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

Resumen en castellano

Breve resumen del trabajo.

Palabras clave

Lista de palabras clave

Abstract

English version of the abstract

Keywords

List of keywords

Contents

Chapter 1

Introduction

1.1 Background

1.1.1 Music Production

When talking about music production we can include or exclude any step in the creation and release of a musical piece. However, its most common definition includes the recording and editing of the music.

Most people, when they think about a music production studio, imagine an audio workstation 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 have DAWs (Digital Audio Workstations) and can have all the functionalities of the classical audio workstation within their computer.

Furthermore, the development of plugins for said DAWs have given the producers the ability to use so many tools to create, analyze and modify tracks that even if there are some imperfections in the recording there is no need to start over since those mistakes are fixable with technology.

Even if almost any mistake is fixable nowadays, not too long ago pitch mistakes were not. In came AutoTune, a system that could detect the pitch of a recording and modify it so that the pitch is correct maintaining the original content and speed.

DAWs

Today's musical landscape would look completely different without the DAWs. The majority of artists working today would rarely record with anything else; such is the impact they've had on music production.

Now, what is a DAW? Simply put, 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.

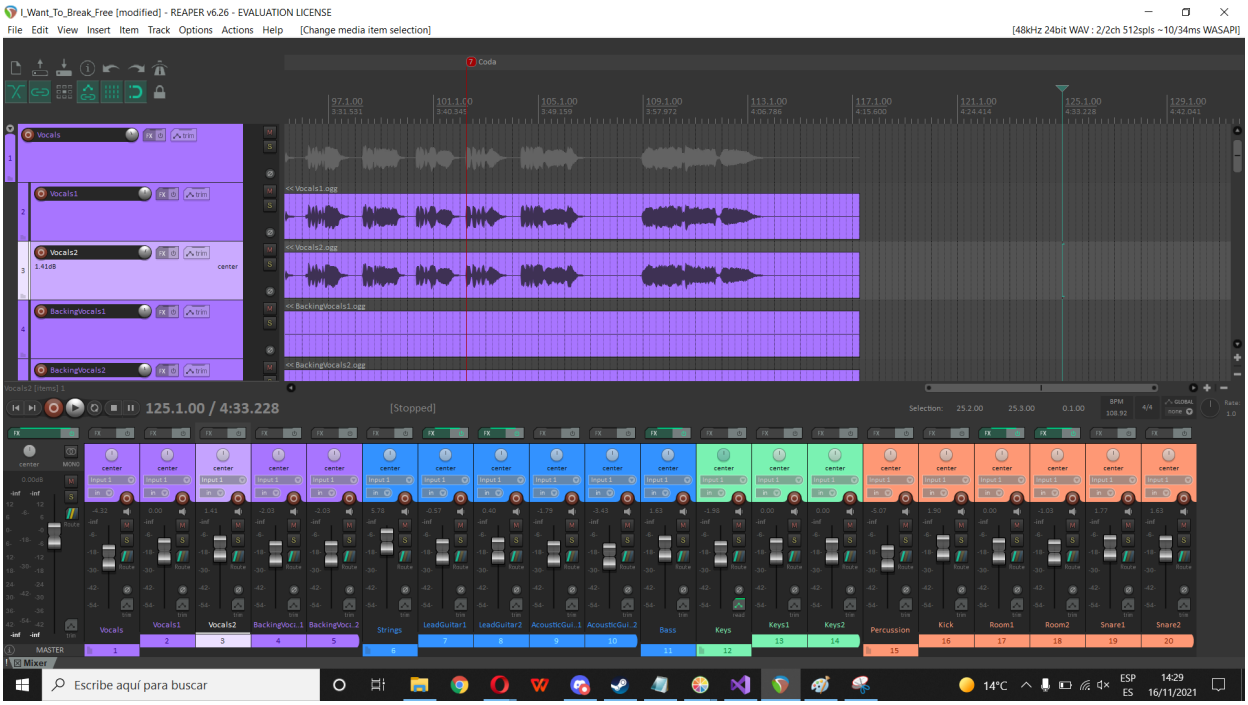

Figure 1.1: *REAPER DAW with multiple tracks*

Digital Pulse Code Modulation (PCM), the digital representation of a sampled analogue signal, was developed by Bell Lab, and while experiments in digital recording were carried out over the following years, it took a few years for the company Soundstream to start working on the first digital recording software. Not much later, the first Computer Musical Instrument (CMI), a digital synthesizer and sampler, was released. To this day, PCM technology is still the basis of most digital recording.

In the early 80s, the idea of software based Audio Workstations became more of a reality thanks to the fact that personal computers had developed processing power that would be able to handle digital audio editing. 1983 saw the first public demonstration of the MIDI

format. Therefore, data could now be scanned, stored and initiated from a computer-based system. The MIDI format also allowed the connection of ones keyboard to the computer and pass the playing data to the software.

Since then DAWs have only been growing and growing. With their evolution came better user interfaces along with many more features packed into a single program. Todays DAWs allow high quality multi-track digital recording with a track count that has grown rapidly and a user friendly interface. With the appearance of audio plugins that could be added into DAWs and into tracks developers started creating their own custom equalizer, reverbs and such. All of these progressive changes along with the release of Cubase VST made it possible for producers to mix within DAWs without outboard gear entirely. As a result, now DAWs are a powerful tools for music production with almost endless possibilities.

Audio Plugins

With the apparition of DAWs, audio plugins came along. This programs can be imported into the DAWs and applied to different tracks. 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.

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 samples.
  - Virtual synthesizers: This plugins use digital signal processing to create sounds. There are algorithms used in virtual synths that emulate real instruments like the Karplus-Strong algorithm which imitates string instruments.
  - Samplers: These plugins are similar to a synthesizer but instead of generating the sounds it utilizes prerecorded sounds cut into chunks that are reproduced in different orders using either a keyboard, a sequencer or another device to choose which chunk to play. A typical example would be drum pad plugins.
- **Effect plugins:** This plugins have the objective of modifying or completely re-shaping a given signal. Auto-Tune would fall into this category. Here are a few other examples of 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 to alter the frequencies of a signal in different ways depending on their value. Simply put, they provide a way to use multiple filters with one single plugin and make the transition between the effect of each filter seamless.
  - 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 the characteristics of the signal being processed. For example, a spectrum analyzer is 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. Some equalizer plugins include a spectrum analyzer to visualize how the filtering parameters affect the output signal.

It is worth mentioning that a plugin can fall into more than one category and said categories are in no way set in stone. This classification is just a simplified representation of the most common uses for plugins.

There are different possible formats, standard or not. Nonetheless, I 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 AutoTune

In the early 90's 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 in an effort 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 groups opinion about what the music industry could need that an electrical engineer might provide. As a joke, one of his friends wife 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 the way to use the same techniques he was applying at Exxon to make a singers 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 singers voice. Armed with this equations, he designed an integrated system that consisted

of a micro controller loaded with a program capable of executing said mathematics. This system was dubbed Auto-Tune.

The patent [12] 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 pitch correct an entire song. It has become so popular that it can be considered an effect in on its own since many musicians use it with certain parameters to achieve a metallic almost robotic sound that has become an identifying style for many of them.

Figure 1.2: *Original AutoTune interface*

1.1.3 Pitch Detection

When a human speaks or sings, a sound is produced. When the vocal folds vibrate they create 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 pitch or the fundamental frequency.

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 of the vocal folds. A child's fundamental frequency averages about 250 Hz with the length of the vocal folds being about 10.4 mm. With age, the length of the male vocal folds grows more than the female vocal folds creating the noticeable differences between male and female

voices since this 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.

Finding the pitch of a sound is a very difficult feat. One big problem is that it can be very challenging to distinguish between speech and "silence" since true silence does not exist in a recording due to the presence of environmental sounds like the act of breathing. This, along with other difficulties, have resulted in multiple different methods for detecting the pitch being proposed throughout the years and compared in numerous papers[15, 17, 18].

Existing pitch detection methods can be divided into three groups:

- Time-domain methods.
- Frequency-domain methods.
- Hybrid methods combining properties of both time-domain and frequency-domain methods.

Time-domain methods are performed directly with the signals information. For this type of pitch detectors the peaks and valleys of waveforms and other information like zero-crossing points (these are points where the signals samples go from positive values to negative values or vice versa) are crucial to find the pitch. On the other hand frequency-domain methods do spectral analysis (normally using some variation of the Fourier Transform and other Fourier Analysis methods) to find the fundamental frequency (the pitch). Both methods proved to be inconsistent for certain applications. To mitigate said inconsistencies methods combining characteristics of both where designed which led to Hybrid methods.

I personally found the time-domain only methods to be very interesting and studied the following methods for the creation of a pitch correction plugin: the method proposed in the AutoTune patent [12], YIN and MPM. All of this methods will be explained in a simple way in this section, to see a more detailed explanation about how the algorithms work see 3.2.

Since AutoTune was the inspiration for this project, I also investigated the more modern implementation that uses a phase vocoder. However, given that the main focus of the project was to use time-domain methods, the phase vocoder was left to the side and will only be seen in this section to provide some information about one of the most popular methods today.

AutoTune Patent

The AutoTune patent [12] describes a time-domain method that revolves around the auto-correlation function. Essentially, the autcorrelation function compares a section of signal

with itself but shifted by a given amount. The idea, is to find the amount that makes the segment match as closely as possible with its shifted version. The amount found would be a close approximation of the period of the signal which translates to its fundamental frequency or pitch. However, autocorrelation is very slow. To speed up the calculations the inventor proposed a series of modifications and adjustments to create a version of the function that was computed a lot faster. In this modifications lies the key to the success of AutoTune where none had succeeded previously.

The AutoTune 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 that can occur. If the variation found in the pitch tracking is found to be higher than expected, then pitch detection is performed again.

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 the pitch shifting and modify 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 determining the period of a note from a musical scale that is closest to the input period.

