Conservatorio di musica di Vicenza
A. Pedrollo

Diploma Accademico Ordinamentale di I livello

Real-time Intelligent Harmonizer Based on AMDF Pitch Estimation

Author:
Mattia Paterna (Matr. 4903)

Supervisor:
Dr. Carmine-Emanuele Cella

A thesis submitted in fulfilment of the requirements for the degree of Bachelor

GRAIM
Scuola di musica elettronica e nuove tecnologie

Anno Accademico 2013/2014
Sessione terza

Itaca tieni sempre nella mente.
La tua sorte ti segna quell'approdo.
Ma non precipitare il tuo viaggio.
Meglio che duri molti anni, che vecchio
tu finalmente attracchi all'isoletta,
ricco di quanto guadagnasti in via,
senza aspettare che ti dia ricchezze.

Itaca t'ha donato il bel viaggio.
Senza di lei non ti mettevi più in via.
Nulla ha da darti più.

Kostantinos Kavafis,

Itaca

CONSERVATORIO DI MUSICA DI VICENZA
A. PEDROLLO

Abstract

Graim
Scuola di musica elettronica e nuove tecnologie

Diploma Accademico Ordinamentale di I livello

Real-time Intelligent Harmonizer Based on AMDF Pitch Estimation

by Mattia PATERNA (Matr. 4903)

The pitch organisation through time is called *harmony* for polyphonic sounds. The pitch of harmonic sounds can be shifted, that is transposing the note and its height. Harmonising a sound means mixing this with several pitch-shifted version of it to obtain specific chords. Pitch shifting consists in scaling the frequency axis of a time-frequency representation of the sound. When a harmoniser is based on several pitch shifter engines, any of this controlled by the input pitch and the harmonic context, it's called *smart* harmony or *intelligent* harmonisation. To achieve this kind of audio effects, an algorithm for pitch estimation is needed.

Acknowledgements

This work was written over a period of some months. This is the first time I attempt on a project of such a complexity. Many of the concepts and the subjects of this work were completely new to me. I learned to write in a programming language as C++, learned to use several softwares. I started thinking in a more logical way and that will help me in non-musical fields. Undoubtedly, it has been an educational path rather than a final project. It's a starting point for what I'm to going to do in the future, or for what I would like to do. I hope for possible collaborations that will help me to continue in this research and will allow a more precise and deeper knowledge of what is necessary to this project.

I am profoundly indebted to many generous people for their numerous suggestions. First, my Supervisor Carmine-Emanuele Cella, for his precious time, and big effort. He made me widen my mind, and see the unknown. I'll keep all his influences for years. I would like to thank Lorenzo Pagliei, my main teacher at GRAIM, for his dedication to the Bachelor course. I'd like to thank Davide Tiso for every tip and for the ambition and perseverance I learned from him. I'd also like to thank my classmates: they made our class a very comfortable place where I spent some of the best moments of these past years. I should thank Marzia, who read all this draft and carefully gave me some suggestions to improve the language. I'd like to thank all my professors, and all the people who walked along with me during this time for their patience, comprehension and love, in particular Alessia and my brothers InSpiral. But above all, I should thank my family for giving me the freedom to lead my life.

Contents

List of Figures

List of Tables

For Emily, whenever I may find her

Simon & Garfunkel

Chapter 1

Project overview

1.1 Introduction

The entire work mentioned hereafter is based on the study of possible ways in which a monophonic signal can be harmonised. In particular it'll be examined the class of *pitch shifters*, from the old Eventide H910 Harmonizer through a vast amount of algorithms based both on time and frequency approach. They usually are used in processes of sound manipulation for transposing the pitch of an audio source, e.g. pitch shifting.

The algorithm chosen to make pitch shifting in this project is based on delay-lines continue modulations [1]. This method, however, doesn't allow for a timbre preservation [2] and modify the original signal in a significant way. One of the aim of this work is to give the user several possibilities to harmonise the audio source in the most natural and musical way. In order to get this, this *Intelligent Harmonizer* uses a pitch estimation algorithm to choose a chord in relation to what degree scale the source is. The user is given the control about choice of the root key and the harmonisation mode.

The instrument is completely developed in Max. The pitch shifting engine allows to select between different algorithms, and so does the pitch detection engine. The external MSP objects (pitchdetect~ and dlshift~) are developed by the Author, together with the Supervisor, and are shown later in 2.

1.2 State of the Art

1.2.1 Pitch Shifting

Some sound transformations are combinations of time-domain and frequency-domain manipulation. In general, when a sound is altered in time, some modification in frequency occurs. If we want to double the duration of a sound, its pitch will be a half of the original one, and viceversa. So, how to change or shift the pitch of a sound without altering its duration? The technique involved in this process is called *pitch shifting*. As said in [2]:

> pitch shifting is the multiplication of every frequency by a transposition factor (i.e., the magnitude spectrum is scaled)

Pitch changing and pitch estimation can be realised by several means, with different degrees of success. For instance, in [3], it's said that pitch shifting

> may be used in audio processing, such as in music synthesis, where the original pitch of musical sounds of a known duration may be shifted to form higher or lower pitched sounds of the same duration.

To get this aim, there are several approaches both in time domain and in frequency domain. Their description is presented in this subsection.

1.2.1.1 Time-based approaches

The *Harmonizer*. The first commercial digital device that was based purely on time-domain techniques was the Eventide H910. It was released in the middle of 1970s and, since there, this kind of effect has become very popular [4]. It consists of a real-time transposition of an incoming signal without altering its duration. Eventide called its device *Harmonizer* so, at the present time, every device that resembles the manipulation of pitch is generally called this way (or *pitch shifter* just because Harmonizer is a trade mark of the Eventide company).

The main limitation of the use of the harmonizer is the characteristic quality that it gives to the processed sounds [2]. The sound quality of a harmonizer is based mainly on the nature of the input signal and on the ratio of pitch change it is asked to perform. For the transposition of the order of a semitone, no objectionable alteration is heard. As the transposition ratio grows toward its extreme values, generally ± 2 octaves, the timbre of the sound obtains the character that is specific to the harmonizer. Moreover,

some commercial devices produce undesirable side effects (such as buzzing and glitches) when used on critical material, such as vocal sounds.

Pitch shifting by resampling. This technique is based on the effect of the analog audio tape recorders. In fact, variable speed replay leads to a compression/expansion of the spectrum of a signal. Particularly, during the faster playback the pitch of a sound is raised and during the slower playback the pitch is lowered [2]. A dual operation in the digital domain corresponds to a sample-rate conversion, i.e. *resampling* the signal [4]. This is essentially the pitch variation technique used in the *wavetable* synthesis - reading the table at higher speeds produces a higher pitch.

In a digital audio system, samples are skipped or doubled in resampling. The number of samples that are skipped is proportional to the amount of pitch shifting specified. The process of skipping samples in resampling is called *decimation* and allows for upward transpositions. On the other hand, downward transpositions are achieved by creating new intermediate samples between existing ones. This means *interpolation* between samples. In particular, to pitch shift by a ratio N/M, we first interpolate by M and then decimate by N. For example, to shift a sound up by perfect fourth (ratio interval: 4/3) we first upsample and interpolate by a factor of 3, then downsample and decimate by 4.

A side effect in applying this simple pitch shifting technique is that the sound duration is altered. In order to rescale the pitch-shifted sound to the original length, a time-stretching algorithm must be applied to it. Furthermore, decimation and linear interpolation can create some artefacts due to discontinuities between two disjoint samples so that a filtering stage is needed to partially remove aliasing. Moreover, all the features of the spectrum (and so, the *formants*) are simultaneously scaled up or down[2].

Pitch shifting by delay-line modulation. Many audio effects are based on mixing the original signal with a copied and delayed version of it [5]. In this case, a variation in reading the delay line produces a dynamic component that is perceived in frequency modulation. So, a continue variation of fixed amount produces a constant transposition of the pitch of a signal coming into the delay line [1].

Generally speaking, a delay-line pitch shifter is based on an overlap-add scheme with *at least* two different delay lines and does not require any fundamental frequency estimation. First, the signal is divided in small chunks (*blocking* process). The chunks are copied into the delay lines and there are read faster or slower in order to obtain an upward or downward transposition. To produce a continuous signal output, two

chunks are read simultaneously with a delay time equal to one half of the block length. Finally, a cross-fade block combines the output: when one chunk is at its end, the cross-fade function is at its zero [2]. This allows to remove possible click due to signal discontinuity at the edge of the block. Since we choose this method for our pitch-shifting engine implementation, more detailed description will be provided later in 2.2.2 and in 2.3.4.

Pitch shifting by Granular Synthesis. Granular synthesis involves breaking an input signal into short duration units, e.g. the *grains*. This is equivalent to the process of *windowing* that occurs during the fast Fourier Transform [4]. The main problem in granulation is that the waveform at the end of a grain may not match the waveform at the beginning of the next one. This makes digital granulation exhibit a periodic clicking sound caused by undesired splicing transients, that don't appear in the original waveform. For that reason, smooth envelopes should be applied to every grain to create a seamless crossfade between grains.

Pitch Synchronous granular synthesis (PGSG) is a better technique designed for the resynthesis (and the transposition too) of a sound that contains one or more formant regions in its spectra, such as vocals one. PSGS involves numerous operations. First, there's a *pitch detection* stage that matches the exactly pitch-period of the incoming signal. This information is used to rescale in real-time the duration of the grain inside the granulator. So, if we have to increase or decrease the reading speed of a single grain for raising or lowering the sound, no alteration in spectral content is done. Pitch synchronisation implies that PGSG more correctly processes monophonic sounds. At each frame time, the system emits a grain that is overlapped and added with the previous. The delay between successive grains is called *hopsize*. This particular process, called *overlap-add* (OA), is implemented in the PSOLA technique. In fact, the PSOLA technique preserves the spectral envelope of a signal [6].

