Dissertation

Masters in Computer Engineering - Mobile Computing

Evolution of Classifiers for Pitch Estimation of Piano Music using Cartesian Genetic Programming

Tiago João Leite Inácio

Leiria, September of 2016

Dissertation

Masters in Computer Engineering - Mobile Computing

Evolution of Classifiers for Pitch Estimation of Piano Music using Cartesian Genetic Programming

Tiago João Leite Inácio

Master Dissertation under the supervision of Doctor Gustavo Reis, Professor at Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Leiria and supervision of Doctor Carlos Grilo, Professor at Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Leiria.

Leiria, September of 2016

Acknowledgement

First of all, I would like to thank to my supervisors, Professor Gustavo Reis and Professor Carlos Grilo, for their support, advices, encouragement and guidance. The effort and dedication that you have put on this work were fundamental.

I would also like to thank to Rolando Miragaia for his input and expertise on signal processing. Your contribution was definitely very important.

I am also deeply grateful to my mother, Fernanda Leite, and my sister, Joana Inácio, and to all my family and friends, who always believed in me.

I cannot end without a special thanks to my girlfriend Bruna for all her love and support.

Thank you.

Resumo

A estimativa de notas musicais, também conhecida como determinação da frequência fundamental (F0), tem sido um tópico bastante popular por muitos anos, e é ainda bastante investigada hoje em dia. O objectivo da estimativa de notas musicais é descobrir a nota ou frequência fundamental de uma gravação digital de um discurso ou música. Desempenha um papel fundamental na transcrição de música, pois permite saber que notas estão a ser tocadas a cada instante.

A estimativa de notas musicais de sons gravados com instrumentos reais é uma tarefa bastante complicada. Cada instrumento tem diferentes características físicas, o que faz com que tenham diferentes características espectrais. Além disso, as condições de gravação podem variar de estúdio para estúdio, e eventuais ruídos de fundo têm de ser considerados.

Esta dissertação apresenta uma nova abordagem para o problema da estimativa de notas musicais, utilizando Programação Genética Cartesiana (PGC). Aproveitamos as vantagens dos algoritmos evolucionários, particularmente da PGC, para explorar e desenvolver funções matemáticas complexas que actuam como classificadores. Esses classificadores são usados para identificar notas de piano num sinal de áudio.

Para nos ajudar com a codificação do problema, foi construída uma *toolbox* de PGC, flexível e genérica o suficiente para codificar diferentes tipos de programas. A *toolbox* é bastante fácil de usar. O algoritmo evolucionário presente na *toolbox* é conhecido como $1 + \lambda$, onde o valor de λ é configurável. A probabilidade de mutação, o número de execuções e gerações são também configuráveis. A representação cartesiana da PGC pode tomar várias formas. Além disso, é capaz de codificar parâmetros para as funções do *function-set*, tem um sistema útil de *callbacks* e está preparada para lidar com diferentes funções de *fitness*: maximização de $f(x)$ e minimização de $f(x)$.

Foram treinados sessenta e um classificadores, correspondentes a sessenta e uma notas de piano. Foram usados conjuntos de sinais aúdio para treinar cada um dos classificadores, em que metade dos sinais tinham uma frequência fundamental igual à do classificador (sinais positivos), e outra metade com frequência fundamental diferente

dos classificadores (sinais negativos). A função de fitness utilizada foi a *F-measure*. Sinais com a mesma nota do classificador, e que foram identificados correctamente pelo classificador, contam como verdadeiros positivos. Sinais com a mesma nota do classificador, e que não foram identificados correctamente pelo classificador, contam como falsos negativos. Sinais com uma nota diferente do classificador, e que foram identificados correctamente pelo classificador, contam como falsos positivos. Sinais com uma nota diferente do classificador, e que não foram identificados correctamente pelo classificador, contam como verdadeiros negativos.
Numa primeira abordagem, foram evoluídos classificadores para a identificação de sinais artificiais, criados por funções matemáticas: onda sinusoidal, onda triangular e onda quadrada. O *function-set* é basicamente composto por operações de filtragem sobre vetores e por operações aritméticas com constantes e vetores. Todos os classificadores identificaram corretamente os sinais positivos e não identificaram os sinais negativos. De seguida, procedeu-se para o treino de classificadores com gravações de áudio reais. Para testar os classificadores, foram escolhidos sinais de áudio diferentes dos utilizados durante a fase de treino. Os resultados obtidos foram muito promissores, mas podiam ser melhorados. Fizemos pequenas alterações na nossa abordagem e o número de falsos positivos reduziu 33%, comparativamente com a primeira abordagem. De seguida, os classificadores evoluídos foram aplicados a sinais de áudio polifónicos. Os resultados indicam que a técnica utilizada é um bom ponto de partida para abordar o problema de estimativa de notas musicais.

Palavras-chave: *estimativa de notas musicais*, *programação genética cartesiana*, *algoritmos evolucionários*, *toolbox de programação genética cartesiana*, *determinação da frequência fundamental*

Abstract

Pitch Estimation, also known as Fundamental Frequency (F0) estimation, has been a popular research topic for many years, and is still investigated nowadays. The goal of Pitch Estimation is to find the pitch or fundamental frequency of a digital recording of a speech or musical notes. It plays an important role, because it is the key to identify which notes are being played and at what time.

Pitch Estimation of real instruments is a very hard task to address. Each instrument has its own physical characteristics, which reflects in different spectral characteristics. Furthermore, the recording conditions can vary from studio to studio and background noises must be considered.

This dissertation presents a novel approach to the problem of Pitch Estimation, using Cartesian Genetic Programming (CGP). We take advantage of evolutionary algorithms, in particular CGP, to explore and evolve complex mathematical functions that act as classifiers. These classifiers are used to identify piano notes pitches in an audio signal. To help us with the codification of the problem, we built a highly flexible CGP Toolbox, generic enough to encode different kind of programs. The encoded evolutionary algorithm is the one known as $1 + \lambda$, and we can choose the value for λ. The toolbox is very simple to use. Settings such as the mutation probability, number of runs and generations are configurable. The cartesian representation of CGP can take multiple forms and it is able to encode function parameters. It is prepared to handle with different type of fitness functions: minimization of $f(x)$ and maximization of $f(x)$ and has a useful system of callbacks.

We trained 61 classifiers corresponding to 61 piano notes. A training set of audio signals was used for each of the classifiers: half were signals with the same pitch as the classifier (true positive signals) and the other half were signals with different pitches (true negative signals). F-measure was used for the fitness function. Signals with the same pitch of the classifier that were correctly identified by the classifier, count as a true positives. Signals with the same pitch of the classifier that were not correctly identified by the classifier, count as a false negatives. Signals with different pitch of the

classifier that were not identified by the classifier, count as a true negatives. Signals with different pitch of the classifier that were identified by the classifier, count as a false positives.
Our first approach was to evolve classifiers for identifying artifical signals, created by mathematical functions: sine, sawtooth and square waves. Our function set is basically composed by filtering operations on vectors and by arithmetic operations with constants and vectors. All the classifiers correctly identified true positive signals and did not identify true negative signals. We then moved to real audio recordings.
For testing the classifiers, we picked different audio signals from the ones used during the training phase. For a first approach, the obtained results were very promising, but could be improved. We have made slight changes to our approach and the number of false positives reduced 33%, compared to the first approach.
We then applied the evolved classifiers to polyphonic audio signals, and the results indicate that our approach is a good starting point for addressing the problem of Pitch Estimation.

Keywords: *pitch estimation*, *cartesian genetic programming*, *evolutionary algorithm*, *cartesian genetic programming toolbox*, *fundamental frequency estimation*

List of Figures

List of Tables

List of Algorithms

Index

Chapter 1

Introduction

Music transcription could be defined as the analysis of an acoustic signal, in order to find the pitch, onset time, duration and source of each sound (see Figure 1.1). Automatic Music Transcription (AMT) is making this process automatic.

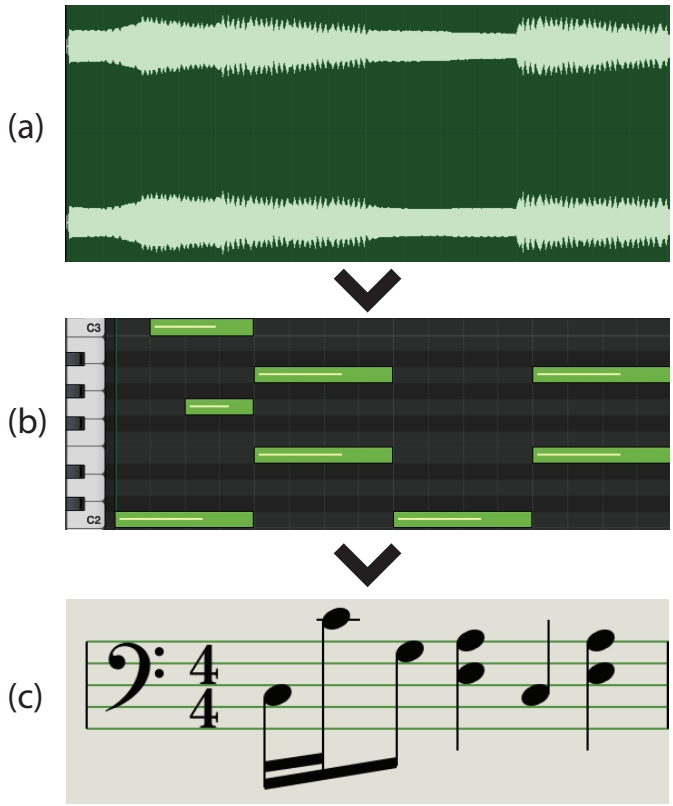


Figure 1.1: General overview of a music transcription system. (a) - Record the sound into the computer. (b) - Apply the transcription technique to obtain a piano-roll representation of the sound. (c) - Convert the piano-roll representation to a partiture.

AMT is a general problem, which comprises several problems of its own, and can be decomposed in: pitch estimation, note onset/offset detection, loudness estimation and quantization, instrument recognition, extraction of rythmic information, and time quantization (Benetos et al., 2013). Pitch Estimation, also known as Fundamental Frequency (F0) estimation, is a sub-problem of AMT; it has been a popular research topic for many years and still is investigated nowadays. The goal of Pitch Estimation is to find the pitch or fundamental frequency of a digital recording of a speech or musical note. It plays an important role, because it is the key to identify which notes are being played and at what time.
Signals where several sounds are played simultaneously are called polyphonic signals, in contrast to monophonic signals, where at most one note is present at a time. Conversely, Single-Pitch Estimation identifies pitches on monophonic signals and Multi-Pitch Estimation identifies multiple pitches in polyphonic signals. Pitch Estimation of real instruments is a very hard task to address. Each instrument has its own physical characteristics, which reflects in different spectral characteristics. Furthermore, the recording conditions can varie from studio to studio and background noise must be considered.

1.1 Objectives and Scope of the Thesis

To the best of our knowledge, there are no Cartesian Genetic Programming (CGP) approaches for addressing the Pitch Estimation problem. This thesis presents a novel approach to the problem of Pitch Estimation, using CGP. We take advantage of the evolutionary algorithms, in particular CGP, to search for complex mathematical functions that act as classifiers. These classifiers are used to identify piano notes pitches in an audio signal. Given an audio recording of a C3 piano note, the classifier for that note, should recognize that a C3 is present in that sound. There will be one classifier for each piano note.

1.2 Thesis Contributions

The main contributions contained within this dissertation are summarized below:

- A Cartesian Genetic Programming Toolbox for Matlab was built and it is freely

available. This toolbox is generic enough to encode different problems with different requirements.

- A novel approach for detecting pitches using CGP is presented. The results show the feasibility of the approach and validate the evolution of classifiers using CGP for Pitch Estimation.

- We wrote an article, where we presented our approach and first results on Pitch Estimation, that was accepted on the 2016 IEEE Symposium Series on Computational Intelligence (IEEE SSCI 2016).

1.3 Outline of the Thesis

This dissertation is organized as follows.

Chapter 2 This chapter starts presenting a brief explanation of several terminology and concepts, from waves and sampling to audio signal processing. Single-Pitch Estimation approaches are presented and the problem complexity for Multi-Pitch Estimation is also discussed.

Chapter 3 In this chapter a literature review of previous studies on multiple-F0 estimation is presented.

Chapter 4 This chapter introduces CGP and the algorithm that it uses. An example of CGP applied to Image Processing is also presented.

Chapter 5 In this chapter we describe the Cartesian Genetic Programming Toolbox that we developed. We show how this toolbox can encode multiple programs, and how configurable it is. An example of the application of the toolbox to a symbolic regression problem is presented.

Chapter 6 In this chapter our approach of applying Cartesian Genetic Programming to the problem of Pitch Estimation is presented. Our work was divided in multiple steps: application of classifiers to signals artificially created by mathematical models; application of classifiers to real audio recordings of monophonic piano

signals; application of classifiers to polyphonic audio signals. The experiments and results for each step is shown and discussed. We also applied those classifiers to polyphonic audio recordings and present the results.

Chapter 7 Finally, this chapter presents our main conclusions. We also present a few suggestions to future work of applying classifiers evolved by Cartesian Genetic Programming to Pitch Estimation.

Chapter 2

Terminology and Concepts

Relevant terminology and concepts about several background topics are presented in this chapter. A brief introduction to sounds, its characteristics and signal processing is presented.

2.1 Waves and Sound

Sound is the propagation of disturbances in a medium, regardless of whether the substance of the medium is gaseous, liquid or solid, some of which can be detected by the human ear. Those disturbances are called sound waves, and propagate by repetitive variations of compression (high pressure) and rarefaction (low pressure) of the medium. The most important properties of sound waves are: wavelength, amplitude and frequency. The wavelength is the distance between any point in the wave and the equivalent point in the next cycle. The amplitude is the strength of a wave signal. The more amplitude the wave signal has, the more loud the volume will sound. Frequency is the number of cycles per second and it is measured in hertz (Hz). Thus, frequency is the number of times the wavelength occurs in one second. The frequency range of the human ear is:

$$20Hz \leq f \leq 20kHz. \tag{2.1}$$

This means that humans can hear vibrations occurring between 20 and 20 000 times per second. Any sound with a frequency below 20 Hz as infrasound and above than 20 kHz is known as ultrasound. The decibel (dB) is a logarithmic unit used to describe the intensity of sound. Our ear has a logarithmic sensitivity, thus the decibel scale is commonly used to measure sound levels.

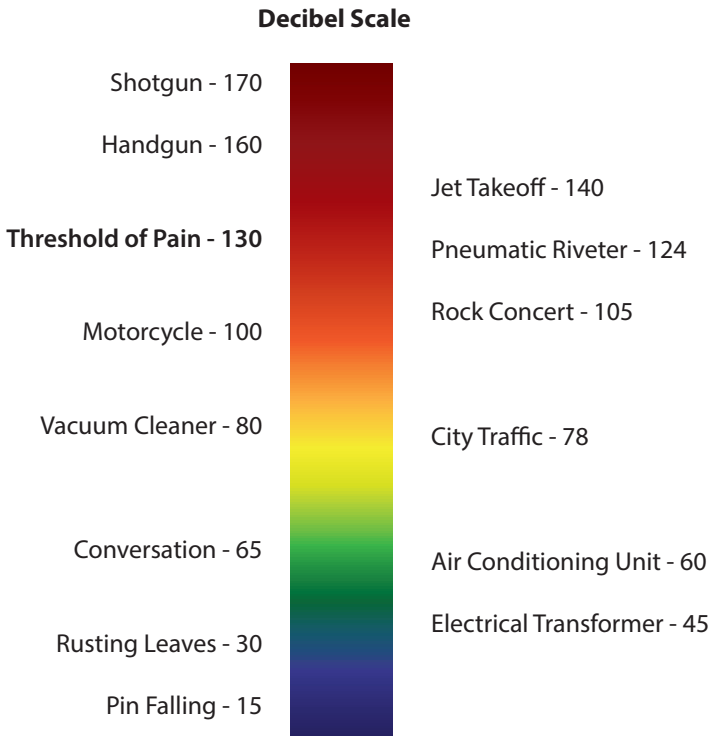


Figure 2.1: Sound intensity measured by the Decibel (dB) unit.

2.2 Digital Audio Recording

The process of recording and playing sound from a digital device, such as a computer, is a very complex task. A brief description of both processes will be introduced, where one of the key elements are the transducers.

2.2.1 AD/DA Converters

Transducers are devices that convert energy from one form to another. The sound of an instrument reaches an acoustic-to-electric transducer (e.g. microphone) and the vibrations are converted into an electric signal which is then amplified. An analog-to-digital converter (ADC) converts the electric signal into digital data which is stored on a hard-drive, CD, or other data storage device (see Figure 2.2).

Figure 2.2: Analog-Digital converter.

To play the recorded sound, the data previously stored is transformed back to an analog signal with a digital-to-analog converter (DAC) (see Figure 2.3). The analog signal is amplified and converted to sound by an electroacoustic transducer (e.g. loudspeaker).

Figure 2.3: Digital-Analog converter.

Microphones convert acoustical energy into electrical energy, sound waves into audio signals. There are different types of microphones based on how they convert the energy, but basically, they all have a diaphragm which vibrates accordingly to the vibrations in the air (sound waves). Those vibrations are then converted into electrical current, which is then amplified.
In order to convert the electrical signal into digital data, a sound card or digital mixer is used. These systems incorporate an AD/DA converter. The analog-to-digital converter samples the input signal periodically (sampling frequency) based on its voltage level.

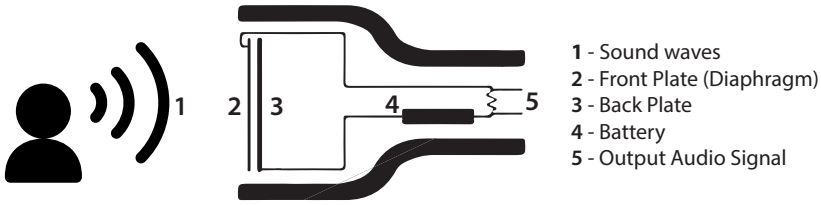


Figure 2.4: Condenser microphone overview.

The voltage level is continuous in time, which means some information is lost, during the sampling process (see Figure 2.5). The digital-to-analog is the opposite, it converts the numbers back into electrical voltage.

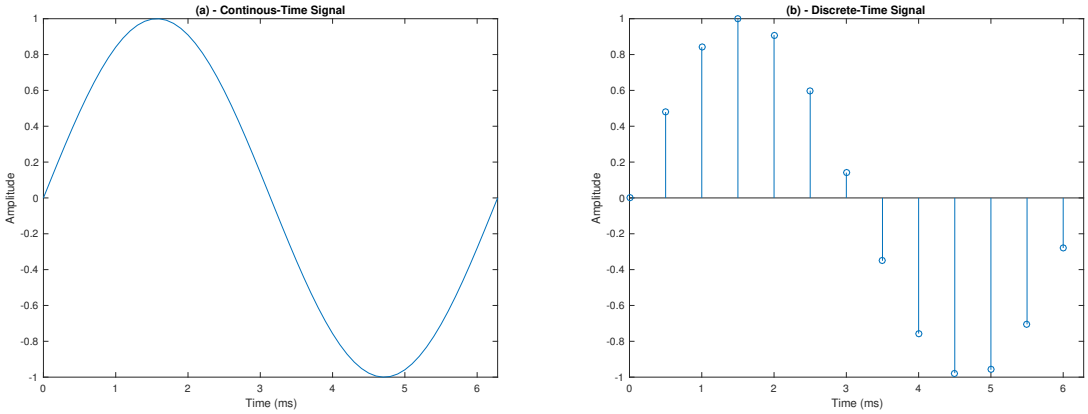


Figure 2.5: Signal sampling. When sampling a continous-time signal, some information is lost, only a few points in time are recorded. (a) - Continuous signal in time. (b) - Sampled signal.

2.2.2 Nyquist Theorem

The Nyquist theorem, also known as the sampling theorem, is a principle that is followed in the digitization of analog signals. For a faithful reproduction of the signal, the analog waveform must be sampled frequently. The number of samples per second is called sampling frequency or sampling rate. The most simple case of an analog signal is a sine wave or a sinusoid. These kind of signals have all the energy concentrated

at one frequency. Most signals consist of different components at various frequencies. Bandwidth is defined as a range within a band of frequencies. For analog signals, bandwith is expressed in Hertz. The highest frequency component determines the bandwith of an analog signal. The Nyquist theorem specifies that in order for all the relevant information in an analog signal to be preserved in the sampling process, the sampling rate must be at least $2 \times F_{max}$, or twice the highest analog frequency component. Mathematically, the theorem can be expressed as:

$$F_s \geq 2 \times F_{max}, \tag{2.2}$$

where F_s is the sampling frequency, and F_{max} is the highest frequency contained in the signal. If a sound wave has a single sine wave at a frequency of 1Hz, the minimum sampling frequency ditacted by the Nyquist theorem is 2Hz. Thus, if this sound wave is sampled at a frequency bigger or equal than 2Hz, there will be more than enough samples on its digitalized version for the human ear to perceive the sound as if it was analogic, and no significant signal information is lost. However, if the signal is sampled at a frequency below than 2Hz, aliasing occurs, because there are not enough samples to capture the significant variations of the signal through time, information will be lost and the result will lead to a different signal being perceived. This is the main reason why industry adopted 44.1 kHz for the CD sampling rate: to cover all the frequencies from the 20 Hz to 20 kHz, which is the highest frequency the human hear can perceive:

$$44.1kHz > 2 \times 20kHz \tag{2.3}$$

2.2.3 Quantization

The sampling process converts a continous-time signal into a discrete-time signal. Each sample or slice, present in the discrete-time signal, contains an amplitude value. The stored amplitude value is the closest to the real one, from a set of possible values. The number of values or levels are expressed in "bits", the binary system. A two bit resolution sampling could store four different values (see Figure 2.6). The most common resolutions are 8-bits, which stores 256 levels, 16-bits, which stores 65536 levels and 32-bits, which stores 4.3 billion levels. The industry adopted 16-bits sampling resolution

for the CD to reproduce a high quality sound.

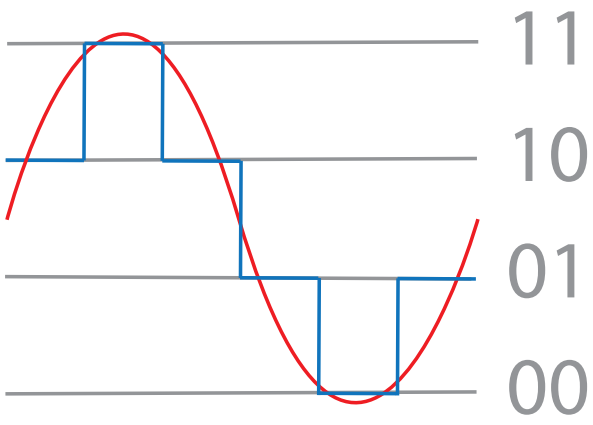


Figure 2.6: 2 bit resolution sampling.

2.3 Music Characteristics

Music is an art resulting from the combination of sounds, that uses rythm, melody, harmony and silence. The Concise Oxford English Dictionary (2002) defines music as:

"the art of combining vocal or instrumental sounds (or both) to produce beauty of form, harmony, and expression of emotion".

As stated before, sounds are vibrations that travel through the air or another medium. Music sounds have four fundamental characteristics: dynamics, duration, timbre and pitch.

Dynamics

The dynamics of a sound is the perception of the amplitude of the sound wave. This is physically related to the amount of energy that is transported by a sound wave, when the particles vibrate in the medium. It is more commonly referred to as the volume or loudness of a sound. The most common dynamic indications in music, which are also referenced by their Italian words, are very soft (pianissimo), soft (piano), loud (forte) and very loud (fortissimo).

Duration

Each sound occurs during a certain period of time. The duration of a sound is the elapsed time between its start time (onset) and end time (offset).

Timbre

In music, timbre is the quality that distinguishes different types of sounds, e.g. a piano from a guitar. The timbre of a sound is determined by the shape of the sound wave.

Pitch

Is the tonal height of the sound. It is related to how low or how high a note sounds. It is a subjective attribute of sound, that is closely related to the frequency, which is an objective physical property. Pitch is an auditory sensation that maps the vibrations of a sound wave to a tone in a musical scale.

2.4 Signals

Signals can be defined as anything that carries information. Examples of signals are gestures, images, human voice, sounds, etc. Technically, signals can be represented as a function of time, space or other observation variable that transfers information. Audio signals carry a representation of a sound, typically an electrical voltage (see Figure 2.7).

2.4.1 Types of Signals

Signals can be classified in a variety of ways, according to their own characteristics. A brief description will be presented of the characteristics which we find more suitable for the undestanding of this dissertation.

Continuous-Time Signals

A signal is continuous in time if the independent variable (t) is continuous in $f(t)$ and will always have a value. Any analog signal is countinous by nature and analog audio signals are not an exception (see Figure 2.5-a).

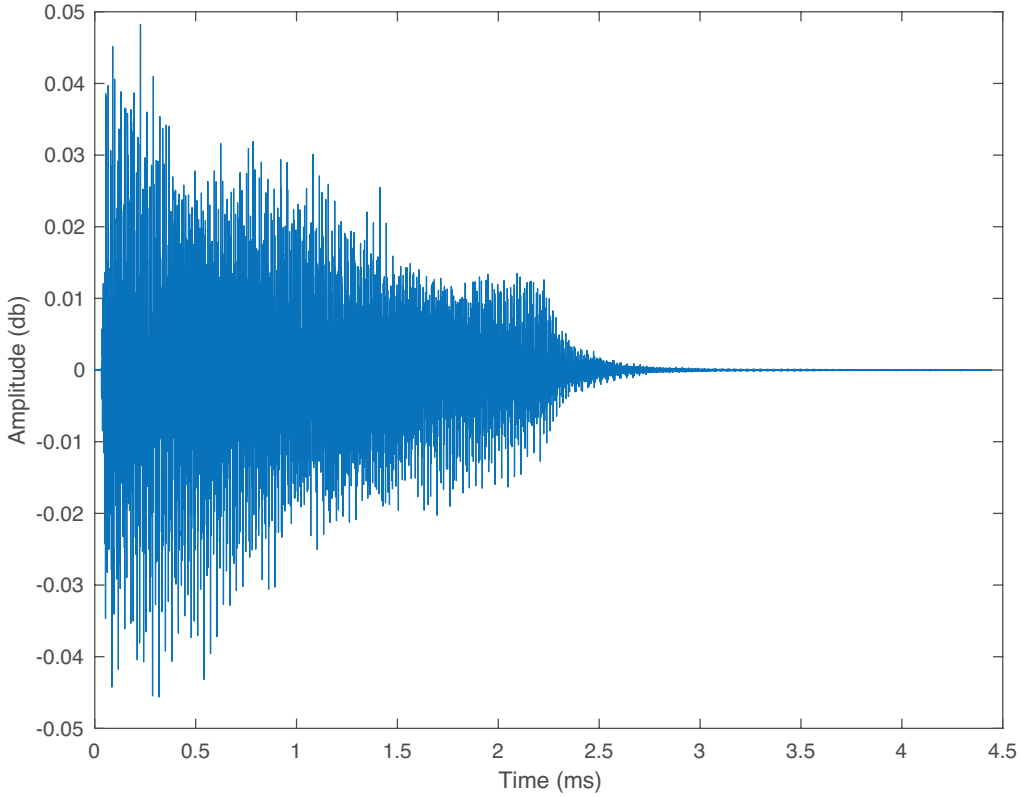


Figure 2.7: Sound signal as a function of time.