For a more in depth explanation including the mathematical equations and the process for reaching those equations see 3.1.2.

YIN

The YIN method[11] proposes another time-domain pitch detection method 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. However, most implementations do use the Fast Fourier Transform to speed up the calculations since it would be too slow otherwise for real time applications.

The YIN method follows a sequence of steps that start with the autcorrelation while the

next steps have the objective of reducing error rates. These are the steps:

1. Autocorrelation: first, the autocorrelation function is used to find an estimation of the pitch. Despite the many efforts to improve its performance this method has proven to not be accurate enough for may applications.

2. Difference function: following some mathematical steps, the end result is a function that calculates three different values of the autcorrelation function. With this step, the error rate falls to 1.95

3. Normalization: because of imperfect periodicity found in real sounds and the existence of the frequencies of harmonics intertwined with the fundamental frequency, the difference function may make mistakes like returning the first formant (F1) because of the strong resonance with the fundamental frequency(F0). The solution proposed is replacing the differnece function with a normalized version (cumulative mean normalized difference function) that prevents this mistakes. The error rate at this step falls to 1.69

4. Threshold: the so called "octave error" (this name is not accurate since it does not need to be an octave) is a common mistake for pitch detection methods that finds a subharmonic instead of the correct fundamental frequency. The solution proposed in the paper is to choose a threshold and choose the smallest value falling bellow that threshold (if none is found the the smallest value overall is chosen). The error rate at this step falls to 0.78

After all these steps are done the best estimate is given as the fundamental frequency of the signal. For a more in depth explanation including the mathematical equations and the process for reaching those equations see 3.1.3.

MPM

The MPM method [13] shares many of the same characteristics of YIN as a base. This method also starts with autcorrelation although it uses what is called the type two autocorrelation instead of type one like YIN (both are almost identical). Then, the difference function is also used. However, since it derives from the type two autocorrelation there are some differences between the one used in MPM and the one used in YIN. Instead of calculating the cumulative mean normalized function the MPM method proposes the normalized square difference function providing a different method for normalizing the results and reducing the error rate. The main difference though, starts at this step.

The MPM method proposes one more step following the normalization: peak picking. Peak picking consists on taking the potential periods found and choosing the highest value

found between two positively sloped zero-crossing points (the range of samples between two points where samples have gone from negative values to positive values). If a positively sloped zero-crossing is found without the corresponding negatively sloped one the highest value found so far is chosen.

For a more in depth explanation including the mathematical equations and the process for reaching those equations see 3.1.4.

Phase Vocoder

The phaser vocoder (PV) analyses a digital sound using frequecy-domain methdos and the 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 blockssf samples. Those blocks are multiplied bay 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. I have found that the Hann windowing is the most used 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.

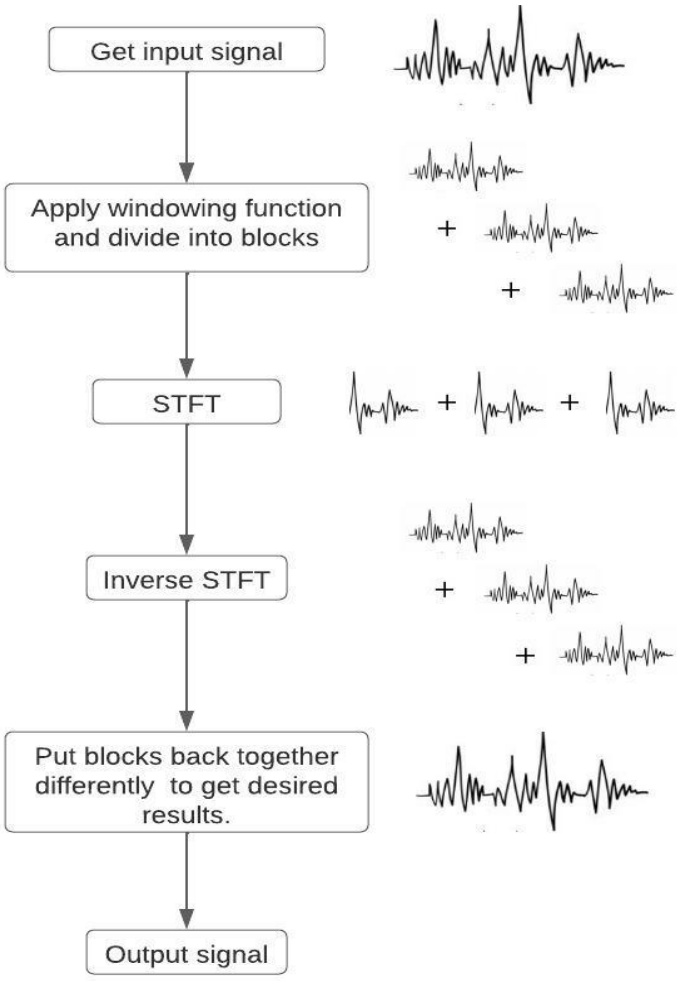

Figure 1.3: *Phase Vocoder steps diagram*

1.2 Motivation and Objectives

Finding the fundamental frequency of a periodic signal is a very complex process. It becomes even more complicated when the signal is not perfectly periodic. Then, when the signal is, for example, the recording of a human voice, finding the fundamental frequency becomes hindered by the existence of many frequencies intertwined with each other called harmonics that add the requirement of distinguishing which is the fundamental frequency among the different frequencies. The investigation of accurate pitch detection methods is an active field of research. Furthermore, while some very accurate algorithms have been created they may not be applicable in real time processing, making them not useful for real time audio

processing applications like plugins used in DAWs. On the other hand, algorithms that do fit the real time processing criteria may not be accurate enough requiring other methods to palliate the mistakes.

I found the idea of finding the fundamental frequency of a periodic signal in real time to be very interesting and decided to pick this field of research and try to create a plugin that could detect the pitch of a recording and correct it just like AutoTune does. By implementing it as a VST plugin it could also be used in any DAW with support for VST format making it very useful for many music production projects. However, since there are a plethora of methods to choose from as seen in previous sections, I decided to focus on the time-domain methods described previously (see 1.1.3) given that most famous pitch corrector (AutoTune) used only time-domain methods in its original implementation (see 1.1.3).

Chapter 2

User Manual

2.1 Plugin

To use the plugin there are three main steps to follow overall: include the plugin in your DAW, add the plugin to a track and configure the plugin to obtain different results. In this manual the DAW chosen is REAPER [5] 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 you have to download the plugin from [10]. 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 you can place the plugin inside the C:\Program Files\Common Files\VST3 directory since this is oune 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 (see 2.1). Then you have to 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.

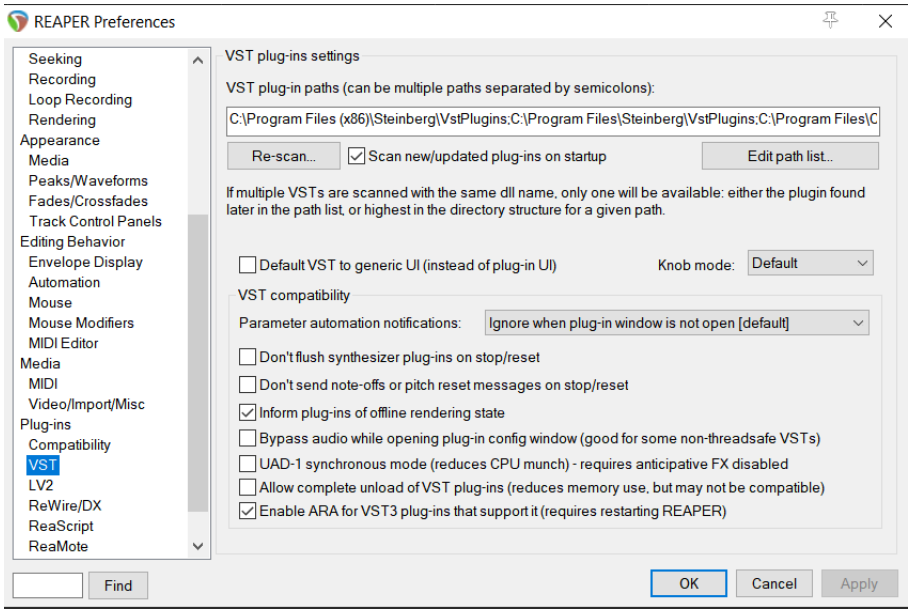

Figure 2.1: *Preferences dialog, VST section*

2.1.2 Adding the plugin to a track

Once the plugin has been added to the DAW you can use it 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.

When opening a new REAPER project the main window has the appearance shown in 2.2.

The main window is divided in two main sections. While many things can be done in both sections they do have two different purposes. The bottom section is the mixer where all the tracks are represented with their own slider for changing the volume of the single track. A master slider is provided in the left that affects all of the tracks simultaneously. The top section also shows the tracks but allows the visualization of the content of each track, either MIDI messages or audio waveforms, on the right. This part of the interface is used to apply effects like cross-fading and panning by manipulating the shapes manually. This allows for a more visual representation of what we are working with and for a more intuitive way of manipulating the behaviour of certain effects.

There are different ways of adding a track using a prerecorded audio (for example, a wavfile). The most straightforward way is by simply dragging your file into the REAPER interface. This will automatically add a new track containing the given audio at the end of the track list. See 2.3 to see the expected result of dragging a file into the empty project.