Revoice is a software developed by Carmine-Emanuele Cella that performs harmonization of human voices by PSGS. The author says that:

> the program can also produce harmonization effects and is based on synchronous granular synthesis and on other complex techniques to detect the pitch of a signal in realtime.

The use of this technique takes into account the possibility to maintain the timbre characteristics in each register during the transposition, for what is called *constant timbre transposition*. The PSOLA technique is the dual operation to resampling in the time domain and can be conveniently used in pitch shifting a speech sound maintaining all its vowel identity [2].

1.2.1.2 Frequency-based approaches

The Phase Vocoder. The phase vocoder is a digital signal processing technique of great significance. Its principal aim is to separate temporal information from spectrum information to achieve possible independent modifications. In fact, the phase vocoder can perform very high-fidelity pitch transposition of such a different variety of sound sources. This signal processing algorithm can be categorised as *analysis-synthesis* technique. The phase vocoder allows for both of these processes. Mathematically, these techniques take an input signal, analyse it and produce an output signal that can be equal to the original or a modified version of it [7]. The parameter values derived from the input analysis can be altered in order to make a useful manipulation of the original signal. The phase vocoder is named so, to distinguish it from the earlier *channel vocoder*. It was first described in an article by Flanagan and Golden [8], but this techniques became popular only in the successive decades.

The phase-vocoder analysis stage can be viewed in two complementary ways. The former consists of a fixed bank of a *bandpass filters* whose outputs are values that show time-varying *amplitude* and time-varying *frequency*. The frequency response of the filters is identical and the number of filters should be sufficient so that each filter describes one component of the signal. One important consideration is that the band edges could overlap each other. So, the aim is to get the sharpest filter cutoff with minimum overlap. The sharper the filter frequency response, the slower the filter responds to changes in input signal. The complementary view consists of a succession of overlapping Fourier transforms in a finite-duration temporal window. Here, the number of filter bands is the number of frequency *bins*. Since most of the signals existing in nature are not perfectly periodic, the Fourier series can be computed by the *Short Time* Fourier transform, assuming that the windowed section repeats indefinitely and out of the window there's no signal. As in the filter interpretation the frequency response can vary the filter response to the signal, in the Fourier interpretation the shape of window influences how the signal spectrum is *smeared* in other close bins. The amount of smearing increases as the window duration gets shorter. These perspectives are complementary because in the latter the attention is focused on the values for *all* the frequency bins at a single instant time, as in the filterbank view the temporal succession of the values for a *single* filter is emphasized. Finally, in either case it's necessary to convert from rectangular to polar coordinates in order to calculate time-varying values both of frequency and amplitude.

This dual interpretation deals with two divergent and equal interpretation of resynthesis. A more recently interest on phase vocoder has focused on its ability to transform

several typologies of sound in such a musical and useful ways. Here, the *pitch transposition* technique is emphasized. In fact, it's possible to change the pitch of a sound without altering its duration with the use of the phase vocoder technique. Specifically, it should *time-scale* the FFTed signal by the pitch-change desired factor (e.g. *pitch ratio*) and then play the buffer at a *wrong* sampling rate. The example below, taken from Carmine-Emanuele Cella's *microCoder* ver. 0.1, clarifies this process.

```
int hspect = fftlen * pitch;
for (int i = 0; i < fftlen; ++i) {
        int idx = ((int) i / pitch);
        if (idx < hspect) {
            ampsyn[i] += amp[idx];
            freqsyn[i] = freq[idx] * pitch;
        }
}
```

A possible complication arises for wide shifts, e.g. an octave interval. In fact, pitch shifting process doesn't alter only the pitch, but also the frequency content so that, for instance, vocal sounds can be modified in their formantic regions characteristics. This causes important modifications both in the nature of sound and in its intelligibility, if speaking of speech signals. To remedy this issue, an additional operation should be inserted into the phase vocoder algorithm. This is a manipulation of the *spectral envelope*.

Unfortunately, using phase vocoder to perform pitch shifting has undesirable side effects, as expressed in [3]. In fact, if we're using time-scale modifications as described above, the phase vocoder can perform only one shift at a time and the computational cost is strictly related to the pitch modification factor. Furthermore, new techniques that allow for more flexible modifications has been developed. These operate solely in the frequency domain, bypassing the resampling stage, and are based on the idea of identifying peaks in the Fourier transform and then translating them to different arbitrary frequencies, allowing also non-linear pitch shifting [9].

1.2.2 State of the Art: Pitch Detection

Nowadays, sound analysis has emerged as a central point in the development of musical technologies. Its importance allows for new and more subtle possibilities in sound transformation. Particularly, the concept of pitch has been proved crucial in many of processing techniques involved in any kind of sound. An application that attempts to find the *fundamental frequency* from an input signal is called *pitch detector* (PD), or

pitch estimator. The concept of pitch is ambiguous itself, because the human perception is not completely understood yet, and what a user asks of a PD is quite difficult.

Numerous algorithms for pitch estimation has been developed. These are based both on time-domain and on frequency-domain approach, while others are hybrid approach. A complete description of this field falls outside the aim of the work. For that reason, we decided to describe only two algorithms in the following paragraphs. The choice focuses on ACF and AMDF, that are very related because the latter is a modified version of the former. We choose AMDF-based algorithm for our purpose because it's easy to implement and gives general efficiency without spending so much time. Moreover, AMDF based method exhibit less computational complexity than the ACF based one [10], even if ACF performs better in case of critical material or very noisy speech.
Before treating the two selected algorithms, we should mention the most important pitch tracking algorithm of either categories. From time domain detection, we remind *zero-crossing* method, well-described in [4]. In Roads, also the *cepstrum analysis* is described. This method, together with the *Harmonic Product Spectrum*, belongs to the frequency domain detection. We also should mention Yin, an algorithm for estimation of the fundamental frequency based on ACF with some modifications, mainly used on speech [11]. A comparative study of several pitch detection algorithm is described in [12]. Finally, in [13] a MSP object based on *maximum-likelihood* detection method, fiddle~, is presented.

Pitch detection may be a prelude to a pitch interpretation within a musical context, or a scalic one - and so it's the case of this Intelligent Harmonizer.

Autocorrelation pitch detection. Correlation function compares two signals. The aim of the correlation is to find the *similarity* between these two considered signals. Correlation function compares signals on a point-by-point analysis; thus the output of a correlation is a signal itself. Generally speaking, if the correlation function is 1, the signals are equal in that point. If it is 0, then the two signals are uncorrelated [4].

Autocorrelation means that the signal is compared with versions of itself delayed by an amount of successive intervals. The point of comparing a signal with a delayed version of itself is to find indicators of periodicity, like repeating patterns. When autocorrelation finds two identical samples in the two versions of the signal, then it can assume the signal is periodic with a period that is exactly the amount of delay from them. Autocorrelation pitch detectors hold part of the signal, storing it into a buffer. To do this, the signal is windowed and the windowed segment is compared with version of itself delayed by one sample, two samples, and so on up to n samples,

where n is the buffer size. If the detector finds a match in the different patterns, this indicates periodicity in the signal. The detector measures the time interval between the two patterns, that is the delay amount value in samples too, and the strongest correlation (i.e. the maximum value from these that tend to be 1) is estimated as the *fundamental* pitch. Obviously, there will not be so many autocorrelation values equal to 1 because every signal existing in nature is not perfectly periodic. In fact, we can speak of *quasiperiodic* signals.

Autocorrelation function (ACF) is one of the most basic pitch detection methods that have been used in a wide range of fields. Since the autocorrelation of a periodic signal is also a periodic signal [14], the period of the autocorrelation function *reflects that of the input signal*. The ACF algorithm is defined as:

$$ACF(\tau) = \frac{1}{N} \sum_{n=0}^{N-1} x(n) \cdot x(n-\tau) \tag{1.1}$$

where $x(n)$ is the signal, N is the length of $x(n)$ and is the total number of points involved in the calculation. τ is the delay or *lag time* and it's defined in the range $0 \leq \tau \leq (N-1)$.

AMDF-based pitch detection. The Average Magnitude Difference function (AMDF) is actually another kind of autocorrelation algorithm. It is a variation of the ACF analysis where, instead of correlating the input signal at various delays (where multiplication and summation are formed at each value), a function of differences is formed between the delayed signal and the original one and, at each delay value, the absolute magnitude is taken [14]. So, the AMDF calculation requires no multiplication and this makes it more suitable for the real-time applications. The AMDF, in its original formula presented in [15] is defined as:

$$AMDF(\tau) = \sum_{n=0}^{N-\tau-1} \mid x(n) - x(n-\tau) \mid \tag{1.2}$$

where $x(n)$ is the processed signal with the length N and τ is the *lag* number which ranges from 0 to $N-1$. As in the autocorrelation function, the signal is also exposed to the *frame-blocking* technique. In practice, the signal is multiplied with a rectangular window of length N. This makes the signal null in every point outside the block. The pitch is defined for the minimum value of the difference function. The τ value that makes AMDF minimum is equal to pitch value.