Discrete-Time Signals

When an analog signal is sampled and converted to bits by an ADC, the signal is represented in small fractions in time. Instead of having a full representation where every instant has a value, we only have a collection of values, depending on the sampling rate and the length of the signal. The signal is represented as $x[n]$, and the independent variable (n) takes on only discrete values (see Figure 2.5-b).

Periodic Signals

A signal is periodic if it repeats itself exactly after some period of time. Some examples of periodic signals are sine waves, square waves, triangle waves, and so on. In continous-time, a signal is periodic if M is an integer and there exists any value T such that:

$$f(t) = f(t + MT). \tag{2.4}$$

The period of the signal is the smallest value of T for which the above relation holds true: the wavelength.
For discrete-time, a signal is periodic if it repeats itself after some period, with one key difference: the period must be an integer. A discrete time signal $x[n]$ is said to be periodic if, both M and N are positive integer values such that:

$$x[n] = x[n + MN]. \tag{2.5}$$

The period of the signal is the smallest value of N for which the above relation holds true.

Quasi-Periodic Signals

Some discrete signals that are almost periodic and can be represented by

$$x[n] \approx x[n + MN] \tag{2.6}$$

are called quasi-periodic signals. These signals, when compared to periodic signals, might not have identical points across periods, but will have very similar points. The general waveshape is nearly the same as if it were a periodic signal.

2.4.2 Signal Processing

Signal processing operates in some fashion on a signal in order to extract useful information. It has many application fields, such as audio signal processing, speech signal processing, image processing, wireless communications and so forth. According to the type of signal, signal processing can be divided into five categories: analog signal processing, continuous-time signal processing, discrete-time signal processing, digital signal processing and nonlinear signal processing.

Digital signal processing (DSP) is the manipulation of signals using a general-purpose computer or digital circuits, in order to analyze, filter, create or compress digitalized signals. DSP applications include, among others, digital image processing,

audio signal processing, sonar and radar signal processing, biomedical signal processing and seismic data processing. It is applied to digital signals and has also been applied during our work. DSP makes use of several transforms, being the most relevant to our work the Fourier Transform, which will be introduced in the next section.

2.4.3 Fourier Analysis

Regardless of the source of the sound wave, the particles in the air move back and forth at a given frequency. As stated before, the **period** of the sound wave is the wavelength, and it is also the inverse of the frequency: a sound wave with high frequency will have a smaller period, whereas a sound wave with low frequency will have a larger period (see Figure 2.8).

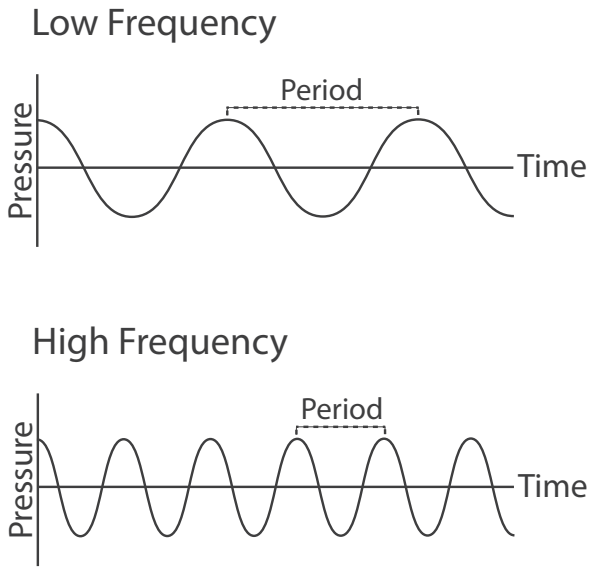


Figure 2.8: Low frequency and high frequency representation.

Jean-Baptiste Joseph Fourier had the insight to see that any continuous function could be represented as an infinite sum of oscillating functions. Fourier analysis is the process of decomposing any periodic signal into the sum of a possibly infinite set of sine and cosine functions or complex exponentials. Fourier synthesis is the process of converting those sines and cosines back into a periodic function (see Figure 2.9). A sine wave or sinusoid is a mathematical curve that describes a smooth repetitive oscillation. A sinusoid is represented as a function of time f(t):

$$y(t) = A \times sin(2\pi ft + \varphi) = A \times sin(\varpi t + \varphi), \tag{2.7}$$

where A is the amplitude of the wave, f is the number os oscillations per second, $2\pi f = \varpi$ is the angular frequency, and φ is the phase.

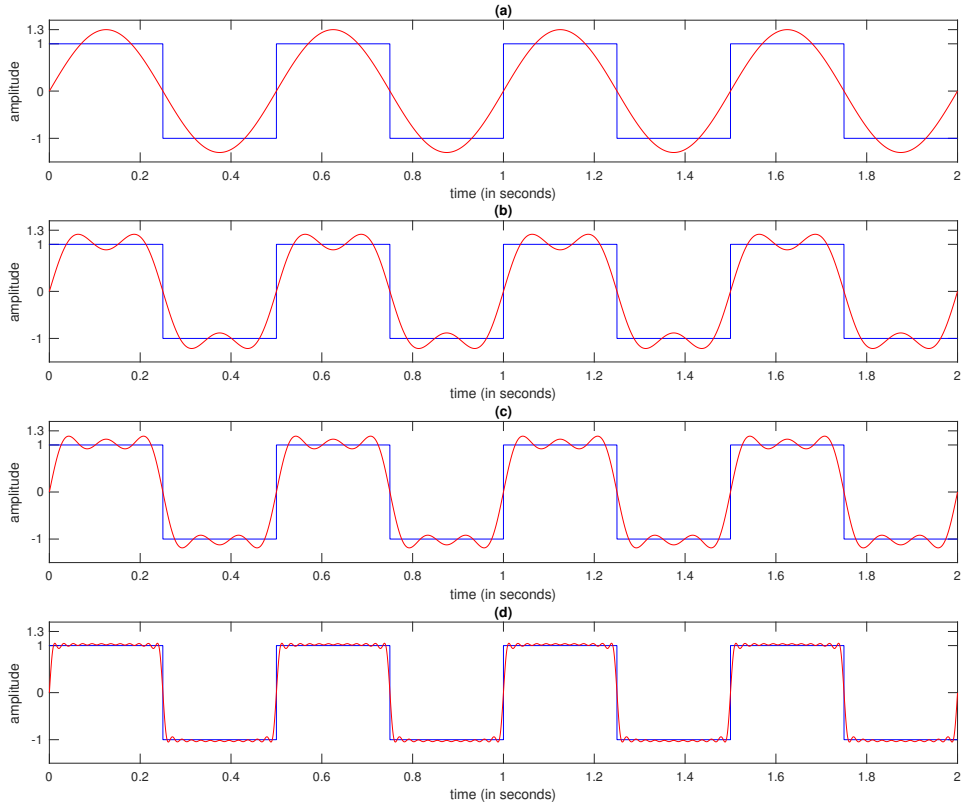


Figure 2.9: Figure (a) Approximates the square wave by a sine function with the same F0 (2Hz). Figure (b) represents a sum of two sine waves oscillating at the F0 and an integer multiple of the F0 (called partial). Figure (c) represents a sum of three sine waves oscillating at the F0 and integer multiples of the F0. Figure (d) represents almost a clear decomposition of the square wave into multiple sine functions.

As stated before, a periodic signal can be expressed as a sum of sines or cosines functions. In particular, a square wave can be approximated by an odd-multiple frequency sine waves at diminishing amplitude. In Figure 2.9-a, the red curve is described by the

Equation 6.8, and the frequency at which it oscillates is equal to the F0 of the square wave underneath:

$$y(t) = 1.3sin(2\pi 2t). \tag{2.8}$$

As we can see in Figure 2.9-b, by summing two sine functions (see Equation 2.9), we can start to see that the result is an approximation of the square wave.

$$y(t) = 1.3sin(2\pi 2t) + 0.42sin(2\pi 6t). \tag{2.9}$$

In Figure 2.9-c, we add a third partial (see Equation 2.10), which makes it even closer to the square wave.

$$y(t) = 1.3sin(2\pi 2t) + 0.42sin(2\pi 6t) + 0.24sin(2\pi 10t). \tag{2.10}$$

Those multiple sine functions are the frequency components (partials) that make up the composed signal. The frequency of each component or partial is an integer multiple of the fundamental frequency. In Figure 2.9-d, the summation of the sine functions is almost at infinite, reproducing the original signal without almost no difference to the human ear. Each wave oscillates at a specific frequency. The lowest frequency of all waves present in a note is defined as the **Fundamental Frequency** (F0). Fundamental frequency is the inverse of the fundamental period or P0 and corresponds to the perceived pitch. All the waves are called **partials**. Frequencies of the partials are mostly limited to integer multiples of the lowest frequency (Fundamental Frequency - F0), forming the **harmonic series**. A **harmonic** is a partial that is exactly an integer multiple of F0. F0 is also considered a harmonic partial, because it is one times itself. Except the fundamental frequency, all the partials that make up the harmonic series are called overtones (over F0).

Fourier Series

The Fourier series is used to represent a periodic signal by a discrete sum of complex exponentials. If a continuous function f(t) is periodic with period T, then it may be approximated by a linear combination of harmonically related exponentials. The

Fourier series representation of a periodic signal f(t) is given by the following synthesis equation:

$$\tilde{x} = \sum_{k=-\infty}^{\infty} a_k e^{jqw_0 t}, k \in \mathbb{Z}, \tag{2.11}$$

where $w_0 = 2\pi F0 = \frac{2\pi}{T_0}$. This equation can be used as a synthesizer to generate a signal as a weighted combination of fundamental frequencies. The corresponding analysis equation for the Fourier series is written as:

$$\tilde{a_k} = \frac{1}{T0} \int_{T0} \tilde{x}(t) e^{jqw_0 t} dt. \tag{2.12}$$

The value ak carries the amplitude and the phase of the frequency content of the signal at kw_0 Hz. The complex exponentials that form a periodic signal occur only at integer multiples (harmonics) of the fundamental frequency w_0. The synthesis Equation 2.11 can be rearrenged into:

$$\tilde{x}(t) = a_0 + \sum_{k=1}^{+\infty} (a_k e^{jkw_0 t} + a_{-k} e^{-jkw_0 t}). \tag{2.13}$$

If we take into account that $a_k^* = a_{-k}$, furthermore, Equation 2.13 can be expressed as:

$$\tilde{x}(t) = a_0 + \sum_{k=1}^{+\infty} (a_k e^{jkw_0 t} + a_k^* e^{-jkw_0 t}). \tag{2.14}$$

The following trigonometric Equation is used to express the Fourier Series of periodic signals and is obtained by reference a_k in its polar form as $a_k = \frac{A_k}{2} e^{j\phi_k}$:

$$\tilde{x}(t) = a_0 + \sum_{k=1}^{+\infty} A_k cos(kw_0 t + \phi_k). \tag{2.15}$$

A harmonic sound is a periodic signal and can be represented by Equation 2.15. Quasi-periodic signals do not have frequencies at multiple locations of its fundamental frequency. Those frequencies are simply referred to as **partials**, instead of harmonic partials, since their frequency is not an exact multiple of the corresponding F0. For an approximation of this type of signals, a finite number of harmonic components H is used:

$$\tilde{x}(t) \approx a_0 + \sum_{k=1}^{H} A_k cos(kw_0 t + \phi_k). \tag{2.16}$$

Fourier Transform - FT

The Fourier transform (FT) is used to represent a periodic signal (function) by a countinous superposition or integral of complex exponentials. It decomposes a signal as a function of time into multiple frequencies resulting in a complex-valued function of frequency. The absolute value represents the frequency band over a range of frequencies present in the original signal and the complex value represents the phase offset of the sinusoid in that frequency. The FT is generally used in signal processing, specially in time-frequency analysis. For a periodic signal, with infinte length, it is defined as:

$$FT_{\tilde{x}}(f) = \tilde{X}(f) = \int_{-\infty}^{+\infty} \tilde{x}(t) e^{-j2\pi ft} dt. \tag{2.17}$$

This equation results in the frequency domain representation of the original signal.

Discrete Fourier Transform - DFT

The Discrete Fourier Transform (DFT) is the equivalent of the continuous Fourier Transform for sampled signals. The DFT is used to perform Fourier analysis in many practical applications, such as digital signal processing. DFT can be achieved in a continous sampled signal by applying the following equation:

$$DFT_{\tilde{x}}[k] = \tilde{X}[k] = \sum_{n=-\infty}^{+\infty} \tilde{x}[n] e^{-j2\pi kn}, \tag{2.18}$$

where k is the spectral bin corresponding to each frequency. Since the computation of

an infinite continous-sampled signal is not efficiently possible, one must restrict the size of the signal. Having N as number of samples, the DFT for finite signals is represented as:

$$DFT_{\tilde{x}}[k] = \tilde{X}[k] = \sum_{n=0}^{N-1} \tilde{x}[n] e^{-j\frac{2\pi}{N}kn}, k = 0, \cdots, N-1. \tag{2.19}$$

The **magnitude spectrum** (see Figure 2.10) is given by $|\tilde{X}[k]|$.
The problem with DFT is that it requires $2N^2$ real multiplications and additions, which makes it really hard to apply in real-time signal processing.

Fast Fourier Transform - FFT

The Fast Fourier Transform (FFT) is an algorithm which optimizes the DFT and was invented by Gauss in 1805, and later re-discovered by Cooley and Tukey in 1965. The FFT applies to signals that have a structured number of samples, such as a power of 2. The first step of the FFT is to decompose an N point time-domain signal into N time domain-signals, where each signal is composed of a single point. Then, the frequency spectra of each N time-domain signals are computed. The last step is to aggregate and synthesize all the N spectra into one single frequency spectrum. Through this method, the FFT only requires $Nlog_2N$ operations, which allows its application in real-time signal processing.

2.4.4 Power Spectral Density

Power Spectral Density function (PSD) represents the strength of variations as a function of frequency and is computed from the squared magnitude value of the DFT of a signal. The unit of PSD is energy per frequency. The PSD is obtained by applying $|\tilde{X}[k]|^2$, assuming that $\tilde{X}[k]$ is the DFT of a signal $x[n]$. By integrating PSD within a specific frequency range, we obtain the energy for those frequencies. The result of the DSP application is the **power spectrum** (see Figure 2.11).

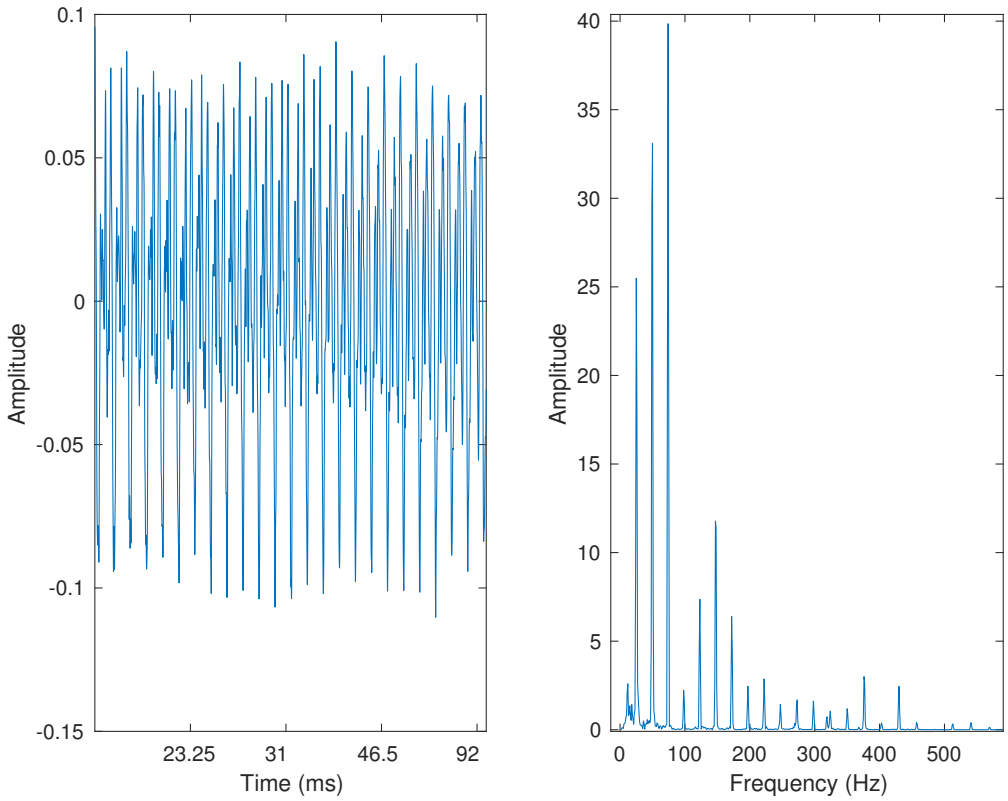


Figure 2.10: The left image shows the signal $x(t)$ in time, whereas the right image shows the magnitude spectrum or absolute value of the DFT - $|\tilde{X}[k]|$.

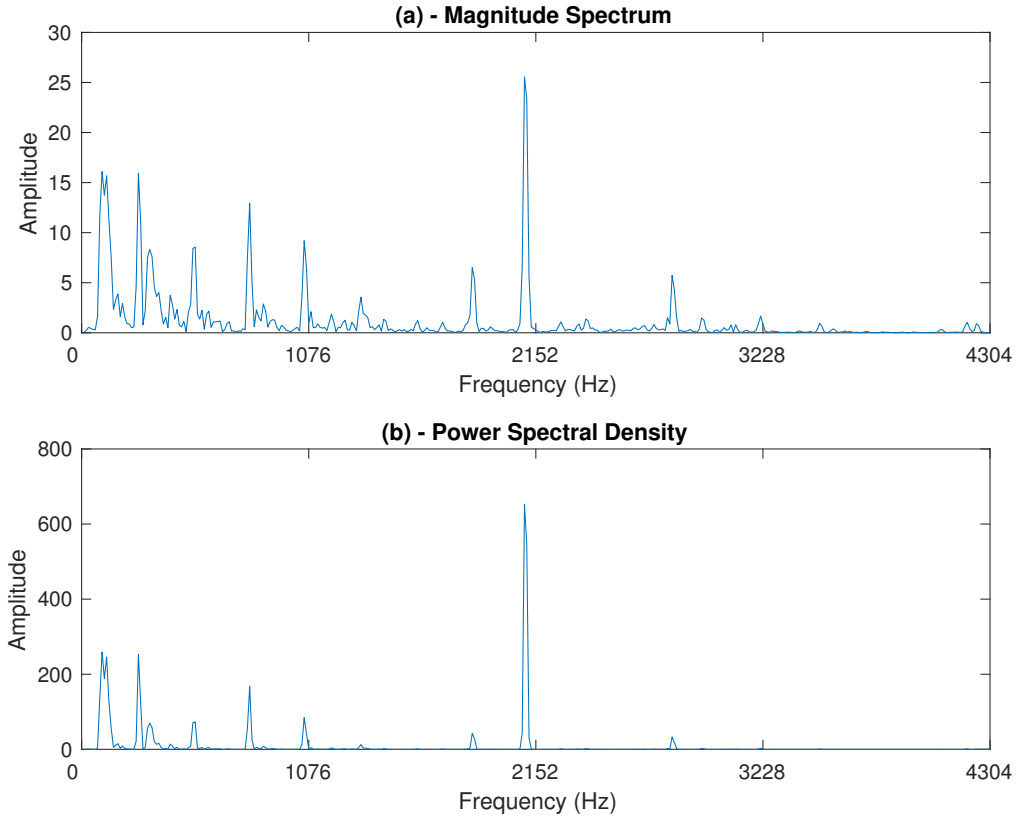


Figure 2.11: This figure compares the magnitude spectrum given by $|\tilde{X}[k]|$, and the Power Spectral Density given by $|\tilde{X}[k]|^2$.

2.4.5 Spectral Leakage

The DFT assumes that the input repeats over and over again (periodic). If a sinewave oscilates at 10 Hz, we would have to calculate the number of samples to work on, to give us the exact period. If the DFT is applied to a signal that is not periodic (the number of samples do not finish on a whole number of periods), discontinuity will occur, and the frequency representation of the signal will be distorted (see Figure 2.12). An effect known as **spectral leakage** occurs when the energy of a frequency bin is leaked or spread across adjacent frequency bins. This effect could interfere with the overall shape of the magnitude spectrum.

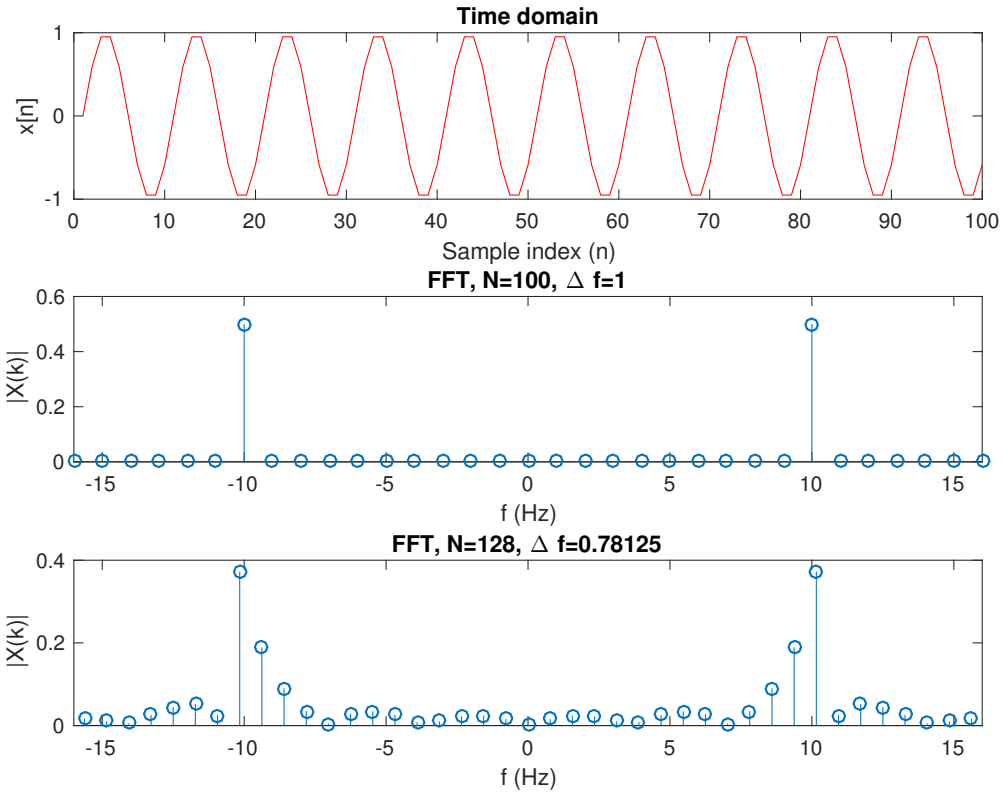


Figure 2.12: The top chart displays a 10hz sinewave, sampled at a 100hz sampling rate. The middle chart displays the DFT applied to the 100 samples of the signal, a whole number of periods. The bottom chart displays a spectral leakage, because the DFT was applied to more samples than the period.

2.4.6 Windowing

In order to minimize the spectral leakage effect, the samples in the frame can be multiplied by a smooth window shape. This will smooth the abrupt edges caused by the truncation of a signal into a single time window. **Windowing** is the process where the input time signal is multiplied by a windowing function (see Figure 2.13). This process is often used for spectral analysis, filter design, and beamforming. When we want to apply the FFT to a signal, we have to choose which interval do we want to analyse. There are multiple types of windows: triangular, Parzen, Hanning, Hamming, Blackman, and so on.

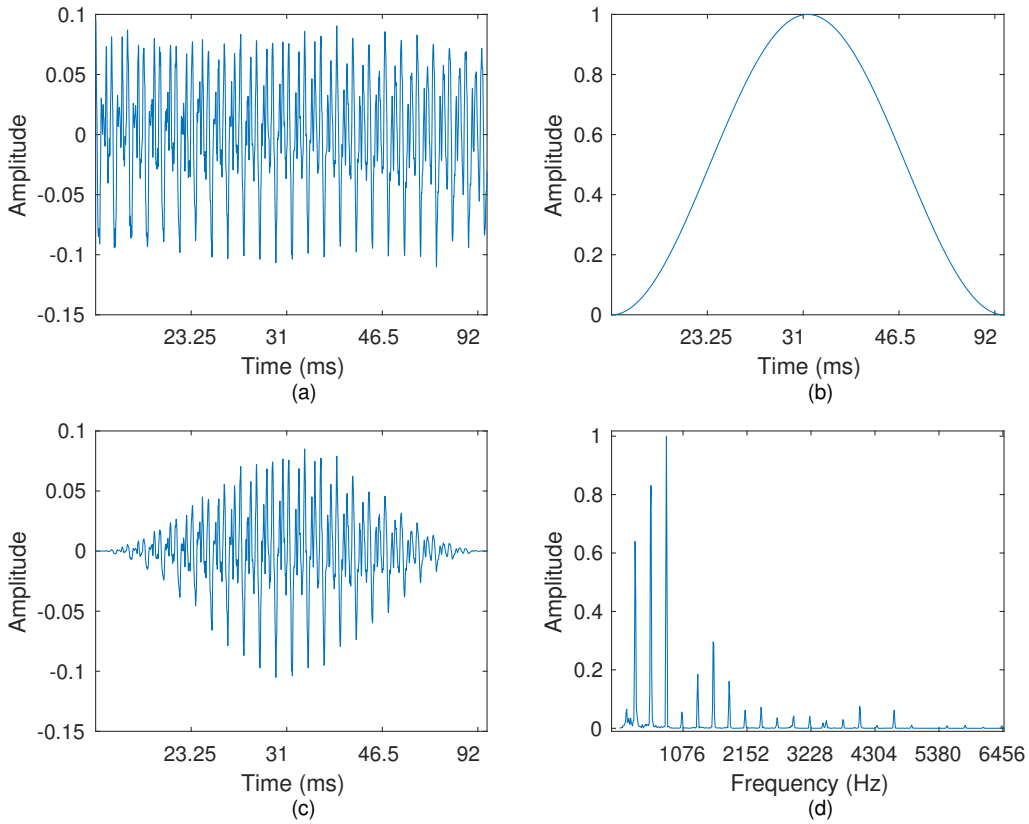


Figure 2.13: Preprocessing process: (a) input time signal piano note, (b) Hanning window, (c) resulting windowed signal, (c) frequency domain signal.

2.4.7 Relation between the signal's properties

It is important to distinguish both time resolution and frequency resolution and the implications that both have in signal analysis. The number of samples of a signal varies with the sampling rate and the seconds of information that we have. Having F_s as the sampling rate and t as the number of seconds recorded, we can calculate n, the total number of samples recorded:

$$n = F_s \times t \tag{2.20}$$

Since the DFT is discrete (Discrete Fourier Transform), this means that the frequency

is also discrete, being k the corresponding Frequency Bin. This way, by analyzing Equation 2.19, we can see that the frequency resolution is related to the number of samples (N). Thus, **frequency resolution** is the distance in Hz between two adjacent frequency bins in the DFT and can be expressed as:

$$\Delta F = \frac{F_s}{N}, \tag{2.21}$$

where N is the DFT window size.
Time resolution, on the other hand, is the minimum time of a signal in seconds that one could extract information from.

$$\Delta t = \frac{N}{F_s}. \tag{2.22}$$

If we have a 2 second music signal recorded at a sampling rate of 22050 samples per second, we would end up with $22050 \times 2 = 44100$ samples. Let us consider that the DFT has a window size of 4096 samples. The frequency resolution is $\frac{22050}{4096}$, which means that each bin, will correspond, approximately, to 5,38 Hz. The time resolution is $\frac{4096}{22050}$, which means that we could only detect musical notes with duration equal or greater then 0.19 seconds. The window size of the DFT must be properly set according to the type of information that we want to focus on, either time information or frequency information. The greater the DFT size is, the shorter the frequency resolution is, which makes it easier to analyse low frequencies, but it will increase the time resolution, which makes harder to analyse shorter periods of time.