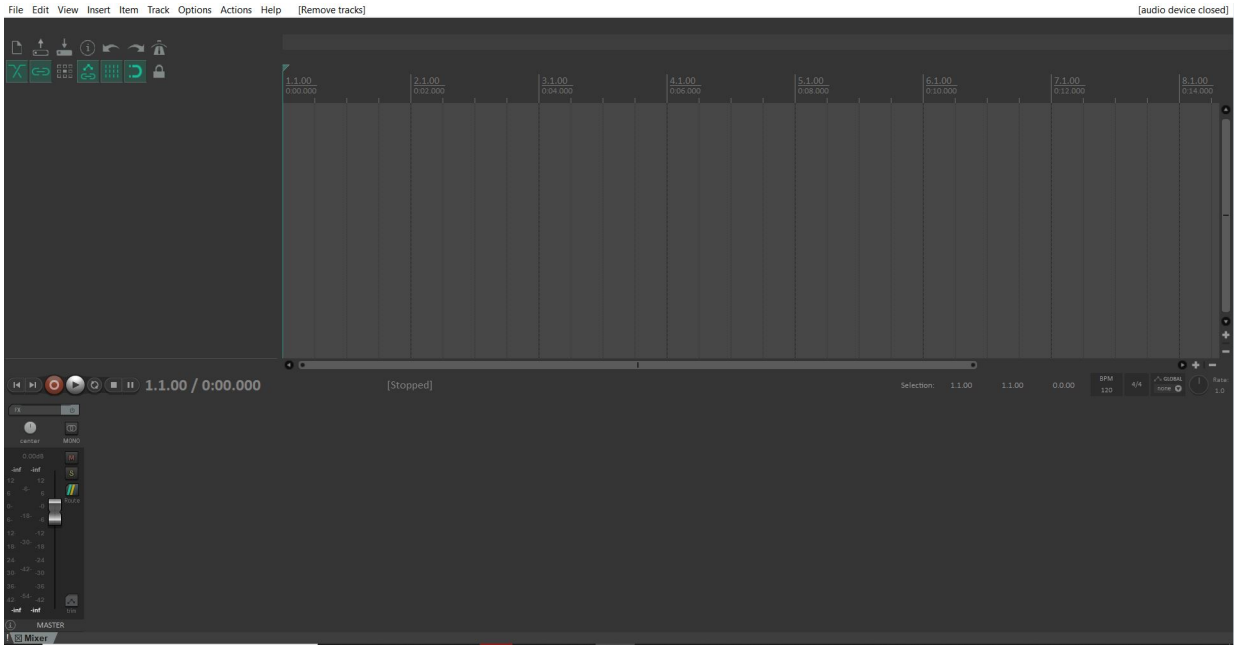

Figure 2.2: *A REPAER empty project*

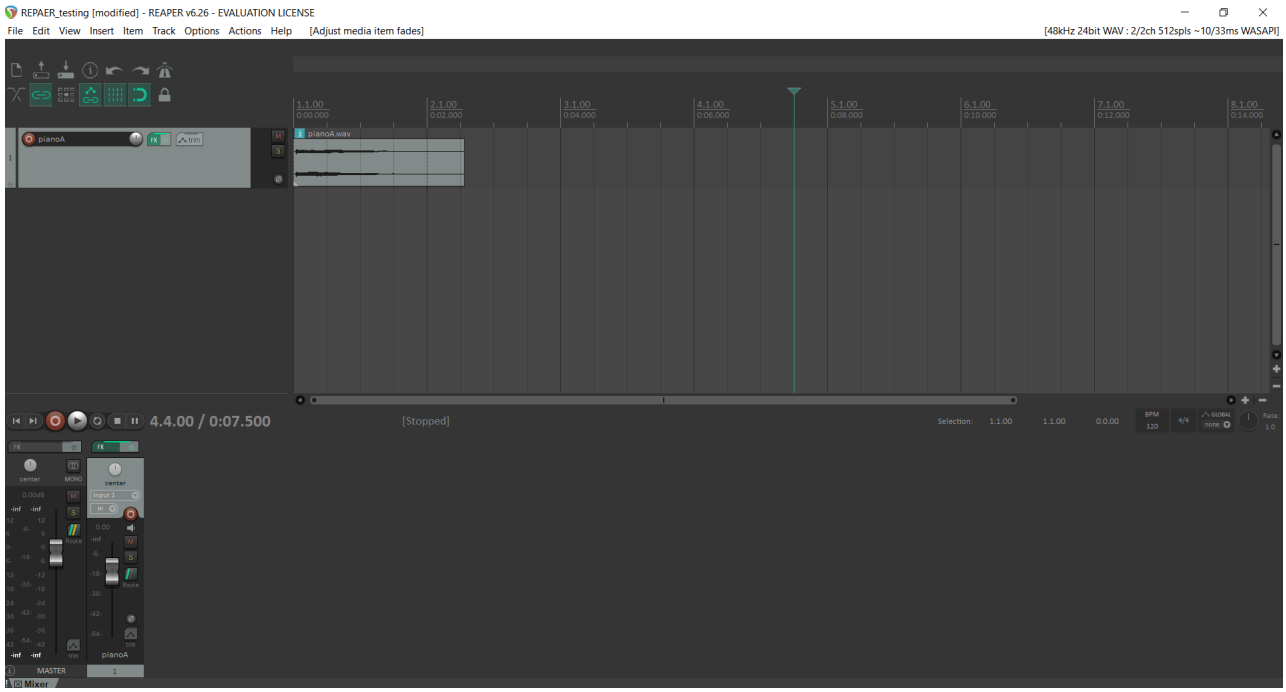

Figure 2.3: *A REPAER project with a single track*

After creating the track with the audio we want to correct then we can add the plugin to the track. To do that press the *FX* button in the corresponding track to manage the tracks plugins. This button can be found in the track in both sections of the interface. If you can't find it see 2.4 for help. 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 and select the plugin VST3: PitchCorrectionPlugin. Then, click the *Add* button to add the plugin to the track (see 2.5). As a result the pitch correction plugin should now appear in the other dialog (see 2.6).

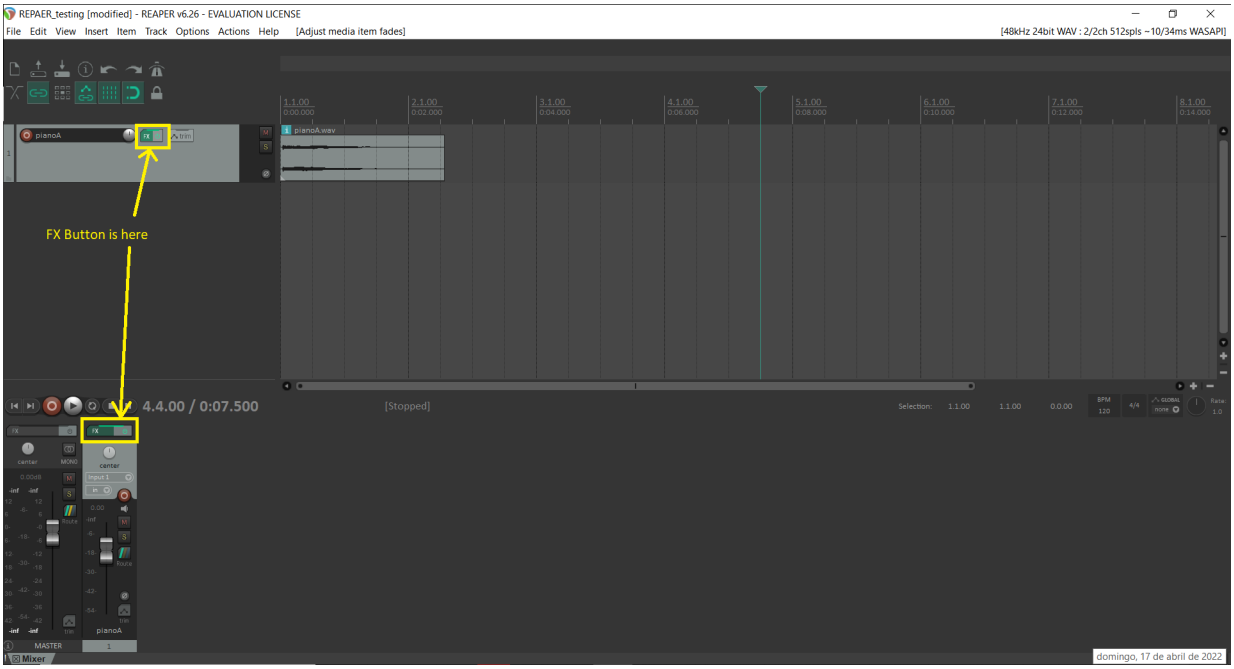

Figure 2.4: *A REPAER project highlighting FX button*

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

2.1.3 Configure the plugin

Now, the only thing left to do is adjust the parameters of the pitch correction to your liking. This is done by modifying the sliders shown in the interface (see 2.6). To access the plugins parameters open the tracks plugin manager again by clicking on the *FX* button of the track (see previous section to do this step if you haven't already).

The parameters offered are:

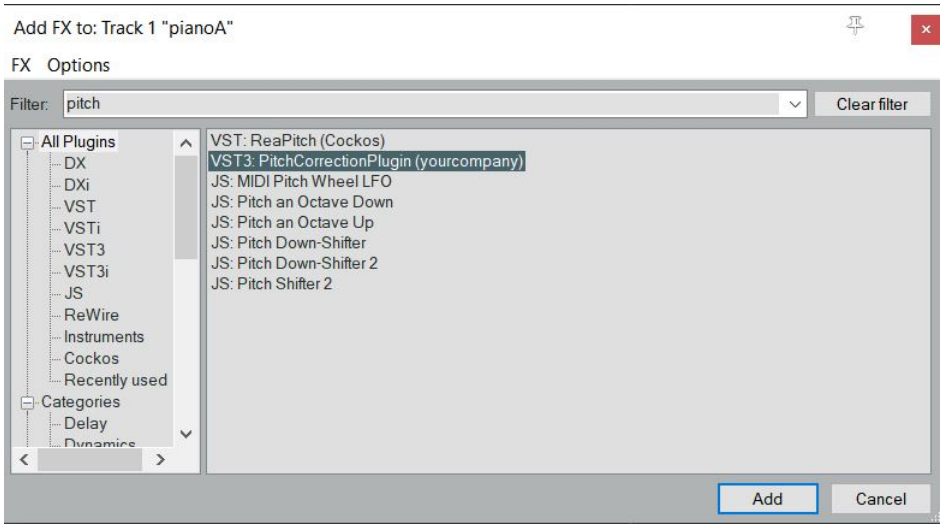

Figure 2.5: *Tracks add plugin dialog*

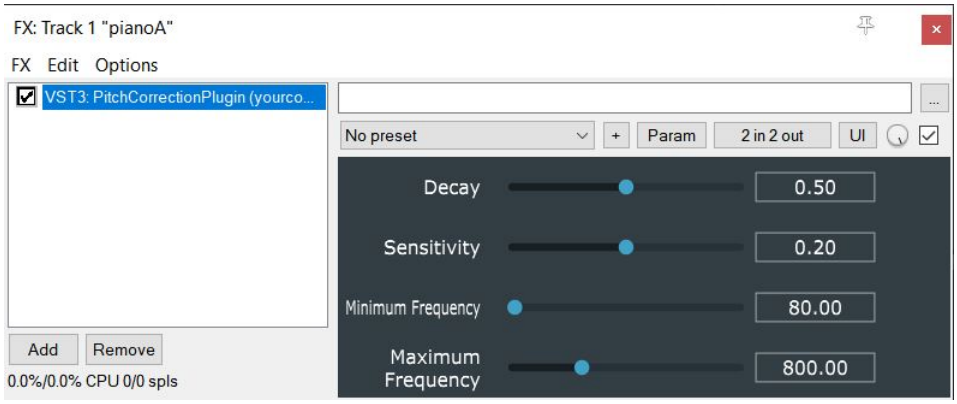

Figure 2.6: *Tracks plugin manager*

- Decay: this represents how smooth the transition between samples is. 0 will create the output samples without doing any cross-fading. 1 is the maximum value creating a progressive smoothing of the transitions.
- Sensitivity: this represents how sensitive the pitch detection step is. Depending on the complexity of the audio better results can be obtained with different levels of sensitivity.
- Minimum frequency: this is the minimum frequency 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.

- Maximum frequency: this is the maximum frequency that we want the plugin to detect. As with the minimum frequency, this value should be adjusted depending on the audio we are working with to get better results.

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 [4].

2.2.1 Load a File

The first step is loading a file. To do that click the *Load File* button (see 2.7). A file explorer will then open. Then, all you have to do is look for your file in your machine and select it. It is important to mention that the application only supports the wavformat and will not find any extension other than wav.

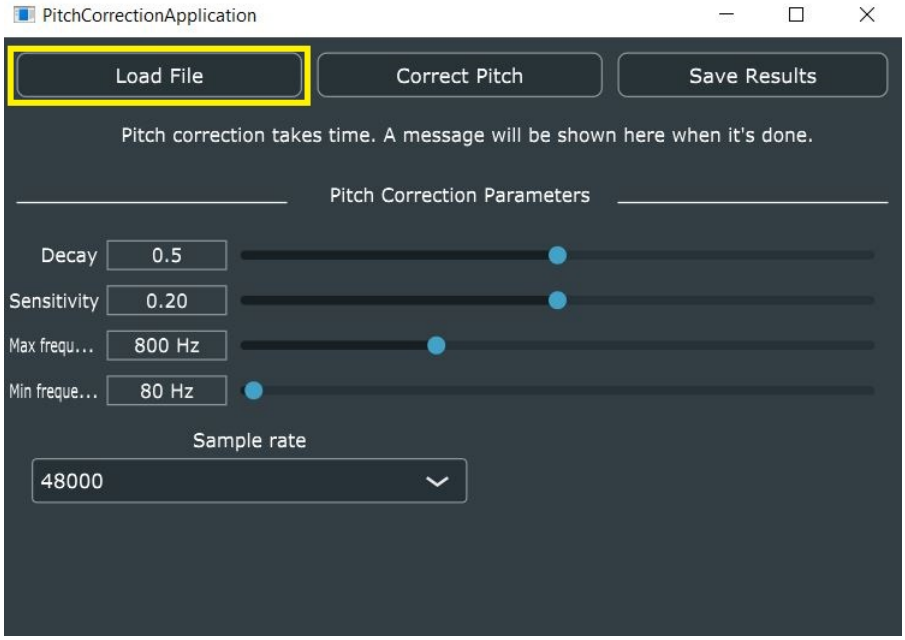

Figure 2.7: *Applications load button*

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 you have to click the button *Correct Pitch* (see 2.8). But first, you may want to adjust the parameters of the pitch correction to your current needs. The parameters can be found bellow the buttons in the form of 4 sliders and a combo box. Each of the parameters have the following purpose:

- Decay: this represents how smooth the transition between samples is. 0 will create the output samples without doing any cross-fading. 1 is the maximum value creating a progressive smoothing of the transitions.
- Sensitivity: this represents how sensitive the pitch detection step is. Depending on the complexity of the audio better results can be obtained with different levels of sensitivity.
- Minimum frequency: this is the minimum frequency that we want the application 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.
- Maximum frequency: this is the maximum frequency that we want the application to detect. As with the minimum frequency, this value should be adjusted depending on the audio we are working with to get better results.
- Sample rate: this allows you to tell the application what is the sample rate of the audio you inputed since the application can not detect the sampling rate by itself. The possible choices are 48000 and 44100.

2.2.3 Save the results

Finally, you can save the results in a wavfile in your machine to later listen to the results. To do that click on the button *Save Results* (see 2.9). Like with the loading of a file, this button opens a file explorer that allows you to browse through your 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.

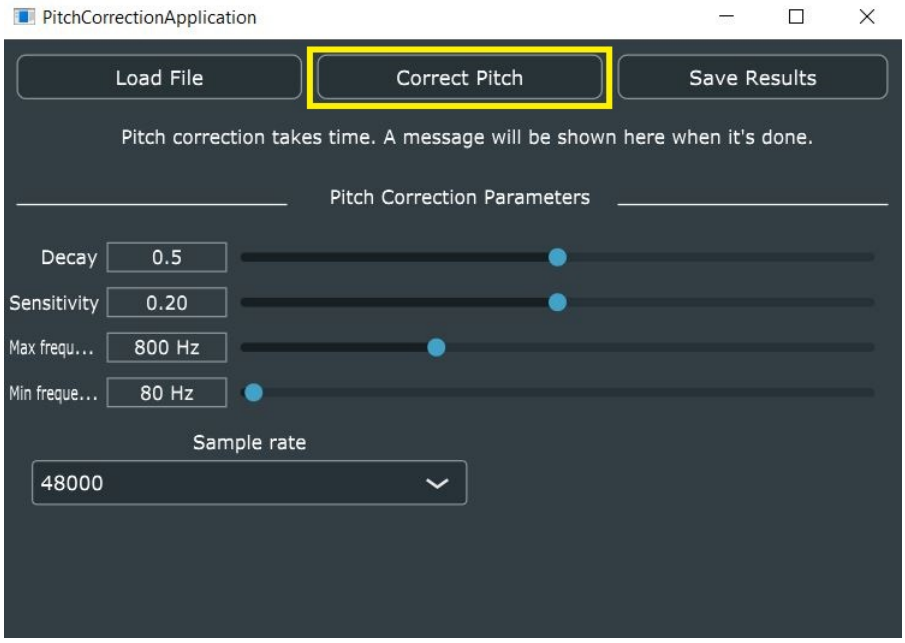

Figure 2.8: *Applications correct pitch button*

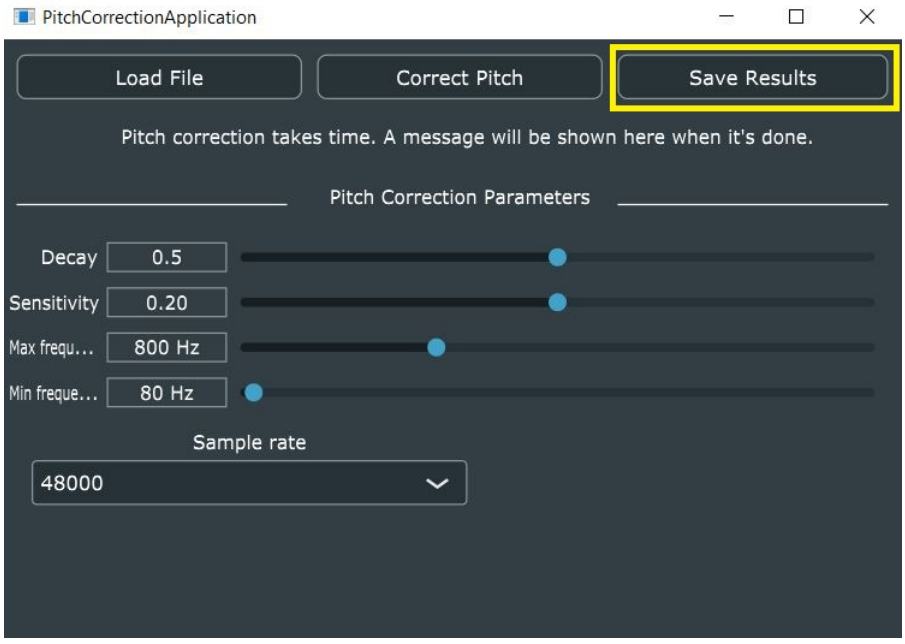

Figure 2.9: *Applications save button*

Chapter 3

Algorithms

This chapter provides a more in depth explanation of the different algorithms both for pitch detection and pitch shifting investigated and implemented (weather fully or partially) at some point including the mathematical equations needed to fully understand how the algorithms work.

3.1 Pitch Detection

3.1.1 Autocorrelation

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.

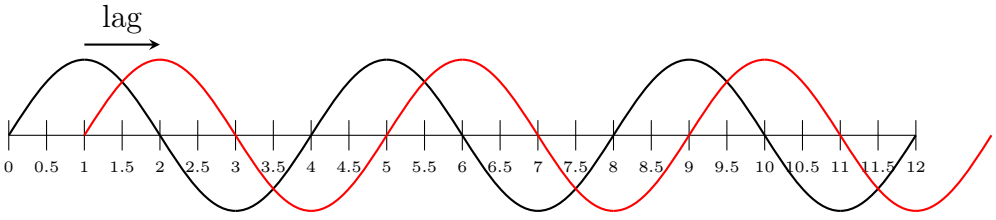

Figure 3.1: *Shifted waveform not matching the original*

There are many uses and implementations of autocorrelation. I have used the variant proposed by Auto-Tune inventor Hildebrand to test its performance. Therefore, an expla-

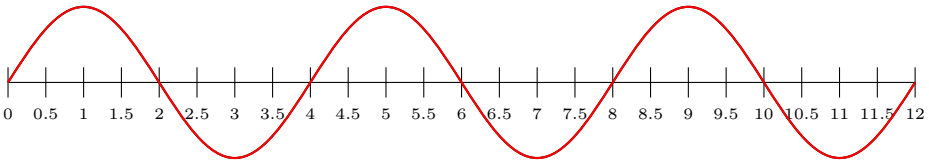

Figure 3.2: *Shifted waveform perfectly matching orginal*

nation of how auto-correlation is used in the original Auto-Tune system will be given in section **??**.

3.1.2 AutoTune Patent

The patent [12] uses an un-normalized version of the auto-correlation function Φ

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

We assume x to be periodic with a fundamental period L. The auto-correlation function is also periodic.

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 is speeding up this process to apply it in real time. To do that, the following functions were derivedt: $E_i(L)$ 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.

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

$H_i(L)$ 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.

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 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.

3.1.3 YIN

While the AutoTune method seemed to work in early implementations with moderate success, the patent for AutoTune [12] made it difficult to fully implement all the suggested

methods to minimize mistakes. As a result, I looked for other more evolved time domain methods that could offer more accurate results.

The original paper for YIN was published in 2002 [11]. In the years after, different variants of the algorithm have been derived. For example, another approach to YIN is the probabilistic YIN algorithm which we will discuss briefly. Other implementations combine the original method with frequency domain techniques like the Fast Fourier Transform (FFT), convolutions, etc.

Here, an overview, taken from the original paper [11], of how the YIN algorithm works is given. 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) 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.

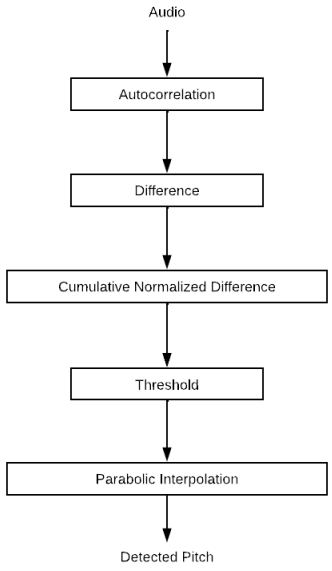

Figure 3.3: *YIN flow diagram*

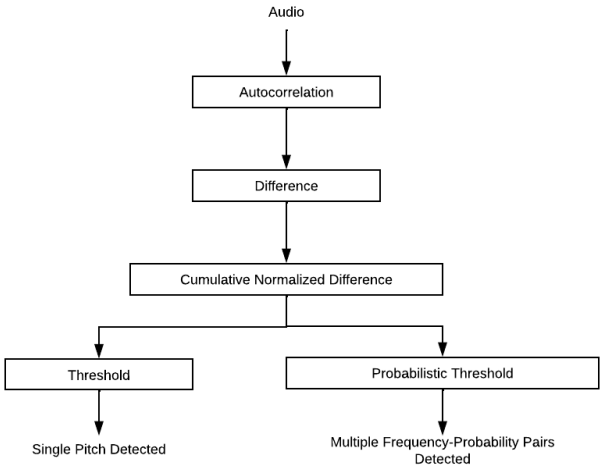

Figure 3.4: *Probabilistic YIN flow diagram*

3.1.4 MPM

Another widely used algorithm that was explored is the one proposed by Philip McLeod, Geoff Wyvill in the paper *A Smarter Way to Find Pitch*. The first few steps are very similar to the YIN algorithm.

There are two main ways used for defining the autocorrelation function (type I and type II). While YIN uses type I most often, 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.

3.2 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 altering 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 needs to be raised. As an example, figure 3.5 can be taken as a representation of a given sound. To make the pitch higher by an octave the frequency has to be doubled and therefore the duration becomes half the original. Figure 3.6 would be the result of doing said modifications.

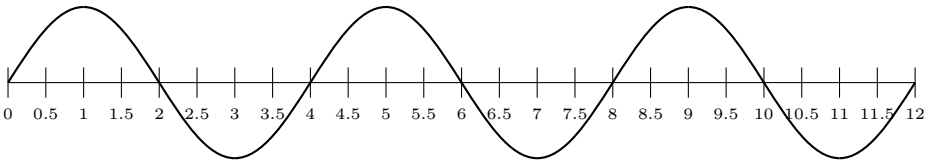

Figure 3.5: *Original waveform*

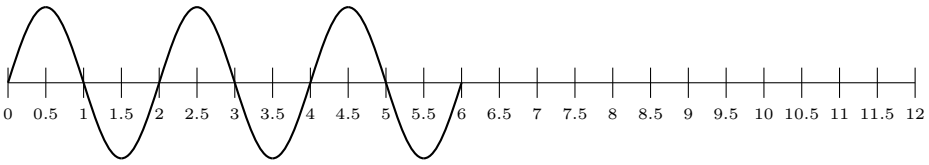

Figure 3.6: *Waveform with modified pitch and duration*

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 resampling techniques to achieve this effect without producing any unwanted artifacts. Retaking the previous example, if the original pitch of the audio where to be shifted to half its original

frequency without altering the speed the ideal result would look like the waveform depicted in figure 3.7.

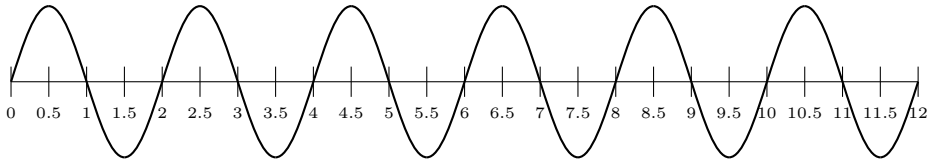

Figure 3.7: *Waveform with modified pitch without altering duration*

By pitch shifting a song by a fixed amount transposing effects are obtained. For example, a piece in C major could be transposed to E major by shifting the piece 3 tones up (see figures 3.8 and 3.9). However, if the piece was shifted 4 and a half tones up the new music scale would be G major (see figures 3.8 and 3.10).

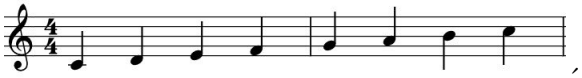

Figure 3.8: *C major*

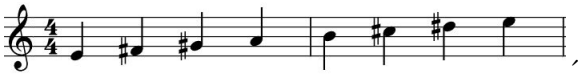

Figure 3.9: *C major shifted by 3 tones becomes E major*

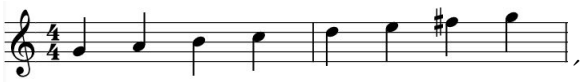

Figure 3.10: *C major shifted by 4 tones and a half becomes G major*

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. The reason why is the need to shift the pitch by a variable amount depending on the moment. 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 without producing artifacts or inconsistencies to the overall sound. This is a very complicated process that is an active field of research nowadays.

In the following subsection the method used for the implementation is explained on a theoretical level. Other methods where explored but this method was chosen for implementation since it is the one suggested in the AutoTune patent [12] and the final implementation uses a simplified version of the method proposed in it.

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 and pitches 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 modification of the current ratio 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 found. 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 waveform shown in figure 3.11 and we want to shift it's pitch to double the original resulting in the waveform shown in figure 3.12. Since these waveforms are represented as data in the computer the original waveform can be represented like shown in figure 3.13. For example, to shifth 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 in figure 3.14.

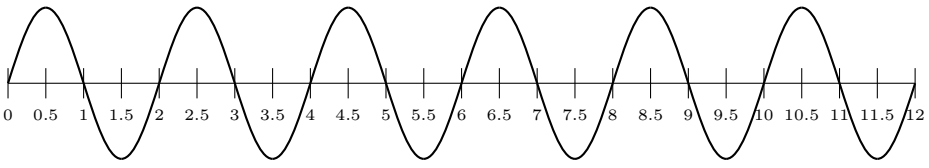

Figure 3.11: *Original waveform*

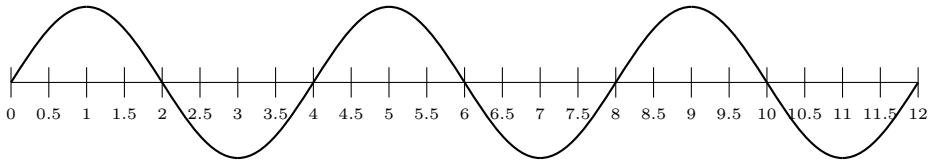

Figure 3.12: *Resulting waveform*

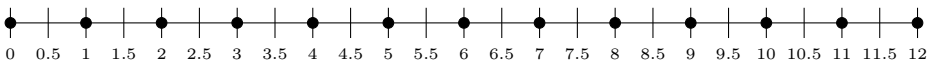

Figure 3.13: *Original data*

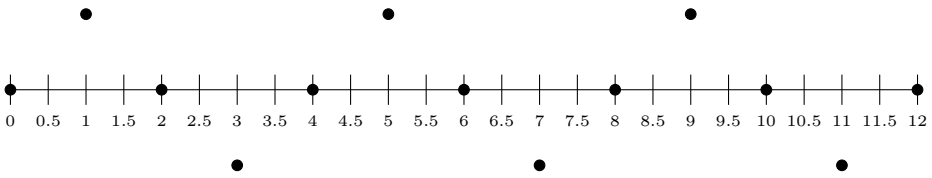

Figure 3.14: *Resulting data*

3.2.2 Pitch-synchronous Overlap Add (PSOLA)

In signal processing, the overlap add method [14] 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 good. This method was studied to be used with the YIN and MPM pitch detection methods. However, it was not useful for the AutoTune method and since yin and mpm ended up being to slow for the objectives this method along with them ended up being discarded in favor of the interpolation method that works well with the AutoTune method.

Chapter 4

Prototypes

4.1 Technologies

4.1.1 Jupyter Notebook

As stated in the website[3], 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.

Some trouble was found while trying to implement the algorithms using C++ and JUCE. Therefore, I ended up creating prototypes in Python to do tests with the pitch detection. To do that I used Jupyter Notebooks since it is a free and easy to use tool that made plotting results, adjusting parameters and testing on different audio recording easier.

4.1.2 SciPy

SciPy [6] 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

The first algorithm implemented in python to test its performance was the method proposed in the AutoTune patent (to see the theoretical explanation on the algorithm see 3.1.2). The prototype follows this steps:

1. Decimation.
2. Computing equations 3.5 and 3.6 for the decimated samples.
3. Finding the lag that satisfies equation 3.8.
4. Approximate the lag for the non downsampled data.
5. Find the correct pitch from the approximated one found.
6. Calculate the ratio for the resampling.
7. Compute the corrected output sample.

To do the decimation, the decimate function from the SciPy library (see 4.1.2) was used. The downsampling factor was set to 8 as described in the patent. The factor 8 decimation passes the samples through an antialiasing filter and then return 1 out of every 8 samples. The end result is an array containing the filtered data with 8 times less samples.

For the autocorrelation equations an array with L positions where L is the maximum possible lag is created. The maximum possible lag is computed using the minimum possible frequency given by the user and the sampling rate. The minimum and maximum lags possible are divided by 8 since the samples used for the autocorrelation functions are the one obtained from the decimation making the possible periods 8 times smaller as well. 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). 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 lag.

```
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 4.1: *Autocorrelation code*

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 indexes contained between index - 8 and index + 8. This is because using decimation makes us loose some information. Since the donwsampling factor is 8, the indexes between index - 8 and index + 8 are not evaluated. Therefore, the function calculates the autocorrelation functions for these indexes and uses interpolation to approximate other values in between to get a better estimation. To do interpolation the implementation from the library SciPy is used (see chapter 4.1.2).

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, I tested the performance of the pitch detection using some simple audio samples and the results were good enough. However, the precision was not as good as expected. I tried using the full implementation proposed in the patent including pitch tracking and other methods to improve accuracy. Unfortunately, the explanation on the patent regarding this methods wasn't clear enough and some of the implemented elements took way too much time and affected the performance greatly. Therefore, I chose to leave the AutoTune version as is for the final implementation.

Since the results where good enough for simple pitch detection I also implemented the

pitch shifting method that uses interpolation to find the output samples. This method was found in the AutoTune patent [12]. 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. A pointer to the input signal is kept throughout the execution, this pointer is updated with the ratio and then the closest values to the pointer are used to interpolate the output sample.

Also, it is important to take into account that the pointer used for interpolating the output sample could advance too fast or too slow. If the ratio is greater than one then it may be too fast. In that case, if the output pointer has become greater than the input index that is reading the samples then it is going too fast so the output pointer is taken back a whole period cycle. On the other hand, we can check if it is going too slow by seeing if the output pointer plus the period is lower than the input which means that we can add a whole cycle period to the output pointer while staying behind the input pointer.

```
if Lmin != 0:
        resample_raw_rate = Lmin / desired_Lmin
        resample_rate2 = resample_raw_rate

output_sample = 0

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

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: *Output sample computation*

4.3 YIN

I also implemented a python prototype for the YIN algorithm. I tested the results of the algorithm at different stages to see if the accuracy was indeed better compared with each new added stage. To do that I compared the results for given recordings of a piano note at each stage. The end result was that the full algorithm always returned a very close value to the frequency of the note being played while the other stages where either less accurate or simply failed. While this algorithm showed that its accuracy was very good, I found the time required to find the pitch with the full algorithm way too long so in the end I ended up using the AutoTune patent algorithm since every method found for making it faster involved the use of the Fast Fourier Transform and the objective was to see what could be achieved without using it.

Next I shall explain how the full algorithm was implemented giving an overview of how the different equations where finally implemented in python and how they where used to find the pitch of the data being analyzed. To see the mathematical equations involved and a more theoretical explanation see 3.1.3. A different function was created for each step described in the paper[11]. First, I implemented the autocorrelation equation (see 3.9) that takes the data of the signal and computes the autcorrelation function with the window size for the lag given at the time step provided in the time argument.

```
def autocorrelation(data, window, time, lag):
    return np.sum(data[time : time+window] * data[lag+time:lag+time+window])
```

Listing 4.3: *YIN autocorrelation function*

Then, the difference function can be calculated using the autocorrelation function like demonstrated in equation 3.13.

```
def difference(data, window, time, lag):
    return autocorrelation(data, window, time, 0)
        + autocorrelation(data, window, time+lag, 0)
        - (2*(autocorrelation(data, window, time, lag)))
```

Listing 4.4: *YIN difference function*

Afterwards, I implemented the cumulative mean normalization equation (see equation 3.14). If the lag is zero the the function returns 1. Otherwise the corresponding calculations are made according to the equation described in the paper.

```
def cumulative(function, window, time, lag):
    if (lag == 0):
        return 1
    return difference(function, window, time, lag) /
        np.sum([difference(function, window, time, j + 1)
        for j in range(lag)])*lag
```

Listing 4.5: *YIN cumulative mean function*

Then, a function to calculate the pitch was implemented. This function gets the values returned by the cumulative mean function for the specified range of possible lags. Then, the function looks for the smallest value that is smaller than the chosen threshold. If a value satisfying this requirement is found then it is used to compute the pitch returned by the function. On the other hand, if no value falling bellow the threshold is found then the minimum value is used instead to obtain the pitch.

```
def detect_pitch(function, window, time, sample_rate,
min, max, threshold = 0.1):

    values = [cumulative(data, window, time, i) for i in range(min, max)]
    sample = None
    for i, val in enumerate(values):
        if val < threshold:
            sample = i + min
            break;

    #If no value found satisfying threshold return minimum
    if sample is None:
        sample = np.argmin(values) + min

    return sample_rate/sample
```

Listing 4.6: *YIN pitch detection with threshold*

Since this method was too slow no further implementation was done to add pitch shifting. An implementation of MPM was started as well but given the similarities between them the computational cost would not have differed much and therfore that algorithm was discarded as well for the final implementation.

Chapter 5

Final Implementation

Once a pitch detection and pitch shifting method where selected I implemented the pitch correction program as a VST plugin and as an application. The final algorithms chosen where the AutoTune patents autocorrelation method for pitch detection with the interpolation method for the pitch shifting (for theoretical details see chapter 3 and for the prototype implementation in python see chapter 4).

5.1 Technologies

5.1.1 JUCE

JUCE[2] is a partially open-source C++ framework used for the development of desktop and mobile 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 needed for basic projects is relatively easy.

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 [2] featuring almost all of its utilities to help new users on the journey of developing with JUCE.

Projucer

The Projucer is JUCE's project management tool 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.

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 functions 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. The code in 5.1 shows the default prepareToPlay given when a project is created.

```
void ExampleAudioProcessor::prepareToPlay (double sampleRate,
int samplesPerBlock)
{
    // Use this method as the place to do any pre-playback
    // initialisation that you need..
}
```

Listing 5.1: *Default prepareToPlay function*

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. It also gives the corresponding MIDI messages provided contained within the MIDIBuffer. The code in 5.2 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 5.2: *Default processBlock function*

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 where mostly used for the JUCE Audio Application development.

Audio Applications

At the final stages of development the final plugin showed some lag when playing the sound in a DAW. As a result a JUCE Audio Application was created to allow the user the possibility of opening an app where they could load a file, correct its pitch and then save the results in another file. This way, I could provide a version of the audio without the added lag caused by the real time processing.

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 Component class which is the most important base class for all JUCE graphical interfaces. In JUCE, practically all visible elements (buttons, sliders, text...) are components deriving from the Component class. The JUCE Audio Application creates a main component owned bay 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 discussed in the previous section, the MainComponent class has two functions that are essential to the interface creation:

- paint(): this function determines how your component should be drawn on the screen. The default code that is created for the paint function shown in 5.3 creates a solid background
- resized(): this function determines what should happen to the component when it is resized.

```
void MainComponent::paint (juce::Graphics& g)
{
    // (Our component is opaque, so we must completely fill the
    //background with a solid colour)
    g.fillAll (getLookAndFeel().findColour
    (juce::ResizableWindow::backgroundColourId));

    // You can add your drawing code here!
}
```

Listing 5.3: *Default paint function*

```
void MainComponent::resized()
{
    // This is called when the MainContentComponent is resized.
    // If you add any child components, this is where you should
    // update their positions.
}
```

Listing 5.4: *Default resized function*

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. I 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 [1] 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 where not well suited for the interpolation of single data points and didn't include the spline interpolation algorithms that I wanted to use for the program so I ended up using this alternative library to do the interpolation of the output samples.

5.1.3 REAPER

REAPER [5] 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 I was already familiar with it. Also, my previous attempts at creating a woring plugin where tested using this DAW which meant that I already knew how to add and use my own plugins in REAPER. 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. Since this behaviour was going pretty much the same for both the plugin and the application I implemented a separate C++ class to manage the different steps while the plugin and application programs simply had to call the different methods included in the class to do the pitch correction of the audio being managed by the program.

The class recieves 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.

Before explaining the details about the implementation of the plugin itself I 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 it in the pitch correction samples, storing the sample into the decimated samples every 8 samples, find a new pitch every 8 samples, calculate the new resampling rate if a new pitch is found, update the pointer for the interpolating and also interpolate the output sample. The reading and writing of samples is managed by the plugin and application.

Therefore, each of those steps where implemented in their own separate public method:

- add_sample(float sample): this function takes a sample, filters it and stores it inside a vector containing the samples being processed.

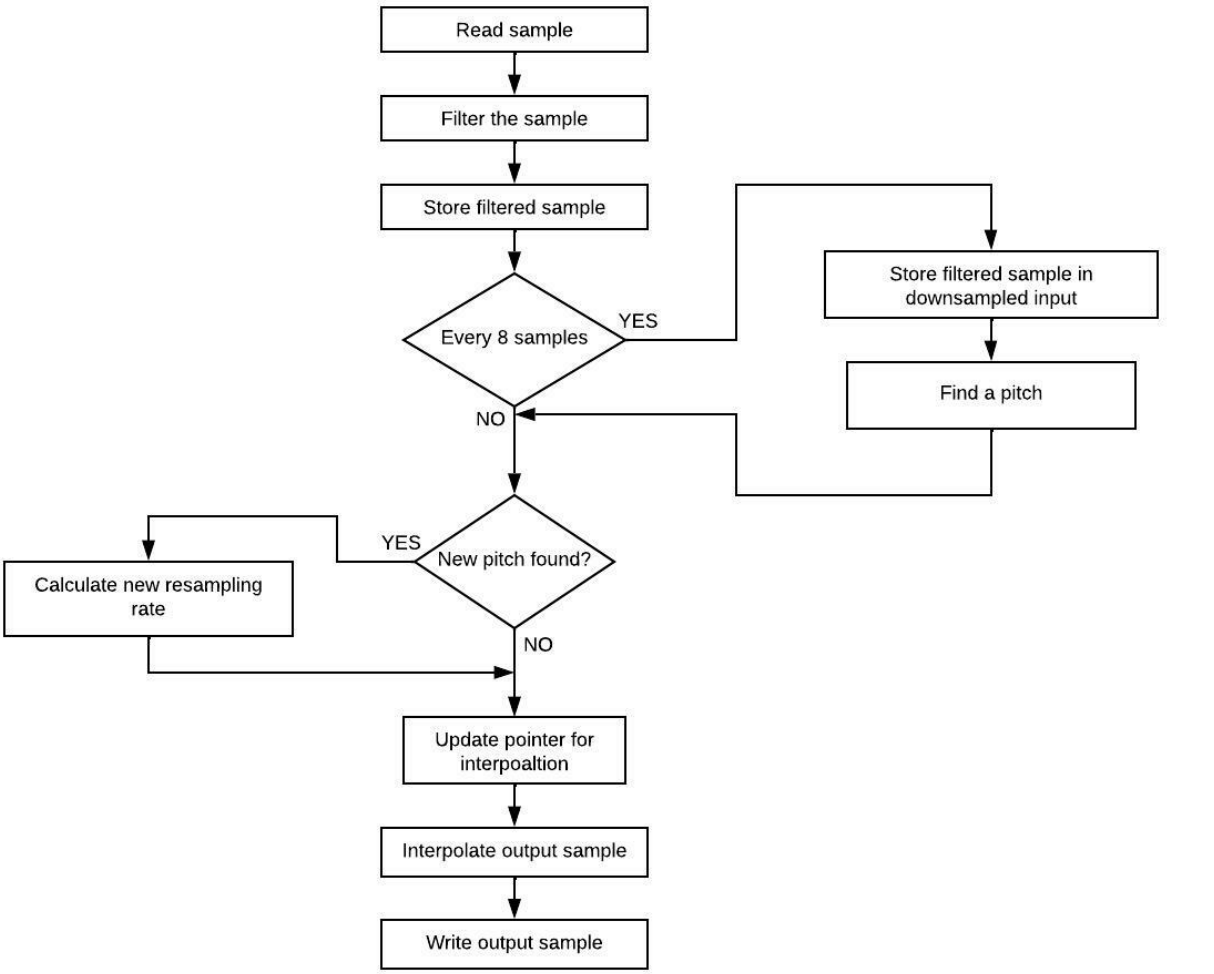

Figure 5.1: *AutoTune simplified flow diagram*

- 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(int L) to find a more accurate lag. Then it returns the fundamental frequency found.
- calculate_resample_rate(float real_freq): this function finds the correct frequency using a private function called get_desired_frequency(double real_freq) and then updates the resampling rate using the detected fundamental frequency and the correct one that was estimated.
- update_output_addr(float Lmin): this function updates the pointer used for interpo-

lation using the period detected.

All of these methods along with the private methods used by them where 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 I wanted to use. Other than that, the implmentation 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, I 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 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.

```
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());

    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);

    for (int i = 0; i < buffer.getNumSamples(); i++) {

        autotune.add_sample(left_data[i]);
        if (i % 8 == 0) {
            autotune.add_decimated_sample();
            freq = autotune.get_fundamental_frequency();
            Lmin = getSampleRate() / freq;
        }

        if (freq != 0) {
            autotune.calculate_resample_rate(freq);
        }

        double output_sample = 0;

        autotune.update_output_addr(Lmin);

        output_sample = autotune.get_output_sample(Lmin);

        left_data[i] = output_sample;
        right_data[i] = output_sample;
    }
}
```

Listing 5.5: *JUCE plugin processBlock*

Finally, I 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 [16] 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 consturctor 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 simple GUI with the parameters shown as slider the createEditor method was modified to instead 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.

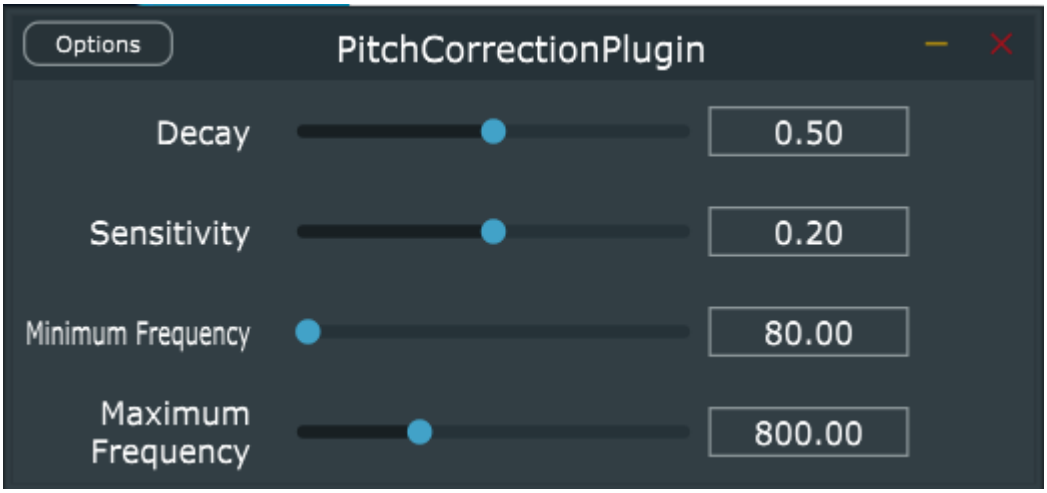

Figure 5.2: *JUCE Plugin GUI*

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.

5.3 Application

Since the plugin performed with lag in the DAW tests. I 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 without lag.

First, I 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 I used the The ComboBox class[7] tutorial, Build an audio player[8] tutorial and The Point, Line, and Rectangle classes[9] tutorial as references. All of these tutorials where provided by JUCE and can be found in their website [2] among many other helpful tutorials. The Build an audio player tutorial also gave me 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 where created using JUCEs TextButton class. First they where declared inside the private section in the MainComponents 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 where 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 5.3). 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. Onche 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. I 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 the calling its method createWriterFor with the file as an argument. Then, if the writer was created successfully, the writers writeFromAudioSampleBuffer can be used to save into the chosen file the contents found in the ouptut audio buffer.

```
void MainComponent::loadFileButtonClicked()
{
    juce::FileChooser chooser("Open a Wav file",
    juce::File::getSpecialLocation(juce::File::userHomeDirectory),"*.wav");

    pitchCorrectionFeedback.setText("Pitch correction takes time.
    A message will be shown here when it's done.", juce::dontSendNotification);

    if (chooser.browseForFileToOpen())
    {
        juce::File audioFile;
        audioFile = chooser.getResult();

        juce::AudioFormatReader* reader =
        formatManager.createReaderFor(audioFile);

        if (reader != nullptr)
        {
            input_audio_buffer.setSize(reader->numChannels,
            reader->lengthInSamples);
            output_audio_buffer.setSize(reader->numChannels,
            reader->lengthInSamples);

            reader->read(&input_audio_buffer, 0,
            reader->lengthInSamples, 0, true, true);
            reader->read(&output_audio_buffer, 0,
            reader->lengthInSamples, 0, true, true);

            startTimer(40);
        }

    }
}
```

Listing 5.6: *loadFileButtonClicked()*

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 5.2). However, some adjustments where made to fit the processing of the whole audio. First, the AutoTune

class described in the previous section (see 5.2) 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 constructur and initialized with a range of values and a default value that falls at a number that I 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 where placed inside the GUI using the resized method (see 5.3) 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 bos is added inside the constructor and initiallized to contain the mentioned values and have 48000 as the default value. Fterwards 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 where configured the final look of the GUI can be seen in figure 5.3.

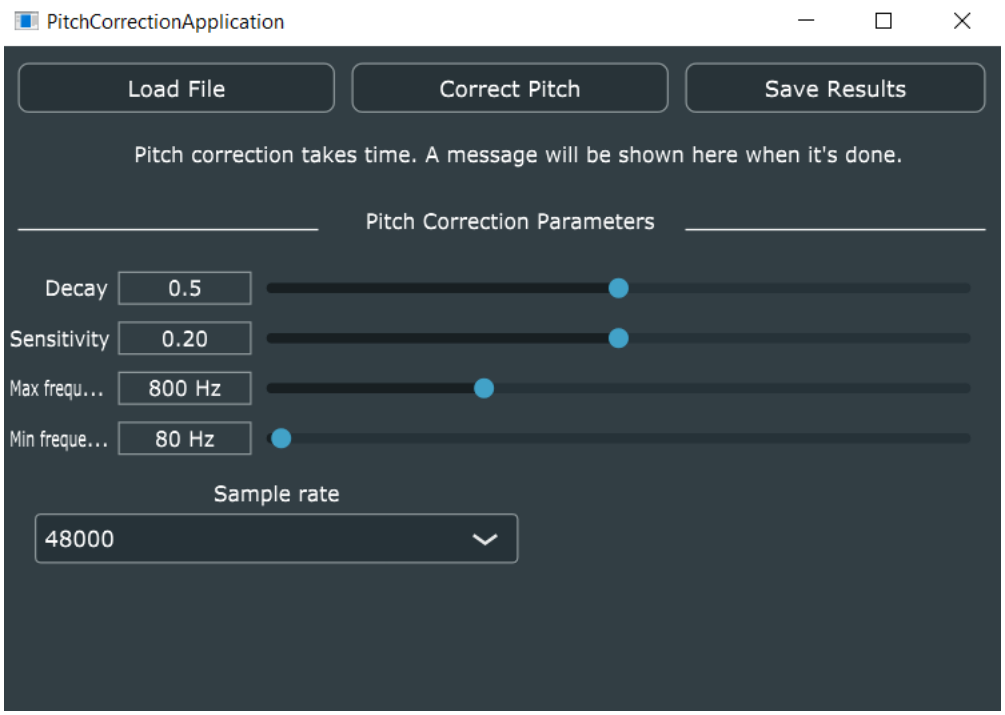

Figure 5.3: *JUCE Application GUI*

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Even though the main objective of creating a working plugin that does pitch correction in real time has been achieved, it has been done without the desired degree of accuracy and performance. During the prototyping phase it became apparent that the methods chosen where either two slow or not accurate enough. While the AutoTune patent seemed promising, the more evolved mechanisms described and the details of certain steps where not fully overviewed resulting in a final product that, while operational, does not perform as desired. Furthermore, original AutoTune had its own digital signal processor that only run the AutoTune program and could add other features to maximize the performance that are not implementable in a plugin. However, the plugin does perform the intended task with a moderate success. The additional feature of letting the user manipulate the parameters allows for a more personalized experience. The adjustable paramaters also let the user find the values that give better results.

Apart from the main objective of developing a plugin, since the final plugin implementation created lag in the DAW, an application has been developed too that performs the same tasks by loading a file and the storing the results in another file to play the results later on. This way, the user can hear the end result without any lag.

While the application can be run in any OS, the VST plugin is only compatible with Windows. Its a shame since the current Projucer does not support VST in Linux and I do not have access to Mac OS.

Having little experience with audio processing, the JUCE framework has proven to be a very useful tool that made the final implementation process easier with its many tutorials and features. On the other hand, using python for the prototyping phase has proven to be

a very good choice. The python language coupled with the usage of Jupyter Notebook and the SciPy libraries have made the process of testing algorithms and visualizing the results for different audio samples easier and faster.

6.2 Future work

I believe that, while the end result can be improved, the desired level of accuracy, speed, performance and versatility can't be achieved with the current algorithm. However, the following modifications could be made to get a better result:

- Implement the storage of samples in the AutoTune class as circular buffers. This would lower the strain on the memory of the program considerably.
- Find better interpolation methods to obtain the output samples. Although the current method works fine, it could be made more accurate for audio waveforms resulting in a better quality of the final sound.
- Including the possibility of choosing which music scale to use would lower the errors due to bad estimations of the correct pitch.
- Including MIDI support to allow the user to manually select the notes that are being played not only would reduce the mistakes of the correct pitch estimation but would also allow the program to choose which parts to correct and which parts to leave as is getting a better performance and avoiding mistakes due to "silence". Furthermore, this would give a new layer of versatility since the user cold choose any note instead of the closest one creating a wide variety of possible results.
- Adding transposing. Since the plugin already does pitch shifting for the correction it would be interesting to also add the possibility of also transposing the song by adding (or subtracting) a given value to the ratio using tones and semitones.
- When the DAW is closed, the parameters chosen for the plugin are returned to their default values. It would be desirable to fix this situation.

While this modifications would end with a better result, the best choice would be to choose another algorithm that uses FFT to get a more useful product that can be used in todays music industry. Right now, even though this implementation is very interesting in a theoretical level, it does not have a great usability in real music production projects.

Chapter 7

Conclusiones y Trabajo Futuro

7.1 Conclusiones

Aunque se ha conseguido el objetivo principal de crear un plugin capaz de corregir el pitch en tiempo real, no se han conseguido los niveles de precisión y rendimiento. Durante la fase de prototipado, resultó evidente que los métodos escogidos eran o demasiado lentos o no eran suficientemente precisos. Aunque la patente de AutoTune parecía prometedora, los mecanismos más avanzados y los detalles de algunos pasos no estban explicados con suficiente detalle; lo que resultó en un producto final que, aunque operativo, no tiene el rendimiento deseado. Además, el AutoTune original tenía su propio procesador dedicado al procesamiento de señales que se encargaba íntegramente de la corrección del pitch, lo que le permitía añadir ciertos mecanismos para maximizar el rendimiento y la precisión del algoritmo que no pueden ser implementados en un plugin. De todas formas, el plugin sí realiza la tarea deseada con un éxito moderado. La opción adicional añadida de permitir al usuario manipular los parámetros de la correción da una experiencia más personalizada. Además los parámetros ajustables permiten al usuario encontrar los valores que den mejores resultados.

Además del plugin, dado que la implementación final creaba retrasos de la reproducción en el DAW, he creado una aplicación que realiza la misma tarea de corregir el pitch. Esta aplicación carga un archivo, corrige el pitch y luego guarda los resultados en un archivo a parte permitiendo al usuario escuchar el resultado sin retrasos.

Aunque la aplicación corre en cualquier sistema operativo, el plugin sólo es compatible con Windows. Es una lástima ya que Projucer actualmente no soporta el formato VST in Linux y no tengo acceso a el sistema operativo Mac Os.

Teniendo poca experiencia con el procesamiento de audio, el framework JUCE ha resul-

tado ser una herramienta muy útil que hizo que la implementación final fuera más fácil con sus múltiples tutoriales y la gran cantidad de utilidades que ofrece. Por otra parte, usar Python para el prototipado ha provado ser una buena elección. El lengua Python unido con el uso the Jupyter Notebook y la librería SciPy ha permitido que la realización de pruebas y la visualización de resultados para distintas muestras de audio fuera más fácil y rápida.

7.2 Trabajo futuro

Creo que, aunque el resultado puede mejorarse, el nivel deseado de precisión, velocidad, rendimiento y versatilidad no puede ser obtenido con el algoritmo actual. De todas maneras, las siguientes modificacions permitirían obtener un mejor resultado:

- Guardar las muestras de audio en la calse AutoTune dentor de un buffer circular. Esto permitiría reducir la carga sobre la memoria del programa.
- Buscar un método de interpolación más apropiado para obtener las muestras de salida. Aunque el método actual da buenos resultados, un método más precioso para audio mejoraría la calida del audio final.
- Permitir al usuario elegir entre distintas escalas musicales reduciendo la cantidad de errores debidos a la estimación del pitch correcto.
- Incluir soporte MID para permitir al usuario seleccionar las notas que deberían sonar. Esto no sólo reduciría los errores de la estmiación del pitch correcto sino que también permitirían al programa discernir entre las secciones que deben ser corregidas y las que no reduciendo la carga, mejorando el resultado y evitando errores por el "silencio". Además permitiría añadir versatilidad ya que el usuario podría no seleccionar la nota más cercana creando un amplio rango the posibles modificaciones fuera de simplemente corregir el pitch.
- Añadir la transposición del audio. Ya que el plugin realiza pitch shifting para la correción sería intersante añadir la posibilidad de transponer la canción medinate la adición (o substracción) a la relación de remuestreo de un valor fijo basado en tonos y semitonos.
- Cuando se cierra el DAW, los parámetros escogidos para el plugin vuelven a su valor por defecto. Sería deseable arreglar esta situación.

Pese a que estas modificaciones darían un mejor resultado, la mejor elección sería elegir un algoritmo que utilize FFT para obtener un producto final que fuera más útil en la industria musical actual. Actualmente, aunque esta implementación es interesante a un nivel más teórico, no tiene una gran aplicación en proyectos de producción musical reales.

Bibliography

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

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

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

[4] Pich correction application download. https://mega.nz/file/FwcyUQLA#t9aqkNqj6g-1FXrpRH11OvC5JctsgGfpHblsn4-rc4E.

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

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

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

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

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

[10] VST plugin download. https://mega.nz/folder/Ys01BIJL#qOfT83PQCfvHvz-zBA0cxQ.

[11] 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.

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

[13] Philip McLeod and Geoff Wyvill. A smarter way to find pitch. 01 2005.

[14] 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.

[15] 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.

[16] 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/.

[17] 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.

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