The advantage of AMDF is the low computational complexity due to the transformation from multiplications to subtractions. However, AMDF contains two possible

calculation errors. In Eq. 1.2 less data is involved to calculate AMDF at higher lags. Therefore AMDF cannot show periodic nature on the later half of the frame and it's often called *falling tendency* in literature. AMDF can also present the so-called *double pitch* error (or *half pitch* error), that is the local minimum appearing at integer multiples of the true pitch, or at halves values [16]. Many strategies have been proposed in order to eliminate these undesired errors and many papers have been written; for instance, in [17] a new method based on circular AMDF is proposed, whereas in [16] and [10] more robust AMDF algorithms are described. Moreover, a modified version of AMDF algorithm is also presented in this paper.

1.3 Conclusion

In this chapter, we presented an overview of the project and its final aims. We also described the state-of-art of both pitch shifting engines and pitch detection methods. For convenience, when treating pitch shifters, we distinguished between time-domain and frequency-domain approach. Later, we discussed the main methods in the field of pitch tracking. We gave a deeper description of AMDF algorithms, that we're going to improve and implement in the next chapter. We did not provide a more detailed description of these several alternatives. For more information about, the reader is suggested to refer to the referenced papers.

Chapter 2

Algorithm design

The aim of this chapter is to give the reader the most important information about the algorithms used in this work, their implementation and their specific differences relative to what has been already said in 1.

Research has been conducted this way: first of all, it has been examined all the literature about a specific technique; then, an own implementation has been done in C++. So it has been produced two main codes: one for the pitch detection, the other for the pitch shifting. These codes, specifically, represent the engines of the relative MSP externals presented here. Before introducing these codes within the tools provided by *Max SDK* (the Cycling 74's Software Developer Kit) in order to build these externals, a command-line version of the codes is made, so to test and improve them easily. Finally, Xcode 6 has been used to realise an **.mxo** object that will be recalled inside the Max 6 patcher built by the Author and the Supervisor together.

2.1 Fundamental decisions

An AMDF-based algorithm for pitch estimation has been chosen because it better suits to the requests of the Intelligent Harmonizer, that are working in real-time and with a small computation cost. In order to prevent and eliminate the problems involved with the use of this technique, several improvements are proposed inside the algorithm. Moreover, it's been added a *median-filter* section after the pitch detection stage to smooth undesired and too fast frequency changes due to the limits of this technique and to the possible issues of the signal.

To achieve pitch transposition, delay-lines based pitch shifter sums up a certain simplicity in the implementation of its code, together with a low cpu amount and a good sound quality - it doesn't require many improvements. Furthermore this approach allows to process the sound without switching to frequency-based domain, as it's the case of the phase vocoder instead.

2.2 Implementation

2.2.1 Pitchdetect~

First, the input signal is passed through a low pass filter to improve the accuracy of this technique [12] (and also to cut off a part of the spectrum not useful for the pitch estimation). Then the AMDF algorithm takes place, as shown in the following listing:

```
for (int n = 0; n < N2; ++n) {
                float diff = 0;
                rms += (buffer[n] * buffer[n]);
                for (int m = 0; m < N2; ++m) {
                        diff += fabs (buffer[m] - buffer[m + n]);
                }
                diff *= pow (n, coeff);
                amdf[n] = diff;
                if (maxAmdf < diff) maxAmdf = diff;
        }
```

We might observe an important revision of the classic algorithm implementation shown in [15, 16, 18, 19]. Unlike traditional AMDF, this version computes two independent loops, one internal to the other. These loops count from zero to half buffer. So, herein, we could suppose the range in which the difference is calculated, isn't the whole buffer length. Despite this supposition, all the buffer samples are taken into account to

compute the minimum difference. In fact, while the external loop increases the value of n, the internal one increases m. So, for instance, when $n = N2-1$ (e.g. half buffer) the range of $[m+n]$ varies from $N2-1$ to $N-1$. We have *in any case* a full comparison within the buffer. This strategy shorten the time in which the computation is done, allowing for better real-time performance without wasting cpu resources.

Threshold and Bias. We might notice a specific parameter in the listing above: *coeff*. What does it mean? It has been previously said that AMDF technique suffers from possible falling tendency [16]. This explains why sometimes the frequency estimated is lower, compared to the expected one. One possible solution is to partially modify the result of summation by well averaging the lag values. In this work, *coeff* means how much the values are averaged. Because *coeff* is the exponent, its right values shouldn't be less than 1. For coeff = 1, the difference value remains the same calculated by AMDF. If we increase this value, we raise the difference value and, obviously, the entire summation. This parameter, inside the external structure, is called *bias*.

```
int minLag = (1. / maxF0) * sr;
        int maxLag = (1. / minF0) * sr;

        if (rms < threshold) return 0;
```

As shown in the listing above, the parameter *threshold* defines when the input signal has to be considered as silence or as sound. The higher the threshold value, the less the estimated values that pitchdetect~ outputs. In fact, at the same time pitch detection is done, the algorithm also computes the RMS (*root mean square*, that is the energy of the signal) on the entire buffer. If this RMS value is less than the threshold one, supposed the energy value is given in linear amplitude, the system will output a zero value: no fundamental frequency is given because there's no harmonic sound in the signal.

The absolute minimum. First of all, the research of the minimum difference is limited inside a specific range, whose side values are given by the user. These are called *min* and *max frequency*. The user should keep in mind that this algorithm, as almost every detection algorithm, could exhibit some issue at its extreme values, so to select an accurate range. For instance, a more precise detection of a 100 Hz fundamental could't be made if the *min frequency* parameter is set to the same value.

```
for (int i = minLag + 1; i < maxLag - 1; ++i) { //safety
                if (amdf[i] < amdf[i - 1] && amdf[i] < amdf[i + 1]) {
                        minimaPos[currentMinimum] = i;
                        minima[currentMinimum] = amdf[i];
                        ++currentMinimum;
                }
        }
```

In this stage of the research, every AMDF value previously calculated (whose index is in the range determined by the user) is compared with the one before and the one after. If this value is the lowest, then we have a local minimum. Its value and its position (i.e. the samples at which we have the minimum) are stored.

```
        for (unsigned int i = 0; i < currentMinimum; ++i) {
                if (minima[i] < min) {
                        min = minima[i];
                        index = i;
                }
        }
```

Next, all the local minimum are compared each other to get the absolute minimum and, in particular, its sample position. In fact, the *index* found refers to the array of indexes *minimaPos*. The value that pitchdetect~ outputs is $sr/minimaPos[index]$, where sr is the sampling rate - that is, a value in Hertz that corresponds to the fundamental frequency for the examined buffer of input signal.

2.2.2 Dlshift~

Dlshift stands for *delay-line* shifting. This approach leads to a quite simple implementation and requires no large CPU resources to let a multiple utilisation possible.

Circular buffers. Generally speaking, the pitcher effect is created by increasing or decreasing the delay by a fixed amount [1]. If looked from a different view, we could think of it as a circular delay line constantly re-written, as it was a wavetable. The term *circular* means that the reading and the writing point of the the delay line are dynamics, changing their position along the line itself. This allows to keep the length of the buffer quite short in order to let the user not to perceive a significant delay of the processed sound relative to the original one. This leads to an important consideration: there always will be a little amount of delayed signal that gives the Harmonizer the classic *phasing* effect [2].

The delay reading point then proceeds through the line with an increment that reflects the desired pitch transposition rate. If its position increment is 1, there's no pitch transposition - and then, no delay change. If the increment is major than 1, we're playing the delay line at a faster rate than the rate at which it was written, so to have an upward transposition. Conversely, if the increment is between 0 and 1 the pitch is shifted downward.

Main features. In this implementation, pitch shifting is achieved using two time-varying delay lines. The continue modulation is provided by a sawtooth-type function [2]. For its similarity to the relative MSP object, this is called *phasor* to remind the principle on what the *phasor*~ is based, that is to generate a sawtooth function expressed in Hz. So, it's required to allocate externally a one-second length buffer to write a sawtooth function in. The length of this buffer depends on the sampling rate of the system. The rate at which the sawtooth function is played is given by the different increment of the reading index of the table. One of the parameters the user is asked to input at the MSP external initialisation, *phasor frequency*, is exactly how much the reading index has to move forward every loop step in order to achieve the desired transposition rate.

Here is the central point of shifter code.

```
out = (frac * D1[next] + (1.0 - frac) * D1[rpi]) * x;
                out += ((frac2 * D2[next2] +
                        (1.0 - frac2) * D2[rpi2])) * (1 - x);
                D1[(int)wptr] = sig[n];
                D2[(int)wptr] = sig[n];
                outbuffer[n] = out;

                ++wptr;
                wptr %= dt;
                tablePos += (pf);
                if(tablePos < 0.) tablePos += len;
                if (tablePos >= len) tablePos -= len;
```

This is included in a main loop that also contains the calculation of *phasor*, *cosine* and *reading point* values. The conditions of this loop are: for $n = 0; n < N; ++n$, where N is the vector size length. We might notice how the algorithm works doing these actions in the following order [1]:

1. both the delay lines are read at their current reading position and summed up together to generate the output;
2. the input signal is written that position on every buffer;

3. the reading positions are incremented.

In particular, the algorithm always checks if the reading point has reached the end of the delay line. In fact, we should keep in mind that the vector size is generally smaller than the delay line - this time, we choose a 50 ms long delay line. It needs the writing position index to be saved before a new buffer of audio signal comes in, so to resume the count during the successive loop. To do that, we pass to the function a *reference* value, and so does for the reading position index of the phasor table. In other words, we are passing the variable itself and not a copy of it and the last value is always kept inside the memory location until it's cleared. Then, the modulus operation guarantees that the variable *wptr* never overcomes the maximum value of delay time (dt).

Linear interpolation. Analogous, but slightly different, is the matter of the increment of the phasor table. Here, we could have a non-integer increment due to the fractional *phasor frequency* value. So how to calculate correctly the value from a table that accepts *only integer* indexes?

```
tableFloor = floor (tablePos);
tableFrac = tablePos - (float) tableFloor;
tableNext = (tableFloor + 1 == len? 0 : tableFloor + 1);
phasor_value = (1. - tableFrac) * table[(int) tableFloor] +
                        tableFrac * table[tableNext];
```

The answer is the linear interpolation, and its formula is:

$$(1 - frac) \cdot x + frac \cdot (x + 1) \tag{2.1}$$

Practically, a linear interpolation makes sense only if we consider two integer values at a time. These are the operations requested:

1. calculate the maximum integer value that is not greater than the fractional one (in this case, we're using the C++ *floor* function);
2. calculate the fractional part of the original value;
3. focus on the integer value next to the one we have already calculated and check if it falls inside the phasor buffer;
4. now, it's possible to do linear interpolation.

All the values involved in linear interpolation let the algorithm calculate the other required variables in the loop: the reading position indexes (through *phasor_value*) and the cosine value corresponding to the phasor table position.

2.3 Issues

In the following section, the main issues observed during the implementation stage are briefly shown. A possible solution, or a discussion for a future improvements will be provided. Some of this enhancements take their name from similar parameters that Cella's *Revoice* exhibits.

2.3.1 Vector size issues and *downsampling*

One of the main problems of both implementations is that they don't have *fixed* vector size values inside. This characteristic initially could not represent such a hard problem but changing the vector size from the Max *Audio Setup* window while the algorithm is working can cause undesired clicks, loss of data and, in some cases, it can suddenly stop working. To fix this, we could think of a buffer independence. First, we could declare a constant called *internal_buffer_size* and give it an average value that best suits any case. For any vector size value selected externally, it could be calculated the algorithm's *downsampling* factor, that is how many buffers the system has to input to completely fill its internal one. Once the internal buffer is filled with signal, the process - whatever it is - can take place.

2.3.2 The *Confidence* parameter

The pitch detection algorithm described before exhibits, in some critical situation, undesired values that make us not very *confident* about their possible use. In particular, when used with vocals or non-tempered instruments it may be output all the pitch-period fluctuation, resulting in microtonal pitch. That's truly correct, but it may give us some problems in using these non-tonal values, for instance in calculating the interval between this note and a root key note.

Pitchdetect~ gives the user a parameter to control these possible undesired values, and this is called *confidence*. In practice, this acts like a pseudo-ratio semitone value. Two separate frequency values are calculated with confidence one, *minpitch* and *maxpitch*. Then, the last detected frequency is compared with these ones. If this value stays

between the smaller and the greater, no *tonal* change in pitch happened in the signal. So, the algorithm doesn't output the the last pitch-period but the previous one and so does until the last detected value falls out of this range. All the process is shown in the listing below.

```
float minpitch = m_oldpitch / m_ratio;
float maxpitch = m_oldpitch * m_ratio;
if ((m_pitch > maxpitch || m_pitch < minpitch) || m_oldpitch == 0) {
        m_oldpitch = m_pitch;
}
```

2.3.3 Using a *median filter*

In order to prevent too fast pitch-period changes, not useful if we're using them in tracking some else musical parameter, pitchdetect~ has been provided with a median filter to smooth the output values curve. It's more likely we're interesting in a well-balanced pitch estimation through audible time segments rather than a more precise estimation at every vector size. Generally speaking, the more a parameter is recognised to belong to the human sphere, the more impressive its musical result will be.

```
for (int i = 0; i < MEDIANSIZE; ++i){
                m_median[i] = m_amdf_tmp[i];
}
bsort (m_median, MEDIANSIZE);
m_pitch = m_median[MEDIANSIZE / 2];
```

Typically used on signals that may contain outliers skewing the values we're waiting for, a median filter takes a list of N values, sorts this in an increasing order and, finally, returns the middle value [20]. From that, a median filter allows a smoother distribution. The only cost is a delay of N vector sizes before it returns the first real value. In fact, it has to wait that all the array designated to be filtered is filled to obtain the first significant value. Moreover, as shown in the listing above, we should create a *copy* of this array to let the pitch output order not be altered from sort operation. The length of this array is chosen out of the algorithm and generally is an odd number.

2.3.4 De-clicking and modulation in pitch shifting

We said that the delay line is finite and circular. So, at some point, either the reading point will overtake the writing point or vice versa [1]. This may happen because the

reading tap is moving with a different increment than the writing one, that is the reading taps is moving through the buffer at its own rate. This will create audible popping sounds and a nasty click in the signal when the sawtooth comes back from its end to its beginning. The best way to avoid the click is to add an envelope (sometimes called a *window*) that goes to zero at that discontinuity. This can be, as it is in the case of this work, a cosine shape, going from 0 at its end to 1 in the middle.

However, the signal is now de-clicked but with an important modulation. The sound periodically disappears when the envelope goes to zero. In order to reduce this *tremolo-like* effect, which is generally undesirable, we can use a second reading point (or, better, a second delay line with its own reading point) that is offset by half a delay length, and enveloped in the same way. So, when one tap is at its discontinuity point (and thus its amplitude is zero), the other tap is at its maximum amplitude. This results in a smoother output, with no click and less disturbing amplitude modulation. The cosine shape curve fits naturally any possible amplitude fluctuation due to the *crossover* stage. This methods comes from the old 2-tap pitch shifter implementation in devices like Eventide H910 and H949. The use of several reading taps, or several delay lines, is perceived as a phasing-like effect artifact that is less annoying anyway [2].

2.4 Conclusion

This chapter has provided a detailed presentation of both the MSP externals built in this work. The main features of pitch detection stage and pitch shifting process have been explained. Later, it has been presented several issues that the Author and the Supervisor have passed through. In particular, we focused on possible issues that AMDF algorithms can exhibit when performing on complex material (i.e. speech or very noisy signal). In those cases, pitch-period values could be wrong or different to what we expected. Moreover, we said that the quality of pitch shifting sound can be altered by some undesired sounds, as periodic clicks. At the present time, our pitch shifting implementation is still presenting some artifacts due to possible errors in algorithm implementation. Surely, its sound quality could be improved using a frequency domain technique, as the phase vocoder [9]. Many of the issues described herein have been fixed, others will be surely dealt with in a future.

Chapter 3

Use cases and performance

In this chapter, a more specific description about the device's programming stage is given. During this, it has been chosen Cycling 74's **Max 7** as programming software for its best suitability for the Author and the Supervisor's ideas. The principal aim on which all the work is based is to provide the user a device that feeds his musical character and ideas. We focused on the ideas of an *effortless* use and a *musical* one. The user neither should care about programming or technical aspects, nor it's required him some practical knowledge about audio effects. Obviously, such a complex device requires some experience. We have tried to overcome this trouble giving the user all the device controls in a nutshell. There are many selection menus, clear parameter visualisation and gain meter for monitoring the input and output signal amplitude. Finally, the user interface is given with a plain harmonisation section: two staves take place. In the former, the estimated pitch of the incoming signal is shown. In the latter, the chord built on the selected harmonisation is shown.

3.1 Musical Aspects

3.1.1 Harmonic context recognition

As said earlier in the abstract, an *intelligent* harmonizer should take into account not only the pitch-period of the incoming signal, but also the harmonic function at a specific moment. In practice, it should know which degree scale every note is and give it the right harmonisation. This means that there are two main parameters the user had to select previously in order to let the harmonizer work correctly: the **root key** and the **scale mode** (see Figure 3.2).

The first, *root key*, is a sort of master key that informs the harmonizer about the *tonality* in which the incoming signal is playing (e.g. a musician singing a piece of melody in B). So, for every estimated pitch-period the device calculates its relative degree, expressed in pitch. For instance, if the estimated frequency is the fundamental its pitch degree is 0, if it's the dominant its pitch will be 7, and so on.

The concept comes from the *pitch set class* theory. For simplifying melodic or harmonic models in atonal music, a new classification was presented in early XX century. In this model, the octave is divided into 12 semitones, as the tempered system, but each semitone is named with a number that ranges from 0 to 11. This model easily fits our purpose because it allows us to make *absolute* chords configuration - in fact, for convenience 0 is always C, if not specified any other note.

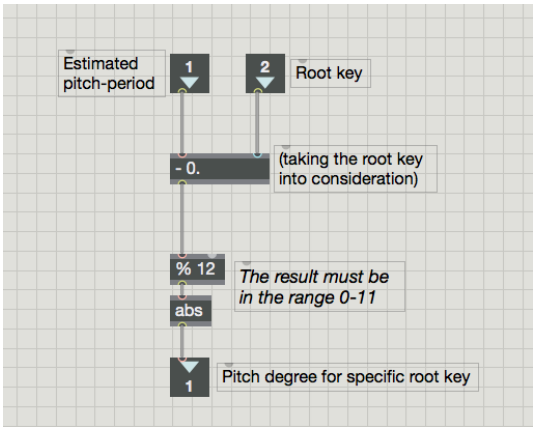

FIGURE 3.1: The *inoctave* Max external.

As shown in Figure 3.1, we *must* take into consideration the selected root key in order to achieve the right degree value. We have already said that pitch value is *absolute*, i.e. tonality is not considered, and so isn't the octave. We should consider whatever octave and give 0 to the lower considered value, that is the *tonic*. In doing that, we

have to convert the root key itself in pitch value and subtract it to the estimated one. For instance, if we have D, or 2 in pitch value, and the root key is C, its degree value will be 2, the supertonic. On the other hand, if the root key is D, we have previously subtracted 2 to the estimated value and so its degree value now is 0, i.e. it's the tonic. The modulus operator let the result always remain in the established range $0 - 11$, as the pitch class set wants. The value found inside this Max external is the pitch period harmonic function and will be used to select the specific chord the Intelligent Harmonizer will use for this note.

3.1.2 Chord selection control stage

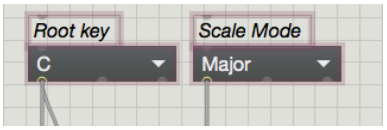

FIGURE 3.2: *Root key* and *scale mode* parameters.

The second parameter shown in Figure 3.2, *scale mode*, determines in which manner the harmonisation takes place. Here, the user can switch between different typologies of harmonisation. Each of this has its own musical character, with its own chordal voices displacement:

- *Major* mode provides a major-scale based harmonisation, while the chords built on the external notes are generally Dominant 7th so to create a like-*tonicization* effect;
- *Minor* mode provides a blended minor-scale based harmonisation. In fact, we take into consideration two of its versions: *natural* minor and *melodic* minor. Furthermore, they are melt together so to give the user more than one scalic combination. Chords for both natural and melodic 6th and 7th degree are provided.
- *Blues* mode provides a harmonisation for degrees that form the so-called *pentatonic* scale with the inclusion of augmented 4th degree in a manner that resembles a classic blues scale harmonisation. No chords for notes that are not part of this scale are given.
- *Parallel* mode gives the user the classic Harmonizer effect. For every octave semitones an equal harmonic function is given, so to create a constant parallel movement of chordal voices.

- *Colours* mode is a slightly different one. In fact, it gives the user very *coloured* chords like suspended chords or add chords, that is a triad with the inclusion of what is not a triadic interval (2nd, 4th or 13th interval for instance). As the name suggests, this kind of harmonisation can be useful when the user is expecting a more subtle and discreet sound, or an ambiguous one.

The *chords* collection. How is the chord correctly chosen? We said previously that a chord configuration is made of 4 pitch values. All the chord configurations are set by software designer and stored into a text file, whose name is *chords.txt*, as shown in Figure 3.3. This text file is loaded immediately at the initialisation of the

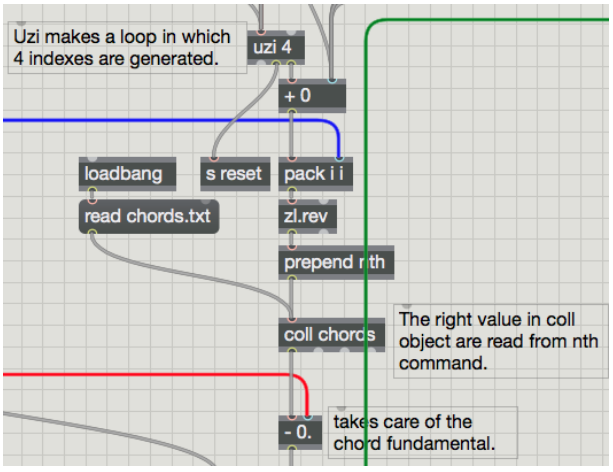

FIGURE 3.3: Harmonisation core stage.

patcher (this is done using the *loadbang* object). The file is made of rows and columns. Each row contains all the different chords that may be selected depending on the *scale mode* choice. The number of rows is equal to the number of semitones that an octave interval can contain. The columns should be read in groups of 4 because we're talking about harmonisation by chords of 4 notes. When the pitch-period harmonic function is recognised, a bang is generated together with the pitch degree value. This bang triggers the *uzi* process, that is to generate a specific number of indexes at the least time amount. These one, together with the pitch degree value, are packed two at a time to make 4 couples. One of these lists contains two integer values: the first refers to the momentary harmonic function of the incoming signal, the second refers to the column in which there's a chord voice it has to be extracted. As the root key and the detected pitch-period, also this voice is in pitch value. The extraction process is made by the *nth* coll argument. It allows for two values, the row one first and the column one then.

Once the desired voice is selected inside the *coll* object, we have to take care about the chord fundamental. For instance, if we have a G note and its harmonic function is dominant, we're expecting messages like *nth 7 0*, *nth 7 1* and so on - for convenience, we're assuming root key is C and scale mode is Major. The pitch values these messages are going to select are 2, 5, 7, 11. These are still not the correct values because we're not taking into consideration that the chord configuration is simply given assuming C as fundamental, as said in 3.1.1. What we must do is subtract the pitch-period scale degree to each selected values. Doing this, the new pitch values will be -5, 2, 0, 4. These are the correct values we could pass to the next stage in order to calculate the ratios for pitch shifting engine. In fact, both the lists refers to the same chord: one with C as fundamental, the other with G - that is our case.

3.1.3 The 'Intellichord' function

One of the main limitation of a so-built harmonisation is the lack of variation in chord succession standing on the same scale degrees. For instance, using the scale modes presented above there's no difference if the user plays a II - V harmonic succession or its inverted V - II. This is quite wrong, because a musician naturally tends to harmonise differently depending on what particular succession he's playing in. For that reason, he'll harmonise differently a II if he comes from a VI rather than if he comes from a IV. The modes discussed before don't allow it. Naming it, inspiration has taken from the similar function implemented in Carmine-Emanuele Cella's *Revoice*. This software has been described in 1.2.1.1.

Here, a different harmonisation mode that tries to care for this musical aspect is shown in Figure 3.4. As we could see, the process of chord selection in *Intellichord* section is somewhat similar to the other one. The main differences are located before the *uzi* process takes place and inside the text file that is read into the coll object.

Let's consider the first different aspect. The detected pitch scale degree comes into the *bucket* object. This allows *Intellichord* to examine every state transition between consecutive chords. In fact, before passing to a next chordal realisation the system takes memory of what scale degree it has just harmonised. Two different states (i.e. degrees) are considered at the same time. The present state defines which harmonic degree the input note is, as it's in the standard section. The previous state determines which chord of a group of possible will be used. Also this collection takes messages with the *nth* argument, and for this purpose lists are created in a similar manner.

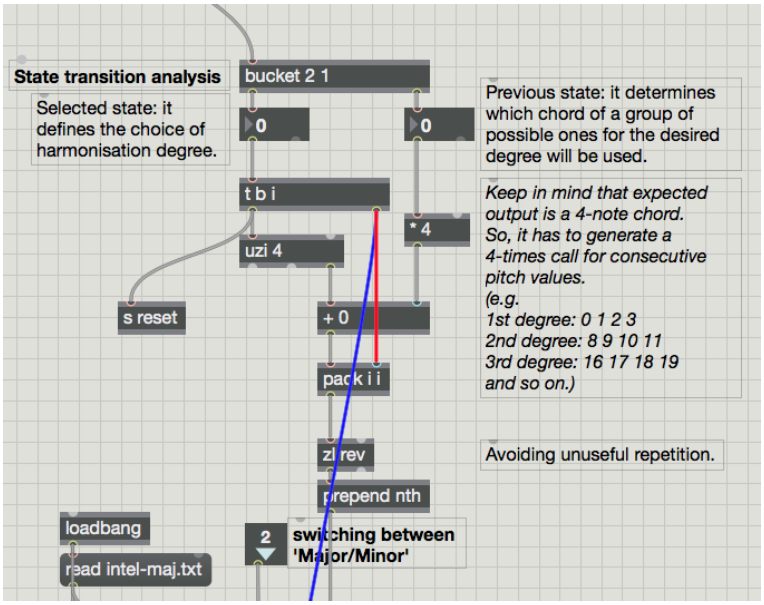

FIGURE 3.4: *Intellicord* engine.

Now let's consider the text file structure:

TABLE 3.1: The *intelli-maj.txt* file structure.

	I	II	III	IV	V	VI	VII
1	I-1	II-1	III-1	IV-1	V-1	VI-1	VII-1
2	I-2	II-2	III-2	IV-2	V-2	VI-2	VII-2
3	I-3	II-3	III-3	IV-3	V-3	VI-3	VII-3
4	I-4	II-4	III-4	IV-4	V-4	VI-4	VII-4
5	I-5	II-5	III-5	IV-5	V-5	VI-5	VII-5
6	I-6	II-6	III-6	IV-6	V-6	VI-6	VII-6
7	I-7	II-7	III-7	IV-7	V-7	VI-7	VII-7

As shown in Table 3.1, the first column indicates which scale degree we are moving from, while the first row indicates which scale degree now has to be considered. There's an implicit clockwise movement, whose final point is the cell that contains the chord that will be used for that specific succession. For instance, if the user would move from II to V, *Intellichord* is going to select the chord inside the cell V-2. If the user was moving from IV to I, *Intellichord* would use the chord in cell I-4. Initials inside every cell show the state transition we're talking. Roman numbers indicate the present state, arabic numbers the previous state.

The main feature of this organisation is that there are 7 possible chords for every harmonic function the device may use, supposing all the state transitions between every scale degree are allowed. This leads to a matrix made of $7 \cdot 7 = 49$ different combinations, and so a greater dynamism in harmonising an incoming signal. Obviously, also the chord configuration in this file is given in pitch values. Differently from the *chords.txt* file, here the chord fundamental is taken into account.

3.2 Interface and performance

Now, let's have a quick look at the controls provided to the user for better helping him during a performance situation.

3.2.1 *General Parameters* control stage.

One aspect the user should notice is that the interface is well-divided into general sections. The first section presented herein, *General Parameters*, contains the most important and basic controls over the entire device. Here, the user can switch between

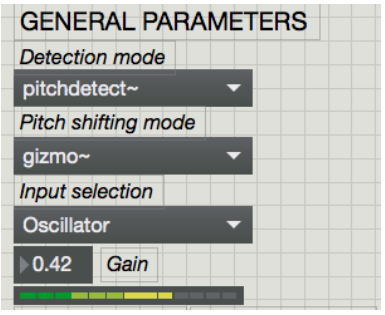

FIGURE 3.5: *General Parameters* interface section.

several different pitch estimation algorithms.
At the present moment, there are two pitch detection algorithms. One, pitchdetect~, is made by the Author and the Supervisor and its features are well-described in 2.2.1. The other, pitch~, was developed in 2001 by Tristan Jehan and is based on another MSP external, fiddle~, originally developed by Miller Puckette in fall '90s [13]. Moreover the user can switch between two different pitch shifting algorithms: gizmo~ and dlshift~. As the second has been discussed previously in 2.2.2, *gizmo* is included directly inside Max 7. This object implements a frequency-domain pitch shifting (see 1.2.1.2).
The concept on which the selection has been done is to give the user the chance to choose and try at least one algorithm for each approach, both for pitch estimation and

for pitch shifting. All the algorithms built and discussed in this project works in the time-based approach, instead the other belong to the frequency-domain approach.

Finally, the user is given the control over three different input signal modes:

- *External* allows for real-time performance. In fact, this choice lets the Intelligent Harmonizer take any signal coming directly from an input device, e.g. a microphone or an instrument. The input device is previously set into the Max *Audio Status* window;
- *Audiofile* is the option the user has to select if he wants to harmonise some audio sample;
- *Oscillator* feeds the device algorithms with a sine oscillator, whose frequency can be manually controlled via MIDI controller or via the device itself. In fact, when choosing this option a window that shows a virtual keyboard will open. The user may change the oscillator frequency selecting the note onto this keyboard.

Also a gain control is provided. The user can adjust the input signal amplitude looking at the level meter and finding the desired one.

3.2.2 Beyond shifting: *delay* and *modulations.*

The following section gives some features that may help the user to achieve a better harmonisation experience.

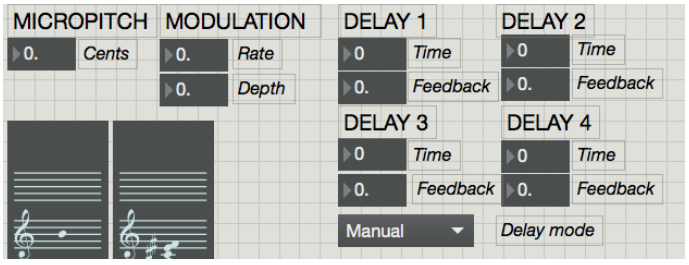

FIGURE 3.6: The *modulation effect* section.

As shown in Figure 3.6, the user is given two effect typologies: *delay* effects and based-*modulation* effects. Delay can be used in several ways. First of all, it can replicate the character of a small vocal group in which the notes of a chord surely don't start at the same moment. It acts producing a *smeared* chord attack. Every singer has his own characteristic vocal emission, and also the *envelope* is involved. Moreover, the singer's

own reaction time can affect the moment in which the vocal emission starts. In order to smooth the chord attack and to allow for a more natural-feel sound, each voice has its own delay with unique *length* and *feedback* controls. The first control is the key for achieving the effect mentioned above, and also true delay effects if the delay time overcomes small values, e.g. those in the range 20 - 80 ms. The second control produces interesting arpeggios effect [1] and can also produces an interesting *blurred* harmony effect. Feedback retains a small portion of the just harmonised note into the delay line, while a new harmonisation is passing through. This generates a blend between notes that belong to different chords and harmonic contexts: a sort of *impressionism* in music. If we're thinking at the delay time parameter in a creative way, it's possible to realise rhythmic patterns simply providing all the delays with different and audible time settings. Finally, a *historical* parameter that comes from Eventide H9 *Quadravox* pitch shifting algorithm has been included. Switching this control from *Manual* to *Fixed*, the four delays aren't independently variable any more. In Fixed mode, the Delay 1 Time affects all the other ones in a manner that makes Delay 2 Time be longer than Delay 1, Delay 3 Time longer than Delay 2, and so on.

Modulation provides an LFO that can be used to achieve a classic *vibrato* effect. It consists of a periodic shifting that tends to simulate the natural imperfections of the vocal emission, and also the particular effect a musician can get on his instrument (for instance, by modulating a guitar string, or using the whammy bar on an electric guitar). We also can think of a vibrato as a vocal technique that comes from a spontaneous nervous tremor in the diaphragm and that can be improved and used on purpose to add expression in music. Since vibrato is typically characterised in terms of amount of pitch variation (that is, the *extent* of a vibrato) and the speed at which the pitch is varied (the *rate* of vibrato), the interface gives the user two independent controls: *depth* and *rate* [21].

Micropitch is what in Eventide's H910/H949 Harmonizer models was called *Fine Control*. In particular, this control allowed for fine adjustment around Unison when the Micro mode was selected. But generally, this control affected the general pitch shifting ratio controls. Here, the concept of micropitch is quite different. The selection of the ratios, that has been described in 3.1.2, is not altered. More specifically, this is an addition to the calculated ratios. The control allows for a ± 50 cents adjustment and all the voices are affected in the same mode.

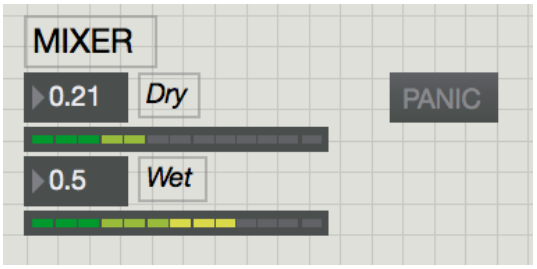

FIGURE 3.7: The *Mixer* section.

3.2.3 Output control stage.

Finally, the interface gives the user the control of the level of harmonised sound (*wet* parameter) and direct signal (*dry* parameter). Both the parameters are given in linear amplitude, that is, they range from 0 (no sound) to 1. No gain in this stage is allowed because the user can directly act both on the input level and on each of harmonisation voices if he wants to raise the overall level. It's also provided a useful *panic* button. If pushed, it stops immediately all the DSP processes and neither input nor output signal can be heard. To restore the correct function of the device, the user has to be start again the DSP engine from the Max *Audio Status* window. In order to prevent undesired sounds when DSP is turning on again, *panic* makes all the input and output levels set to their minimum.

3.3 Issues and future enhancements

Harmonisation process is a very personal one. Numerous treatises on music harmony have furnished many students different ways of harmonic realisation. There has been no definitive formalisation until now. As Piston wrote in [22]:

> True harmonisation, then, means a consideration of the *alternatives* in available chords, the reasoned selection of one of these alternatives, and the tasteful arrangement of the texture of the added pans with due regard for consistency of style. (p. 152)

Surely, the idea of harmonisation including specific musical style codified aspects is not the aim of this work. Generally, a Harmonizer has often been seen as a way of getting a quickly harmonisation and there has been not so much attention on the quality of its realisation, considering not the technical but rather the musical aspect. As shown in 3.1.2, also this device does not pay attention in the choice of arrangement and in the choice of a better alternative chord relative to the context. In fact, what our Harmonizer does is to read a table and get out the selected pitch values. If the *scale mode* is

changing, the selected chord will be different. But, internal to a specific harmonisation mode, there's no alternatives in available chords. This means that, for the same note, the same chord will always be selected. Moreover, there's no attention to any melodic lines as it should be in a classic four-voice writing. The concept of *mutual connectibility* exposed by Piston in his own manual is not considered. In practice, most of the musical aspects and heuristics for an appropriate harmonisation are missing. This is a normal issue regarding many harmonizers, due to the fact these devices should be first very accessible to a vast range of utilisers and are affected by all the computing and technology limitations.
A first attempt to solve this lack of musical considerations can be seen in the development of our *Intellichord* function, as shown above in 3.4. This harmonisation mode does not care about style or any arrangement taste, but focuses on the recognition of the state transitions in order to make a reasoned selection inside a range of several alternative chords.

3.3.1 Probabilistic networks?

In the recent time, the research focused on non-deterministic approaches and on the use of logical and mathematical assumptions in order to obtain more sophisticated harmonic realisations in a manner that can take into account a pseudo human experience. In fact, Artificial Intelligence is not able to capture the musical style of a human composer [23], and all his symbolic representations. This symbols make a particular rule be effectively good to describe a specific musical character, that is the *style*. So, more and more diffuse is the use of probabilistic frameworks or the use of neural systems that allow to create a harmonisation system which "*learns* from examples" [24] and experience. In particular, we'll provide a short description of two of these.
The first uses a data set of chorale harmonisations to train a Hidden Markov Model. This learning task allows to build a model of harmonic processes that can be used to compose novel harmonisations that follow particular aesthetic preferences of a wanted style, due to the clearly understanding of the basic rules of harmonisation [24]. The second uses neural networks both in training task and in the harmonisation process. In particular, it's been designed a learning system based on the Boltzmann Machine which allows to learn a particular harmonisation style [23]. The same system can be used with adjustment and constraints (that is, the harmonic *rules*) to harmonise music via completion.

We can observe how the examples are focused mainly on two characteristics. First, both of the processes are non-deterministic and based on probabilistic networks. This means

that there are not pre-configured tables of chords, as it's our case. The system makes a choice between different combinatoric possibilities, looking for the one that better suits the requested harmonisation. This process depends both on the nature of the melody and on the stochastic nature of the models involved (Boltzmann distribution, or Markov Model for instance). We can speak in some way of a real harmonic *synthesis*. This also makes the system perform different results even if the melody is the same. Secondly, both of the examples allows for machine-learning music harmonisation systems. Given the advance in machine learning, it's desirable to apply its techniques to music harmony so to build a system that learns to harmonise and to distinguish between different musical styles. So, the great advantage of this networks is that they are capable of learning internal representations and are also able to solve difficult combinatoric problems. Doing so, the chord construction and its displacement over four voice is truly a real-time task.

Perhaps, one issue about Boltzmann Machine is that it could stop learning correctly. Learning in general Boltzmann Machine is quite impractical, so it could be made useful to properly constrain the connectivity. This means to insert external constraints in order to let the process efficient enough, whereas the Hidden Markov Model example learns its harmonisation constraints directly from training data. These external constraints are vital to the success of the Boltzmann Machine harmonisation system. Moreover, neural networks are less compute-intensive, but these harmonisation systems retain a pre-programmed knowledge base [24]. They take a large set of rules written by a human, and so could penalise undesirable combinations of notes so that they can be filtered out. Instead, Markov models don't need predetermined tune or implicit control knowledge.

In conclusion, several system have applied probabilistic networks to harmonisation. Using these kind of system allows to perform more efficient inference to generate new harmonisations. It's also possible to let the system itself learn from adequate examples provided by an external user. In this manner, learning processes and techniques takes importance as much as the harmonisation process itself. The choice of right notes to complete a specific chord and the choice of which harmonic formulae the system should use in a specific content should be made by stochastic models. They allow for non-repetitive patterns, even if the harmonised material doesn't change. In a musical view, these characteristics would be a truly improvement for this kind of device. They perhaps require lots of non-musical knowledge, as most of the concepts come from mathematics and statistic. Further information about HMM-based harmonisation model can be found in [25], as in [26] different representation for musical chords that are presented.

3.4 Conclusion

In this chapter, we focused mainly on the musical aspects of this project. First, we described the process that lets this Harmonizer be *intelligent*, that is to recognise every pitch-period harmonic function so to give it a specific scale degree, depending on the selected root key. Then, we described the different typologies of harmonisation that the device provides the user, and how the chords are selected. A more accurate harmonisation model, *Intellichord*, is also shown. Later in the chapter, the user interface was shown in details. All its section were described, and also all the effects and controls. Finally, it's been provided a briefly discussion about possible enhancements so to realise a more complete, ready-to-use and sophisticate device.

At the end of this paper, we can observe that all this work is undoubtedly acceptable. Nevertheless, several improvements and a change for the better are recommended. In particular, the realm of harmonisation system should be enhanced to give it a more flexible behaviour. The Author wishes for a possible work of revision and improvement during the successive years.

Appendix A

Source code

Here are presented, for documentation, the codes of the two realised algorithms, pitchdetect~ and dlshift~. Both of them are included in the header file *dsp.h* and are the core of their own MSP externals.

All this codes are written using C++ programming language.

A.1 AMDF-based pitch detection

```
float amdf_pitch (float* buffer, float* amdf, float* minima,
                  float* minimaPos,int N, float sr, float minF0,
                  float maxF0, float threshold, float& rms, float coeff) {

    // amdf ----------------------------------------
    float maxAmdf = 0;
      int N2 = N / 2;

    for (int n = 0; n < N2; ++n) { // run twice on half buffer
        float diff = 0;
        // contexually compute RMS (only on half buffer)
        rms += (buffer[n] * buffer[n]);
        for (int m = 0; m < N2; ++m) {
                diff += fabs (buffer[m] - buffer[m + n]);
        }
        diff *= pow (n, coeff);
        amdf[n] = diff; // no normalization, not needed
        if (maxAmdf < diff) maxAmdf = diff;
    }
    rms = sqrt (rms / N2);
```

```
    // pitch detection ----------------------------
    int minLag = (1. / maxF0) * sr;
    int maxLag = (1. / minF0) * sr;

    if (rms < threshold) return 0;

    // detect local minima
    int currentMinimum = 0;
    // discard first and last lag (safety)
    for (int i = minLag + 1; i < maxLag - 1; ++i) {
        if (amdf[i] < amdf[i - 1] && amdf[i] < amdf[i + 1]) {
            minimaPos[currentMinimum] = i;
            minima[currentMinimum] = amdf[i];
            ++currentMinimum;
        }
    }

    float min = minima[0];
    int index = 0;
    // location of absolute minimum
    for (unsigned int i = 0; i < currentMinimum; ++i) {
        if (minima[i] < min) {
            min = minima[i];
            index = i;
        }
    }

    // if minimum is > 50% of max energy then is too high
    if (min > maxAmdf * .5) return 0;
    return sr / minimaPos[index];
}
```

Listing A.1: AMDF-based pitch detection

A.2 Delay-line base pitch shifting

```
void pitch_shift (float* sig, float* outbuffer, float* table,
                  float* D1, float* D2, float* env, int dt, int N,
                  int len, float& tablePos, int& wptr, float pf){

    float out;
    float frac, frac2;
    float rptr, rptr2; //reading pointers (set to D1, D2 beginning)
    int rpi, rpi2; //integer writing position indexes
    int next, next2; //successive samples for delay interpolation
    double phasor_value; //interpolated phasor value
    float tableFrac;
    int tableNext;
    int tableFloor; //integer part of the phasor position index
    float x; //phasor-dependent cosine value

    for (int n = 0; n < N; ++n){

        /*calculate interpolated phasor value*/
        tableFloor = floor (tablePos);
        tableFrac = tablePos - (float) tableFloor;
        tableNext = (tableFloor + 1 == len? 0 : tableFloor + 1);
        phasor_value = (1. - tableFrac) * table[(int) tableFloor] +
                               tableFrac * table[tableNext];

#ifdef DEBUG
        post ("%g - %d", phasor_value, dt);
#endif

        /*table position cosine corresponding calculation*/
        x = (1. - tableFrac) * env[(int)tableFloor] +
        tableFrac * env[tableNext];

        /*calculate reading positions
        first index*/
        rptr = wptr - ((float) dt * phasor_value);
        while (rptr < 0) rptr += dt;
        while (rptr >= dt) rptr -= dt;
        rpi = floor(rptr); // position integer index
        frac = rptr - (float) rpi; // fractional part
        next = (rpi == dt ? 0 : rpi + 1);

        /*second index*/
        rptr2 = rptr - ((float) dt / 2.);
        while (rptr2 < 0) rptr2 += dt;
        while (rptr2 >= dt) rptr2 -= dt;
        rpi2 = floor(rptr2);
        frac2 = rptr2 - (float) rpi2;
        next2 = (rpi2 == dt? 0 : rpi2 + 1);

        /*variable delay*/
        out = (frac * D1[next] + (1.0 - frac) * D1[rpi]) * x;
        out += ((frac2 * D2[next2] +
```

```
                (1.0 - frac2) * D2[rpi2])) * (1 - x);
        D1[(int)wptr] = sig[n];
        D2[(int)wptr] = sig[n];
        outbuffer[n] = out;

        /*update position indexes*/
        ++wptr;
        wptr %= dt;
        tablePos += (pf);
        if(tablePos < 0.) tablePos += len;
        if (tablePos >= len) tablePos -= len;
    }
}
```

Listing A.2: Delay-line base pitch transposition

Appendix B

MSP external code

Here, the main methods of the two externals built in Xcode 6, labelled *perform*, are presented. They are excerpts of a longer codes that include all the information about MSP external construction, initialisation and all the methods with which the user can set the object and modify its behaviour.
Generally, everyone of these codes contains a section relative to the class making. In fact, each externals has an internal class that specifies all the member variables used by its own functions and all the methods. For simplicity, the DSP class method is always called *perform*. It's a void function that is called when the audio turns on in Max.
Finally, every class contains two function named as the class itself. One is the initialisation function, the so-called *constructor*. The other is the *destructor*, that is the function acting when the external is removed into the Max patcher. The purpose of this function is to free the memory previously occupied by interal buffers.

All this codes are written using C++ programming language.

B.1 Pitchdetect~

```
// pitchdetect~.cpp
//

#include "dsp.h"
#include "maxcpp6.h"
```

```
const int MAXVSIZE = 4096;
const int MEDIANSIZE = 7;

class PitchDetect : public MspCpp6<PitchDetect> {
public:
    // default signal processing method is called 'perform'
    void perform (double **ins, long numins, double **outs,
                  long numouts, long sampleframes) {
        /*ins and out are array of arrays because
         *there could be more than one inputs or outputs
         *e.g. stereo signal*/
        double* input = ins[0];
        double* output1 = outs[0];
        double* output2 = outs[1];

        for (int i = 0; i < sampleframes; ++i) {
            m_tmp1[i] = input[i];
        }

        /*filtering the signal*/
        filter(m_tmp1, m_z1, sampleframes, m_b0, m_a1);

        /*pitch detection and rms extraction*/
        float rms = 0;
        float f = amdf_pitch(m_tmp1, m_amdf, m_minima, m_minimaPos,
                             sampleframes, m_sr, m_minF0, m_maxF0,
                             m_threshold, rms, m_bias);

        if (f > 0) {
            m_amdf_tmp[m_index] = f;
            for (int i = 0; i < MEDIANSIZE; ++i){
                m_median[i] = m_amdf_tmp[i];
            }

            /*median filter of last three pitch values------*/
            bsort (m_median, MEDIANSIZE);

            /*keep median*/
            m_pitch = m_median[MEDIANSIZE / 2];

            /*falling and octave error controls*/
            if (m_pitch == m_oldpitch * 2) {
                m_pitch /= 2.;
            }
            if (m_pitch == m_oldpitch / 2.) {
                m_pitch *= 2.;
            }

            /*compare with old pitch*/
            float minpitch = m_oldpitch / m_ratio;
            float maxpitch = m_oldpitch * m_ratio;
            if ((m_pitch > maxpitch || m_pitch < minpitch) || m_oldpitch == 0) {
                m_oldpitch = m_pitch;
            }
```

```
            m_index++;
            m_index %= MEDIANSIZE;
        }
        else {
            m_pitch = 0;
        }

        for (int i = 0; i < sampleframes; ++i) {
            output1[i] = m_oldpitch; // replicate f0 to all samples of frame
            output2[i] = rms;
        }
    }

        void dsp(t_object * dsp64, short *count,
                 double samplerate, long maxvectorsize, long flags) {
        // specify a perform method manually:
        m_sr = samplerate;
        filtcoeff();
        REGISTER_PERFORM(PitchDetect, perform);
        post ("sr = %g", samplerate);
        post ("vector size = %d", maxvectorsize);
    }
}
```

Listing B.1: The pitchdetect~ *perform* function

B.2 Dlshift~

```
/*
 *Dlshift~.cpp
 *pitch transposition based on
 *continue delay modulation
 *
 */

#include "dsp.h"
#include "maxcpp6.h"

const int MAXVSIZE = 4096;

class DLShift : public MspCpp6<DLShift> {
public:

    void perform (double **ins, long numins,
                  double **outs, long numouts, long sampleframes) {
        double* input = ins[0];
        double* output = outs[0];

        for (int i = 0; i < sampleframes; ++i) {
            m_tmp1[i] = input[i];
```

```
        }

        filter(m_tmp1, m_z1, sampleframes, m_b0, m_a1);
        pitch_shift(m_tmp1, m_tmp2, m_table, m_del1,
                    m_del2, m_env, m_dt, sampleframes,
                    m_len, m_tablePos, m_wptr, m_pfreq);

        for (int i = 0; i < sampleframes; ++i) {
            output[i] = m_tmp2[i];
        }
    }

        void dsp(t_object * dsp64, short *count,
                 double samplerate, long maxvectorsize, long flags) {
        // specify a perform method manually:
        m_sr = samplerate;
        m_dt = (int) (m_dtime * (m_sr/1000.));

        REGISTER_PERFORM(DLShift, perform);
        post ("sr = %g", samplerate);
        post ("vector size = %d", maxvectorsize);
        post("dsp dt = %d", m_dt);
    }

protected:

C74_EXPORT int main(void) {
    // create a class with the given name:
    DLShift::makeMaxClass("dlshift~");
    REGISTER_METHOD(DLShift, bang);
    REGISTER_METHOD(DLShift, clear);
    REGISTER_METHOD_FLOAT(DLShift, ratio);
}
```

Listing B.2: The dlshift~ *perform* function

Bibliography

[1] Victor Lazzarini and Richard Boulanger. *The Audio Programming Book*. MIT Press, 2010.

[2] Udo Zölzer. *Digital Audio Effects, 2nd Edition*. Wiley, 2011.

[3] J. Laroche and M. Dolson. Phase-vocoder pitch-shifting, April 15 2003. URL `http://www.google.com/patents/US6549884`. US Patent 6,549,884.

[4] Curtis Roads. *The Computer Music Tutorial*. MIT Press, 1996.

[5] Sascha Disch and Udo Zölzer. Modulation and delay line based digital audio effects. *2nd Workshop on Digital Audio Effects DAFx, Proceedings of*, 1999.

[6] Eric Moulines and Jean Laroche. Non-parametric techniques for pitch-scale and time-scale modification of speech. *Speech communication*, 16(2):175–205, 1995.

[7] Mark Dolson. The phase vocoder: A tutorial. *Computer Music Journal*, Vol. 10 (No. 4):pp. 14–27, Winter, 1986.

[8] James L Flanagan and RM Golden. Phase vocoder. *Bell System Technical Journal*, 45(9):1493–1509, 1966.

[9] Jean Laroche and Mark Dolson. New phase-vocoder techniques for pitch-shifting, harmonizing and other exotic effects. *Applications of Signal Processing to Audio and Acoustics, 1999 IEEE Workshop on*, pages 91–94, 1999.

[10] K Abdullah-Al-Mamun, F Sarker, and G Muhammad. A high resolution pitch detection algorithm based on amdf and acf. *Journal of Scientific Research*, 1(3): 508–515, 2009.

[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):1917–1930, 2002.

[12] Lawrence Rabiner, Md Cheng, Aaron E. Rosenberg, and C. McGonegal. A comparative performance study of several pitch detection algorithms. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 24(5):399–418, 1976.

[13] Miller Puckette, Theodore Apel, et al. Real-time audio analysis tools for pd and msp. 1998.

[14] Li Tan and Montri Karnjanadecha. Pitch detection algorithm: autocorrelation method and amdf. 2003.

[15] M Ross, H. Shaffer, Andrew Cohen, Richard Freudberg, and H. Manley. Average magnitude difference function pitch extractor. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 22(5):353–362, 1974.

[16] Ghulam Muhammad. Noise robust pitch detection based on extended amdf. *Signal Processing and Information Technology, 2008. ISSPIT 2008. IEEE International Symposium on*, pages 133–138, 2008.

[17] Wenyao Zhang, Gang Xu, and Yuguo Wang. Pitch estimation based on circular amdf. *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*, 1:341–344, 2002.

[18] Goangshiuan S Ying, Leah H Jamieson, and Carl D Mitchell. A probabilistic approach to amdf pitch detection. *The Journal of the Acoustical Society of America*, 95(5):2817–2820, 1994.

[19] Huan Zhao and Wenjie Gan. A new pitch estimation method based on amdf. *Journal of Multimedia*, 8(5):618–625, October, 2013.

[20] Nicolas Devillard. Fast median search: an ansi c implementation. *ndevilla AT free DOT fr*, 1998.

[21] Alessandro Cipriani and Maurizio Giri. *Musica Elettronica e Sound Design, volume 1*. ConTempoNet, 2009.

[22] Walter Piston. *Harmony 4th Ed. revised and expanded by Mark Devoto*. London: Victor Gollancz LTD, 1978.

[23] M. I. Bellgard and C. P. Tsang. Harmonizing music the boltzmann way. *Connection Science*, 6:281–297, 1994.

[24] Moray Allan and Christopher KI Williams. Harmonising chorales by probabilistic inference. *Advances in neural information processing systems*, 17:25–32, 2005.

[25] Yushi Ueda, Yuuki Uchiyama, Takuya Nishimoto, Nobutaka Ono, and Shigeki Sagayama. Hmm-based approach for automatic chord detection using refined acoustic features. *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, pages 5518–5521, 2010.

[26] Jean-François Paiement, Douglas Eck, and Samy Bengio. Probabilistic melodic harmonization. *Advances in Artificial Intelligence*, pages 218–229, 2006.

[27] Julius O Smith. *Physical audio signal processing*. W3K Publishing, 2010.

[28] Dennis Gabor. Acoustical quanta and the theory of hearing. *Nature*, 159(4044): 591–594, 1947.

[29] Adrian von dem Knesebeck, Pooya Ziraksaz, and Udo Zölzer. High quality time-domain pitch shifting using psola and transient preservation. *Audio Engineering Society Convention 129*, November, 2010.

[30] Norbert Schnell, Geoffroy Peeters, Serge Lemouton, Philippe Manoury, and Xavier Rodet. Synthesizing a choir in real-time using pitch synchronous overlap add (psola). *Proceedings of the International Computer Music Conference*, pages 102–108, 2000.

[31] John William Gordon and John Strawn. An introduction to the phase vocoder. *CCRMA, Department of Music, Stanford University*, 1987.

[32] Curtis Roads. *Microsound*. MIT Press, 2002.

CONSERVATORIO DI MUSICA DI VICENZA
A. PEDROLLO

Abstract

Graim
Scuola di musica elettronica e nuove tecnologie

Diploma Accademico Ordinamentale di I livello

Real-time Intelligent Harmonizer Based on AMDF Pitch Estimation

by Mattia PATERNA (Matr. 4903)

Viene definita *armonia* l'organizzazione delle altezze di un suono polifonico all'interno di un intervallo temporale. L'altezza di un suono armonico (inteso come suono scomponibile nelle sue componenti tramite la serie di Fourier) può essere trasportato. Questo avviene tramite la tecnica del *pitch shifting*, ovvero la modifica del contenuto frequenziale di un segnale ottenuta moltiplicandolo per una *ratio* specifica. In generale, un suono viene *armonizzato* se ad esso vengono aggiunte differenti versioni trasportate al fine di generare un accordo. Si intende armonizzazione intelligente, o *smart harmony*, un processo che tiene conto sia del contesto armonico sia del fondamentale del segnale. A tal scopo, è necessario l'utilizzo di un algoritmo per la stima della frequenza fondamentale del segnale.