2.4.8 Missing Fundamentals

A missing fundamental occurs when we perceive a fundamental frequency in a sound, that does not have that frequency. The missing fundamental or phantom fundamental, may be created by the overtones present in the signal such that, together, suggest a frequency that does not exist. The brain perceives a pitch by the fundamental frequency or the periodicity of an audio signal. If a signal has two pure tones at 1000 Hz and 1300 Hz, we might perceive a missing fundamental by hearing those two frequencies and an additional one, created by the difference of the two signals. We would end up

with one additional pitch correspondent to the frequency of 300 Hz, because of the form of the waveform (see Figure 2.14).

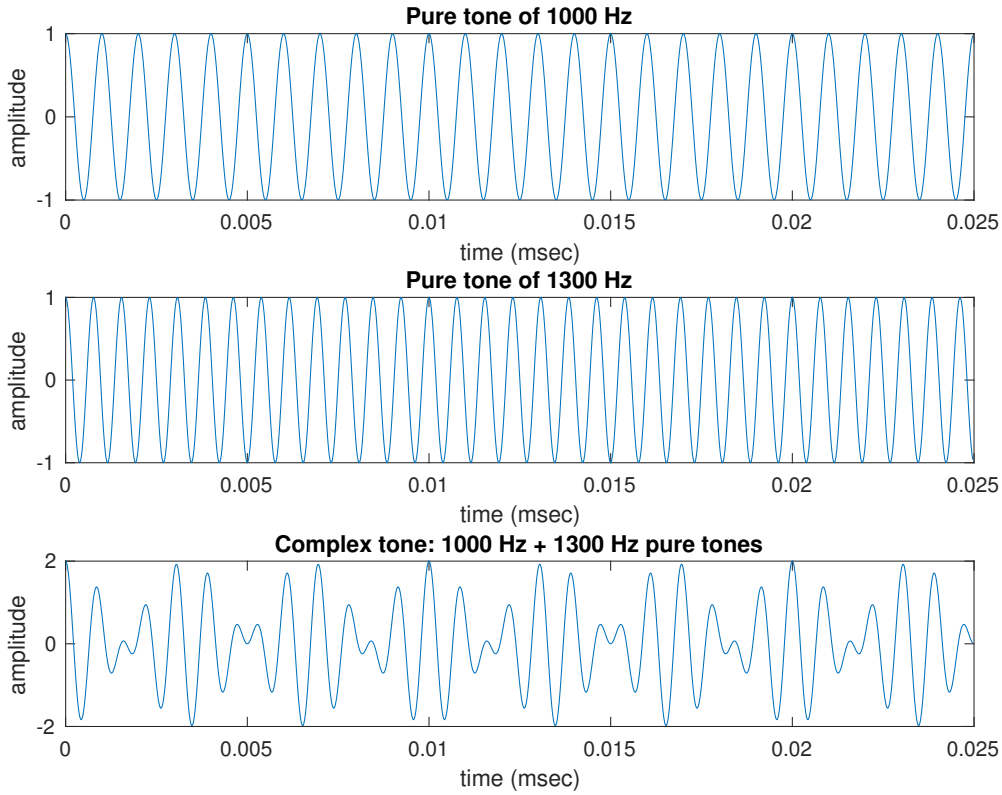


Figure 2.14: Complex tone with a phantom frequency at 300Hz.

2.4.9 Pitch

Each musical note is composed by an harmonic series: its fundamental frequency (F0) and the corresponding partials. What the human ear perceives as Pitch is the fundamental frequency (i.e.: lowest harmonic partial) of each harmonic series or musical note. A sound wave that vibrates at a specific frequency, will be mapped internally by our brain to a certain pitch. Each pitch is related to a musical note (see Figure 2.15). If, for instance, we hear a sound wave vibrating at, approximately, 262 Hz, our brain will map internally that sound wave to the pitch C4 (middle C in a 88 keys keyboard), because the fundamental frequency of C4 is, approximately, 262 Hz. Since pitch is

an auditory sensation, if the previous sound wave vibrates at 260 Hz or 262 Hz, our brain probably would still map it to a C4 pitch. One octave higher (pitch C5) has the fundamental frequency of, approximately, 523 Hz. Comparing this two signals, pitch C5 will sound 'higher' than pitch C4. The distance perceived between C2 and C3 (66 Hz) is the same as the distance between C3 and C4 (131 Hz), since the human ear perceives pitch in a logarithmic scale: for each octave, the frequency doubles. Note that A2, A3, A4 and A5 have 110Hz, 220Hz, 440Hz and 880Hz respectively.

NOTE	OCTAVE								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
C	16	33	65	131	262	523	1047	2093	4186
C#	17	35	69	139	278	554	1109	2218	4435
D	18	37	73	147	294	587	1175	2349	4699
D#	20	39	78	156	311	622	1245	2489	4978
E	21	41	82	165	330	659	1319	2637	5274
F	22	44	87	175	349	699	1397	2794	5588
F#	23	46	93	185	370	740	1475	2960	5920
G	25	49	98	196	392	784	1568	3136	6272
G#	26	52	104	208	415	831	1661	3322	6645
A	28	55	110	220	440	880	1760	3520	7040
A#	29	58	117	233	466	932	1865	3729	7459
B	31	62	124	247	494	988	1976	3951	7902

Figure 2.15: Pitch to frequency relationship. C4 has the frequency of 262 Hz.

2.4.10 Pitch vs Fundamental Frequency

Pitch detection and F0 detection are two different processes that are easily confused. Several algorithms have been developed to address the single-pitch and multi-pitch estimation problems. In the overall, what all try to achieve is the pitch transcription,

what notes are being played. Fundamental frequency estimation approaches, try to identify exactly the frequency of the signals. After knowing which F0 or F0s are present in the signal, they are mapped to pitches or notes. Pitch estimation approaches, try to identify the pitch or pitches present in the signal, without the need of knowing exactly what is the exact fundamental frequency. In this dissertation, the problem which we are trying to solve is the pitch estimation, not the fundamental frequency estimation.

2.5 Single-Pitch Estimation

Signals where several sounds are played simultaneously are called *polyphonic* signals, in contrast to *monophonic* signals, where at most one note is present at a time (Klapuri and Davy, 2006). Yeh (2008) states that, without loss of generality, a monophonic signal can be expressed as a sum of a quasi-periodic part $\tilde{x}[n]$ and the residual $z[n]$:

$$x[n] = \tilde{x}[n] + z[n] \approx \sum_{h=1}^{H} A_h \cos(h\omega_0 n + \phi_h) + z[n]. \tag{2.23}$$

The goal is to extract the periodicity part of $x[n]$, and not to minimize the residual $z[n]$. The most common errors are harmonically related to the correct F0: **subharmonic errors** and **super-harmonic errors**. Subharmonic errors are errors in which the results are *unit fractions* of the correct F0 and super-harmonic errors are errors in which the results are *multiples* of the correct F0. Single-F0 estimation algorithms can be classified as time domain approaches or spectral domain approaches. Temporal domain methods try to find the fundamental period, as opposed to frequency-domain methods which rely on the spectral analysis.

2.5.1 Spectral-location Approaches

Time domain methods look for a similar repetitive waveform in $x[t]$ through pattern matching between $x[t]$ and a delayed version of $x[t]$. Pattern matching in time domain can be carried out through multiplication or subtraction between patterns.

2.5.1.1 Autocorrelation

The **autocorrelation** function (ACF) allows to measure the similarity between a signal and delayed versions of itself at different points in time. It corresponds to the cross-correlation of a signal with itself for a given **lag** or delay. Mathematically, the autocorrelation function can be calculated as the sum of the product between a signal $x[n]$ of finite duration L and its delayed version $x[n+\tau]$, for each lag τ:

$$ACF[\tau] = \frac{1}{L} \sum_{n=0}^{L-\tau-1} x[n]x[n+\tau]. \tag{2.24}$$

For quasi-peridic signals, correlation will be higher when τ equals the period or a multiple of the period. Nonetheless, this technique is sensitive to resonance in music signals.

2.5.1.2 Magnitude difference

Ross et al. (1974) evaluate the distance between two patterns by comparing the dissimilarity of $x[n]$ and $x[n+\tau]$. This method is called the **Average Magnitude Difference Function** (AMDF). It is used often for real time applications as it involves less computation. Analytically, it is represented by:

$$AMDF[\tau] = \frac{1}{L-\tau} \sum_{n=0}^{L-\tau-1} |x[n] - x[n+\tau]|. \tag{2.25}$$

For quasi-periodic signals, the result of the AMDF is particularly small at delays corresponding to the period or integer multiples of the period. The AMDF is not very accurate when the signal has background noise. Ghulam (2011) extended this technique to address this issue. A similar technique called **Squared Difference Function** (SDF) measures the dissimilarity by the *squared difference*:

$$SDF[\tau] = \frac{1}{L-\tau} \sum_{n=0}^{L-\tau-1} (x[n] - x[n+\tau])^2. \tag{2.26}$$

de Cheveigné and Kawahara (2002) adapted this function for the YIN algorithm, by

normalizing SDF with its average over shorter-lag values. It is commonly addressed as **Cumulative Mean Normalized Difference Function** and avoids super-harmonic errors. Both methods are related to the autocorrelation function. Hess (1983) demonstrated that both those methods are error prone when submited to intensity variations, noise and low-frequency spurious signals.

2.5.1.3 Cepstrum

The **cepstrum** is the result of taking the Fourier Transform (FT) of the logarithm of the power spectrum of a signal. This results in a complex cepstrum, a real cepstrum, a power cepstrum and a phase cepstrum. This technique is useful to measure the periodicity between peaks in the frequency domain. The real cepstrum or the power cepstrum is calculated by applying the following equation:

$$c(\tau) = IDFT\{\log|DFT(x[n])|)\}. \tag{2.27}$$

Schroeder, in 1962, proposed the application of the power cepstrum for F0 estimation based on the first cepstral analysis paper on echoes resulting from earthquakes and bomb explosions. Noll (1967) proposed, later, a short-time power cepstrum analysis for pitch determination of human speech. The spectral envelope information if given by the lower-quefrency components in the cepstrum. The period candidates correspond to the sharp cepstral peaks components.
Figure 2.16 shows three time-domain salience functions applied to a baritone sax signal of $T0 = 2.3ms$.

2.5.2 Spectral-interval Approaches

In spectral domain approaches, F0 estimation is performed by extracting the periodicity from the spectrum, after applying a Fourier Transform. The resulting spectrum contains the spectral information of the harmonic, including its partials at almost integer multiples of the fundamental frequency. One approach of the spectral domain techniques is to measure the space between dominant peaks and assume it as the F0 of the signal. Another approach is to extract the F0 based on a function of hypothetical partials. Based on this assumptions, Yeh (2008) states that fundamental frequency can

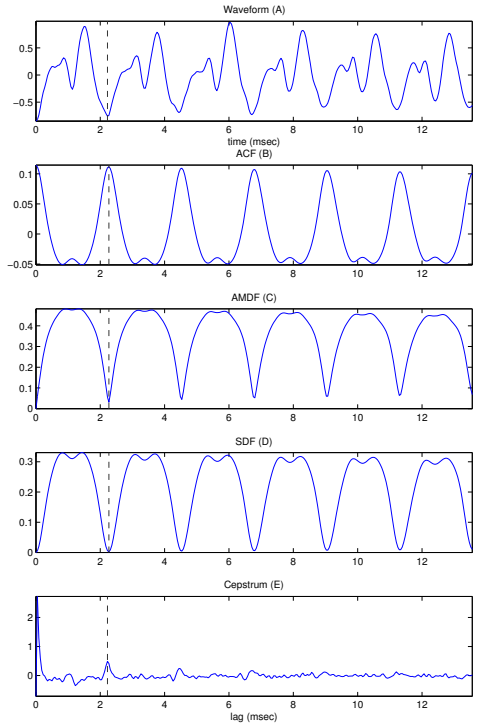


Figure 2.16: (A) signal waveform; (B) autocorrelation function; (C) average magnitude difference function; (D) squared difference function; and (E) cepstrum. Figure taken from Reis (2012), page 26, with permission.

also be defined as the greatest common divisor of the frequencies of all the harmonics.

2.5.2.1 Spectral Autocorrelation

Lahat et al. (1987) showed that since the autocorrelation function searches for repetitive patterns in the time domain, it can also be applied to the spectral domain. The periodicity is obtained by pattern matching between the spectrum and its shifted versions. The ACF function applied to the magnitude spectrum is calculated as:

$$ACFS(m) = \frac{2}{K-2m} \sum_{k=0}^{\frac{k}{2}-m-1} |X[k]||X[k+m]|, \tag{2.28}$$

where $X[k]$ is the spectrum and $X[k+m]$ are its shifted versions. When the shift

m is equal to F0, the ACFS should result in the maximal spectral autocorrelation coefficient. The product between the spectrum and the shifted spectrum is attenuated when the shift m is not equal to F0 or multiples of F0, since the partial peaks are not aligned.

2.5.2.2 Harmonic Matching

Harmonic matching or pattern matching makes use of harmonic spectral patterns to match the observed spectrum. These harmonic spectral patterns can either be a specific spectral model or a **harmonic comb** without specifying the amplitudes of the harmonics. A harmonic comb is a series of spectral pulses with equal spacing defined by a F0 hypothesis. Specific spectral models are often used in multi-pitch signals, whereas harmonic comb is often used on single-pitch estimation. In the works of (Martin, 1982) and (Brown, 1992), a F0 hypothesis can be evaluated based on the correlation between the harmonic comb and the observed spectrum. In Goldstein (1973) and Duifhuis and Willems (1973) a F0 hypothesis is evaluated based on the minimization of the distance between the frequencies of the harmonics and the frequencies of the matched peaks.

2.6 Multi-Pitch Estimation

Multi-pitch estimation algorithms are used for short-time signals that can have more than 1 harmonic source at the same time. Yeh (2008) stated that those signals can be expressed as a sum of harmonic sources $Y_m[n]$ plus a residual $z[n]$, where M is the number of harmonic sources:

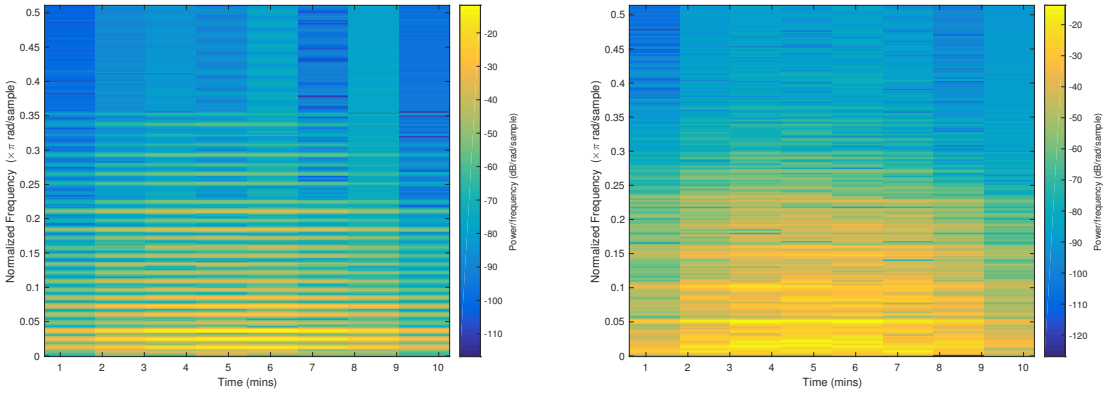
$$y[n] = \sum_{m=1}^{M} Y_m[n] + z[n], M > 0. \tag{2.29}$$

The goal of multiple-F0 estimation algorithms is to infer the number of sources and the related F0s. The residual $z[n]$ is not related to the sinusoids but can be explained by background noise, spurious components or inharmonic partials. Equation 2.30 represents this model by the Fourier Series.

$$y[n] = \sum_{m=1}^{M} \{ \sum_{k=1}^{\infty} A_{m,k} \cos(k\omega_m n + \phi_{m,k}) \}. \tag{2.30}$$

The complexity of polyphonic music signals is far superior to monophonic music signals.

Figure 2.17: Comparison between two spectrograms of monophonic and polyphonic signals.



(a) Spectrogram of a monophonic signal, recorded from a piano.

(b) Spectrogram of a polyphonic signal, recorded from a piano.

Figure 2.17a shows the representation of a spectrogram of a monophonic signal, recorded from a piano and Figure 2.17b is the representation of a spectrogram of a polyphonic signal with 4 harmonic sources, recorded from a piano. As we can see, the spectrogram of a polyphonic signal has more frequency components than a monophonic signal. In polyphonic music we need to infer the number of harmonic sources, whereas in monophonic music there is no such need, because we only deal with one harmonic source. Extracting the correct multiple F0s from a music piece is very difficult, due to the overlapping partials, transients, reverberation and the different spectral characteristics of the musical instruments.

2.6.1 Overlapping Partials

For polyphonic signals, different harmonic sources may overlap or interfere with one another, in time and in frequency. Different sources, in polyphonic signals, with fundamental frequencies F_a and F_b are harmonically related when they can be represented by Equation 2.31.

$$F_a = \frac{m}{n} F_b, \quad n, m \in \mathbb{N}. \tag{2.31}$$

As demonstraded by Klapuri (1998), every n^{th} partial of the source a overlaps every m^{th} partial of source b. This frequently happens when the sources are harmonically related to each other, since it could result in partial colisions. One of the issues when dealing with multi-pitch signals is that most of the musical notes are harmonically related, which results in a high probability of partial overlapping. Another issue is that when fundamental frequencies of two notes are multiples of each other, the partials of the higher note may overlap completely with those of the lower note (Yeh, 2008). The frequencies, amplitudes and phases of the overlapping partials of harmonic sources are thus disturbed. Parsons (1976) addressed the problem of separating the voice of a vocalist speech, by trying to detect the overlapping components, based on three tests: spectral peak symmetry, distance and well-behaved phase. This technique is restricted to two voices and relied on the sinusoidality of stationary sinusoids and is not suitable for modulated sinusoids. As highlighted by several authors, it still remains very difficult to decompose the overlapping partials into their original sources, even if the number of concurrent sources is known beforehand (H. Viste and G. Evangelista (2002);Virtanen (2003);Every and Szymanski (2004);Yeh and Roebel (2009)).

2.6.2 Spectral Characteristics

Since polyphonic music signals could have multiple instruments playing at the same time, the diverse spectral characteristics of each one, increases the complexity of the transcription problem.

2.6.2.1 Spectral Envelopes

Spectral envelope denotes a countour that passes through the peaks of the spectrum. Generally, those peaks are the partials of the signal. Many musical instruments have smooth spectral envelopes but differ immensely in their shapes. The spectral shape also evolves with time, by having partials decaying at different rates. Furthermore, pianos, bassoons, oboes and guitars often produce relatively weak fundamentals on the lower frequencies. A universal model that generalizes musical instruments still has to be developed, according to previous studies (Jensen (1999); Loureiro et al. (2004); Burred et al. (2006)).

2.6.2.2 Inharmonic Partials

Because of the physical properties of instruments, most of them do not produce sounds with harmonic partials, but partials slightly deviated from the ideal frequency. Those partials are called **inharmonic** partials, and occur often in string instrument sounds. The measure of how much inharmonic partials are deviated from their ideal frequencies, is called inharmonicity deviation. For strectched strings, the frequencies of the partials are given by the Equation 2.32, where F is the fundamental frequency, h is the partial number, and β is the inharmonicity factor (Fletcher and Rossing, 2008).

$$f_h = hF\sqrt{1+\beta(h^2-1)}, \tag{2.32}$$

A harmonic model needs to allow for certain inharmonicity in order to explain the frequency deviation from each partial. If that is not the case, additional sources may be needed to explain the inharmonic partials.

2.6.2.3 Spurious components

Some instruments have dominant frequency components excited along with the partials, called phantom partials. These phantom partials, observed in string instruments, are related to the tension variation of the plucked strings and appear close to the frequencies of the partials (Conklin, 1999).

2.6.3 Transients

Transients can be simply stated as an event or zone of short duration where a fast variation of the sound signal occurs (Rodet and Jaillet, 2001). These variations can occur at note onsets as fast attacks or at note offsets with fast releases. Due to its highly non-stationary nature, it is very difficult to estimate the correct F0 within transients. As demonstrated by McIntyre et al. (1983), sometimes the attack transient also excites subharmonics on bowed and woodwind instruments. Transients often have high energy which results in spectral collisions with other sound sources. Recent research deal with transients as a specific signal component. Rodet and Jaillet (2001), Röbel (2003) and Bello et al. (2005) detect transients by applying non-parametric approaches, as oposed to Molla and Torrésani (2004) and Daudet (2004), which applied parametric approaches.

2.6.4 Reverberation

Reverberation prolongs preceding sounds by overlapping them with the following sounds, and also increases the complexity of the task of F0 estimation. A recorded signal becomes a mixture of multiple sounds, such as direct sounds, reflected sounds and reverberated sounds. As studied by Beauchamp et al. (1993), Baskind and De Cheveigné (2003) and Röbel et al. (2006), even a record of a monodic instrument in a reverberant environment can be polyphonic. The reverberated parts are often non-stationary, adding even more complexity to the analysis of the signal.

Chapter 3

Related Work

At the time of this dissertation, a general-purpose transcription system does not exist. Over the years, there has been a lot of research on Pitch Estimation.

Yeh (2008) categorized the approches to Automatic Music Transcription as joint estimation or iterative estimation. Iterative estimation approaches typically find the most predominant F0 value, cancel the detected F0 and iterate again to find the next fundamental frequency. The cancellation technique is applied to the detected F0, in order to remove its harmonics and subharmonics, otherwise it would infer noise and could lead to false results on future detection processes. These algorithms often have a small computational cost, but at each iteration, tend to acumulate errors. Join Estimation approaches often have a greater computational cost compared to the iterative approaches. Instead of evaluating one fundamental frequency at each step, they evaluate F0 combinations, increasing the results accuracy.

According to Su and Yang (2015), multi-pitch estimation could be categorized depending on wether a training dataset with ground-truth pitch and instrument annotations is used. Most of the work done earlier, used unsupervised learning, while more recently, several approaches use supervised learning. For this purpose, a few datasets are available, such as Emiya et al. (2010a), Goto et al. (2002), Fritts (2006).

Given that the most recent approaches are mostly joint approaches, Benetos et al. (2013) categorize the current multi-pitch detection systems into three groups, according to the core techniques employed: feature-based, statistical model-based or spectrogram factorisation-based. Feature-based techniques do not use a specific model, but try to

devise measures of pitch salience and criteria for selecting and scoring pitch candidates from time-frequency representations. Statistical model-based techniques use probabilistic methods to model the spectral peaks or envelops. Spectrogram factorisation-based techniques use templates of spectral patterns of different pitch combinations and then decompose an input magnitude spectrogram according to the activation of different templates. Next we will use this type of categorization in our description.

3.1 Feature-based multi-pitch detection

3.1.1 Hypothetical Partial Sequence

Yeh et al. (2010), Yeh (2008) present a frame-based system for estimating single-channel polyphonic music signals based on the STFT representation. The system classifies the spectral peaks into sinusoids and noise with an adaptive noise level estimation. The Rayleigh distribution is used to model the spectral magnitude distribution of noise (Yeh and Roebel, 2006). The plausibility of a set of F0 hypotheses is jointly evaluated, in order to match as many sinusoidal peaks as possible, taking into consideration the overlapping partials. For each hypothesis, the frequencies and the amplitudes of their **hypothetical partial sequences** (HPS) are calculated by partial selection, using a harmonic matching technique and an overlapping partial treatment. The joint estimation algorithm is based on the characteristics of harmonic instrument sounds: harmonicity, the smoothness of spectral envelope and synchronous evolution of partial amplitudes. The score function is a linear combination of four criteria: harmonicity (HAR), mean bandwidth (MBW), spectral centroid (SPC) and the standard deviation of mean time (SYNC). The HAR criterion evaluates the harmonic matching between the combination of the HPS and the observed spectral peaks. The MBW criterion evaluates the frequency of the envelope of a HPS by its bandwidth. The SPC criterion is used to prevent subharmonic errors. The SYNC criterion estimates the mean time for each individual peak, in order to evaluate the synchronicity of the temporal evolution of the partials in a HPS. These criteria are then combined by the sum of the peak salience of the related partials. A F0 hypothesis is considered a valid estimate if it either explains significant energy or improves the spectral smoothness of the set of the valid F0 estimates.

3.1.2 Cancellation by Spectral Models

Klapuri (2003) presented an iterative estimation approach based on harmonicity and spectral smoothness. The input signal is preprocessed by a RASTA-like technique (Hynek Hermansky, 1993) on a logarithmic frequency scale such that the spectral magnitudes are compressed and the additive noise is removed. The preprocessed spectrum is spllited into multiple frequency bands. At each subband, F0 weights are calculated by normalizing the sum of their partial amplitudes. Those weights are then combined taking inharmonicity into account. The predominant F0 source is smoothed and subtracted from the signal spectrum in order to avoid its corruption after multiple iterations of direct cancellation. After the subtraction, the overlapping partials still persist in the remaining sources. The method described uses the average amplitude within one octave band in order to smooth out the envelope of an extracted source, and is called the **bandwise smooth model**. The process repeats itself by computing the weights of each candidate and extracting the F0 candidate until the maximum weight related to signal-to-noise ratio (SNR) is below a fixed threshold. A perceptually motivated multiple-F0 estimation method is presented by Klapuri (2005). The input signal is splitted into multiple frequency bands by using a bank of bandpass filters, which models the frequency selectivity of the inner ear. Each subband signals are compressed, half-wave rectified and low-pass filtered. The magnitude spectra is summed across channels and used to perform the harmonic matching to extract the predominant F0. A **1/k smooth model**[1] is used to remove the predominant source from the mixture, while keeping the energy of higher partials for the next iterations. Klapuri (2006) presents a spectral model which attempts to generalize a variety of musical instrument sounds. An input signal is first spectrally flattened in order to suppress timbral information. Then, the salience of a F0 candidate is calculated as a weighted sum of the amplitudes of its harmonic partials. Santoro and Cheng (2009) present an algorithm for multiple F0 estimation in the transform domain, based on Klapuri's work, to function in the Modified Discrete Cosine Transform (MDCT) domain.

3.1.3 Combined Frequency and Period Domains

Peeters (2006) and Emiya et al. (2007) use both frequency and lag domain features to tackle the problem of single-pitch estimation. They multiply a spectral and a temporal representation of the input audio signal to determine the likelihood of a pitch

[1]Partial amplitudes are approximately inversely proportional to the partial index.

candidate. Bello et al. (2006) extended this ideia to multi-pitch estimation of piano music and presents a system which uses a hybrid method, where the frequency domain approach is improved by a time–domain recognition process. This method takes into account the information contained in phase relationships that are lost when only the magnitude spectra of sounds are analyzed. Su and Yang (2015) extends Peeters (2006) and Emiya et al. (2007) work for multi-pitch estimation and propose an unsupervised feature-based approach referred to as Combined Frequency and Periodicity. This system detects pitches according to both harmonic series in the frequency domain and a subharmonic series in the quefrency domain. The log-scaled amplitude spectrum is used for the frequency representation of the signal, which is then pseudo-whitened to spectrally flatten the signal as in Klapuri (2003). The generalized cepstrum is employed for the temporal representation of the signal. After thresholding both signals, a peak picking process is applied to all local maxima and discards other non-peak terms. The presence of a true pitch is identified by three conditions: a prominent harmonic series in the frequency representation, a prominent subharmonic series in the temporal representation and the fundamental frequency of the harmonic series and the fundamental period of the subharmonic series match at the same fundamental frequency. Criteria to deal with missing fundamental frequencies and stacked harmonics are presented. False positives are reduced by sparcity contraints. In post-processing, pitches above C5 that leave any other pitches in the affinity of 0.1 seconds by more than one octave are discarded. Then, a comparison is done between the pitch estimates in neighbor frames for temporal smoothness which connects non-continuous estimates and removes isolated notes shorter than 0.12 seconds.

3.1.4 Neural Networks

Marolt (2004), presented a connectionist approach to automatic transcription of polyphonic piano music, using a partial tracking technique, based on a combination of an auditory model and adaptive oscillator networks. The input audio signal is converted to a time-frequency representation through the use of an auditory model. The auditory model, which imitates the functionality of human cochlea, has two parts: the Patterson-Holdsworth gammatone filterbank (Patterson and Holdsworth, 1996) and the Meddis hair cell model (Meddis, 1986). The first, applies a series of logarithmically spaced gammatone filters to the acoustic signal, which splits the input signal into several frequency channels, in order to model the movement of basilar membrane in the inner ear. The output from each filter is processed, using Meddis model (Meddis, 1986) of hair cell transduction, which simulates several of the cell's characteristics, like

half-wave rectification, saturation and adaptation. The auditory model results in a quasi-periodic impulsive signal that represents the firing patterns of inner hair cells. To achieve partial tracking, Large and Kolen (1994) adaptive oscillators were used to synchronize the frequency and phase from the output channels of the auditory model. Phase and period of the oscillators are updated by minimizing an error function, according to the modified gradient descent rule. A partial is detected if a synchronization occurs. This model was extended to track multiple harmonically related partials. Partial groups are tracked using 88 networks of adaptive oscillators, corresponding to 88 piano tones (A0-C8). Each network consists of up to ten interconnected oscillators, and the frequency of each oscillator in the network is initially set to an integer multiple of the fundamental. Networks of oscillators are more resistant to noise and are more robust on indicating the presence of a tone than an individual oscillator. To perform note recognition, a set of 76 neural networks was used to recognize notes from A1-C8. The inputs of each network consist on the output values of oscillator networks, amplitude envelopes of signals in frequency channels of the auditory model, and a combination of amplitude envelopes and oscillator network's outputs. The neural network model used is the **time-delay neural network** (Waibel et al., 1990). The partial tracking model and time-delay neural networks were integrated into SONIC, a system for transcription of piano music (Marolt, 2001). An onset detector, and a module for detecting repeated notes were also included in this system.

3.1.5 Blackboard Systems

A blackboard system is an artifical intelligence application designed to handle complex problems, where the solution is the sum of its parts (Nii, 1986b). According to Nii (1986a), a blackboard-system application consists of three major components: knowledge sources, blackboard and control shell. The knowledge sources are a diverse group of specialists, each one being a self-cointained expert on some aspects of the problem which can contribute to the solution independently of the particular mix of other specialists (Corkill, 1991). The blackboard is a common knowledge base, shared repository of problems, partial solutions, suggestions, and contributed information. It is constantly being updated by the knowledge sources, in order to achieve a solution. The control shell is responsible for controlling the flow of problem-solving activity in the system, organizing the common knowledge sources in the most effective way. Martin (1996) presents a blackboard approach to automatic music transcription where a new system is proposed, based on the log-lag correlogram (Ellis, 1996). Bello and Sandler (2000), Bello et al. (2000) present a system based on a top-down approach. The

blackboard system is composed of three hierarchical levels. The inputs are the result of a segmentation routine in the form of an averaged STFT matrix. The blackboard has a hypotheses database, a scheduler and knowledge sources. One of those sources is a neural network, trained for chord recognition, which allows the system to output more than one note hypothesis at a time.

3.2 Statistical Model-Based Multi-Pitch Detection

3.2.1 Maximum a Posteriori Estimation Approach

Emiya et al. (2007) address the problem of single-pitch estimation with a technique based on a Weighted Maximum Likelihood principle. The signal is decomposed into a sum of sinusoidal components and a colored noise. A moving average process is assumed for the noise while the spectral envelope of the partials is modeled by an autoregressive model. The fundamental frequency is calculated by following a Weighted Maximum Likelihood principle, which simultaneously whitens both noise and sinusoidal sub-spectrums. This technique is extented for multi-pitch estimation, by jointly evaluate multiple F0's at the same time. Emiya (2007) incorporates an onset detector presented by Alonso et al. (2005). For each segment, the first frames are analised and the largest peaks will result in a set of F0 candidates. Then, for each frame and for each combination of notes among the selected candidates, the likelihood of the spectrum is derived, according to the previous work in Emiya et al. (2007). The maximum likelihood estimation is embedded into a Hidden Markov Model framework. Finally, detected pitches in consecutive frames and segments are merged together. Emiya et al. (2010) extend this approach to multipitch estimation of multiple concurrent pitches in piano sounds. A new spectral model is employed where the inharmonic distribution is taken into account and adjusted for each possible note. A smooth autoregressive model is introduced to model the spectral envelope of the overtones and a low-order moving-average (MA) process is used for the residual noise. Goto (2004) propose a method called *PreFEst* to estimate the most predominant F0 of melody and bass lines in audio signals. Maximum A Posteriori Probability estimation is employed to represent every possible F0 as a probability density function, by using the expectation-maximisation algorithm (Dempster et al., 1977). Kameoka et al. (2007) present a multipitch analyzer called the harmonic temporal structured clustering method. This method jointly estimates multiple fundamental frequencies, onsets, offsets and dynamics. The harmonic

structure model is an extension of the Goto (2004) work. The frequency of each partial is modelled using a Gaussian distribution function and the spectra is obtained by the constant Q transform. The synchronous evolution of partials is modelled by Gaussian mixtures.

3.2.2 Time-domain Bayesian Approach

Davy et al. (2006) extends the polyphonic time-domain Bayesian harmonic model previously presented in (Walmsley et al., 1998), (Walmsley et al., 1999), (Davy and Godsill, 2003) and (Godsill and Davy, 2002), for multi-pitch transcription, with slight modifications. The authors use a Gabor representation of nonstationary signals and a Markov chain Monte Carlo method for the parameter estimation algorithm. The model supports time-varying amplitudes and inharmonicity. Peeling and Godsill (2011) propose a new method for solving the problem of multi-pitch estimation using novel statistical models. The partial frequencies in the frequency domain are modeled by an inhomogeneous Poisson process. Koretz and Tabrikian (2011) addresses the problem of multi-pitch estimation using a combination of the maximum likelihood and maximum a posteriori probability criteria. Each of the fundamental frequencies is modeled by a Markov process. The dominant signal is modeled as a harmonic source and the remaining sources are modeled as Gaussian interference. The dominant source is estimated and removed from the mixture, and the process is applied to the next harmonic source.

3.2.3 Maximum-Likelihood Approach

Duan et al. (2010) presents a maximum likelihood approach to multiple fundamental frequency (F0) estimation. The audio signal is splitted into multiple frames. To each frame, the Short Time Fourier Transform, hamming window and zero-pagging are applied, to obtain a power spectrum. Spectral peaks are detected using the peak detector presented in Duan et al. (2008). The parameters for the model are learned from monophonic and polyphonic data. The system models the power spectral density as both spectral peaks and non-peak regions. The peak likelihood aims to find F0s that have harmonics that explain peaks and the non-peak likelihood aims to avoid F0s that have harmonics in non-peak regions. Yoshii and Goto (2012) present a statistical method called infinite latent harmonic allocation (iLHA) to deal with multiple fundamental

frequencies (F0s) estimation in polyphonic music audio signals. The method relies on hierarchical nonparametric Bayesian models that can deal with complex models of multiple F0's and their harmonic structures.

3.3 Spectrogram Factorisation-Based Multi-Pitch Detection

3.3.1 Genetic Algorithms

Reis and Vega (2007), Reis et al. (2007) consider music transcription as a search problem and present a new method for multi-pitch estimation on piano recordings, using genetic algorithms. Altough the authors show the feasibility of the approach, the genetic algorithm tends to create additional notes in harmonic locations of the original notes to minimize the timbre differences between the original audio signal, and the internal samples. In order to avoid harmonic overfitting, Reis et al. (2008) add harmonic gains in the algorithm, which boost or cut the value of the first 20 harmonic peaks, on each note harmonic. Reis (2012); Reis et al. (2012) present a method which consists in a genetic algorithm aided by two major components: an adaptative spectral envelope modeling and a dynamic noise level estimation. The system starts with an onset detector, based on Martins (2009) with slight modifications, which splits the input signal into multiple segments. For each segment, a genetic algorithm, is launched to perform the transcription. After applying the genetic algorithm, all the segments are joined into one whole transcription, and an Hill-Climber algorithm is used to merge consecutive notes. Each individual represents a candidate transcription, which encodes each musical note, with its start time, duration, MIDI note and MIDI velocity. An adaptative threshold component encodes the dynamic noise level estimation, by adjusting the noise level for each frequency bin in the spectrum. The spectral envelope component encodes the internal synthesizer, by using the gain of its harmonics, expressed in dB, and its inharmonicity deviation for each partial, in order to best match the input piano in the original signal. A general system to encode the harmonic deviation of each partial was adopted to work with other kinds of pitched instruments. Each solution is first rendered by an internal synthesizer. Then, the fitness function, based on the log spectral distance, compares the input audio signal with the generated transcription, in the frequency domain.

3.3.2 Non-Negative Matrix Factorisation

Most recent multi-pitch detection techniques use and expand spectrogram factorisation techniques (Benetos et al., 2013). Smaragdis and Brown (2003) present a methodology for analyzing polyphonic music based on Non-negative Matrix Factorization (NMF) of magnitude spectra. Non-negative Matrix Factorization is a useful decomposition for multivariate data Lee and Seung (2000). The method proposed by Cont (2006) for Automatic Music Transcription, extends the work done previously by Sha and Saul (2005). The method uses unsupervised learning to reconstruct the realtime audio, using previously learned pitch structures of an instrument. A modified sparse Non-negative Matrix Factorization algorithm is used for realtime pitch observation. Vincent et al. (2010) incorporates harmonicity constraints in the NMF model. Each basis spectrum is modeled as a weighted sum of narrowband spectra representing a few adjacent harmonic partials. The spectral envelope is adapted to each instrument to enforce harmonicity and spectral smoothness. Bertin et al. (2010) present a NMF in a Bayesian framework applied to polyphonic music transcription, which uses a model of superimposed Gaussian components. The likelihood function incorporates spectral smoothness constraints. A space-alternating generalized expectation-maximization (SAGE) algorithm is applied to estimate the parameters. Ochiai et al. (2012) proposes a NMF for estimating simultaneously basis spectra and activations, detecting note onsets and duration, and determining beat locations. The rythmic structure is used to constrain NMF by parametrically modeling each note activation with a Gaussian mixture. They also developed an algorithm which iteratively updates the model parameters.

Chapter 4

Cartesian Genetic Programming

Genetic Programming is a type of evolutionary algorithm whereby programs are evolved. Evolutionary Algorithms are methods of parallel search and optimization, that mimick the processes of Darwinian evolution and molecular genetics such as selection, recombination and mutation, the so called genetic operators. The selection operator gives preference to better candidate solutions by stochastically passing them to the next generation of the algorithm. The crossover operator takes more than one parent solution and recombine their genes. The goal is for the algorithm to take the better parts of each parent solution and produce a candidate solution better than the previous ones. The mutation operator encourages genetic diversity amongst candidate solutions and tries to prevent the algorithm from converging to a local minimum. These techniques are used to exploration of the search space and exploitation of "good" zones.
In each generation (iteration) exists a population of possible solutions (candidate solutions) to the problem, which are referred as individuals. During each iteration, all the individuals are evaluated by an evaluation function, often referred to as fitness function. After the evaluation, they are submitted to a process of selection, where the best individuals are preferably chosen. Those individuals can be recombined and suffer mutations. The resulting individuals will constitute the next population in the new generation.

4.1 Genetic Programming

Genetic Programming is concerned with the automatic evolution of computational structures in the form of Lisp like parse trees (Koza, 1992). Those computational structures (mathematical equations, computer programs, digital circuits, etc.) are evolved and applied to solve problems for which it is very difficult for humans to design solutions. Complex problems generally require larger population sizes to solve and crossover is most often used as the primary method of developing new candidate solutions from the previous generations (Koza, 1994). In contrast, Evolutionary Programming (EP) tends to promote the importance of the mutation operator (Fogel et al., 1966), (Fogel, 1995). They both share the same underlying structure, where programs are represented as parse trees, without having a distinction between genotype and phenotype (Miller and Thomson, 2000).

4.2 Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) grew out of the work of Miller et al. (1997), as a method of evolving digital circuits. However, the term "Cartesian Genetic Programming" appeared two years later in Miller (1999). CGP is now considered as a general form of Genetic Programming (Miller and Thomson, 2000). It is being applied to many fields, such as machine learning, neural networks, data mining, financial prediction, function optimization, classification, electronic circuit design, and so on. According to Miller (1999), CGP is more efficient than standard GP methods in learning Boolean functions.

CGP is *Cartesian* because it encodes programs as a two-dimensional grid of nodes that are addressed in the *Cartesian* coordinate system (see Section 4.2.2). In its classic form, it uses a very simple integer based genetic representation of a program in the form of a directed graph instead of a tree. Graphs are very useful program representations, more general than trees, and can be applied to many domains (e.g. electronic circuits, neural networks).

4.2.1 Programs

CGP Programs have three major components: program inputs, computational nodes and program outputs. **Computational nodes** are structures organized and composed by input connections and a function. The input connections of a node have their origin in any program input or other precedent nodes. The function is among the ones previously defined in a look-up table and it takes as arguments the values received through the node's inputs. The node itself is indexed by an integer value so that it can be referenced by other node input connections. The computational nodes, organized in a two-dimensional grid of nodes, are numbered sequentially and linked directly between them in a feed-forward manner (see Figure 4.1). A program can have several inputs, named **program inputs**. **Program outputs** are indexes that link to some nodes. For example, if the program's output is the number 4, the result of the program is the value computed by node 4's function (see Figure 4.1). Program inputs and nodes are referenced by sequential numbers. The idea is best explained with a simple example. In Figure 4.1 we can see that the program has two inputs, four nodes and one output.

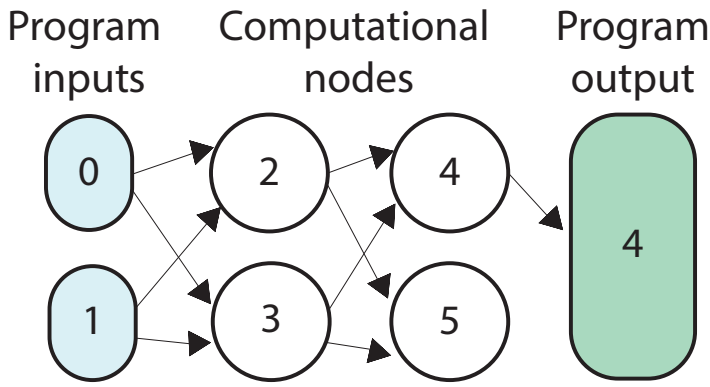


Figure 4.1: Overall strucuture of a CGP program. Program inputs and computational nodes are numbered sequentially. The program outputs can link to any computational node or program input.

4.2.2 Genotype

The genotype is the codification of a program as it is used and manipulated by the CGP algorithm. It describes what are the programs inputs, computational nodes, program outputs and how they are connected together. In general, it is a list of genes where each gene is an integer. As we have seen earlier, program inputs and nodes are referenced by their index. Since a node is a structure with input connections and a function, each node has multiple genes (see Figure 4.2). The genetic structure that encodes a

node first references the function value and then the values of the node's connections sources. In Figure 4.2, the list of genes to encode the node are: 2 3 4.

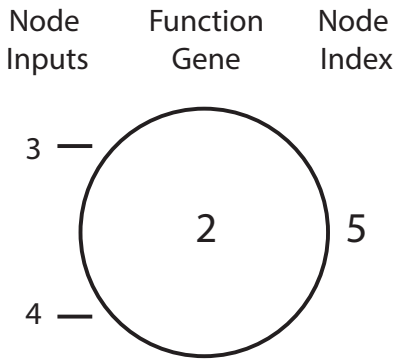


Figure 4.2: Example of a node that has two connection genes: node 3 and node 4. It will compute the function number 2 in the function-set. The node is referenced by the number 5.

Each node has a function gene which is an address in a look up table of functions. Usually, all functions have as many inputs as the maximum function arity and unused connections are ignored. This introduces an additional redundancy into the genome. In the example of Figure 4.2, the node 5 will have nodes 3 and 4 has inputs and it will apply the function number 2 defined previously. If function 2 represents a *sum*, node 5 would compute the following:

$$y = c_1 + c_2, \tag{4.1}$$

where c_1 is the value comming through the first connection and c_2 is the value comming through the second connection. If $c_1 = 2$ and $c_2 = 1$, the value of node 5 would be $2 + 1 = 3$ (see Figure 4.3).

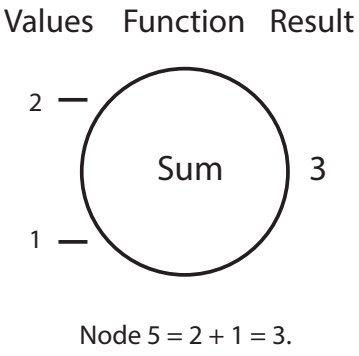


Figure 4.3: The result of node 5 will be $2 + 1 = 3$.

There are a few number of parameters that we need to define in order to encode a CGP

Table 4.1: Parameters of the program illustrasted in Figure 4.4

Parameter	**Value**
Number of Inputs (ni)	3
Number of Outputs (no)	1
Number of Rows (nr)	1
Number of Columns (nc)	3
Inputs (i_i)	0,1,2
Functions (f_i)	5,3,1
Outputs (O_i)	4
Genotype	512 303 102 4
Phenotype	512 303 4

program. The number of program inputs is given by ni and the number of program outputs is given by no. Given that nodes are organized in a tabular way, the number of columns is given by nc and the number of rows by nr. For example, the program in Figure 4.4 has the following attributes: $ni = 3$, $no = 1$, $nc = 3$, $nr = 1$ and the genotype is the following list of integers: 512 303 102 and 4. Knowing that $ni = 3$, the genotype encodes the first node at index 3, since the first three indexes represent the program inputs and the first index is 0. The first node in the genotype, node 3, computes function 5, and its connections are the program input 1 and program input 2. Node 4 computes function 3, and its connections are the program input 0 and the value of node 3. The output of that program is the value of node 4. We point that there are no program outputs nor nodes whose input connections reference node 5. This means that this node cannot influence the program output. More on this in Section 4.2.4.

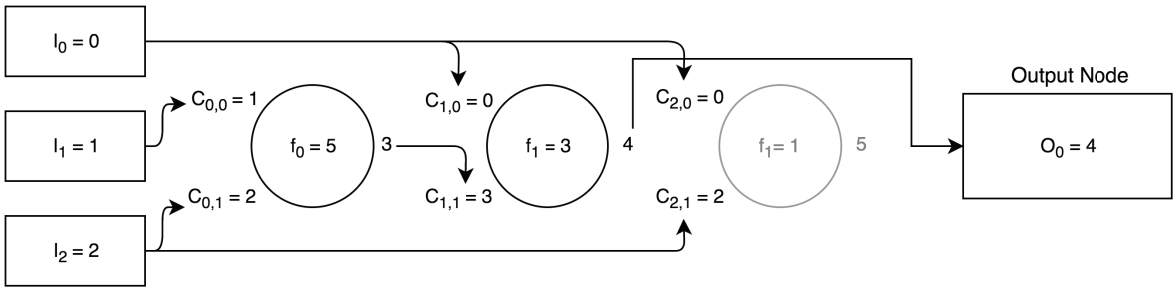


Figure 4.4: CGP graph, where $ni = 3$ and $no = 1$. The grid has $nc = 3$ (columns) and $nr = 1$ (row).

Table 4.1, enumerates a few parameters of the program illustrasted in Figure 4.4. Figure 4.4 shows the general form of a CGP graph, with $nr = 1$ and $nc = 3$.

4.2.3 Allelic Constrains

There are some allelic constrains, that the genotype must respect. The alleles (values) of function genes f_i must take valid address values in the look-up table of primitive functions. Let nf represent the number of allowed functions. Then f_i must obey to the following range:

$$0 \leq f_i < nf. \tag{4.2}$$

There is another parameter called **levels-back** l, which determines how many previous columns of nodes may connect to a node in the current column. When $nr = 1$ and $l = nc$, any node can have input connections coming from any program input and any node on its left, which allows unrestricted connectivity. However, if $nr > 1$, nodes cannot connect to other nodes in the same column. Then, having a node in column j, and $j \geq l$, node connections, C_{ij}, must obey to the following range:

$$ni + (j - l)nr \leq C_{ij} \leq ni + j \times nr. \tag{4.3}$$

If $j < l$, then the following condition must be met:

$$0 \leq C_{ij} \leq ni + j \times nr. \tag{4.4}$$

Program output genes O_i can connect to any node or program input:

$$0 \leq O_i < ni + Ln, \tag{4.5}$$

where Ln is the number of nodes in the genotype, computed by the following:

$$Ln = nr \times nc. \tag{4.6}$$

This representation is very simple, flexible and convenient for many problems.

4.2.4 Genotype-Phenotype Mapping

One of the key characteristics of CGP is the genotype-phenotype mapping. The genotype is of fixed-length but the phenotype is not, duo to the fact that the genotype can have inactive genes. Thus, they are redundant because they cannot influence the programs output. The corresponding genes are called **non-coding genes** or **inactive genes**. This means that we can have a phenotype different from the genotype because non-coding genes are not expressed in the phenotype, that is, the program that will run in practice.
The output or outputs of the CGP are nodes that point to other nodes (connection genes) and so on. Decoding the program is recursive in nature and works from the program output genes first. To decode the program outputs, the active nodes should be identified. The process begins by looking at which nodes are directly connected to the output genes. Then these nodes are examined to find out which nodes are directly linked to them. Since non-coding genes are not addressed, they present little computational overhead.

4.3 Algorithm

The evolutionary strategy widely used for CGP is a special case of the $\mu + \lambda$ (Hansen et al., 2015) where $\mu = 1$ (Algorithm 1). This means that, in this special case, the population size is always one. At each iteration (generation), λ new offspring are generated from the current one through mutation. Then, the best among the current individual and the offspring becomes the current individual in the next iteration. An offspring can become the current individual in the next iteration when it has the same fitness as the current individual and there is no other individual with a better fitness.

4.4 Genetic Operators

The selection operator is mainly expressed in step 8 of Algorithm 1 above, where the best among the current individual and the λ offspring is chosen as the next iteration individual.

Algorithm 1 Algorithm $((1+\lambda)\,EA)$

1: $t \leftarrow 0$;
2: Set current individual I_0 as the best of λ individuals created randomly;
3: **while** a stop condition is not fulfilled, **do**
4: **for** i = 1 to λ **do**
5: Create a copy x_i of current individual I_t;
6: Mutate each gene of x_i with probability p;
7: **end for**
8: Set new current individual I_{t+1} as the best of $I_t \cup \{x_1, \ldots, x_\lambda\}$;
9: $t \leftarrow t + 1$;
10: **end while**

The mutation operator used in CGP is a point mutation operator and it is very simple to implement. One merely has to allow changes to the genes which respect either the functional constraints or the constraints imposed by levels-back. If the gene to mutate is a function gene, we have to make sure that the new chosen value is valid in the function-set table. If a connection gene is chosen for mutation, then a valid value is the address of the output of any previous node, respecting the levels-back parameter, or of any program input. If the gene to mutate is an output gene, we choose any random node.
Crossover operators don't usually receive much attention in CGP. In (Miller and Thomson, 2000), a one-point crossover operator was used but they found it to be disruptive to the subgraphs within the chromosome, which affected the performance of CGP.

4.5 Example - CGP applied to Image Processing

Harding et al. (2013); Hardingb et al. (2013) present a technique based on CGP, that allows the automatic generation of computer programs using a subset of the OpenCV image processing library functionality. This approach is refered to as Cartesian Genetic Programming for Image Processing (CGP-IP) and it is applied to several domains, such as basic image processing, medical imaging, and object detection in robotics. They also present a framework combining computer vision and machine learning for the learning of object recognition in humanoid robots using CGP.

4.5.1 Object Detection - Classification Problem

One of the domains that CGP-IP is used for is **object detection**, where the goal is to target a certain object in an image. To train CGP-IP, a number of images were collected from a few cameras. For each collected image, the target object was repositioned. The training set is implicitly composed by multiple views of the target object with different angles, scales and lighting conditions. The target object was then hand segmented for each image in the traning set (see Figure 4.5). A set of filters are trained to produce

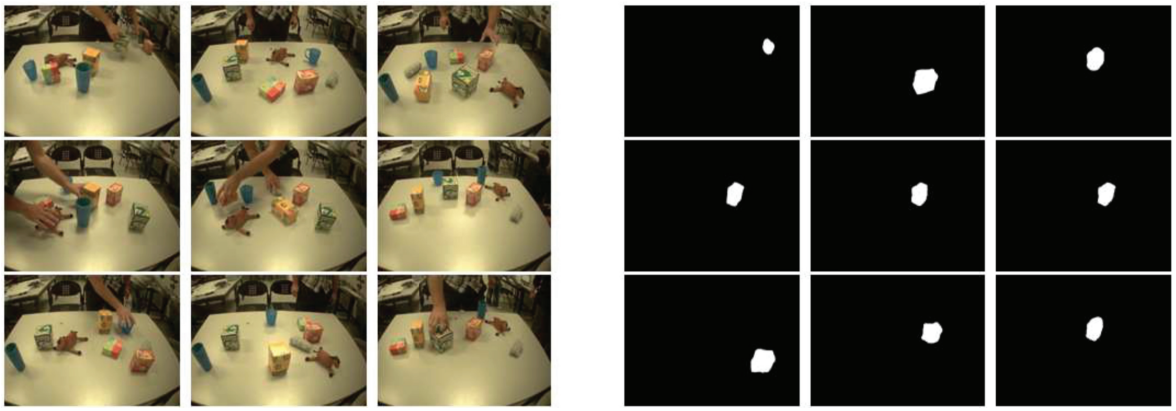


Figure 4.5: The set of the left shows each collected images with a target object. The set on the right shows the binary classification, determined by a human, where a particular box is highlighted in white. Image was taken from Harding et al. (2013), page 11.

a binary classification on new images, different from the ones used in the training set. The program inputs are based on the training-set images: the camera image in grayscale, the image's red, green and blue (RGB) channel, as well as, its hue, saturation and brightness value (HSV) channels. Figure 4.6 shows the application of an evolved filter running in real time.

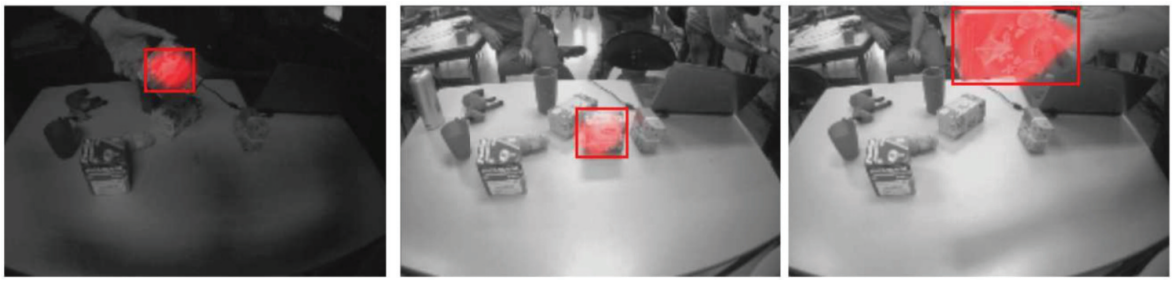


Figure 4.6: Examples of an evolved filter running in real time. Image was taken from Harding et al. (2013), page 11.

The next two sections describe the two features of CGP-IP that were applied to our work and that will be discussed later in the next chapters.

4.5.2 Parameters

CGP-IP uses a large function-set composed by primitive functions and high level image functions. These functions often require one or more parameters and are sensitive to their type and range (e.g. to subtract each image pixel the value of the first parameter p_0). The total number of parameters is represented by np, and each parameter is denoted by $p_0, p_1, \ldots, p_{n_p-1}$. Compared to classical CGP, CGP-IP encodes 5 additional values (parameters) into each node. p_0 is a real number, typically used as a constant value. p_1 and p_2 are integers in the range -16 to $+16$, used as a parameter to an image operation. p_3 is an integer in the range 0 to 16, used as a parameter for Gabor filter operations. Finally, p_4 is an integer in the range -8 to 8, used as a parameter for Gabor filter operations. Figure 4.7 shows a node with two parameters.

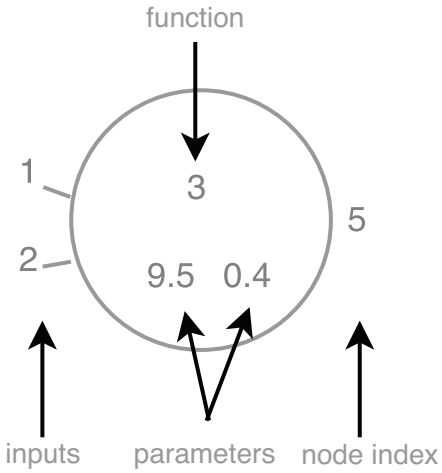


Figure 4.7: Representation of a node with two parameters, where $n_p = 2$, $p_0 = 9.5$ and $p_1 = 0.4$.

4.5.3 Threshold

For the classification problems (e.g. object detection), the output of the CGP program (an image) is thresholded and treated as a binary image. Each pixel in the image is treated as a binary classification test case. This is then compared to the target image using the Matthews Correlation Coefficient (a measure of the quality of binary classifications).

The threshold has a range of 0 to 255. It performs a binary threshold of an image by outputting a white pixel if a value is more than p_0 and a black pixel otherwise. This threshold suffers mutation, and is modified with a 1% probability: uniform noise of +/-10% is added. It has an important role in the classification of the test cases.

Chapter 5

Cartesian Genetic Programming Toolbox

As the first step on applying Cartesian Genetic Programming to sound processing, we decided to create a Matlab Toolbox for this task. The idea was to have a highly flexible toolbox, configurable throughout parameters and function callbacks. Then, we moved to the problem of Pitch Estimation by applying and configuring the same toolbox to our particular case. This toolbox is open source and freely available at: https://github.com/tiagoinacio/cgp-toolbox.
The toolbox's architecture will be introduced throughout this chapter. Then, each component will be explained in detail. Lastly, we will show an example of the toolbox applied to a sixth order polynomial symbolic regression problem.

5.1 Architecture

The CGP Toolbox is very simple to use and allows to quickly encode a problem. The structure of classic CGP is reproduced in the toolbox. One of the main goals was to have a generic toolbox that could help us to encode from smaller to bigger problems. With that in mind, a few design decisions were made that will be explained next.

5.1.1 Overview

All the combinations of rows and columns are possible, considering that $nr > 0$ and $nc > 0$. The allelic constrains are generated dynamically, depending on the cartesian representation of the nodes. *Levels-back* was also taken into consideration. Additionally, the toolbox is prepared to use parameters in the genotype. There are no limits for the number of parameters. The fitness function is any function provided by the user (it is explained in more detail in Section 5.2.11). The toolbox is prepared to receive one or more program inputs of any types and values. The number of program outputs can be one or more, in order to address different problem requirements. The function-set is also provided by the user and the look-up table is automatically generated. Furthermore, there is a system of callbacks which is discussed in Section 5.2.1. The Evolutionary Algorithm (EA) used is the $1 + \lambda$, referred previously in Section 4.3.

5.1.2 Evolutionary Algorithm

A detailed representation of the encoded EA is presented in Algorithm 2. The goal is to have a toolbox as generic as possible, so a few parameters for the evolutionary process were chosen to be configurable.

Algorithm 2 Algorithm $((1 + \lambda)\, EA)$ encoded with multiple runs

$r \leftarrow 0$;
while $r < mr$ **do**;
    $g \leftarrow 0$;
    Set current individual I_0 as the best of λ individuals created randomly;
    $F_g \leftarrow$ fitness of current individual;
    while $g < mg$ or $F_g < f$ **do**
        for i = 1 to λ **do**
            Create new individual x_i from the current individual I_g;
            Mutate each gene of x_i with probability p;
        end for
        Set new current individual I_{g+1} as the best of $I_g \cup \{x_0, \ldots, x_\lambda\}$;
        $g \leftarrow g + 1$;
        $F_g \leftarrow$ fitness of current individual;
    end while
    Save the best individual of run r (I_g) as B_r;
    $r \leftarrow r + 1$;
end while

The number of **offspring** (λ) is defined by the user. This is useful because there can be

some problems that require a small number of offspring and others that require a bigger number of offspring. The **mutation rate** (p) is also configurable. This is the mutation probability for each gene. The maximum number of **runs**, mr, and maximum number of **generations**, mg, are also required parameters. Finally, the last parameter is the maximum or minimum **fitness**, f, for a solution to be considered valid, depending if we want to maximize or minimize the fitness function. The EA needs to know when a candidate solution can be considered as a valid solution for the problem, in order to stop the evolutionary process.

5.1.3 Components

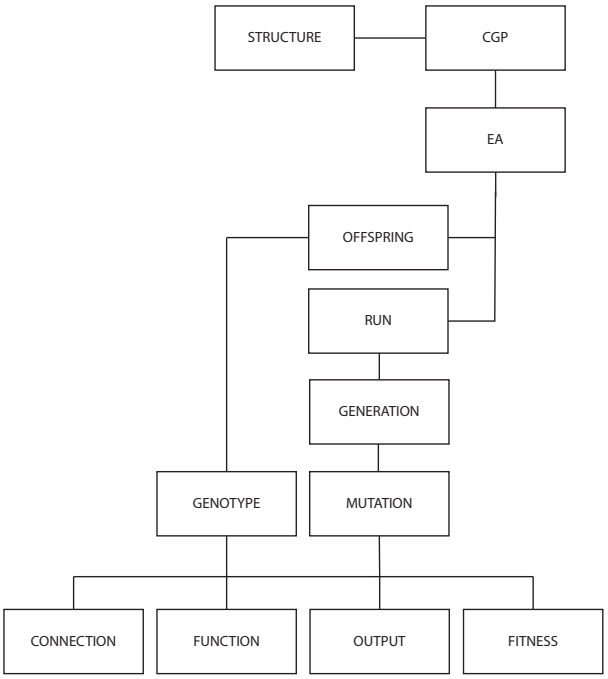


Figure 5.1: Components that are part of the toolbox.

The toolbox is divided into several components. Each one has its purpose and special role. The first one is the **CGP** component. It exposes all the functionality to encode an application built on top of the toolbox. This component comunicates with the **EA** and **Structure** components. The Structure is just an helper, which stores the positions of the genes according to the type of gene (connection, function, program output and parameter) and it will be explained in detail in Section 5.2.2. The EA component is responsible for initializing the runs in the evolutionary algorithm. It starts with a certain number of **Offspring**, created by the **Genotype** component which, in turn, is composed by the **Connection**, **Functions**, **Ouputs** and **Fitness** components. **Run** is connected to the **Generation** component, by executing it multiple times. In each

generation, **Mutation** can occur, which will change the genotypes (using the Connection, Functions, Outputs and Fitness components). Figure 5.1 shows the overall structure of the toolbox's components. Each component will be addressed in detail next.

5.2 Implementation

The toolbox was built using Object Oriented Programming methodology of Matlab (version R2016a). All the *classes* that compose the toolbox will be introduced in the next sections. For some classes, a detailed explanation of the most relevant properties and methods is also presented.

5.2.1 CGP

The *CGP* class provides access to an API that lists all the features needed to encode a program. It is the "core" of the toolbox and its primary component. The *CGP* class lets the user add the program inputs, provide the fitness function, add parameters and define the function-set. The constructor takes a configuration object. This object will contain all the configuration necessary for the CGP and for the EA.
For the CGP, the parameters are divided into: number of rows, number of columns, number of levels back and number of program outputs. Since some CGP approaches assume that the output node is the last node of the graph, this option was also taken into consideration. So, if we pass the value *last* to the *output_ type*, the last node of the genotype will be considered the program output. This option only works when the number of program outputs is set to 1, otherwise it will be ignored. Having these parameters configurable, the user has total control of the grid layout of the generated program (genotype).
For the EA, the parameters are: maximum number of generations, maximum number of runs, number of offspring, mutation rate, the fitness threshold and the fitness operator. The fitness threshold is the limit for which a candidate solution is considered a valid solution to the problem. This allows the evolutionary process to stop or skip to the next run. In some kind of problems, the goal is to minimize an error rate, where 0 would be the best value for the fitness. Also, there are other problems where the goal is to maximimze the fitness function, as we have in our approach. The *fitness_ solution*

property covers that necessity. However, the operator to use in the comparison between fitness values also needs to be configurable, because the optimization of those values is different. The fitness operator, O, is the operator to use when comparing the new fitness candidate solution with the parent's fitness, and can take the following values: '>', '<', '>='and '<='. For example, consider the parent's fitness as f_0 and an offspring fitness as f_1: if $O =$ '>', $f_0 = 0.5$ and $f_1 = 0.6$, then the offspring will replace the parent in the new generation; if $O =$ '<', $f_0 = 0.5$ and $f_1 = 0.6$, the parent will remain as it have the best fitness. This operator is also used for checking if a solution is a valid solution for the given problem. Therefore, it is also used for comparison between a solution's fitness and the *fitness_solution* value, also configurable. Figure 5.1 describes every possible field for the configuration.

Table 5.1: Configuration table with the fields that the structure should have, the type of value and the description of each one.

Key	**Type**	**Description**
rows	double	number of rows
columns	double	number of columns
levels_back	double	number of levels-back
outputs	double	number of outputs
output_type	string	set the program output as the last node (last, random)
runs	double	number of runs
generations	double	number of generations
offspring	double	number of offspring
mutation	double	probability of mutation
fitness_solution	double	fitness for a solution to be considered valid
fitness_operator	string	fitness operator ('>','<','>=' or '<=')

At the time of instantiaton, the *CGP* class will verify if all the required settings were passed in the configuration object.

This class also exposes the functionallity of adding program inputs. Each problem requires a specific set of program input or inputs. Some may require one integer as input, others may require an array, or even a complex type of object. To address this abstraction, the input provided for the CGP toolbox is of type **struct** (structure). Each field in the structure is a program input. Therefore, the program inputs can be of any type: integers, strings, structs, arrays, matrix, etc. The number of fields present in the structure indicates the number of inputs that the toolbox needs to set in the genotype, which is dynamically set: there is no need to specify how many program inputs this program will have.

The fitness function is passed by callback (function pointer) to the program.

The toolbox reads the function set from a specific directory provided by the user.

This directory should have all the functions that could be used in the genotype. All the functions should receive as many inputs as the maximum function arity. This is a requirement for the program to work. Besides the maximum function arity, if the user used added parameters to the genotype, these should be also passed to each function. This method will iterate through all the Matlab files in the directory passed as argument, and it will create a function handle for each one.
Some specific signal processing functions might require special arguments like ranges or constants to be executed (e.g.: a low pass filter needs to know which percentage of the original signal will be attenuated). Those parameters might need to evolve through time, because their best values for the contribution to the solution of the problem is unknown beforehand. The genotype can encode those parameters and add them to the evolutionary process. Parameters should have integer or double values. Each parameter is encoded by a structure with a name, a callback function for the initialization of the parameter value, and another callback function for mutating the value. The initialization and mutation functions should return an integer or a double. The mutation function should also accept an argument, that is the value of the current parameter to mutate. When running the algorithm, there are a number of events from the evolutionary process that can be useful to handle, for running additional scripts or simply to add some kind of report. In order to have that range of possibilities, the user is able to pass optional callbacks, each of which, will fire at the following events: the configuration has been set, a fittest solution is achieved after a run, a fittest solution is achieved in a generation, a new solution is created, a new generation starts, a new run starts and a genotype is mutated. After adding all the program inputs, fitness function, parameters and callbacks, the configuration callback is fired, with a few useful parameters about the configuration of the program. All the methods and properties of the *CGP* class are listed in figure 5.2.

5.2.2 Structure

There are several components that need to know how many genes are in the genotype, or if a specific gene is a function-gene or a connection gene. Instead of having to determine those properties multiple times and at different stages, this information is only computed once, in this class. The *Structure* class serves as an helper throughout the entire evolutionary process. The main goal is to classify each gene *a priori*, according to its type: connection, paremeter, program output or function. For example, if we have 3 genes per computational node and our genotype starts at number 1 (Matlab does not accept zero-based vectors), we know in advance that gene 1 will represent a

Figure 5.2: Properties and methods for the CGP class.

CGP
- config_ - callbacks_ - functions_ - inputs_ - parameters_
+ addCallbacks(callbacks : struct) + addFitnessFunction(fitness_function: function_handle) + addFunctionsFromPath(path: char array) + addInputs(inputs : struct) + addParameters(parameters : struct) + run() - isValidConstructor_(configuration : struct) - configuration_() - areValidParameters_(parameters : struct)

function and genes 2 and 3 will both correspond to connections. Since this class will be responsible for defining the type of genes, it needs to know a few parameters, such as: the number of genes, the number of genes per node, the connection genes per node, the number of computational nodes and the number of parameters. Figure 5.3 lists the properties and methods of the class.

Figure 5.3: Properties and methods for the *Structure* class.

Structure
+ connectionGenes + parameters + programOutputs + functionGenes
- setConnectionGenes_(genes : double, genes_per_node : double, connections : double) - setParameters_(genes : double, genes_per_node : double, connections : double, nodes : double, parameters : double) - setFunctionGenes_(genes : double, genes_per_node : double) - setProgramOutputs_(genes : double, genes_per_node : double, nodes : double)

5.2.3 EA

The *EA* class is responsible for starting the evolutionary process. It iterates for the maximum number of runs, defined in the configuration of the CGP, storing the fittest

candidate solution of each one.

Figure 5.4: Properties and methods for the *EA* class.

EA
- fittestSolution_ - solutionOfAllRuns_ - configuration_
+ getSolutions() + run() - updateSolutions_(functions : cell array, run : double) - initSolutions_()

If the callback *RUN_ENDED* is provided, it will be fired after each run, with a few parameters, such as the genes of the fittest solution and their fitness.
Figure 5.4 lists the properties and methods of this class.

5.2.4 Run

The *Run* class is responsible for initializing a run. First, it generates a few candidate solutions. Then, it will start the evolutionary loop over the generations. The class stores the best candidate solution, while evaluating if a solution for the problem was found. The properties and methods for the Run class are listed in figure 5.5.
The *Run* class contains two callback events. The FITTEST_SOLUTION occurs when a candidate solution has better fitness than the previous stored solution. The GENERATION_ENDED occurs each time a new generation ends.

Figure 5.5: Methods and properties of the *Run* class.

Run
- configuration_ - fitnessOfAllGenerations_ - fittestSolution_
+ getFittestSolution() + initGenerations() + initOffspring() - solutionNotFound_(fitness_solution : double, fitness_operator : char array) - maxGenerationNotReached_(currentGeneration : double, maxGenerations : double) - fireCallback_(callbackName : char array, activeNodes : array, genes : array)

5.2.5 Generation

The *Generation* class is responsible for initializing a new generation. It starts with the previous fittest candidate solution (parent), and generates a few mutated versions, according to the configuration provided. If the λ chosen in the configuration phase is 4, it will generate four mutated versions of the parent solution. All the new genotypes are evaluated, and the fittest solution is stored. Figure 5.6 lists the methods and properties of this class.

Figure 5.6: Methods and properties of the *Generation* class.

Generation
- configuration_ - fittestSolution_
+ getFittestSolution() + mutate() - isThisSolutionFitterThanParent_(solution_fitness : double, fitness_operator : char array)

The *Generation* class contains two callback events: NEW_SOLUTION_IN_GENERATION and FITTEST_SOLUTION_OF_GENERATION. The first, occurs everytime a new solution is generated. The last one, occurs each time a new solution is generated and has a better fitness than the parent.

5.2.6 Offspring

The *Offspring* class is responsible for the initialization of a specific number of offspring, previously defined, at random, before iterating through the generations. It initializes randomly different genotypes which are then evaluated. The fittest solution is stored and used as the parent solution, for the generation loop initialization.
Figure 5.7 lists the properties and methods of the class.

Figure 5.7: Methods and properties of the *Offspring* class.

Offspring
- configuration_ - candidateSolutions_ - fittestSolution_
+ createOffspring() + getFittestSolution()

5.2.7 Genotype

The *Genotype* class is responsible for the creation of a genotype, restricted to the configuration provided: number of columns, number of rows, number of program inputs, parameters, and so on. First, the function genes are added to the genotype. Then, the connection genes are randomly generated, as well as the parameters and program outputs.
After the genotype is created, the active nodes are recursively found by analysing the program outputs. For each output, the connection nodes are retrieved and stored in an array. For each of those, their connections are also saved in that array, and so on. This process stops until there are no more nodes to analyse. Lastly, the fitness of this new candidate solution is computed.
Figure 5.8 lists the properties and methods of the class.

Figure 5.8: Methods and properties of the *Genotype* class.

Genotype
- configuration_ - activeNodes_ - genes_ - fitness_
+ createGenotype() + getActiveNodes() + getFitness() + getGenes() - findActiveNodes_(sizes : struct, connectionGenes : struct) - createParameters_(sizes : struct, parameters : struct) - createProgramOutputs_(genes : double, outputs : double, nodes : double, shouldBeLastNode : bool) - createFunctionGenes_(functionGenes: array, functionSet : double) - createConnectionGenes_(structure : struct, sizes : struct)

5.2.8 Connection

The *Connection* class is responsible for generating a random and valid connection for a specific node. It receives the connection gene index as argument. The class first finds which node belongs the connection gene. This is done by subtracting the number of program inputs from the gene index and dividing that value by the number of genes per node. Then, it finds all the possible connections for that node. This is achieved by recursively iterating through the previous nodes, taking into account that nodes in the same row cannot be connected between each other, and also taking into account the number of levels-back. Lastly, it randomly pick one connection from the possible connections.
Figure 5.9 lists the properties and methods of the class.

Figure 5.9: Methods and properties of the class.

Connection
-configuration_ - newConnection_ - nodeIndex_ - possibleConnections_
+ createConnection() - findPossibleConnections_(sizes : struct) - findWhichNodeBelongs_(sizes : struct, gene : double)

5.2.9 Functions

The *Functions* class is responsible for randomly generating the function genes for the genotypes. It takes into account the number of functions present in the function-set, to be able to generate valid function genes. It can generate one function gene at a time or multiple function genes. This is is useful, because we find where all the function-genes are positionated in the genotype, and call this class once, which returns function genes to all those positions. If we have 10 nodes, we have to generate 10 function-genes in the genotype. If our function-set is composed by 5 functions, this class will generate 10 random values between 1 and 5, each corresponding to a function-gene mapped to one of the functions in the function-set.
Figure 5.10 lists the properties and methods of this class.

Figure 5.10: Methods and properties of the *Functions* class.

Functions
- configuration_
+ createFunctions()

5.2.10 Output

The *Output* class is responsible for generating a valid program output. Depending on the settings provided initially, this class can pick the last node to be the program output, or randomly pick any program input or computational node in the genotype.
Figure 5.11 lists the properties and methods of the class.

Figure 5.11: Methods and properties of the *Output* class.

Output
- configuration_
+ createOutput()

5.2.11 Fitness

The *Fitness* class is responsible for calling the fitness callback provided in the configuration phase. A few properties are passed to that callback, such as the genes in the genotype, active nodes, function-set, program inputs and others. It has a validation of the type returned by the function, which should return an integer or double value. The returned value, is stored and used as the fitness of that particular candidate solution. Figure 5.12 lists the properties and methods of the class.

Figure 5.12: Methods and properties of the *Fitness* class.

Fitness
- fitness_
+ getFitness()

5.2.12 Mutation

The *Mutation* class receives a genotype and iterates over its genes. All the genes have the same mutation probability. For a gene being mutated, we first find what type of gene it is: connection, function, parameter or program output. If it is a program output, the *Output* class is used. If it is a connection gene, the *Connection* class is used. If it is a function gene, the *Functions* class is used. Recall that when we add parameters to the CGP, we must provide an initialization function and a mutation function. If it is a parameter gene, the mutation function provided will be called.
After iterating all genes, the active nodes are found again, and the fitness is recalculated. If the GENOTYPE_MUTATED callback is provided, it will be called, having as arguments the genes before the mutation, the genes after the mutation and the index of the mutated genes.
Figure 5.13 lists the properties and methods of the class.

Figure 5.13: Methods and properties of the *Mutation* class.

Mutation
- configuration_
- activeNodes_
- genes_
- fitness_
+ createMutation()
+ getGenes()
+ getActiveNodes()
+ getFitness()
+ mutate_()
+ findActiveNodes_()

5.3 Symbolic Regression (Example)

We now show how the toolbox can be used using, as an example, the following classic symbolic regression problem:

$$y(t) = x^6 - 2(x^4) + x^2. \tag{5.1}$$

To start enconding one application built on top of the toolbox, we must import the package **cgptoolbox** into our Matlab workspace. The necessary steps for running the toolbox are the following (see Figure 5.14):

- configure the CGP;
- add the program inputs;
- provide a fitness function;
- add the function-set.

Also, there are two optional steps:

- add callbacks to be executed during the evolutionary process;
- add parameters to the genotype.

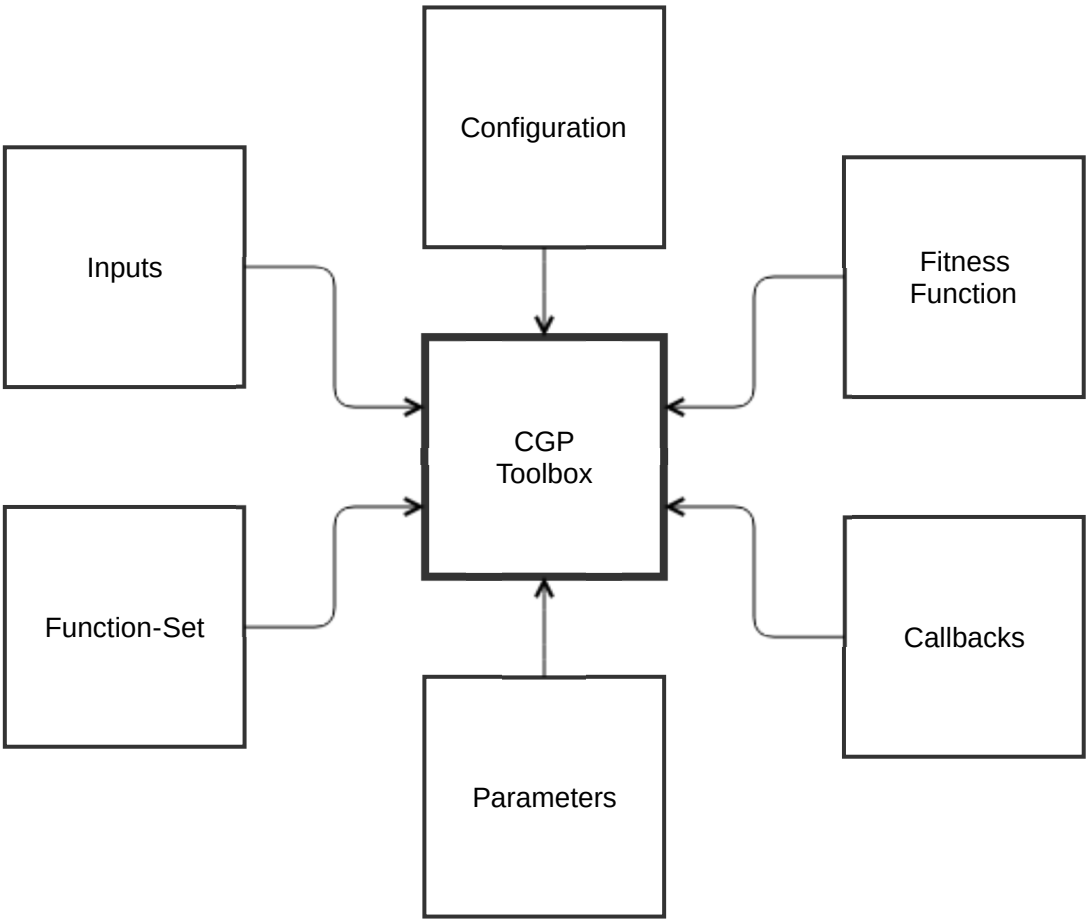


Figure 5.14: Components to provide to the CGP Toolbox.

5.3.1 Configuration

The method for instantiating a new CGP application is by referencing the toolbox **cgptoolbox** and the *CGP* class. Listing 5.1 illustrates one possible configuration (as in Miller and Thomson (2000)), and instantiation of the *CGP* Class:

Listing 5.1: Example of initializing the CGP Class.

```
% create a configuration struct
configuration = struct(
    'rows', 1,              ...
```

```
    'columns', 10,          ...
    'levels_back', 10,      ...
    'output_type', 'random', ...
    'runs', 100,            ...
    'outputs', 1,           ...
    'generations', 8000,    ...
    'offspring', 9,        ...
    'mutation', 0.02,       ...
    'fitness_solution', 0.01, ...
    'fitness_operator', '<=' ...
);

% initialize a CGP instance with a custom configuration
cgp = cgptoolbox.CGP(configuration);
```

5.3.2 Inputs

Since we are trying to solve a symbolic regression problem, we will create 50 points, between -1 and 1, with Equation 5.1. Those points will constitute the program input (see Listing 5.2).

Listing 5.2: Example of adding inputs to the CGP.

```
% initialize CGP instance
cgp = cgptoolbox.CGP(configuration);

% create 50 points of x^6 - 2*(x^4) + x^2, between -1 and 1.
a = zeros(1, 50);
b = zeros(1, 50);
index = 1;
for x = -1:2/50:1
    a(index) = x^6 - 2*(x^4) + x^2;
    b(index) = x;
    index = index + 1;
end

% add program inputs
cgp.addInputs( ...
    struct( ...
```

```
        'points', struct('x', b, 'y', a) ...
    ) ...
);
```

5.3.3 Parameters

The following listing shows how to add two parameters to the genotype. The first is a double, and the second is an integer. This example would not be used in the symbolic regression problem, because there is no need for additional parameters in the genotype in that case.

Listing 5.3: Example of adding two parameteres to the genotype.

```
% initialize CGP instance
cgp = cgptoolbox.CGP(configuration);

% add the parameters to the genotype
cgp.addParameters(
    struct(
        'name', 'some-parameter',
        'initialize', @()rand(),
        'mutate', @(x) x + rand()
    ),
    struct(
        'name', 'constant',
        'initialize', @()randi([-10, 10]),
        'mutate', @mutateParameter
    )
);
```

These parameters will be encoded in the genotype and share the mutation probability with the rest of the genotype's genes. Listing 5.4 shows a possible mutation function for a parameter.

Listing 5.4: Example of one function that doubles the parameter value at every mutation.

```
function newValue = mutateParameter(parameter)
    % mutate the old parameter value to a new one
```

```
    newValue = parameter * 2;
end
```

5.3.4 Function Set

All the functions from the function-set should be under the same directory. The path to this directory is passed to the CGP's public method *addFunctionsFromPath* (see Listing 5.5).

Listing 5.5: Example of adding the path to the function-set.

```
% initialize CGP instance
cgp = cgptoolbox.CGP(configuration);

% set the directory of the function-set
cgp.addFunctionsFromPath('./my-path/function-set/')
```

Listing 5.6 lists four functions used for the symbolic regression problem. Each function is in a separated file, under *'./my-path/function-set/'*.

Listing 5.6: Example of four functions that receive two inputs and do some action with those.

```
% function that sums the first and second input
function result = Sum(x, y)
    result = plus(x, y);
end

% function that subtracts the first and second input
function result = Subtract(x, y)
    result = x - y;
end

% function that divides the first and second input
function result = Divide(x, y)
    if y == 0
        result = x;
    else
        result = x / y;
```

```
    end
end

% function that multiplies the first and second input
function result = Times(x, y)
    result = x * y;
end
```

5.3.5 Fitness Function

The fitness function is provided by callback, which means that we pass the reference of this function to the toolbox. The public method for this is *addFitnessFunction* (see Listing 5.7).

Listing 5.7: Example of passing the Fitness function to the program.

```
% initialize a CGP instance with a custom configuration
cgp = cgptoolbox.CGP(configuration);

% pass the fitness function as reference
cgp.addFitnessFunction(@myFitnessFunction);
```

The fitness function receives a struct with a series of fields that help with the decodification of the phenotype (see Section 5.2.11). The fitness function should return a double or integer for the fitness value. Listing 5.8 shows how to decode a phenotype from the symbolic regression problem, step by step. The goal of the symbolic regression fitness function is to minimize the difference between the output of a candidate program and the required output. The fitness (f) is computed by applying the absolute sum of the errors:

$$f = \sum_{t=1}^{50} |e_t|. \tag{5.2}$$

The lower the fitness, the lower the error and the better this candidate solution.

Listing 5.8: Example of a fitness function.

```
function score = fitness(args)
    score = 0.0;
    values = zeros(1, args.config.sizes.nodes);

    % for each data point or test case
    for j = 1:50
        values(1) = args.programInputs.points.x(j);

        % iterate through active nodes
        for i = args.config.sizes.inputs + 1:size(args.activeNodes, 2)
            % get current active node that we want to decode
            currentActiveNode = args.activeNodes(i);

            % get the gene that points to the function-gene of the active node
            functionGeneOfActiveNode =
                args.config.structure.functionGenes(currentActiveNode);

            % get the function gene of the active node
            currentFunctionGene = args.genes(functionGeneOfActiveNode);

            % get the genes index
            geneFirstConnection =
                args.config.structure.connectionGenes{1}(currentActiveNode);
            geneSecondConnection =
                args.config.structure.connectionGenes{2}(currentActiveNode);

            % get the nodes index
            nodeFirstConnection = args.genes(geneFirstConnection);
            nodeSecondConnection = args.genes(geneSecondConnection);

            % get the values of the connections
            firstConnection = values(nodeFirstConnection);
            secondConnection = values(nodeSecondConnection);

            % call the function which index is given by currentFunctionGene
            values(currentActiveNode) =
                args.functionSet{currentFunctionGene}(firstConnection,
                secondConnection);
        end

        % compute the sum of squared error
```

```
        score = score + abs(values(args.activeNodes(end)) -
            args.programInputs.points.y(j));
    end
end
```

If the genotype is encoded with parameters, those are very easy to extract. Consider the node connection inputs as ci, the number of parameters as np and the number of genes per node as g. If $ci = 2$ and $np = 2$, then $g = 5$: one gene for the function gene, two genes for the node's connection inputs, and two additional genes for the parameters. To recall, the genotype is a sequence of numbers, the genes composing the nodes and the program outputs. Each node starts with the function gene, then the connection inputs and lastly are the parameters. So, if the user wants to extract the parameters relative to node 2, all it needs is to address the last genes from the node. Listing 5.9 is the continuation of Listing 5.8, extended to decode the parameters from the genotype and pass them to each function call. The functions from the function-set should know which parameters to use and which to ignore.

Listing 5.9: Example of decoding the parameters of the current node.

```
for i = args.config.sizes.inputs + 1:size(args.activeNodes, 2)
    % decode the firstConnection and secondConnection
    ...

    % find how many genes per node
    genesPerNode = 3 + args.config.sizes.parameters;

    % genes of the active node
    lastParameter = currentActiveNode * genesPerNode;

    % gene of the first parameter
    firstParameter = lastParameter - args.config.sizes.parameters + 1;

    % get all parameter genes
    allParameters = firstParameter:lastParameter;

    % get the value of each parameter
    for k = 1:size(allParameters, 2)
        parameters(k) = args.genes(allParameters(k));
    end
```

```
    % pass the parameters to the functions
    values(currentActiveNode) =
        args.functionSet{currentFunctionGene}(firstConnection,
        secondConnection, parameters);
end
```

5.3.6 Callbacks

All callbacks receive a struct object with specific properties, relevant to each event.

5.3.6.1 Generation Ended

After running a generation, the callback *GENERATION_ENDED* is fired. This is very useful for knowing the genes present or the fitness value at each generation. The callback accepts a structure with a few fields, such as the current generation and the fitness value. In Listing 5.10, the function will print to the output window the generations and corresponding fitness.

Listing 5.10: Example of passing the GENERATION_ENDED callback function to the program.

```
function myGenerationCallback(args)
    % print current generation and fitness
    fprintf('%d - %.16f\n', args.generation, args.fitness);
end

% initialize a CGP instance with some custom configuration
cgp = cgptoolbox.CGP(configuration);

% pass the callback function as reference
cgp.addCallbacks(struct(
        'GENERATION_ENDED', @myGenerationCallback
));
```

5.3.6.2 Run Ended

At the end of a run, the callback *RUN_ENDED* is fired. This is very useful for knowning the fitness value of each run. The callback accepts a structure with a few fields, such as the current run. In Listing 5.11, the function will print to the output window the run that is currently being processed.

Listing 5.11: Example of passing the RUN_ENDED callback function to the program.

```
function myRunCallback(args)
    % print current run
    fprintf('run: %d\n', args.run);
end

% initialize a CGP instance with some custom configuration
cgp = cgptoolbox.CGP(configuration);

% pass the callback function as reference
cgp.addCallbacks(struct(
        'RUN_ENDED', @myRunCallback
));
```

5.3.6.3 New Solution In Generation

When a new solution is generated a *NEW_SOLUTION_IN_GENERATION* event is fired. This is useful to know exactly which offspring is being evaluated at each time. The callback accepts a structure with a few fields, such as the current fitness and current offspring. In Listing 5.12, the function will print to the output window the offspring that is currently being processed.

Listing 5.12: Example of passing the NEW_SOLUTION_IN_GENERATION callback function to the program.

```
function myNewSolutionCallback(args)
    % print current offspring
    fprintf('current offspring: %d\n', args.offspringIndex);
end

% initialize a CGP instance with some custom configuration
cgp = cgptoolbox.CGP(configuration);

% pass the callback function as reference
cgp.addCallbacks(struct(
        'NEW_SOLUTION_IN_GENERATION', @myNewSolutionCallback
));
```

5.3.6.4 Genotype Mutated

Everytime a solution is created and the genotype is mutated, a *GENOTYPE_MUTATED* event is fired. This event is useful in order to know which genes were mutated. The callback accepts a structure with three fields: genes before mutation, genes after mutation and index of the mutated genes. Example 5.13 shows a function that prints the mutated genes.

Listing 5.13: Example of writing to a file the genes before and after mutation.

```
function genotypeMutatedCallback(args)
    % open a file and store in someFileHandler variable
    before = args.genesBeforeMutation(args.genesMutated);
    after = args.genesAfterMutation(args.genesMutated);
    dlmwrite(someFileHandler, [args.genesMutated(:), before(:), after(:)],
        '-append');
end
```

5.3.6.5 Fittest Solution Found In A Run

When a new solution is better then a previous achieved one, the *FITTEST_SOLUTION* event is fired. The callback accepts a structure with a few fields, such as the current

fitness and current run. Example 5.14 prints to the output window all the runs that obtained a better candidate solution.

Listing 5.14: Example of passing the FITTEST_GENERATION callback function to the program.

```
function myFittestSolutionCallback(args)
    % print current generation and fitness
    fprintf('new fittest solution at run: %d \n', args.run);
end

% initialize a CGP instance with some custom configuration
cgp = cgptoolbox.CGP(configuration);

% pass the callback function as reference
cgp.addCallbacks(struct(
        'FITTEST_SOLUTION', @myFittestSolutionCallback
));
```

5.3.6.6 Fittest Solution Of A Generation

When iterating through the offspring, if a solution is better than the previous one, the event *FITTEST_SOLUTION_OF_GENERATION* is fired. The callback accepts a structure with a few fields, such as the current fitness and current offspring. Example 5.15 prints to the output window the new fittest offspring index.

Listing 5.15: Example of passing the FITTEST_SOLUTION_OF_GENERATION callback function to the program.

```
function myFittestSolutionOfGenerationCallback(args)
    fprintf('new fittest offspring: %d \n', args.offspringIndex);
end

% initialize a CGP instance with some custom configuration
cgp = cgptoolbox.CGP(configuration);

% pass the callback function as reference
cgp.addCallbacks(struct(
        'FITTEST_SOLUTION_OF_GENERATION',
            @myFittestSolutionOfGenerationCallback
));
```

Chapter 6

CGP approach to Pitch Estimation

In our CGP approach to Pitch Estimation, we have multiple inputs and we have only one row of graph nodes, one output (the result of the corresponding classifier), and *levels-back* = *nc*. To perform pitch detection using CGP, we developed a system where some important decisions and tasks were made besides the CGP. We had to define what kind of inputs to use from the original piano audio signal, through a pre-processing task. We also had to develop a process to reach a binary output in order to perform our fitness function.

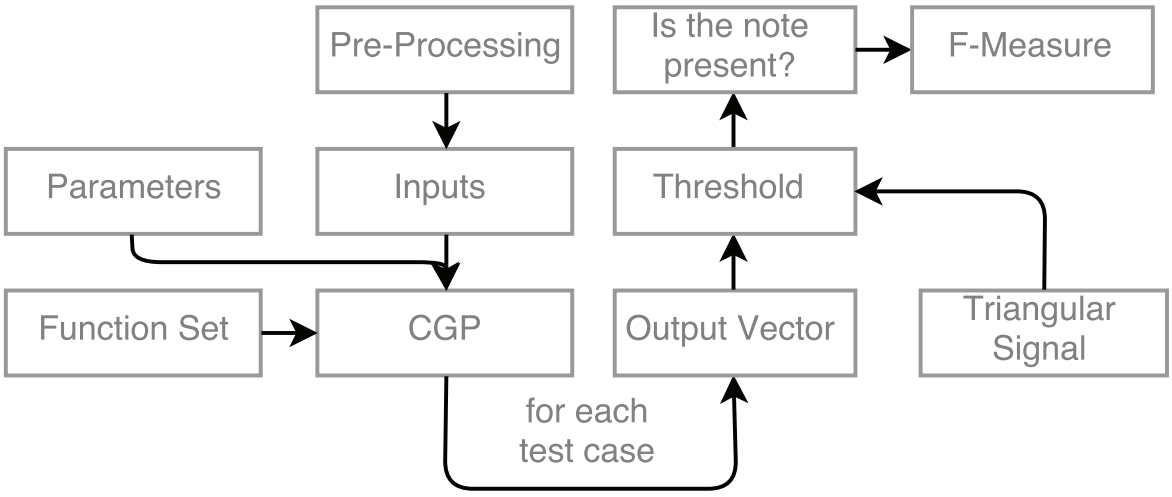


Figure 6.1: System architecture.

The block diagram of our proposed system is much more than a simple CGP process

and is depicted in Figure 6.1. Our goal is to train 61 classifiers, each one corresponding to one pitch or piano note: from C1 to C6. To train one classifier, we first start with a set of learning cases: a group of audio signals corresponding to the pitch that we want to identify and a group of audio signals without that pitch. Those audio signals are pre-processed in order to extract some important features that will be used as program inputs like for example, the magnitude spectrum. The computational nodes in the genotype have two connection inputs, one function and two parameters. Each program is an evolved mathematical function, which is applied to each of the learning cases. The output of that function is compared to a triangular signal, where a threshold is applied, for binary classification. After the binary classification of all learning cases, the fitness function is applied.
To tackle the problem of Pitch Estimation, we started by addressing a simplified version of the problem, and then we increased its difficulty. First, we started with simple mathematical models: sine waves, sawtooth waves (triangular wave) and square waves. Then, we moved to real audio monophonic recordings. Lastly, slight changes were made to improve the results obtained previously, which are discussed in the last section. The following sections describe our three approaches (from working with simple mathematical signals to dealing with real audio) in detail and present the obtained results.

6.1 General Approach

The common configurations used for all the experiments, is described carefully throughout this section.

6.1.1 Inputs

The algorithm inputs are obtained from the preprocessing system (described in each of the experiments sections), which returns a set of audio signals used in the training phase of our approach. Those are vectors in time with a sample rate of 44.100 samples per second. Each audio signal is split into time frames and transformed into frequency domain using a DFT. The time-frame was set to 4096 samples. With this settings, we have a frequency resolution of $\frac{44100}{4096} = 10.76$Hz and a time resolution of $\frac{4096}{44100} = 93$ milliseconds. This lets us identify notes in small portions of time while having a good

frequency resolution to extract the frequency components.
By applying the DFT, we get frames with 2048 frequency bins with complex domain numbers, each one representing a time frame of a signal with 96 milliseconds duration. From this vector $X[k]$ in the frequency domain we may use two different representations of a complex number, cartesian and polar:

1. $\Re\{X[k]\}$
2. $\Im\{X[k]\}$
3. $\measuredangle\{X[k]\}$
4. $||X[k]||$

We have a couple of vectors each one with two components making four usable program inputs. Two from the cartesian representation: the real and imaginary parts, and two from the polar representation: magnitude and phase. By having redundant information, regarding the four inputs, we ensure the CGP system has a variety of representations of the same data, so it can be able to choose the one which best fits the problem.

6.1.2 Individual Encoding

Our proposed algorithm contains four program inputs and one single program output. We used a single row CGP configuration. The number of columns differ from experience to experience. Each computational node contains five genes (see Figure 6.2): two connection inputs, one function gene and two function parameters.

6.1.3 Function parameters

The functions of the instruction set may have up to two parameters, because most functions need real parameters to perform their tasks. These parameters also evolve during the training process. We used two real parameters for each computational node. This way, each function has its particular parameters with a particular meaning. Each parameter r_1, r_2 of each function has its own range. Those intervals are normalized

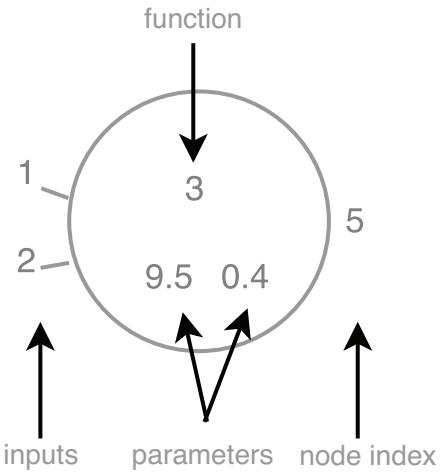


Figure 6.2: Node Genes(5): inputs, code function and real parameters.

into $[0, 1]$: all intervals are transformed from $[a, b]$ to a normalized one $[0, 1]$. By using this technique, the actual value of any parameter can bee seen as a number between 0 and 1 or a percentage of the interval.

6.1.4 Program Output

Since our system is a classifier, the final program output must return a binary value. In order to transform the output vector into a binary value, our system uses a threshold after a comparison between the program output and a base signal. This base signal is a triangular signal centered on frequency bin of the corresponding F0 of the note classifier. The threshold function has a constraint to ensure that it is the last function before the binary output. This threshold is also evolved: parameter θ mutates from an initial configurable value with an also configurable mutation probability.

6.1.5 Mutation

In CGP, mutation plays a major role on the evolution process. Here, each gene may be subject to mutation according to a configurable parameter: the mutation probability. This parameter represents the probability of each gene to be mutated. For instance, $p = 0.01$ means that each gene mutates with a probability 0.01. In our case, different mutation processes are used according to the gene type and domain: if a function gene happens to be mutated, then a valid value must be chosen for selecting a new function

in the function set lookup table; if a mutation occurs in a connection gene, then a valid value is the index of any previous node in the genotype or any program input; the valid values for the program output are the index of any node in the genotype or the address of a program input. All these mutations happen according to the uniform probability distribution function for integers. Two additional genes can also mutate: the real parameters used by the functions of the function set. According to each function, each parameter has a specific meaning and also has its own domain range where it can variate. In this case, we take any value in the normalized interval $[0, 1]$ and transform it into the real interval $[a, b]$. The mutation of the real genes (function parameters) is done using the normal distribution in order to address the entire range:

$$f(x) = \frac{e^{-(x-\mu)^2/(2\sigma^2)}}{\sigma\sqrt{2\pi}}, \tag{6.1}$$

where $f(x)$ represents the density function of x variable, with a normal distribution. This function is also represented as $N(\mu, \sigma)$, where μ is the mean and σ is the standard deviation. To perform the mutation of a function parameter, r_{old}, we generate a new random r_{mutate} using the normal distribution $N(\mu = r_{old}, \sigma)$, with σ being configurable in our system. This way, we ensure that when a mutation occurs in a real parameter, all the parameter interval is reachable, but with higher probability to mutate to closer values.

As previously described in Section 6.1.4, the program output is obtained by using a threshold value in order to have a binary output. This threshold mutates independently from the genotype genes, with a different probability. It has 50% chances of being mutated. Based on a few experiments, we set the initial value of the threshold at 0.5. This value is mutated with step increments of 0.01. If it is marked for mutation, it has 50% chances for increment and 50% for decrement its value (this was later adapted to evolve using a normal distribution - see Section 6.4).

6.1.6 Fitness Function

The main goal of the proposed system is to evolve a classifier for each pitch. The output of each classifier is binary: when the corresponding note is detected the output is 1, otherwise it is 0. During the evolutionary process, we used as program inputs an amount of signals with the desired pitch (fundamental frequency) and the same amount of signals with different pitches. Thus, for each classifier, we used 50 signals

with the desired pitch and 50 signals with different pitches. If a signal with the desired pitch is identified by the classifier, it counts as a true positive (tp). If a signal with the desired pitch is not identified by the classifier, it counts as a false negative signal (fn). If a signal with different F0 from the desired pitch is identified by the classifier, it counts as a false positive signal (fp). If a signal with different F0 from the desired pitch is not identified by the classifier, it counts as a true negative signal (tn). During the evolutionary process, the evaluation of each individual (classifier) is done using F-measure, Equation (6.2)

$$F_{measure} = 2 \times \frac{recall \times precision}{recall + precision}, \tag{6.2}$$

where:

$$precision = \frac{tp}{tp + fp}, \; recall = \frac{tp}{tp + fn}. \tag{6.3}$$

One of our system peculiar characteristics is the binaritazion process, since the CGP output is a signal vector processed and filtered. In order to accomplish a binary output, where 1 means the presence of the corresponding pitch in the analyzed frame and 0 means the opposite, we used a comparison process between the CGP ouput vector normalized in amplitude $O_{CGP}(n)$ and a base signal with the frequency corresponding to the pitch of the estimator, $B_{F0}(n)$. The first step is the normalization of the output vector in amplitude. This way all the elements of the vector fall in the interval [0,1]. The base signal is obtained with a triangular mask on frequency domain around the F0 of the estimator. We used a triangular mask with three configurable points in both in size and amplitude. We then generate the following scalar:

$$x = \sum_{n=0}^{N} O_{CGP}(n) * B_{F0}(n), \tag{6.4}$$

where x measures the interception between the two discrete time signals. If we approximate these signals to continuous time we could see x as the intersected area between the two signals. Finally, we used a threshold function to accomplish the binary result:

$$T(x) = \begin{cases} 1, & \text{if } x > \theta \\ 0, & \text{if } x <= \theta \end{cases} \tag{6.5}$$

where θ is the threshold value. Since both signals are normalized, the max value for x is:

$$x = \sum_{n=0}^{N} B_{F0}(n). \tag{6.6}$$

6.2 Pitch Estimation of Mathematical Functions: Sine, Square and Sawtooth Waves

Our first approach started with the application of the CGP to simple mathematical models. These are artificial signals, which were not recorded from real world instruments. We first wanted to validate that the application of CGP to Pitch Estimation is a valid method and that our approach works with simple and pure signals. Those type of signals were chosen to be analysed first, because its characteristics, such as the harmonic structure, are well known. There are no problems with reverberation, transients and background noise; although we added some white noise in the pre-processing stage, it is a controlled noise (later discussed in Section 6.2.1).

6.2.1 Preprocessig

We used three types of signals: sine wave, sawtooth (triangular) wave and square wave. To recall what we presented in Section 2.4.3, the sine wave is given by the following equation:

$$y(t) = A \times sin(2\pi ft + \varphi), \tag{6.7}$$

where A is the amplitude of the wave, f is the number os oscillations per second and φ is the phase. In our experiments, A is set to 1 and f is the fundamental frequency of the pitch that we want to classify. To train the classifier of pitch 60, which has as fundamental frequency 261.6 Hz, the Equation 6.8 takes the form of:

$$y(t) = sin(2\pi \times 261.6 \times t + \varphi). \tag{6.8}$$

The sawtooth wave can be approximated by the Fourier series:

$$y(t) = \frac{A}{2} - \frac{A}{\pi} \sum_{k=1}^{\infty} (-1)^k \frac{\sin(2\pi k f t)}{k}. \tag{6.9}$$

To train the classifier of pitch 60, the Equation 6.10 takes the form of:

$$y(t) = 0.5 - \frac{1}{\pi} \sum_{k=1}^{\infty} (-1)^k \frac{\sin(2\pi k \times 261.6 \times t)}{k}. \tag{6.10}$$

The square wave is given by the following equation:

$$y(t) = \frac{4}{\pi} \sum_{k=1}^{\infty} \frac{sin(2\pi(2k-1)ft}{2k-1}. \tag{6.11}$$

Adapting for the classifier of pitch 60, we have:

$$y(t) = \frac{4}{\pi} \sum_{k=1}^{\infty} \frac{sin(2\pi(2k-1)261.6 \times t}{2k-1}. \tag{6.12}$$

Figure 6.3 shows the three types of waves signals used.

In order to increase the difficulty of the algorithm, we apply an additive white Gaussian noise (AWGN), which is a basic noise model used to mimic the effect of many random processes that occur in nature. It adds noise with uniform power across the frequency band to our signal.
Figure 6.4 shows the three types of waves signals used after the application of an AWGN.
To accomplish a good base for signal processing common tasks, we transform the domain signals from the time domain to the frequency domain, using the Discrete Fourier Transform (DFT), shown in Equation 2.19. In order to do so, each frame is

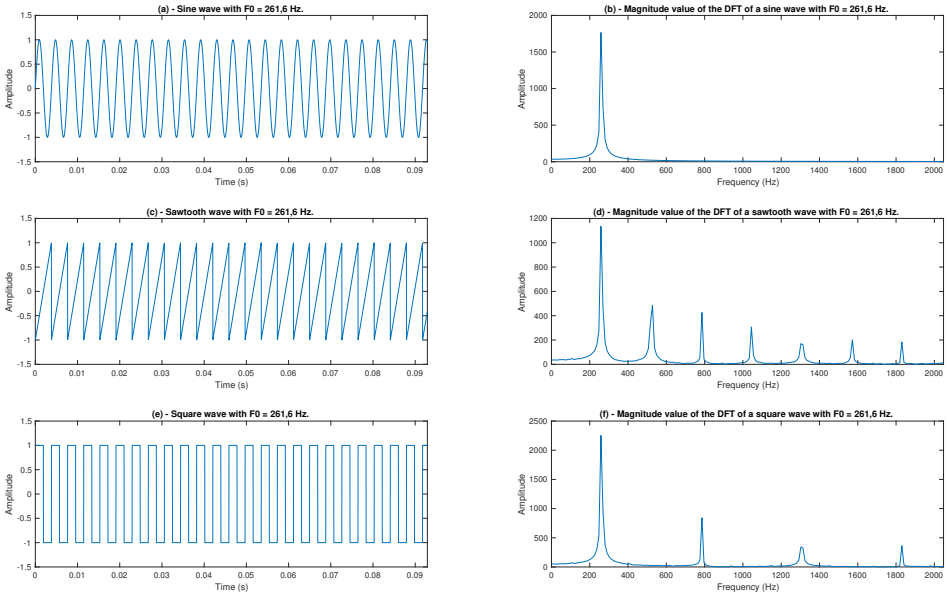


Figure 6.3: Types of signals: (a) sine wave in time-domain, (b) sine wave in frequency domain, (c) sawtooth wave in time domain, (d) sawtooth wave in frequency domain, (e) square wave in time domain, (f) square wave in frequency domain.

windowed (see Equation 6.13) using an Hanning window (see Equation 6.14) to avoid spectral leakage. Then, the DFT is applied, obtaining signal frames in the frequency domain. Figure 6.5 shows the steps for preprocessing.

$$x_w[n] = w[n].x[n]. \tag{6.13}$$

$$w[n] = 0.5\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right). \tag{6.14}$$

Recall that any signal cannot be uniquely represented for frequencies above $\frac{f_s}{2}$, where f_s is the sampling frequency of the sequence (see Section 2.2.2). Above $\frac{f_s}{2}$ all signal energy is reflected back into the frequency range $-\frac{f_s}{2}$. Between $\frac{f_s}{2}$ and f_s, the reflection is in reverse order, which gives rise to a DFT frequency domain period of $[0, f_s]$.
Due to the DFT spectrum properties, the symmetry of the real part and the anti-symmetry of the imaginary part relative to the Nyquist frequency $\frac{f_s}{2}$, we may only use half of the resulting signal of the DFT, so only the first 2048 frequency bins were considered.

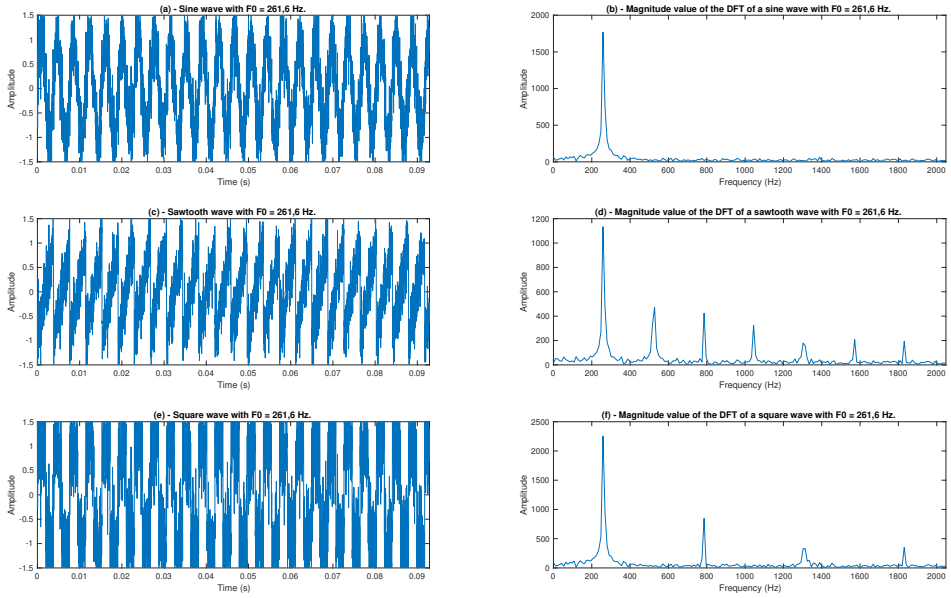


Figure 6.4: Types of signals with AWNG applied: (a) sine wave in time-domain, (b) sine wave in frequency domain, (c) sawtooth wave in time domain, (d) sawtooth wave in frequency domain, (e) square wave in time domain, (f) square wave in frequency domain.

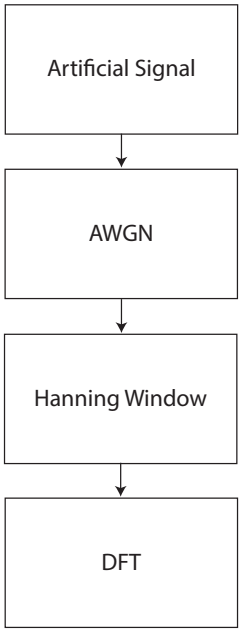


Figure 6.5: First, the mathematical models are created. Then, the additive white Gaussian noise is added, the Hanning window is applied and the DFT transforms the time domain signal into the frequency domain.

6.2.2 Function set

Each function node has a gene that indicates which function of the function set should be used on that node. Table 6.1 shows the function set or look-up table that was used in the experiments. The maximum function arity is 2. Note that all the functions are prepared to receive one or two vectors, and all of them return a vector. The returned vector is then normalized between 0 and 1.

Table 6.1: Function set - lookup table.

Index	Function	Description
1	SPConvolution	Convolution
2	SPCos	Cosine Function
3	SPDivide	poit to point Division
4	SPFFT	Absolute value of the DFT
5	SPGaussfilter	Gaussian filter
6	SPIFFT	Absolute value of Inverst DFT
7	SPLog	Natural logarithm
8	SPLog10	Common logarithm
9	SPMedFilter	Median filter
10	SPMod	Remainder after division
11	SPMulConst	Multiplication by constant
12	SPNormalizeMax	Normalization maximum
13	SPNormalizeSum	Normalization sum
14	SPPeaks	Find peaks
15	SPSin	Sine Function
16	SPSubtract	Subtraction
17	SPSum	Sum
18	SPSumConst	Sum with a constant
19	SPThreshold	tresholding
20	SPTimes	Multiplication

Our function set is basically composed of filtering operations on vectors and of arithmetic operations with constants and vectors. Each function may use one or two real parameters that are encoded in genotype as well. There are functions that do not need such parameters, so they are ignored.

SPConvolution performs a convolution between the first input vector and the second

input vector.

SPCos applies the cosine function to each data point in the first vector and returns it.

SPDivide divides each point in the first input vector by each data point in the second input vector.

SPFFT applies the DFT to the first input vector, and returns its absolute value.

SPGaussfilter applies a Gaussian filter to the first input vector. Sigma (first parameter) can evolve between 1 and 10, depending on the interval parameter encoded in the node.

SPIFFT applies the Inverse DFT to the first input vector, and returns its absolute value.

SPLog returns the natural logarithm of the first input vector.

SPLog10 returns the absolute value of the common logarithm applied to the first input vector.

SPMedFilter applies a median filter to the first input vector. The percentage of signal that will be filtered is in a range between 3% to 10%, depending on the first parameter.

SPMod returns the remainder after division of the first input vector by the second input vector.

SPMulConst return the multiplication of the first vector by a constant (second parameter).

SPNormalizeMax returns the normalization of the first input vector by the maximum value in the vector.

SPNormalizeSum returns the normalization of the first input vector by sum.

SPPeaks returns the first input vector with all non-peak values set to 0.

SPSine applies the sine function to each data point in the first vector and returns it.

SPSubtract returns the subtraction of the first input vector by the second input vector.

SPSum returns the sum of the first input vector with the second input vector.

SPSumConst returns the sum of the first input vector with a constant (second parameter).

SPThreshold sets all values of the first vector below a certain threshold (first parameter) to zero, and returns its value.

SPTimes multiplies the first vector by the second vector.

6.2.3 Experiments and Results

Since this is our first approach to the problem of Pitch Estimation using CGP, our main goal is to show the applicability of CGP on this problem. We evolved one classifier for each pitch which is represented by the corresponding MIDI note number, being 60 the MIDI note number corresponding to the C4 musical note (the middle C). Table 6.2 shows the values of the configurable parameters for our system.

Table 6.2: List of parameters used in the experiments.

Parameter	Value
Frame Size	4096
Fitness Threshold	0.5
Positive Test Cases	50
Negative Test Cases	50
Program Outputs	1
Rows	1
Columns	50
Levels Back	50
Offspring	4
Mutation Probability	5%
Runs	10
Generations	500

The evolutionary process consisted of 10 runs with 500 generations each, using 50

positive and 50 negative cases. The classifiers were evaluated using the F-measure (Equation 6.2). Table 6.3 lists all the classifiers and their best runs. All achieved fitness of 1, which is the maximum possible fitness value. That is, all the evolved classifiers give a correct answer for all the given inputs.

6.3 Moving to Real Audio Recordings

Since we obtained good results with the application of CGP to simple mathematical models, we moved to real monophonic signals. We used the piano samples (audio signals) from the MAPS database Emiya et al. (2010b). This is a huge dataset with multiple piano samples, chords and melodies in wave format.
Classifying mathematical signals is different from classifying real sounds: we have to take into consideration the noise and the harmonic structure.

6.3.1 Approach to Real Audio Signals

The piano sound signals are vectors in time with a sample rate of 44.100 samples per second. For the preprocessing task, we split each piano sound signal into frames of 4096 samples width, corresponding to 96 milliseconds. We followed the same preprocessing tasks from the previous experiments: application of the Hanning window, followed by the DFT.

Table 6.3: Training results for 61 classifiers for pure signals.

classifier	generation	fitness
24	68	1
25	35	1
26	145	1
27	61	1
28	71	1
29	19	1
30	15	1
31	71	1
32	57	1
33	19	1
34	18	1
35	79	1
36	48	1
37	49	1
38	54	1
39	61	1
40	183	1
41	69	1
42	120	1
43	31	1
44	213	1
45	100	1
46	46	1
47	54	1
48	22	1
49	196	1
50	120	1
51	105	1
52	107	1
53	23	1
54	20	1
55	165	1
56	54	1
58	24	1
59	16	1
60	56	1
61	32	1
62	76	1
63	48	1
64	42	1
65	21	1
66	89	1
67	76	1
68	43	1
69	34	1
70	153	1
71	11	1
72	27	1
73	16	1
74	24	1
75	13	1
76	8	1
77	7	1
78	9	1
79	9	1
80	6	1
81	9	1
82	9	1

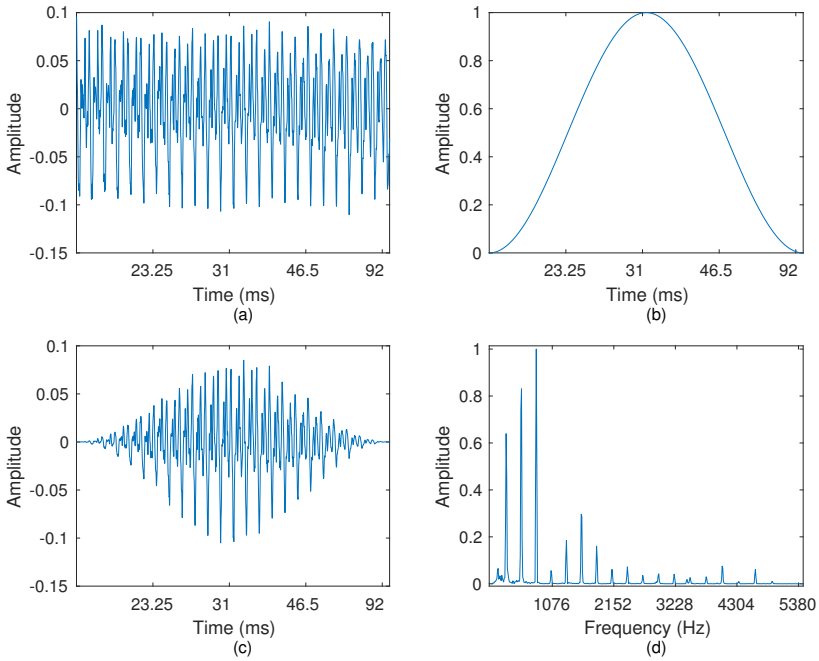


Figure 6.6: Preprocessing process: (a) input time signal piano note, (b) Hanning window, (c) resulting windowed signal, (d) frequency domain signal.

Figure 6.6 shows the preprocessing stages for a real audio piano note, from the input time signal to one of the program inputs in the frequency domain: the magnitude spectrum. As we can see, the frequency spectrum is very different from a pure sine wave, square wave or sawtooth wave. Many musical instruments produce sounds with smooth spectral envelopes but differ immensely in their shapes. In Figure 6.6, we can observe that the strongest harmonic is the third one, not the F0.
The problem complexity rises and we have to deal with different spectral envelopes from different pianos, played in different conditions, inharmonic partials, spurious components, transients and reverberation (see Section 2.6). The program inputs are obtained in the same way as in the previous approach (see Section 6.1.1).

Table 6.4: Function set lookup table

Index	Function	Description
1	SPBPGaussFilter	band pass Gaussian filter
2	SPHighPassFilter	high pass filter
3	SPLowPassFilter	low pass filter

The function-set used in the experiments with artificial signals was extended, by adding three more filters, to give more options for tackling those problems (see Figure 6.4).

The filters are a band pass Gaussian filter, a high pass filter an a low pass filter. These can be helpful to filter out frequency components that are not relevant for the identification of the fundamental frequency.

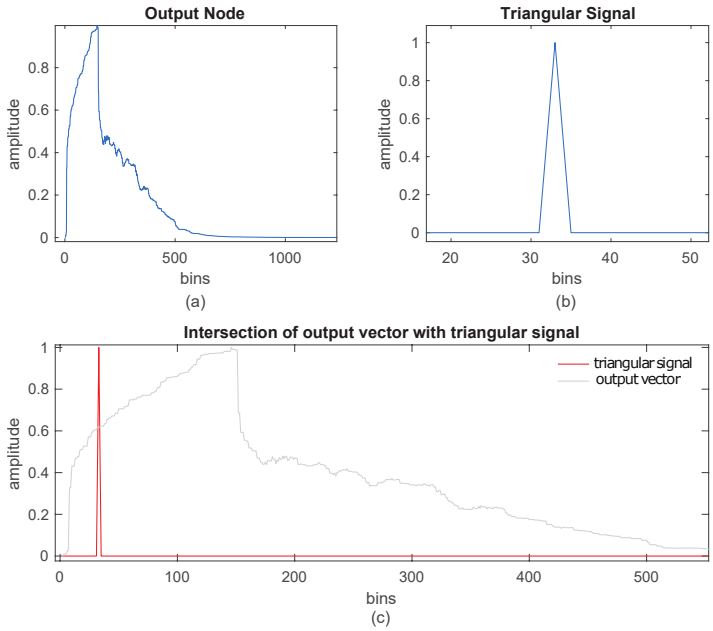


Figure 6.7: (a) CGP output signal, (b) base triangular signal (c) computing intersection for threshold.

The method for computing fitness is the same. All the fitness computation process including the sum and the thresholding is illustrated in Figure 6.7 with tree graphs.

6.3.2 Experiments and Results

Table 6.7 shows the values of the configurable parameters for our system. The evolutionary process consisted of 30 runs with 5000 generations each, using 50 positive and 50 negative cases. The number of computational nodes is increased, from 50 to 100. The classifiers were evaluated using the F-measure (Equation 6.2).

Table 6.5: List of parameters used in the experiments.

Parameter	Value
Frame Size	4096
Fitness Threshold	0.5
Positive Test Cases	50
Negative Test Cases	50
Outputs	1
Rows	1
Columns	100
Levels Back	100
Offspring	4
Mutation Probability	5%
Runs	30
Generations	5000

Figure 6.8: Training results obtained during 30 runs for pitch 60. Fitness values were calculated using F-measure.

Figure 6.8 shows the individual results of each run (i.e. evolved pitch estimator) for classifying note 60. The best generated classifier was chosen as the pitch estimator for note 60.

Figure 6.9 depicts the resulting program of this classifier. It shows the functions used in each node as well as the inputs of each node. For instance: node 101 contains the Gaussian filter function and the function arguments (inputs) are the outputs of the nodes 36 and 87. The resulting program is a set of mathematical functions, with

```
Node 1 = input (1)
Node 2 = input (2)
Node 3 = input (3)
Node 4 = input (4)
Node 5 = SPConvolution ( 1 , 1 )
Node 6 = SPFFT ( 1 , 4 )
Node 7 = SPConvolution ( 4 , 4 )
Node 8 = SPTimes ( 5 , 7 )
Node 10 = SPSum ( 2 , 1 )
Node 11 = SPIFFT ( 10 , 4 )
Node 12 = SPPeaks ( 8 , 11 )
Node 13 = SPPeaks ( 11 , 10 )
Node 14 = SPSum ( 3 , 10 )
Node 15 = SPSubtract ( 1 , 13 )
Node 16 = SPAbs ( 13 , 12 )
Node 17 = SPLog10 ( 16 , 10 )
Node 18 = SPThreshold ( 5 , 16 )
Node 19 = SPCos ( 17 , 18 )
Node 20 = SPLog ( 8 , 2 )
Node 23 = SPNormalizeSum ( 6 , 14 )
Node 24 = SPAbs ( 13 , 13 )
Node 33 = SPDivide ( 12 , 23 )
Node 36 = SPHighPassFilter ( 19 , 20 )
Node 87 = SPSum ( 15 , 33 )
Node 101 = SPGaussfilter ( 36 , 87 )
Node 104 = SPIFFT ( 24 , 101 )
```

Figure 6.9: Evolved classifier code for pitch 60.

specific parameters, over vectors - our phenotype.

After the training process, each classifier was tested with a different test set. Each test set consisted in 144 negative notes (48×3) and 5 positive notes, comprising a total of 149 piano sound samples. Table 6.6 shows our results. As a first approach with real signals, these results are very encouraging, since for almost all notes we achieved a classifier with F-Measure values greater than 70%.

Table 6.6: Test results for 61 classifiers

classifier	tp	tn	fp	fn	f-measure
24	5	138	6	0	0.63
25	5	127	17	0	0.37
26	5	127	17	0	0.37
27	5	127	17	0	0.37
28	5	127	17	0	0.37
29	5	124	20	0	0.33
30	4	122	22	1	0.26
31	5	108	36	0	0.22
32	5	132	12	0	0.46
33	4	138	6	1	0.53
34	5	111	33	0	0.23
35	5	139	5	0	0.66
36	5	140	4	0	0.71
37	5	121	23	0	0.30
38	5	140	4	0	0.71
39	5	113	31	0	0.24
40	4	130	14	1	0.35
41	4	138	6	1	0.53
42	4	124	20	1	0.28
43	4	138	6	1	0.53
44	5	112	32	0	0.24
45	5	135	9	0	0.53
46	3	138	6	2	0.43
47	4	119	25	1	0.24
48	5	135	9	0	0.53
49	5	136	8	0	0.55
50	5	140	4	0	0.71
51	5	127	17	0	0.37
52	5	138	6	0	0.63
53	5	142	2	0	0.83
54	5	128	16	0	0.39
55	5	138	6	0	0.63
56	5	128	16	0	0.39
57	5	139	5	0	0.67
58	5	139	5	0	0.67
59	5	137	7	0	0.59
60	5	142	2	0	0.83
61	4	142	2	1	0.73
62	4	144	0	1	0.88
63	4	144	0	1	0.88
64	5	138	6	0	0.63
65	5	141	3	0	0.77
66	5	139	5	0	0.67
67	5	141	3	0	0.77
68	5	141	3	0	0.77
69	5	141	3	0	0.77
70	5	142	2	0	0.83
71	5	142	2	0	0.83
72	5	142	2	0	0.83
73	5	144	3	0	0.77
74	5	146	1	0	0.91
75	5	142	5	0	0.66
76	5	142	5	0	0.66
77	5	146	1	0	0.91
78	5	143	4	0	0.71
79	5	144	3	0	0.77
80	5	143	4	0	0.71
81	5	147	0	0	1
82	5	147	0	0	1
83	5	146	1	0	0.91
84	5	145	2	0	0.83

In order to compare our results with the ones by other authors we generated the graph depicted in Figure 6.10 where, besides F-measure, we can see the error rate in percentage for the data-set test with 96ms frames. The three main monophonic pitch estimators are: Parametric F0 estimator (Emiya et al. (2007)), the Non-parametric F0 estimator and the YIN estimator (De Cheveigné and Kawahara, 2002) and those estimators have mean error rates of 2.4%, 3.0% and 11.0%, respectively. Our Cartesian Genetic Programing approach to F0 estimation reaches the mean error rate of 6%.

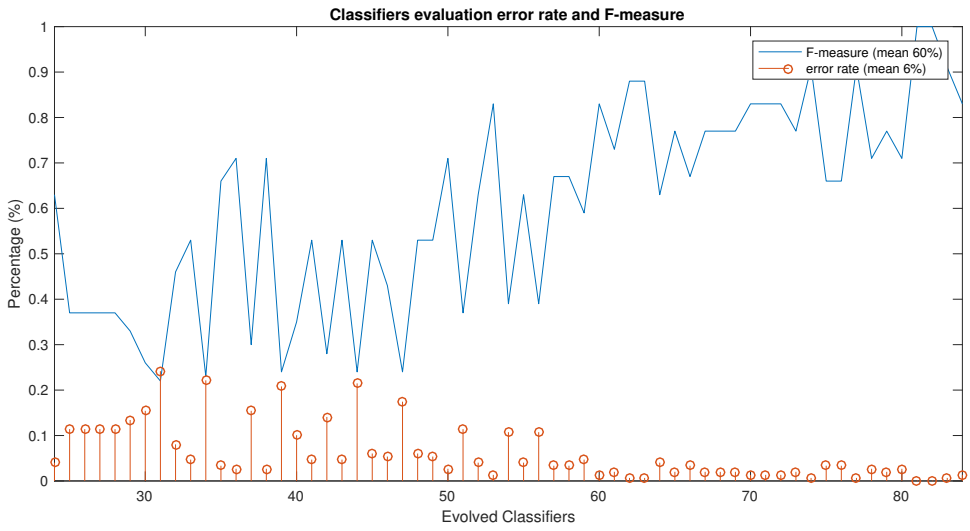


Figure 6.10: Graph with 61 classifiers evaluation results in error rate and F-measure.

6.4 Improvements on Real Audio Recordings

As can be seen in Table 6.6, we only have 12 false negatives, which means that, from the 305 positive test cases (61 classifiers x 5 positive test cases for each classifier), only 12 were not correctly identified. In contrast, we have a bigger number of false positives, specially on the lower notes. This is the major problem with the results obtained in this first approach with real signals. One possible reason for this, is that in the binary threshold, we compare the output vector with a base signal, and this base signal is a triangular signal with its peak on the F0 of the current classifier. That is, the F0 peak on the base signal is being matched by harmonics in pitches different from the classifier.

In order to mitigate the problem identified after the first experiments with real signals, we decided to add a second triangle to the base signal, with a peak in the second harmonic. The first triangular mask remains the same, with values [0.5; 1; 0.5] and the second has an amplitude of [0.2; 0.3; 0.2].

Figure 6.11 shows the intersection of the normalized magnitude spectrum of one piano note with pitch 60 that serves as one of the program inputs to CGP, and its base signal. Two triangles are centerered on its F0 (261.6Hz) and second harmonic (532.2 Hz). The intersection of those two vectors gives a value of 1.42. As we increase the base signal, we have to increase also the threshold. Before, its initial value was set to 0.5. Now, the initial value of the threshold is set to 1.5. The goal of CGP is to evolve a mathematical function that salients the peaks on F0 and second harmonic of the signal.

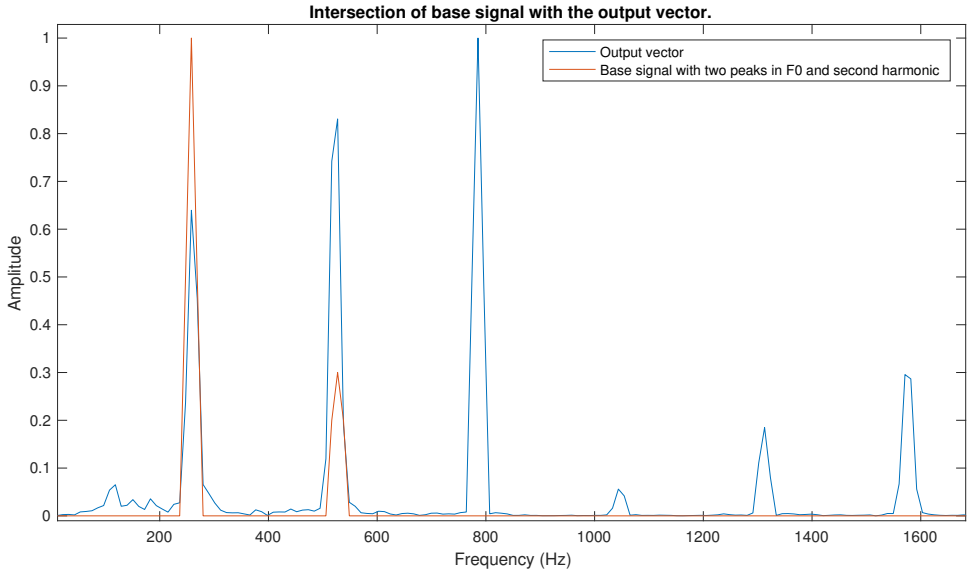


Figure 6.11: Intersection of magnitude spectrum of one piano note with pitch 60 and normalized, with its base signal. Two triangles are centerered on its F0 (261.6Hz) and second harmonic (532.2 Hz).

In the first approach, the mutation of the threshold was a simple step increment of +0.01 and -0.01. Besides, adding a second triangle to the base signal, we also changed the step increment to a normal distribution in order to verify if this could lead to some improvement in the results (see Equation 6.1).

6.4.1 Experiments and Results

The experiments conditions are listed in Table 6.7. Each classifier takes almost a day to evolve with 30 runs. Due to short time, we could not fullfill the table with all 61 classifiers. We concentrate mainly on the lower pitches, where the amount of false positives is higher.

Table 6.7: List of parameters used in the experiments.

Parameter	Value
Frame Size	4096
Fitness Threshold	1.5
Positive Test Cases	50
Negative Test Cases	50
Outputs	1
Rows	1
Columns	100
Levels Back	100
Offspring	4
Mutation Probability	5%
Runs	30
Generations	5000

Table 6.8 lists a few experiments and the results are encouraging, where the lowest notes reduce drastically the number of false positives, when compared to the first approach. The mean F-measure of the same classifiers, from the previous experiments, was 0.48, whereas in current experiments is 0.57: it was improved, approximately, 9%. The number of false positives was reduced from to 201 to 135, which corresponds, approximately, to 33%. These are very good indicators that the CGP can have a valid application for Pitch Estimation.

Table 6.8: Test results for classifiers by intersecting the output vector with two triangles.

classifier	tp	tn	fp	fn	f-measure	f-measure of previous experiments
24	5	139	5	0	0.66	0.63
26	5	129	15	0	0.40	0.37
29	4	131	13	1	0.36	0.33
30	5	116	28	0	0.26	0.26
31	5	129	15	0	0.40	0.22
32	5	137	7	0	0.59	0.46
34	5	127	17	0	0.37	0.23
39	5	122	22	0	0.31	0.24
48	5	141	3	0	0.77	0.53
53	5	143	1	0	0.91	0.83
55	5	139	5	0	0.66	0.63
57	5	141	3	0	0.77	0.67
60	5	143	1	0	0.91	0.83

6.5 Applying Classifiers to Polyphonic Audio Recordings

This section was not one of the objectives of this thesis, but we find interesting to apply the classifiers, trained with monophonic audio signals, to polyphonic audio signals. For this, we also used the piano samples from the MAPS database Emiya et al. (2010b). Our test set is composed of 20 audio recordings of chords, where the classifier's pitch is present and 75 chords without the classifier's pitch. As we can see in Table 6.9, the results show room for improvements and future work, however, they also validate that our particular approach is a good starting point for evolving classifiers using CGP.

Table 6.9: Test results for classifiers applied to polyphonic recordings.

classifier	**tp**	**tn**	**fp**	**fn**	**f-measure**
24	15	50	25	5	0.50
26	17	42	33	3	0.49
29	8	50	25	12	0.30
30	17	38	37	3	0.46
31	15	63	12	5	0.64
32	12	51	24	8	0.43
34	9	61	14	11	0.42
39	6	70	5	14	0.39
48	15	65	10	5	0.66
53	18	59	16	2	0.67
55	15	69	6	5	0.73
57	10	71	4	10	0.59
60	15	74	1	5	0.83

Chapter 7

Conclusions and Future Work

The objective of this thesis consisted on the application of Cartesian Genetic Programming to the problem of Pitch Estimation of piano notes. Our goal was to train 61 classifiers from C1 to C6 piano notes, that could identify when those pitches are present in small time-frames of an audio signal. In order to do this, we have developed a toolbox to help us encoding this problem under a classic CGP approach. The experiments for the Pitch Estimation were broken down as follows: application of classifiers to signals artificially created by mathematical models; application of classifiers to real audio recordings of monophonic piano signals; application of classifiers to polyphonic audio signals.

7.1 CGP Toolbox

Pitch Estimation is a very complex problem, so we decided to take incremental steps for addressing this problem. We wanted to apply CGP for evolving pitch classifiers. Our first step for this was to create a toolbox to help us with the codification of the problem. Instead of creating a toolbox specific for audio processing, we made that toolbox generic enough to encode different kind of problems. The first main reason was to validate that our CGP code works. The second reason was that, with a toolbox, we can clearly define what code belongs to the CGP itself, and what code belongs to the pitch estimation problem. We could debug and track better what was hapenning during the evolutionary process.
The toolbox is very simple to use. We only need to provide a small configuration, define

program inputs, a fitness function and the path to the function-set. The EA encoded is the $1 + \lambda$, and we can choose the value for λ. Settings such as the mutation probability, number of runs and generations are configurable. The cartesian representation of CGP can take multiple forms, because the number of rows, columns, levels-back and program outputs are customizable. It is prepared to handle with different type of fitness functions: minimization of $f(x)$ and maximization of $f(x)$. It can receive multiple program inputs, of any type. The toolbox is also prepared to receive parameters for each node. Those parameters are encoded in the genotype and can also mutate. Furthermore, it has a useful system of callbacks. This callback system let us handle multiple events, such as knowing when a new genotype was created, a new solution candidate was found or a run ended. Each callback receives useful information about the event itself: genotype, active nodes, fitness of the current candidate solution, and so on.

7.2 Pitch Estimation

During the first phase of our experiments, we applied our algorithm to artifical signals, created with mathematical models: sine, square and sawtooth waves. All the evolved classifiers found the correct pitch on true positive signals, and did not match true negative signals. Those were very promising results, and led us one step further. We then went into real audio monophonic signals. The function-set was updated to include more filters to help the algorithm deal with the harmonic structure of real recordings. After the results, we could see that the first approach could be improved specially on the lower pitches, because the classifiers were identifying a few false positives. On the other hand, only a short ammount of false negatives was found, which means that the classifiers were identifying correctly almost all notes that had the same F0 as the classifier. To decrease the false positives, we changed the mutation of the threshold, and add a second harmonic to the base signal. The experiments showed that false positives decreased and the overall F-measure of the classifiers increased. Altough, there are still classifiers to test with this new approach; for lack of time, we concentrate mainly on lower pitches. For testing purposes, we also applied those classifiers to polyphonic signals (chords). The results were encouraging, not great, but a good starting point for future work (we stress that the classifiers were not evolved to classify polyphonic signals). With those results, we have shown the feasibility of our approach to the problem of Pitch Estimation, using CGP to evolve classifiers.

7.3 Future Work

We have taken a little step into de world of evolving classifiers using CGP for Pitch Estimation, but more can be taken. For future work, we propose the following experiments:

- the application of polyphonic signals in the training phase of the classifiers. These could lead to better results in multi-pitch estimation.

- add more program inputs to increase the data for the function-set to work on. Cepstrum and power spectrum would be suitable candidates for this. They both add relevant information about frequency and periodicity.

- the function-set could be improved with more filters and signal processing functions.

- instead of using the product of the base signal with the output vector, we could use a normalized autocorrelation.

- another suggestion would be to try to use the Matthews correlation coefficient (MCC) (Matthews, 1975) in the fitness function.

- our approach can detect pitches in small time-frames, which let us also infer the onset and offset of a note. It would be interesting to, in the future, include dynamic perception, to extract the amplitude or velocity of each note.

We think that this dissertation is a contribute to the resolution of the Pitch Estimation problem, adding one more technique to the existing ones.

Bibliography

Alonso, M., G. Richard, and B. David (2005, July). Extracting note onsets from musical recordings. In *2005 IEEE International Conference on Multimedia and Expo*, pp. 4 pp.–.

Baskind, A. and A. De Cheveigné (2003, June). Pitch-Tracking of Reverberant Sounds, Application to Spatial Description of Sound Scenes. In *AES 24th conference "Multichannel Audio - The New Reality"*, Banff, Canada, pp. –. cote interne IRCAM: Baskind03a.

Beauchamp, J. W., R. C. Maher, and R. Brown (1993, Mar). Detection of musical pitch from recorded solo performances. In *Audio Engineering Society Convention 94*.

Bello, J., L. Daudet, S. Abdullah, C. Duxbury, M. Davies, and M. Sandler (2005, September). A tutorial on onset detection in music signals. *IEEE Transactions on Speech and Audio Processing 13*(5).

Bello, J. P., L. Daudet, and M. B. Sandler (2006, Nov). Automatic piano transcription using frequency and time-domain information. *IEEE Transactions on Audio, Speech, and Language Processing 14*(6), 2242–2251.

Bello, J. P., G. Monti, M. B. Sandler, et al. (2000). Techniques for automatic music transcription. In *ISMIR*.

Bello, J. P. and M. Sandler (2000). Blackboard system and top-down processing for the transcription of simple polyphonic music. In *In Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFx-00*, pp. 7–9.

Benetos, E., S. Dixon, D. Giannoulis, H. Kirchhoff, and A. Klapuri (2013). Automatic music transcription: challenges and future directions. *J. Intell. Inf. Syst. 41*(3), 407–434.

Bertin, N., R. Badeau, and E. Vincent (2010, March). Enforcing harmonicity and smoothness in bayesian non-negative matrix factorization applied to polyphonic

music transcription. *IEEE Transactions on Audio, Speech, and Language Processing 18*(3), 538–549.

Brown, J. C. (1992). Musical fundamental frequency tracking using a pattern recognition method. *The Journal of the Acoustical Society of America 92*(3), 1394–1402.

Burred, J., A. Roebel, and X. Rodet (2006, December). An Accurate Timbre Model for Musical Instruments and its Application to Classification. In *Workshop on Learning the Semantics of Audio Signals*, Athens, Greece, pp. 1–1. cote interne IRCAM: Burred06a.

Conklin, H. A. (1999). Generation of partials due to nonlinear mixing in a stringed instrument. *The Journal of the Acoustical Society of America 105*(1), 536–545.

Cont, A. (2006). Realtime multiple pitch observation using sparse non-negative constraints. In *in International Conference on Music Information Retrieval*.

Corkill, D. D. (1991). Blackboard systems. *AI Expert 6*, 40–47.

Daudet, L. (2004). Sparse and structured decompositions of audio signals in overcomplete spaces. In *Proc. Int'l Conference on Digital Audio Effects (DAFx), Naples, Italy*, pp. 22–26.

Davy, M. and S. Godsill (2003). Bayesian harmonic models for musical signal analysis. In J. M. Bernardo (Ed.), *Bayesian Statistics VII*. Oxford University Press.

Davy, M., S. Godsill, and J. Idier (2006, April). Bayesian analysis of polyphonic western tonal music. *The Journal of the Acoustical Society of America 119*(4), 2498—2517.

de Cheveigné, A. and H. Kawahara (2002). Yin, a fundamental frequency estimator for speech and music. *J Acoust Soc Am 111*, 1917–1930.

De Cheveigné, A. and H. Kawahara (2002). Yin, a fundamental frequency estimator for speech and music. *The Journal of the Acoustical Society of America 111*(4), 1917–1930.

Dempster, A. P., N. M. Laird, and D. B. Rubin (1977). Maximum likelihood from incomplete data via the em algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B 39*(1), 1–38.

Dictionary, C. O. E. (2002). *Concise Oxford English Dictionary - Thumb Index Edition* (tenth ed.). Oxford University Press.

Duan, Z., B. Pardo, and C. Zhang (2010, November). Multiple fundamental frequency estimation by modeling spectral peaks and non-peak regions. *IEEE Transactions on Audio, Speech, and Language Processing 18*(8), 2121–2133.

Duan, Z., Y. Zhang, C. Zhang, and Z. Shi (2008, May). Unsupervised single-channel music source separation by average harmonic structure modeling. *IEEE Transactions on Audio, Speech, and Language Processing 16*(4), 766–778.

Duifhuis, H. and L. Willems (1973). Measurement of pitch in speech: An implementation of goldstein's theory of pitch perception. *The Journal of the Acoustical Society of America 71*, 1568–1580.

Ellis, D. P. W. (1996). *Prediction-driven Computational Auditory Scene Analysis*. Ph. D. thesis, Cambridge, MA, USA. AAI0597425.

Emiya, V. (2007). Multipitch estimation and tracking of inharmonic sounds in colored noise.

Emiya, V., R. Badeau, and B. David (2007, September). Multipitch estimation of quasi-harmonic sounds in colored noise. In *10th Int. Conf. on Digital Audio Effects (DAFx-07)*, Bordeaux, France.

Emiya, V., R. Badeau, and B. David (2010). Multipitch estimation of piano sounds using a new probabilistic spectral smoothness principle. *Audio, Speech, and Language Processing, IEEE Transactions on 18*(6), 1643–1654.

Emiya, V., N. Bertin, B. David, and R. Badeau (2010a, July). MAPS - A piano database for multipitch estimation and automatic transcription of music. Research report.

Emiya, V., N. Bertin, B. David, and R. Badeau (2010b). Maps-a piano database for multipitch estimation and automatic transcription of music.

Emiya, V., B. David, and R. Badeau (2007). A parametric method for pitch estimation of piano tones. In *2007 IEEE International Conference on Acoustics, Speech and Signal Processing-ICASSP'07*, Volume 1, pp. I–249. IEEE.

Every, M. and J. Szymanski (2004, 10). A spectral-filtering approach to music signal separation. pp. 197–200.

Fletcher, N. and T. Rossing (2008). *The Physics of Musical Instruments*. Springer New York.

Fogel, D. B. (1995). *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. Piscataway, NJ, USA: IEEE Press.

Fogel, L., A. Owens, and M. Walsh (1966). *Artificial Intelligence Through Simulated Evolution*. John Wiley & Sons.

Fritts, L. (2006). Available: http:// theremin.music.uiowa.edu/mis.html

last accessed: 2016-06-20 University of Iowa Musical Instrument Samples. http://theremin.music.uiowa.edu/.

Ghulam, M. (2011). Extended average magnitude difference function (eamdf) based pitch detection. *Int. Arab J. Inf. Technol. 8*(2), 197–203.

Godsill, S. and M. Davy (2002). Bayesian harmonic models for musical pitch estimation and analysis. *Acoustics, Speech, and Signal Processing, 2002. Proceedings. (ICASSP '02). IEEE International Conference on 2*, 1769–1772.

Goldstein, J. L. (1973). An optimum processor theory for the central formation of the pitch of complex tones. *The Journal of the Acoustical Society of America 54*(6), 1496–1516.

Goto, M. (2004, September). A real-time music-scene-description system: predominant-F0 estimation for detecting melody and bass lines in real-world audio signals. *Speech Communication 43*(4), 311–329.

Goto, M., H. Hashiguchi, T. Nishimura, and R. Oka (2002, October 13-17). Rwc music database: Popular, classical and jazz music databases. In *Proceedings of the Third International Conference on Music Information Retrieval*, Paris, France, pp. 287–288. `http://ismir2002.ismir.net/proceedings/03-SP04-1.pdf`.

H. Viste and G. Evangelista (2002, Sept.). An Extension for Source Separation Techniques Avoiding Beats. In *Proc. of Digital Audio Effects Conference (DAFx '02)*, Hamburg, Germany.

Hansen, N., D. V. Arnold, and A. Auger (2015). *Evolution Strategies*, pp. 871–898. Berlin, Heidelberg: Springer Berlin Heidelberg.

Harding, S., J. Leitner, and J. Schmidhuber (2013). Cartesian genetic programming for image processing. In R. Riolo, E. Vladislavleva, M. D. Ritchie, and J. H. Moore (Eds.), *Genetic Programming Theory and Practice X*, Genetic and Evolutionary Computation, pp. 31–44. Springer New York.

Hardingb, S., P. Ch, M. Franka, and J. Trieschc (2013). Learning visual object detection and localisation using icvision.

Hess, W. (1983). *Pitch determination of speech signals : algorithms and devices*. Springer series in information sciences. Berlin, New York: Springer-Verlag. Includes index.

Hynek Hermansky, Nelson Morgan, H.-G. H. (1993, April). Recognition of speech in additive and convolutional noise based on rasta spectral processing. *Proceed-*

ings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP) 2, 83–86.

Jensen, K. (1999). Timbre Models of Musical Sounds Kristoffer Jensen. *Computer* (99).

Kameoka, H., T. Nishimoto, and S. Sagayama (2007, March). A multipitch analyzer based on harmonic temporal structured clustering. *IEEE Transactions on Audio, Speech, and Language Processing 15*(3), 982–994.

Klapuri, A. (1998, Sept). Number theoretical means of resolving a mixture of several harmonic sounds. In *Signal Processing Conference (EUSIPCO 1998), 9th European*, pp. 1–5.

Klapuri, A. (2005). A perceptually motivated multiple-f0 estimation method. In *in Proc. IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, pp. 291–294.

Klapuri, A. (2006). Multiple fundamental frequency estimation by summing harmonic amplitudes. In *in ISMIR*, pp. 216–221.

Klapuri, A. and M. Davy (Eds.) (2006). New York: Springer.

Klapuri, A. P. (2003, Nov). Multiple fundamental frequency estimation based on harmonicity and spectral smoothness. *IEEE Transactions on Speech and Audio Processing 11*(6), 804–816.

Koretz, A. and J. Tabrikian (2011, September). Maximum a posteriori probability multiple-pitch tracking using the harmonic model. *Trans. Audio, Speech and Lang. Proc. 19*(7), 2210–2221.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press.

Koza, J. R. (1994). *Genetic programming. 2. , Automatic discovery of reusable programs*. Cambridge (Mass): The Mit Press.

Lahat, M., R. Niederjohn, and D. Krubsack (1987, Jun). A spectral autocorrelation method for measurement of the fundamental frequency of noise-corrupted speech. *IEEE Transactions on Acoustics, Speech, and Signal Processing 35*(6), 741–750.

Large, E. W. and J. F. Kolen (1994). Resonance and the perception of musical meter. *Connection Science 6*(2-3), 177–208.

Lee, D. D. and H. S. Seung (2000). Algorithms for non-negative matrix factorization. In *In NIPS*, pp. 556–562. MIT Press.

Loureiro, M. A., H. B. Paula, and H. C. Yehia (2004). Timbre classification of a single musical instrument. In *In Proc. Intl. Conf. on Music Information Retrieval (ISMIR.*

Marolt, M. (2001). Sonic: Transcription of polyphonic piano music with neural networks. In *Audiovisual Institute, Pompeu Fabra University*, pp. 217–224.

Marolt, M. (2004, June). A connectionist approach to automatic transcription of polyphonic piano music. *IEEE Transactions on Multimedia 6*(3), 439–449.

Martin, K. D. (1996). Automatic transcription of simple polyphonic music: . . .

Martin, P. (1982, May). Comparison of pitch detection by cepstrum and spectral comb analysis. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '82.*, Volume 7, pp. 180–183.

Martins, L. G. (2009, February). *A computational Framework for Sound Segregation in Music Signals.* Ph. D. thesis, University of Porto, Porto, Portugal.

Matthews, B. W. (1975). Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochim. Biophys. Acta 405*, 442–451.

McIntyre, M. E., R. T. Schumacher, and J. Woodhouse (1983). On the oscillations of musical instruments. *The Journal of the Acoustical Society of America 74*(5), 1325–1345.

Meddis, R. (1986). Simulation of mechanical to neural transduction in the auditory receptor. *The Journal of the Acoustical Society of America 79*(3), 702–711.

Miller, J., P. Thomson, and T. Fogarty (1997). Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study. *219*.

Miller, J. F. (1999). An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2*, GECCO'99, San Francisco, CA, USA, pp. 1135–1142. Morgan Kaufmann Publishers Inc.

Miller, J. F. and P. Thomson (2000). Cartesian genetic programming. In *Genetic Programming*, pp. 121–132. Springer.

Molla, S. and B. Torrésani (2004). Determining local transientness of audio signals. *IEEE signal processing letters 11*(7), 625–628.

Nii, H. P. (1986a, June). Blackboard systems. *Report No. KSL 86-18 & AI Magazine Vols. 7-2 and 7-3.*

Nii, H. P. (1986b). Blackboard systems, part one: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine 7*(2), 38–53.

Noll, A. M. (1967). Cepstrum pitch determination. *The Journal of the Acoustical Society of America 41*(2), 293–309.

Ochiai, K., H. Kameoka, and S. Sagayama (2012, March). Explicit beat structure modeling for non-negative matrix factorization-based multipitch analysis. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 133–136.

Parsons, T. W. (1976). Separation of speech from interfering speech by means of harmonic selection. *The Journal of the Acoustical Society of America 60*(4), 911–918.

Patterson, R. D. and J. Holdsworth (1996). A functional model of neural activity patterns and auditory images. In *Advances in Speech, Hearing and Language Processing*, Volume 3, pp. 547–558. JAI Press.

Peeling, P. H. and S. J. Godsill (2011, Oct). Multiple pitch estimation using non-homogeneous poisson processes. *IEEE Journal of Selected Topics in Signal Processing 5*(6), 1133–1143.

Peeters, G. (2006, May). Music pitch representation by periodicity measures based on combined temporal and spectral representations. In *2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings*, Volume 5, pp. V–V.

Reis, G. (2012). *Una Aproximación Genética a la Transcripción Automática de Música*. Ph. D. thesis, New York, NY, USA.

Reis, G., F. F. de Vega, and A. Ferreira (2012, Oct). Automatic transcription of polyphonic piano music using genetic algorithms, adaptive spectral envelope modeling, and dynamic noise level estimation. *IEEE Transactions on Audio, Speech, and Language Processing 20*(8), 2313–2328.

Reis, G., F. Fernandez, and A. Ferreira (2008). Genetic algorithm approach to polyphonic music transcription for mirex 2008. *MIREX (2008), multiple f0 estimation and tracking contest.(Cited on pages 163 and 164)*.

Reis, G., N. Fonseca, and F. Ferndandez (2007, Oct). Genetic algorithm approach to polyphonic music transcription. In *Intelligent Signal Processing, 2007. WISP 2007. IEEE International Symposium on*, pp. 1–6.

Reis, G. and F. F. Vega (2007). Electronic synthesis using genetic algorithms for

automatic music transcription. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, New York, NY, USA, pp. 1959–1966. ACM.

Rodet, X. and F. Jaillet (2001). Detection and modeling of fast attack transients.

Ross, M., H. Shaffer, A. Cohen, R. Freudberg, and H. Manley (1974, Oct). Average magnitude difference function pitch extractor. *IEEE Transactions on Acoustics, Speech, and Signal Processing 22*(5), 353–362.

Röbel, A. (2003). Transient detection and preservation in the phase vocoder.

Röbel, A., X. Rodet, and C. Yeh (2006, May). Multiple f0 tracking in solo recordings of monodic instruments. In *Audio Engineering Society Convention 120*.

Santoro, C. A. and C. I. Cheng (2009, October 26-30). Multiple f0 estimation in the transform domain. In *Proceedings of the 10th International Society for Music Information Retrieval Conference*, Kobe, Japan, pp. 165–170. `http://ismir2009.ismir.net/proceedings/PS1-19.pdf`.

Sha, F. and L. K. Saul (2005). Real-time pitch determination of one or more voices by nonnegative matrix factorization. In L. K. Saul, Y. Weiss, and L. Bottou (Eds.), *Advances in Neural Information Processing Systems 17*, pp. 1233–1240. MIT Press.

Smaragdis, P. and J. C. Brown (2003, Oct). Non-negative matrix factorization for polyphonic music transcription. In *Applications of Signal Processing to Audio and Acoustics, 2003 IEEE Workshop on.*, pp. 177–180.

Su, L. and Y. H. Yang (2015, Oct). Combining spectral and temporal representations for multipitch estimation of polyphonic music. *IEEE/ACM Transactions on Audio, Speech, and Language Processing 23*(10), 1600–1612.

Vincent, E., N. Bertin, and R. Badeau (2010, March). Adaptive harmonic spectral decomposition for multiple pitch estimation. *IEEE Transactions on Audio, Speech, and Language Processing 18*(3), 528–537.

Virtanen, T. (2003). Algorithm for the separation of harmonic sounds with time-frequency smoothness constraint. In *In: Proc. Int. Conf. on Digital Audio Effects*, pp. 35–40.

Waibel, A., T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang (1990). Readings in speech recognition. Chapter Phoneme Recognition Using Time-delay Neural Networks, pp. 393–404. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Walmsley, P. J., S. J. Godsill, and P. J. W. Rayner (1998). Multidimensional optimisation of harmonic signals. pp. 2033–2036.

Walmsley, P. J., S. J. Godsill, and P. J. W. Rayner (1999). Polyphonic pitch tracking using joint Bayesian estimation of multiple frame parameters. *Applications of Signal Processing to Audio and Acoustics, 1999 IEEE Workshop on*, 119–122.

Yeh, C. (2008). Multiple fundamental frequency estimation of polyphonic recordings. *Thése de doctorat, University Paris 6 (UPMC)*.

Yeh, C. and A. Roebel (2006, Mars). Adaptive noise level estimation. In *Workshop on Computer Music and Audio Technology (WOCMAT'06)*, Taipei, Taiwan.

Yeh, C. and A. Roebel (2009, April). The expected amplitude of overlapping partials of harmonic sounds. In *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 3169–3172.

Yeh, C., A. Roebel, and X. Rodet (2010, August). Multiple fundamental frequency estimation and polyphony inference of polyphonic music signals. *Trans. Audio, Speech and Lang. Proc. 18*(6), 1116–1126.

Yoshii, K. and M. Goto (2012, March). A nonparametric bayesian multipitch analyzer based on infinite latent harmonic allocation. *Trans. Audio, Speech and Lang. Proc. 20*(3), 717–730.

Appendices

Cartesian Genetic Programming applied to pitch detection of piano notes

The authors names and affiliations were hidden for blind review purposes.

***Abstract*—Pitch Estimation, also known as Fundamental Frequency (F0) estimation, has been a popular research topic for many years, and is still investigated nowadays. This paper presents a novel approach to the problem of Pitch Estimation, using Cartesian Genetic Programming (CGP). We take advantage of the evolutionary algorithms, in particular CGP, to explore and evolve complex mathematical functions that act as classifiers. These classifiers are used to identify piano notes pitches in an audio signal. For a first approach, the obtained results are very promising: our error rate outperforms two of three state-of-the-art pitch estimators.**

1. Introduction

Pitch estimation on sound signals, also known as F0 detection, is a very important task of Automatic Music Transcription. This is a process in which a computer program writes the instrument's partitures of a given song or an audio signal. Usually, only pitched musical instruments are considered. Music transcription is a very difficult problem from both musical and computational points of view: although there has been much research devoted to it, it still remains an unsolved problem.

Given that Cartesian Genetic Programming (CGP) has already demonstrated its capabilities for synthesizing complex functions capable of extracting main features from images and performing image segmentation [1], we wanted to see its capabilities, when applied to audio processing, specially on Pitch Estimation. To tackle the problem of Pitch Estimation by using CGP, and future problems related to audio signal processing, we created the CGP-AP toolbox for Matlab. CGP-AP stands for *Cartesian Gentic Programming for Audio Signal Processing*. Then, by using this toolbox, we created a CGP System to synthesize mathematical expressions which act as classifiers capable of identifying the Pitch of all piano keys.

The rest of this section explains the related work. Section 2 presents the Cartesian Genetic Programming features, Section 3 explains our approach, on Section 4 we show our experiments and results and finally on section 5 we present our conclusions and future work.

1.1. Related Work

Over the years, there has been a lot of research on Pitch Estimation. However, to the best of our knowledge, there are no Cartesian Genetic Programming approaches for addressing this problem. Yeh et al. [2] proposed an algorithm based on the short-time Fourier transform (STFT) representation, by applying an adaptive noise level estimation algorithm and an harmonic matching technique. Klapuri [3], proposed an iterative approach algorithm, based on harmonicity and spectral smoothness. Reis et al. [4], used a genetic algorithm approach which relies on an adaptive spectral envelope modeling and dynamic noise level estimation. Marolt [5], presented a connectionist approach where he uses a partial tracking technique, based on an auditory model, which converts the acoustic signal into time-frequency space, and uses adaptive oscillators to detect periodicities in the output of the auditory model. Benetos and Weyde [6], based on probabilistic latent component analysis and supporting the use of sound state spectral templates, proposed an efficient, general-purpose model for multiple instrument polyphonic music transcription.

2. Cartesian Genetic Programming

Cartesian Genetic Programming is an increasingly popular and efficient form of Genetic Programming [7], [8] proposed by Julian Miller in 2000 [9]. In its classic form, it uses a very simple integer based genetic representation of a program in the form of a directed graph.

The genotype is a list of integers (and possibly real parameters) that represent the program primitives and how they are connected together (see Fig. 1). The programs are represented as graphs in which there are non-coding genes. The genes are addresses in data (connection genes), addresses in a look up table of functions and additional parameters. This representation is very simple, flexible and convenient for many problems. In the Figure 1 is shown the general form of a CGP graph. Usually, all functions have as many inputs as the maximum function arity and unused connections are ignored.

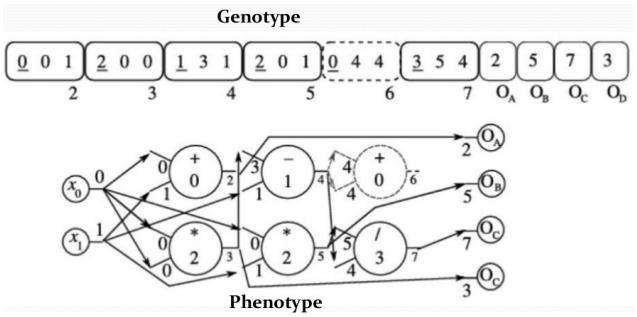


Figure 1. CGP general form. It is a grid of nodes whose functions are chosen from a set of primitive functions.There are 2 inputs and 3 outputs. The grid has $n_c = 3$ (columns) and $n_r = 2$ (rows).

CGP is *Cartesian* because it considers a grid of nodes that are addressed in a *Cartesian* coordinate system. Each CGP graph node may contain additional genes for encoding additional parameters that might be necessary for specific functions (eg.: a threshold value). The CGP uses individuals as possible problem solutions, a set of individuals is usually called population, to evaluate the quality of an individual or solution a fitness function is used, with this simple process all the solutions are evaluated quantitatively and the best solution is found. Each problem iteration or generation contains a set of possible solutions, population. The e population changes each generation pursuing the best problem solution.

Algorithm 1 General CGP Algorithm

1: Generate initial population at random (subject to constraints)
2: **while** stopping criterion not reached **do**
3: Evaluate fitness of genotypes in population
4: Promote fittest genotype to new population
5: Fill remaining places in the population with mutated versions of the fittest
6: **end while**

The CGP algorithm shown in Algorithm 1, begins with the generation of the initial population, then it uses a fitness function to evaluate the individuals of the population. The evolutionary strategy chooses the fittest one (best individual) and promote it directly the next generation. The remaining places in the population are mutated versions of the fittest individual. The algorithm stops when the stopping criterion is reached.

3. CGP approach to Pitch Estimation

The CGP general form is a grid of nodes whose functions are chosen from a set of primitive functions. The grid has n_c (columns) and n_r (rows), the idea is illustrated in Figure 1. