSOLA)

In signal processing, the overlap add method [25] is an efficient way to manipulate the pitch and duration of a signal. This method identifies repeating periods of the audio signal, slices out periods of the signal to get the desired number and then smooths transitions to eliminate artifacts and discontinuities.

For example, the idea to get a higher pitch is to window each period to obtain segments and the bring the segments closer together. Then, the different segments are added to create a single signal that has a reduced speed. This would be an overview of the method needed to shorten a signal. To stretch a signal the segments are separated and close segments are used to fill in the space by copying them in the middle. Then they are added like before to create a stretched single signal.

This method works very well without producing any unwanted artifacts if the pitch detection is accurate. This method was meant to be used with the YIN and MPM pitch detection methods since they work with windows and have a higher accuracy than the AutoTune method.

3.2.3 Zero-Crossing

When applying filtering to an audio signal we can allow the fundamental frequency through while also attenuating all the other frequencies. The resulting filtered samples should have two zero-crossing points per period. A zero-crossing point is the point where the values of the samples go from positive to negative and vice-versa.

The zero-crossing method used in this project [22] has three parts: a pitch tracker using that finds the length in samples of the period by counting the samples every two zero-crossing points, a compressor/expander and a pitch shifter. The compression/expansion is used to compensate for the modification of the length of the source caused by the pitch shifting. To lengthen a sound the individual cycles found with the zero-crossing points can be repeated in the output signal at the appropriate points. On the other hand, to shorten a sound cycles can be removed. The method proposed avoid introducing higher frequency artifacts involves windowing techniques.

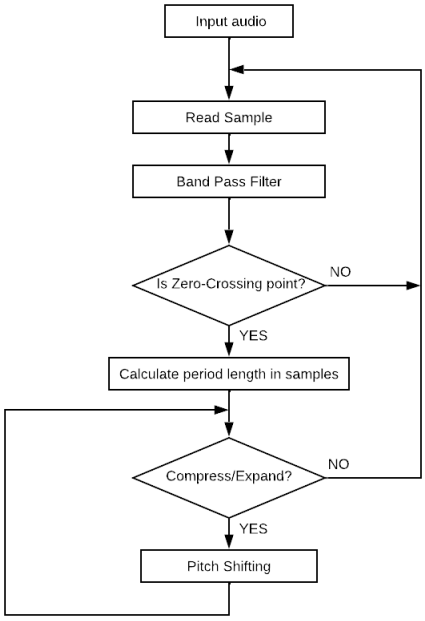


Figure 3.14: Zero-Crossing method diagram

Chapter 4

Prototype

Although different algorithms were implemented (either partially or fully) with the technologies mentioned in the following section (section 4.1) to test their performance, in the end the final implementation was based on the AutoTune's patent method [21]. The reason why is the lack of speed that the other algorithms presented. For example, a full implementation of the YIN algorithm gave very accurate results for more complex waveforms while the patent's method made some high frequency mistakes. However, it took a long time to identify a single pitch making it unusable for real time processing. A similar situation was found when testing MPM.

The only solution would have been to use frequency-domain techniques to speed up the calculations. Furthermore, the MPM paper [24] already suggests the usage of Fast Fourier Transforms (FFTs) to speed up the autocorrelation calculations. However, the objective of this project was using time-domain methods so we left these algorithms to the side and chose the one we originally proposed.

Before implementing the plugin, an later on the application, a prototype was created using the Python programming languages along with other technologies (see section 4.1). Given the lack of details found in certain parts of the patent, the prototype has allowed us to test out many different alternatives for the implementation and to compare the different versions. In this chapter, we shall explain the prototype created for the AutoTune pitch correction method.

4.1 Technologies

4.1.1 Jupyter Notebook

As stated in the website [7], the Jupyter Notebook is an open-source web application that allow data scientists to create and share documents that integrate live code, equations, computational output, visualizations and other multimedia resources, along with explanatory text in a single document. We decided to use it because it made plotting results, adjusting parameters and testing the results on different audio recordings easier.

4.1.2 SciPy

SciPy [12] provides algorithms for optimization, integration, interpolation, eigenvalue problems, algebraic equations, differential equations, statistics and many other classes of problems. This library was used in the Python prototypes to manage all the audio related elements like read a wavfile, decimation, interpolation, etc.

4.2 AutoTune Patent Prototype

To implement AutoTune's algorithm we followed these steps:

1. Decimating the audio.
2. Computing equations 3.5 and 3.6 (both can be found in page 17) for the decimated samples.
3. Finding the lag that satisfies equation 3.8 (page 18).
4. Approximating the lag for the non downsampled data.
5. Finding the correct pitch from the approximated one found.
6. Calculating the ratio for the resampling.
7. Computing the corrected output sample.

Decimation

The decimation step essentially consists in filtering the audio with an anti-aliasing filter and the downsampling by 8 (we keep 1 out of every 8 samples). To do the decimation, the `decimate` function from the SciPy library (see 4.1.2) was used. The end result is an array containing the filtered data with 8 times less samples.

Since the autocorrelation equations are computationally expensive, by reducing the size of the audio to be 8 times smaller we get two benefits. First, the amount of samples to analyze is 8 times less. Second, the range of possible periods to consider is also reduced. This results in a considerable impact on the computational cost, making it suitable for a real time implementation of the algorithm.

In the original patent, the hardware had an A/D converter that included the anti-aliasing filter. There are different possible filters that can be used for anti-aliasing. Normally, a low-pass filter is used. Since we are working with a software only implementation instead we get the original audio from a recording and the apply the `decimate` function that uses an order 8 Chebyshev type I filter as the anti-aliasing filter.

According to the *Fatigue Testing and Analysis* book [18] anti-aliasing filters are applied so that when a set of sample points is inspected, the input frequency components that are present are uniquely related to the input data. Otherwise, there would many possible input frequencies (the aliases), all of which can produce the same data points. The anti-aliasing filters theoretically should remove all but the wanted input frequencies.

Aliasing errors occur when the digitizing sampling rate is too low. As the input signal frequency nears the digitizing frequency, some components of the signal can be lost. Since here we are reducing the number of samples by 8 the same problem can be found since the data would be equivalent to an 8 times smaller sampling rate. Therefore, the same technique is used to avoid this problem.

Computing equations 3.5 and 3.6 for the decimated samples

Since we are using equations 3.5 and 3.6, we use the previously calculated value to find the current one. To do that we use two arrays with L positions, where L is the maximum possible lag where we will store the current value when it is computed so that it can be used in the next calculation.

Since we are working with samples, the lag is an integer value representing one possible cycle period. This value is, essentially, the amount of samples that are inside the cycle period. Therefore, we can calculate the lag by dividing the sample rate (number of samples per second) by the theoretical fundamental frequency to find how many samples would have been taken in the corresponding period.

Taking this into account, the maximum possible lag is computed using the minimum possible frequency given and the sampling rate. For this step, the lag is the divided by 8 since we will be working with the decimated samples making the periods 8 times smaller as well. Similarly, the minimum possible lag is computed using the maximum possible frequency and the sampling rate.

For each decimated sample, the E and H equations are computed using the simplified version proposed in the patent (see equations 3.5 and 3.6).

Finding the lag that satisfies equation 3.8

Since we are already going through all the possible values of L, the test to see whether L satisfies equation 3.8 is done once the E and H values have been updated for the current lag. To test if the results satisfy equation 3.8 we use the *eps* variable. The *eps* variable corresponds to the ϵ described in equation 3.8.

Approximating the lag for the non downsampled data

When a lag that satisfies equation 3.8 is found, the actual lag has to be approximated since the one found corresponds to the downsampled data.

To do that a separate function receives the non downsampled data, the index and the lag and uses the equations 3.3 and 3.4 to calculate the value for the non downsampled samples contained between $estimatedlag - 8$ and $estimatedlag + 8$. This is because using decimation makes us loose some information. Since the dowmsampling factor is 8, the lags between $estimatedlag - 8$ and $estimatedlag + 8$ are not evaluated. Therefore, the function calculates the autocorrelation functions for these lags.

To get a better estimation we also use interpolation to find other values in between and then find the one that minimizes the autocorrelation value. To interpolate, the implementation from the library SciPy is used. The final value found is the estimated cycle period for the original data which means that we have now found an approximation of the pitch.

Finding the correct pitch from the approximated one found

With the estimated frequency found the desired correct pitch can be estimated using an array containing the frequencies of real musical notes and finding the closest note to the incorrect pitch.

At this point, we have implemented the pitch detection step of the algorithm. If we do steps 2 to 5 for every decimated sample we will find the original pitch and the desired pitch of the audio every 8 samples. A reduced version of the code that implements the pitch detection can be seen in listing 4.3.

Now, we move on to the pitch shifting steps. The next step is taken whenever a new pitch has been found. On the other hand, the last step is taken for every **original** sample.

Calculating the ratio for resampling

We implemented the pitch shifting method that uses interpolation to find the output samples. This method was found in the AutoTune patent [21]. First, the ratio of the found lags (correct and incorrect) has to be computed 3.21. Then, to smooth the pitch shifting using the decay parameter, the ratio is modified using the previous ratio and the decay value to obtain cross-fading effect 3.22.

Computing the corrected output sample

A pointer to the input signal is kept throughout the execution (`output_addr`), this pointer is updated with the ratio and then the closest values to the pointer are used to interpolate the output sample. To do the interpolation a simple two point Lagrange interpolation was used [19]. The code that implements the pitch shifting can be seen in listing 4.2.

This pointer could advance too fast or too slow. If the ratio is greater than one, we see if the output pointer has become greater than the input pointer that is reading the samples. If that is the case `output_addr` is taken back a whole period cycle. On the other hand, we can check if it is going too slow by seeing if the ratio is lower than one and then checking if the `output_addr` plus the period is lower than the input. If that is the case we can add a whole cycle period to the output pointer while staying behind the input pointer to let the output pointer catch up. The code that implements the adjusting of the `output_addr` can be seen in listing 4.1.

```
output_addr = output_addr + resample_rate2

if resample_rate2 > 1:
    if output_addr > input_addr:
        output_addr = output_addr - Lmin
else:
    if output_addr + Lmin < input_addr:
        output_addr = output_addr + Lmin
```

Listing 4.1: Adjusting the output addr

```
if Lmin == 0:
    output_sample = 0
elif resample_rate2 == 1 and output_addr - 5 >= 0:
    output_sample = wav[index]
elif output_addr - 6 >= 0 and output_addr - 4 < len(wav):
    x = output_addr - 5
    x1 = math.floor(x)
    x2 = math.ceil(x)
    y1 = wav[x1]
    y2 = wav[x2]
    output_sample = y1 + (((y2 - y1) / (x2 - x1)) * (x - x1))

self.output[index] = output_sample
```

Listing 4.2: Pitch Shifting code

```
min_L = int(44100.0/(float(max_freq)*8))
max_L = int(44100.0/(float(min_freq)*8))

# max_L + 1 because we use values from min_L to max_L included
E_Table = np.zeros(max_L + 1)
H_Table = np.zeros(max_L + 1)
# Containes E - 2*H
sub_Table = np.zeros(max_L + 1)

for L in range (min_L, max_L + 1):

    E_Table[L] = E_Table[L] + decimated_input[decimated_index]**2 -
     ↪ decimated_input[decimated_index-2*L]**2
    H_Table[L] = H_Table[L] + decimated_input[decimated_index]*decimated_input[
     ↪ decimated_index-L] - decimated_input[decimated_index-L]*
     ↪ decimated_input[decimated_index-2*L]
    sub_Table[L] = E_Table[L] - 2 * H_Table[L]

    if sub_Table[L] < eps * E_Table[L]:

        #The function get_real_freq finds the aproximated true lag
        Lmin,freq = self.get_real_freq(wav, decimated_index, L)
        desired_freq = self.find_desired_freq(freq)
        desired_Lmin = 44100/desired_freq
        break
```

Listing 4.3: Pitch Detection code

Chapter 5

JUCE Implementation

In this chapter the final implementation will be explained. To implement it, the JUCE framework was used along with the C++ Spline library. Both of these technologies will be introduced in the following section.

The pitch correction program was implemented as a VST plugin [1] and as a standalone application application [2], both using the JUCE framework. We take the Python prototype as a starting point for this final implementation.

5.1 Technologies

5.1.1 JUCE

JUCE [6] is a partially open-source C++ framework used for the development of desktop and mobile applications (mostly audio audio applications). The main interest in JUCE for this project lies in its GUI and plugin libraries which make the process of creating an audio application or plug-in that runs on any OS easier. It is especially useful for developing VST plugins for DAWs since you only have to set up the project and build the program to create the VST plugin.

Since JUCE uses C++ it takes advantage of its object oriented features to provide a wide collection of classes and utilities. All the building blocks needed for the development are given to the user in such a manner that familiarizing yourself with the fundamentals

[1] A VST plugin is an audio plugin that can be imported into any DAW with support for the VST format

[2] A standalone application is an application that runs locally on the device and doesn't require anything else to be functional.

needed for basic projects is relatively easy. Given all the utilities provided by the framework this is a professional environment that has been used to develop many audio plugins (for example, Neural DSP [8] uses JUCE for the development of their very successful plugins).

Also, these utilities make the process of setting up audio playback, user interfaces and such very straightforward. This means that creating and designing all of the common elements of a working audio application or plugin doesn't take as much time. Furthermore, a basic user interface is created with just a few instructions so testing is easy to do in the early stages.

JUCE provides many tutorials on its website [6] featuring almost all of its utilities to help new users on the journey of developing with JUCE. Although JUCE makes the development of simple plugins very easy, testing and perfecting more complex plugins can be difficult. We found some trouble when trying to find the source of mistakes in the implementation. Also, certain types of applications requiring a high amount of samples to work require some adaptation to work with JUCE's buffer sizes.

Projucer

The Projucer is JUCE's project management tool included with the distribution that allows the user to create, manage and configure JUCE projects. To make the setting up faster the Projucer offers a few templates divided in 3 main categories. Included in said templates we can find:

- Application templates: Blank, GUI, Audio, Console, Animated, OpenGL
- Plug-In templates: Plug-in
- Library templates: Static Library, Dynamic Library

Each of these archetypes are configurable in different ways, and are initialized to include the fundamental code for each type of project. For example, when creating a Plug-in project the different formats in which the project will be built are specified using the Projucer.

Audio Plugins

For Audio Plugin projects, the Projucer can export several plugin formats, such that one build process compiles and exports all the plugin formats configured. The most important

formats included are VST and Standalone. The Standalone format allows the usage of the plugin without a DAW. However, it is worth mentioning that building VST formats is only available on Windows.

The base audio plugin project provides the developer with two fundamental classes. These two classes allow the separation of the audio processing (*PluginProcessor*) from the GUI side of the plugin (*PluginEditor*).

The *PluginProcessor* takes care of the audio processing aspects of the plugin. This is where tasks like MIDI management, audio rendering, audio editing and such take place. Many methods are included within its files. However, the main interest lies in two of them: `processBlock` and `prepareToPlay`.

`prepareToPlay` is a function used to do any initialization needed. Essentially, this function is meant to be used as a setting up function before we start processing the audio.

On the other hand, `processBlock` is the function where the developer must write their audio processing code. This function is a callback function that progressivley gives the user blocks of the audio being processed inside the Audiobuffer [3]. The code in listing 5.1 shows the content of the default `processBlock` function given when the project is created (some comments have been removed to make it shorter).

```
void ExampleAudioProcessor::processBlock
(juce::AudioBuffer<float>& buffer, juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto totalNumInputChannels  = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
        buffer.clear (i, 0, buffer.getNumSamples());

    for (int channel = 0; channel < totalNumInputChannels; ++channel)
    {
        auto* channelData = buffer.getWritePointer (channel);

        // ..do something to the data...
    }
}
```

Listing 5.1: Default processBlock function

[3] processBlock also provides the corresponding MIDI messages contained within the MIDIBuffer.

The *PluginEditor* is the object that builds the user interface and responds to interaction with graphical elements. It has two functions that are crucial for the design of the GUI: `paint`, the function that dictates the on screen layout of the component and `resized`, which contains the reactionary logic for when the component is resized. Both functions are callbacks, which means they are invoked automatically upon triggering events. These functions will be discussed further in the next section since they were mostly used for the JUCE Audio Application development.

Audio Applications

At the final stages of development a JUCE Audio Application was also created to allow the user the possibility of opening an app where they can load a file, correct its pitch and then save the results in another file.

The JUCE Audio Application template creates a minimal JUCE application but automatically adds all the setup code that is needed to easily get audio input and output. In this case, the files created are: Main.cpp and the *MainComponent* cpp and header files. All the code modfications are to be made in the *MainComponent* class.

MainComponent is a class that extends the *MainComponent* class which is the most important base class for all JUCE graphical interfaces. In JUCE, practically all visible elements are components deriving from the *MainComponent* class. The JUCE Audio Application creates a main component owned by the main application window that represents the window's content. All other components added to the main component will be children of it. Like with the Audio Plugin projects, the *MainComponent* class has two functions that are essential to the interface creation. First, we have the `paint` function, this function determines how your component should be drawn on the screen. The default code that is created for the `paint` function shown in **??** creates a solid background. Then, we also have the `resized` function, this function determines what should happen to the component and it's children when it is resized.

Since this is an audio application, the function `prepareToPlay` is also included by default along a getNextAudioBlock function that is called repeatedly to fetch subsequent blocks of audio data. After calling the `prepareToPlay` function this the `getNextAudioBlock` callback will be made each time the audio playback hardware (or any other destination being used) needs another block of data. The managing of this functions is similar to the one described in the previous section. We will not go into anymore details since these functions will not be used because there is no audio playback in the final developed application.

5.1.2 C++ Spline

C++ Spline [3] is an open source library that interpolates gid point with cubic, hermite or monotonic splines. It is light weight since, simple to use (consists of a single header file), has no dependecies and has an efficiency of O(n) for the spline generation and an efficiency of O(log(n)) for the evaluation of the spline at a point. The interpolation classes provided by JUCE were not well suited for the interpolation of single data points and didn't include the spline interpolation algorithms that we wanted to use for the pitch shifting part of the program (see section 3.2.1) so we ended up using this alternative library to do the interpolation of the output samples.

5.1.3 REAPER

REAPER [11] is a DAW that offers a multitrack audio and MIDI recording, editing, processing, mixing and mastering toolset.

This DAW was previously used in one of the electives at this university so and other previous attempts at creating a working plugin were tested using this DAW making it the perfect candidate for testing in this project. Furthermore, REAPER is relatively easy to learn for beginners who don't know anything about music production and the learning curve is quite smooth.

5.2 Plugin

The pitch correction essentially follows the steps shown in figure 5.1. This flow diagram is a representation of the different steps taken for correcting each sample. The steps are very similar to the ones taken in the prototype (see 4.2). Since this behaviour was going to be pretty much the same for both the plugin and the application, we implemented a separate C++ class to manage the different steps while the plugin and application programs simply had to call the different methods to follow this sequence of steps.

The class receives a series of parameters with its constructor to set up any previous initialization needed. The parameters given are the sensitivity (epsilon in equation 3.8), the minimum and maximum frequencies detectable, the sampling rate and the decay used for the smoothing of the resampling rate (see equation 3.22). The minimum and maximum frequencies are used to find the minimum and maximum lags possible that need to be checked for the autocorrelation functions E and H. Same as with the prototype, this minimum and maximum lags are divided by 8 because the autocorrelation is applied to the decimated samples making the periods 8 times smaller.

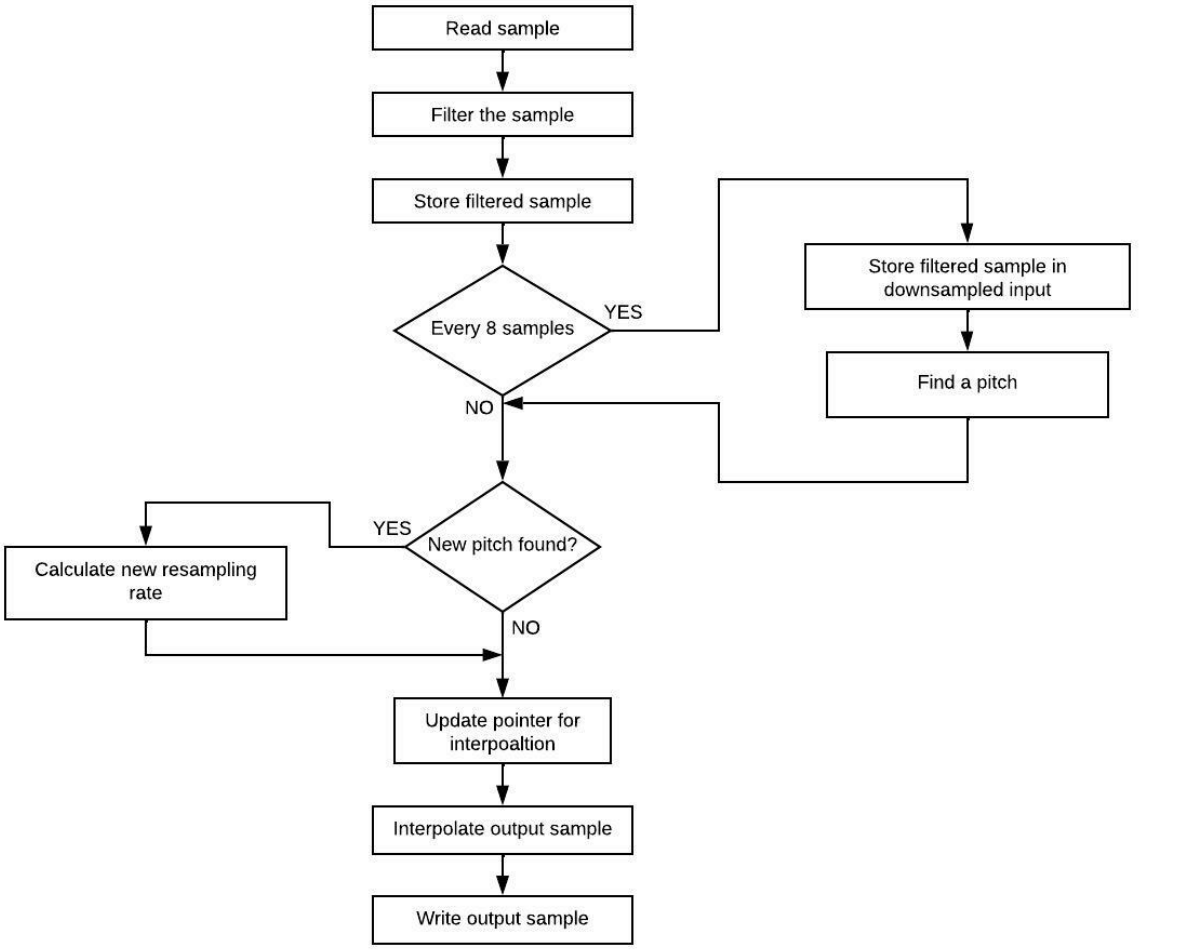


Figure 5.1: AutoTune simplified flow diagram

Before explaining the details about the implementation of the plugin itself we will proceed to explain the implementation of the class. The class is called AutoTune and is created using both a cpp and header files. As seen in the diagram (see 5.1), the class had to provide methods for filtering a sample, storing the sample to use it later the step that finds the more approximated estimation, storing the sample every 8 iterations into the vector used for the autocorrelation functions, finding the pitch every 8 iterations, calculating the new resampling rate if a new pitch is found, updating the pointer for the interpolating and also interpolating the output sample. The reading and writing of samples is managed by the plugin and application.

Each of those steps were implemented in their own separate public method:

- `add_sample`: this function takes a sample, filters it and stores it inside a vector containing the samples being processed.
- `add_decimated_sample`: this function stores the last sample that was added with add_sample into the decimated samples being processed.
- `get_fundamental_frequency`: this function computes the E and H functions, finds a lag satisfying equation 3.8 and then uses a private function called `get_real_lag` to find a more accurate lag. Then it returns the fundamental frequency found.
- `calculate_resample_rate`: this function finds the correct frequency using a private function called `get_desired_frequency` and then updates the resampling rate using the detected fundamental frequency and the correct one that was estimated.
- `update_output_addr`: this function updates the pointer used for interpolation using the period detected.

All of these methods along with the private methods used by them were implemented in the same way as seen in the Python prototype (see 4.2). The main differences lie in the usage of the C++ Spline library (see 5.1.2) for the interpolation used when estimating a more accurate lag and the need to implement the filtering and downsampling needed for decimation separatly, since JUCE did not provide any utilities for decimation and the interpolators provided did not include the spline interpolators that we wanted to use. Other than that, the implementation is virtually the same once the main program does the appropriate calls.

Now, let's get into the details of the main programs implementation. First of all, we will discuss the audio processing involved in the program. The program does the pitch correction using the AutoTune class by declaring it inside the *PluginProcessor.h* file with a constructor using intermediate values. As seen in previous sections (see 5.1.1), a JUCE plugin has two main methods inside the *PluginProcessor* class that manage the auido processing: `prepareToPlay` and `processBlock`.

Since all the initializations and modifications needed are made inside of the AutoTune class, the `prepareToPlay` in this case was only used to initialize the sampling rate using the methods argument sampleRate with the rest of the parameters being intermediate values until the user given parameters could be read in `processBlock`.

Then the main audio processing code was written inside the `processBlock` method. In this method, the first thing to do is to read the parameters given by the user from the interfaces parameters. Then this values are used to update the values inside the AutoTune class using the specific public method `set_parameters`. The choice of parameters given to the user include the senstivity (epsilon), the minimum and maximum frequencies and the decay value. Once the AutoTune class has been updated the audio processing

resumes. Since `processBlock` is a callback that progressively gets chunks of audio data and gives them to the developer inside the buffer argument, the steps shown in the flow diagram (see 5.1) are taken for every sample inside the buffer. A loop goes through every sample of the chunk and calls the AutoTune method `add\_sample` giving it he current sample inside the buffer to filter it and store it inside the AutoTune class. Then, if the current reading index divided by 8 has module 0 the methods `add_decimated_sample` and `get_fundamental_frequency` are called to store the decimated sample and find the current detected pitch. Since the `get_fundamental_frequency` returns 0 when the pitch detection failed then if the fundamental frequency stored in variable freq is not 0 a pitch has been found and the method `calculate_resample_rate` is called with this new fundamental frequency as argument. Then, the pointer for interpolation is updated using the `update_output_addr` method with the current lag found as argument and the output sample is interpolated using the `get\_output\_sample` method. Afterwards, the output sample is written inside the buffer.

Finally, we shall explain how the user interface was set up. Unlike with the application, the plugins interface was very straight forward to implement. The tutorial provided in the Audio Ordeal website [27] was very useful for simplifying this process. The four parameters are first declared inside the *PluginProcessor.h* file as AudioParameterFloats. Then the four of them are added inside the *PluginProcessor* constructor with the method `addParameter` declaring for each of them their identifier, their name and the range of values allowed for each as well as the default value. Finally, to create a GUI with the parameters as sliders the `createEditor` method was modified to return a *GenericAduioProcessorEditor* using the *PluginProcessor* as argument. This class takes the *PluginProcessor* and prints the AudioParameterFloats as sliders in the GUI. The end result is the interface shown in 5.2.

The performance of the plugin in a real DAW was done using REAPER. While the plugin works as intended the computations are too slow for the real time requirements imposed by the DAW and the resulting sound has lag when being played.

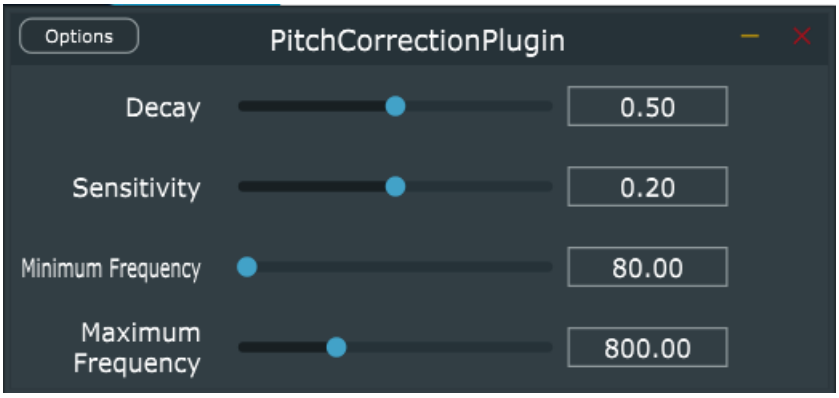


Figure 5.2: JUCE Plugin GUI

```
void PitchCorrectionPluginAudioProcessor::processBlock(juce::AudioBuffer<float>&
    ↪ buffer, juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto totalNumInputChannels  = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
        buffer.clear (i, 0, buffer.getNumSamples());
    //Update the parameters of the pitch correction
    autotune.set_parameters(sensitivity_parameter->get(), min_frequency_parameter
    ↪ ->get(), max_frequency_parameter->get(), decay_parameter->get());

    double freq = 0;
    double Lmin = 0;
    auto* left_data = buffer.getWritePointer(0);
    auto* right_data = buffer.getWritePointer(1);

    //Go through all the samples in the buffer/chunk
    for (int i = 0; i < buffer.getNumSamples(); i++) {
        //Store the sample in the vector containing all the samples
        autotune.add_sample(left_data[i]);
        if (i % 8 == 0) { //Every 8 samples . . .
            //Store the sample in the decimated vector
            autotune.add_decimated_sample();
            //Find the current pitch
            freq = autotune.get_fundamental_frequency();
            Lmin = getSampleRate() / freq;
        }
        //Update the resampling rate if a pitch was found
        if (freq != 0)
            autotune.calculate_resample_rate(freq);

        double output_sample = 0;
        //Update the output pointer for the interpolation
        autotune.update_output_addr(Lmin);
        //Interpolate the output sample
        output_sample = autotune.get_output_sample(Lmin);
        //Write the sample into the buffer
        left_data[i] = output_sample;
        right_data[i] = output_sample;
    }
}
```

Listing 5.2: JUCE plugin processBlock

5.3 Application

We decided to create JUCE Audio Application to provide a tool that allows the user to load a file, correct its pitch and then save the results inside another file that can be heard separately. Later on it had the added perk of allowing the user to listen to the end result without lag since the plugin introduces lag when playing in a DAW.

First, we set up the interface. The GUI has 4 sliders representing the same parameters used in the plugin, a combo box that allow the user to provide the sampling rate and three buttons. The buttons let the user load a file, tell the program to correct the pitch of the file and then save the results in a different file.

To create all the different components we used the The ComboBox class [14] tutorial, Build an audio player [15] tutorial and The Point, Line, and Rectangle classes [16] tutorial as references. All of these tutorials were provided by JUCE and can be found in their website [6] among many other helpful tutorials. The Build an audio player tutorial also gave us the tools to create the load and save buttons so that a file chooser is opened allowing the user to choose which file to load the data from or save the results into.

The buttons were created using JUCEs *TextButton* class. First they were declared inside the private section in the *MainComponent*s header file. Then, an auxiliar method was also declared to hold the code to be executed for each button when they are clicked. Then they were configured inside the constructor in the *MainComponent*.cpp file. To do that the method `addAndMakeVisible` was called for each of the buttons and the method to be executed when the button is clicked was defined. Afterwards, their positioning in the interface was specified inside the resized function (see 5.1.1). Each button was placed in the same y coordinate while the x coordinate was chosen by dividing by three the width of the window and placing the buttons in each of the 3 sections created.

The load and save buttons have a similar behaviour. To manage files and formats JUCEs AudioFormatManager class was used by declaring a format manager in the header and the initializing by usign the method `registerBasicFormats` inside the constructor. Once the format manager is set up it can be used in both the file loading method and the file saving method to create the appropriate audio reader and writer. To give the user the choice of which file to use for both in a familiar manner the FileChooser class was used. Since the application is meant for wavfiles the file chooser is initialized to choose this format. Now, to use the file chooser to choose a file for loading the method `browseForFileToOpen` is used. On the other hand, to choose a file for saving the method called is `browseForFileToSave` with the argument true to warn the user about overwriting files. In both the loading and saving methods when and udio file has been chosen the file is saved in a File by calling the method `getResult`. Up to this point, the implementation for both the loading and saving of files is pretty similar. It is here where different steps need to be taken for each.

To load a file, the *AudioFormatReader* class is used. We take the formatManager and call the method `createReaderFor` and pass the chosen file as the argument. If the reader is created correctly, it can be used to read the file and store the samples inside an audio buffer. To do that an *AudioBuffer* was declared in the header file to store the input samples while another one was declared to store the output. First, the input and output audio buffers have to be resized to match the number of channels and number of samples detected using the reader. Then the samples are put into both the input and output buffers by using the readers `read` method. The samples are stored in both the input and output buffers to make sure that if the user where to save results without doing any pitch correction the output samples would match the input samples and the resulting file would simply be a copy of the original.

To save a file, the *AudioFormatWriter* was used to create a writer for the file chosen by the user. To do that a writer in the wav format is used by declaring a *WavAudioFormat* and then calling its method `createWriterFor` with the file as an argument. Then, if the writer was created successfully, the writers `writeFromAudioSampleBuffer` method can be used to save into the chosen file the contents found in the ouptut audio buffer.

Finally, the pitch correction occurs when the correct pitch button is clicked. The pitch correcting method is very similar to the one created for the plugin (see 5.1.1). However, some adjustments were made to fit the processing of the whole audio. First, the AutoTune class described in the previous section (see 5.1.1) is used to manage the pitch detection and pitch shifting methods like with the plugin. The constructor is called at the start of the method with the different values being taken from the slider and combo box components added in the GUI to allow the user to manually choose which values to use. Then, the audio processing can begin. In this case, a loop goes through all the samples inside the output audio buffer and performs the same steps seen in the `processBlock` of the plugin. For each sample, the output sample that was interpolated using the AutoTune class is stored inside the output audio buffer at the current index. When the pitch correction is finished the text of a label in the GUI is modified to tell the user that the pitch correction has finished and the results can be saved correctly.

To add the sliders to the GUI for the decay, sensitivity and frequency parameters, each of the parameters had to have their own declaration inside the header with the corresponding Slider an Label. Each Slider needs to also have a Label to show the name of the parameter so the user knows which sliders corresponds to each parameter. Then, each slider is added inside the constructor and initialized with a range of values and a default value that falls at a number that we have found to give acceptable results in most cases. Then, the corresponding label is also added and then attached to the slider so it is placed alongside it when printing the GUI. The text for the label is also chosen inside the constructor. To finish, the sliders were placed inside the GUI using the resized method (see 5.1.1) so the sliders appeared one after the other bellow the label marking the start of the parameters section.

The last component to be added is the combo box that allows the user to choose between a sampling rate of 48000 and a sampling rate of 44100. Since the JUCE application does not provide the sampling rate when reading a file the user has to input the sampling rate manually. 48000 is chosen as the default value since nowadays it is the most common sampling rate. To add the combo box the *ComboBox* class is used by first declaring the combo box and its label inside the header file (same as with the sliders, the label is necessary to tell the user what the combo box is for). Then, The combo box is added inside the constructor and initialized to contain the mentioned values and have 48000 as the default value. Afterwards the label is added and attached to the combo box like with the sliders. The component is then placed bellow the sliders using the resized method.

Once all of the components were configured the final look of the GUI can be seen in figure 5.3.

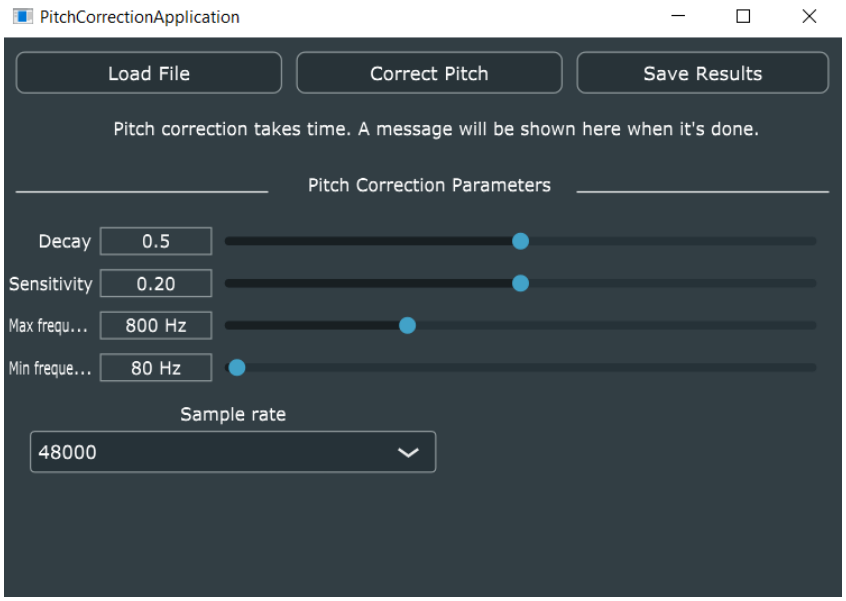


Figure 5.3: JUCE Application GUI

Chapter 6

Conclusions and Future Work

6.1 Conclusions

For the development of the pitch correction tools, we researched a wide variety of pitch detection and pitch shifting algorithms. We have found that, although all the algorithms were meant for pitch correction, not all of them were suitable for detecting the pitch of a singers voice. Many algorithms were meant to be used only for speech sounds. These sounds do not include such a wide range of possible fundamental frequencies and have a lower harmonic complexity. After finding some algorithms both in the time domain and in the frequency domain suitable for detecting the pitch of a singers voice, we have studied them on a theoretical level. We then did the same with pitch shifting algorithms. In this case we also were looking for algorithms that could be paired with the previously studied pitch detection algorithms to create a pitch correction algorithm. The research of the pitch detection and pitch shifting algorithms led to the study of other algorithms and concepts widely used (whether in audio programming or in other applications). For example, we also had to look into the Fast Fourier Transform and how it is applied in spectral analysis, decimation, interpolation, filtering, aliasing, windowing, etc. As a result we achieved deeper understanding of the projects subject (pitch correction) and audio programming as a whole.

Since we decided to use a time-domain method because of AutoTune's patent [21], we took a special interest in finding alternative time-domain algorithms and go into more detail when explaining them in section 3.1.1. The two other main algorithms found, apart from AutoTune's method, for finding the pitch of a singer's voice were YIN [20] and MPM [24]. We found YIN to be a especially common choice for pitch detection (for example, the popular library Aubio [1] uses YIN for its pitch detection methods).

To choose which algorithm to use for the implementation of the plugin, and later on the application, we decided to create simple prototypes to test the performance of the different

algorithms to determine which one gave the most accurate results in real time. Both YIN and MPM start with an autocorrelation function and then go on to add different steps to improve the accuracy of the estimation (see section 3.1.1). We tested their performance separately at each stage to see how they measured up against each other and AutoTune's method. When using a perfectly periodic signal generated with an oscillator all the algorithms gave perfect estimations. However, when including alterations to the waveforms like attenuation we found that the full implementations of YIN and MPM provided the most accurate results. Although it became clear that YIN and MPM were the most accurate, they did not meet the time-domain constraints without using frequency-domain techniques to improve their speed (for example, Aubio's fast implementations of YIN use the Fast Fourier Transform to speed up the autocorrelation calculations). As a result, we went on to compare partial implementations with AutoTune's and found AutoTune's method to provide approximately the same accuracy with less computational cost. Therefore, the final algorithm chosen was AutoTune's method.

To do some more tests on the performance of the full pitch correction algorithm we extended AutoTune's prototype to include the pitch shifting. We used the pitch shifting method found in the patent since it was the most suitable for its pitch detection method (see section 3.2.1). Once we had achieved an operational pitch correction prototype we tested it with real audio recordings and then we went on to implement the plugin and application.

We implemented the pitch correction tool both as a plugin and as an application using the JUCE framework, a professional environment used for the creation of a lot of popular commercial audio plugins and applications (see section 5.1.1). The plugin works in real time as intended and is in the VST format making it compatible with a lot of DAWs (download from [17]). Since the application was created also with JUCE it runs in any operative system (download from [10]).

It is worth mentioning that choosing to use time-domain methods, while proving to be very interesting in theory and very accurate, has given us many problems in real time processing. Both YIN and MPM required the usage of frequency-domain methods to work in real time. The application and plugin implementations of the pitch correction tool using AutoTune's method introduces high-frequency mistakes due to errors in the pitch detection. The plugin also introduces some lag. When implementing AutoTune's mechanisms for improving the estimation of the pitch and reducing the computational cost we ran into some limitations due to the nature of the original system. AutoTune's patent describes a hardware dedicated to running its pitch correction program, improving its performance compared to a software system like ours. Also, the original system handles the arrival of samples as interruptions, giving it the opportunity of using the time in between to perform pending autocorrelation computations reducing the computational cost. Furthermore, we found a lack of details overall in the description of the patent. We strongly suspect that this was done on purpose in order to not let other replicate its results.

Although we have achieved the original objective of creating a VST plugin that performs pitch correction in real time (the plugin is available at [17]) with the addition of also implementing the algorithm as an application (the application is available at [10]), the behavior, as expected given the limitations found, isn't good enough for using it in real projects. We provide some original audio recordings at [9] and the result of using the application to correct their pitch at [2].

Finally, the prototype's code and the plugin and application code can be seen in a GitHub repository at [5].

6.2 Future work

At this point we see two possible alternatives:

1. **Abandon the time-domain only restriction.** The YIN and MPM algorithms gave very accurate results. We could include the frequency-domain methods needed to improve their performance and use them as the pitch detection algorithms to get rid of the high-frequency mistakes. We would also need to change the pitch shifting method. A possibility would be using the PSOLA method (see section 3.2.2). Seeing the performance of other implementations using these methods, we should also get rid of the lag found in the plugin. Another option would be to implement the pitch correction tool as a phase vocoder (a very popular implementation nowadays), although that would require abandoning all the other work done so far (see section 3.1.2).

2. **Improve the current performance**. If we were to go on with the current implementation, we could do the following modifications to improve its performance:

    - Add MIDI support to allow the user to choose which pitch is the desired one. This should reduce some of the correction mistakes due to the estimation of the correct pitch.
    - Add a selection tool that gives the user the chance to specify which sections need to be corrected and which sections do not. This would allow the user to manually ignore sections with high-frequency mistakes.
    - Add a way to let the user give the program specific ranges of possible frequencies for different sections. This way, we would restrict the range of possible frequencies reducing the high-frequency mistakes.
    - Add a transposing functionality. Since we are already doing pitch shifting, it could be useful to allow the user to pitch shift the whole recording by a certain amount of tones and semitones.

- Find a way to use threads in JUCE to improve the performance of the pitch detection algorithm.

Chapter 7

Conclusiones y Trabajo Futuro

7.1 Conclusiones

Para el desarrollo de la herramienta de corrección de afinación investigamos varios algoritmos de detección y modificación de la afinación. Descubrimos que, aunque todos los algoritmos tienen el mismo objetivo, no todos son utilizables para la detección de la afinación de la voz cantada. Muchos algoritmos estaban pensados para la voz hablada. Estos sonidos no tienen un rango tan amplio de posibles frecuencias fundamentales y tienen menor complejidad armónica. Tras encontrar algunos algoritmos, tanto en el dominio del tiempo como en el dominio de la frecuencia, adecuados para su uso con voces cantadas, los estudiamos con mayor profundidad a nivel teórico. Después hicimos lo mismo con algoritmos de modificación de pitch. En este caso, también buscábamos algoritmos que pudieran emparejarse con los algoritmos de detección de pitch de forma que la combinación resultante pudiera ser utilizada para la corrección del pitch. El estudio de estos algoritmos conllevó el aprendizaje de otros algoritmos y conceptos muy utilizados (tanto en programación de audio como en otras aplicaciones). Por ejemplo, tuvimos que investigar sobre la Transformada Rápida de Fourier, el filtrado de frecuencias, la interpolación, el alisasing, etc. Como resultado, obtuvimos una comprensión más profunda de la esencia del proyecto (la corrección del pitch) y de la programación de audio en su totalidad.

Dado que decidimos utilizar un método en el dominio temporal debido a la patente de AutoTune [21], tomamos un especial interés en encontrar algoritmos en el dominio temporal alternativos y entramos más en detalles a la hora de explicarlos en la sección 3.1.1. Los otros dos algoritmos encontrados, aparte del método de AutoTune, para estimar la afinación de la voz de un cantante fueron YIN [20] y MPM [24]. Hemos visto que YIN es una elección especialmente común (por ejemplo, la popular librería de audio Aubio [1] utiliza YIN para sus métodos de detección de pitch).

Para elegir qué algoritmo utilizar en la implementación del plugin, y más tarde en la aplicación, decidimos crear prototipos en Python para probar el rendimiento de distintos algoritmos con el fin de determinar cuál proporcionaba los resultados más precisos en tiempo real. Tanto YIN como MPM empiezan con la autocorrelación como base y luego añaden más pasos para mejorar la estimación del pitch (see section 3.1.1). Probamos el rendimiento de cada etapa por separado para ver cómo se comportaban respecto al resto y al método de AutoTune. Al utilizar una señal perfectamente periódica generada con un oscilador todos los algoritmos estimaban correctamente el pitch. Sin embargo, al introducir alteraciones como atenuación a la señal encontramos que las implementaciones completas de YIN y MPM eran las más precisas. Sin embargo, ambos algoritmos requerían demasiado tiempo para obtener la estimación y requerían el uso de técnicas como la Transformada Rápida de Fourier o convoluciones para acelerar su ejecución (por ejemplo, la implementación de Aubio de YIN utiliza FFT para acelerar el cálculo de la autocorrelación). Como resultado, pasamos a comparar las etapas anteriores con el método de AutoTune y descubrimos que todos obtenían una estimación similar y que el algoritmo de AutoTune era un poco más rápido. Por lo tanto, el algoritmo elegido finalmente fue el propuesto en la patente.

Con el fin de realizar tests sobre el algoritmo de corrección de pitch completo, extendimos el prototipo de AutoTune para incluir la modificación del pitch. Utilizamos el método encontrado en la patente (ver sección 3.2.1 ya que era el más adecuado para el algoritmo de detección de pitch. Una vez obtuvimos un prototipo funcional hicimos varias pruebas con grabaciones reales y después pasamos a la implementación del plugin y la aplicación.

Implementamos la herramienta de corrección de pitch tanto como plugin como aplicación utilizando el framework JUCE, un entorno profesional utilizado para la creación de múltiples plugins y aplicaciones comperciales populares de audio (ver sección 5.1.1). El plugin funciona en tiempo real y está en formato VST tal y como se quería por lo que es compatible con muchos DAWs (se puede descargar en [17]). Dado que la aplicación se creó también con JUCE corre en cualquier sistema operativo (se puede descargar en [10]).

Cabe mencionar que el elegir utilizar algoritmos en el dominio temporal nos ha dado muchos problemas al procesar en tiempo real. Tanto YIN como MPM requerían el uso de técnicas como la Transformada de Fourier para funcionar en tiempo real. La aplicación y el plugin implementados con el método de AutoTune introducen errores de alta frecuencia debido a errores en la detección de pitch. El plugin además introduce un poco de lag. Al implementar los mecanismos propuestos en la patente para mejorar la estimación del pitch y reducir el coste computacional encontramos una serie de limitaciones debidas a la naturaleza del sistema original. La patente describe un hardware dedicado a ejecutar el programa lo que mejora su rendimiento frente a una implementación software como la nuestra. Además, el sistema original gestiona la llegada de samples como interrupciones aprovechando el tiempo entre llegadas para hacer cálculos pendientes de autocorrelación reduciendo así el coste. Por otro lado, encontramos una falta de detalles generalizada en la descripción de la patente. Sospechamos que esto es intencionado para dificultar la replicación del sistema por terceros.

Aunque hemos conseguido el objetivo original de implementar un plugin VST que hace corrección de pitch en tiempo real (el plugin está disponible en [17]) con el añadido de haber implementado una aplicación que también realiza esta tarea (la aplicación está disponible en [10]), el comportamiento, como cabía a esperar dadas las limitaciones descritas, no es suficientemente bueno como para que sea utilizable en proyectos reales de producción musical. Proporcionamos grabaciones originales de audio en [9] y el resultado de utilizar la aplicación para corregir el pitch en [2].

Para finalizar, el código del prototipo y el código del plugin y la aplicación está disponible en un repositorio de GitHub en [5].

7.2 Trabajo futuro

Llegados a este punto vemos dos posibles alternativas:

1. **Abandonar la restricción del dominio temporal.** Los algoritmos de YIN y MPM dieron resultados muy precisos. Podríamos incluir los métodos del dominio de frecuencias para mejorar su rendimiento y utilizarlos como el algoritmo de detección de pitch y de esa manera eliminar los errores de alta frecuencia. Esto requeriría cambiar también el algoritmo de modificación del pitch. Una posible alternativa sería utilizar el algoritmo PSOLA (ver sección 3.2.2). Viendo el funcionamiento de otras implementaciones que utilizan estos métodos, debería ser posible eliminar de esta manera el lag introducido por el plugin al reducirse el tiempo de cálculo de la autocorrelación. Otra opción sería implementar la herramienta como un phase vocoder (muy popular hoy en día) aunque esto requeriría abandonar el trabajo de implementación hecho hasta ahora (ver sección 3.1.2).

2. **Mejorar el rendimiento actual.** Si quisiéramos continuar con la implementación actual, podríamos hacer las siguiente modificaciones para mejorar su funcionamiento:

   - Añadir soporte MIDI para permitir al usuario elegir el pitch deseado, Esto reduciría algunos de los errores debidos a la estimación del pitch correcto.
   - Añadir un sistema de selección que permita al usuario seleccionar qué partes quiere corregir y qué partes no. De esta manera, el usuario podría ignorar manualmente secciones conflictivas.
   - Añadir una manera de permitir al usuario seleccionar manualmente rangos de posibles frecuencias fundamentales específicos para diferentes partes de la grabación. De esa manera, se reduciría el rango de posibles frecuencias que explora el algoritmo reduciendo los errores de alta frecuencia.

- Añadir una funcionalidad de transporte, Ya que se está haciendo modificación del pitch podría ser útil permitir al usuario corregir la afinación y además transportar la grabación especificando el número de tonos y semitonos con una sola herramienta.
- Buscar una manera de utilizar hilos en JUCE para mejorar el rendimiento de la detección de pitch.

Bibliography

[1] Aubio. https://aubio.org.

[2] Corrected audio recordings. https://mega.nz/folder/cwlBAQ5R#oeabmi7-s9w8KEqw4nlJyQ.

[3] C++Spline website. https://kluge.in-chemnitz.de/opensource/spline/.

[4] Fast Fourier Transformation FFT - Basics. https://www.nti-audio.com/en/support/know-how/fast-fourier-transform-fft.

[5] Github repository. https://github.com/LaCabraDeLaLegion/TFG-PitchCorrection.git.

[6] JUCE website. https://juce.com.

[7] Jupyter website. https://jupyter.org.

[8] Neural dsp. Plguins web page https://rb.gy/agjxzs.

[9] Original audio recordings. https://mega.nz/folder/UltkwBxA#jcnZLGPNPFOJdcHLnmxlXQ.

[10] Pich correction application download. https://mega.nz/file/V411iJhI#aKg5mOj4hVcjzrK8OKHPOC9vPetDC4s9oMXuQxCsaSU.

[11] REAPER website. https://www.reaper.fm.

[12] SciPy website. https://scipy.org.

[13] Soundstream. https://soundstream.com.

[14] Tutorial: Build an audio player. https://docs.juce.com/master/tutorial_combo_box.html.

[15] Tutorial: Build an audio player. https://docs.juce.com/master/tutorial_playing_sound_files.html.

[16] Tutorial: The point, line, and rectangle classes. https://docs.juce.com/master/tutorial_point_line_rectangle.html.

[17] VST plugin download. https://mega.nz/folder/ph1BGRqA#bTk32KZAQPEzF0KBk4OJkA.

[18] In Yunk-Li Lee, Jwo Pan, Richard B. Hathaway, and Mark E. Barkey, editors, *Fatigue Testing and Analysis*, pages 395–402. Butterworth-Heinemann, Burlington, 2005.

[19] Branden Archer and Eric W. Weisstein. Lagrange interpolatiing polynomial. From MathWorld–A Wolfram Web Resource. https://mathworld.wolfram.com/LagrangeInterpolatingPolynomial.html.

[20] Alain de Cheveigné and Hideki Kawahara. Yin, a fundamental frequency estimator for speech and music. *The Journal of the Acoustical Society of America*, 111 4:1917–30, 2002.

[21] Harold A. Hildebrand. Pitch detection and intonation correction apparatus and method, Oct 1999. US5973252A.

[22] Keith Lent. An efficient method for pitch shifting digitally sampled sounds. *Computer Music Journal*, 13(4):65–71, 1989.

[23] Charlet Reedstrom Mark Husband, Adan Galvan and Richard Baraniuk. OpenStax CNX, 2011.

[24] Philip Mcleod and Geoff Wyvill. A smarter way to find pitch. In *In Proceedings of the International Computer Music Conference (ICMC'05)*, pages 138–141, 2005.

[25] Eric Moulines and Francis Charpentier. Pitch-synchronous waveform processing techniques for text-to-speech synthesis using diphones. *Speech Communication*, 9(5):453–467, 1990. Neuropeech '89.

[26] L. R. Rabiner, M. J. Cheng, A. E. Rosenberg, and C. A. Mcgonegal. A comparative performance study of several pitch detection algorithms. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 24(5):399–418, 1976. Cited By :534.

[27] Alex Rycroft. How to build a VST - Lesson 1: Intro to JUCE website. https://audioordeal.co.uk/how-to-build-a-vst-lesson-1-intro-to-juce/.

[28] Lyudmila Sukhostat and Yadigar Imamverdiyev. A comparative analysis of pitch detection methods under the influence of different noise conditions. *Journal of Voice*, 29(4):410–417, 2015.

[29] Peter Veprek and Michael S Scordilis. Analysis, enhancement and evaluation of five pitch determination techniques. *Speech Communication*, 37(3):249–270, 2002.

[30] Eric W. Weisstein. "fast fourier transform." from mathworld–a wolfram web resource. https://mathworld.wolfram.com/FastFourierTransform.html.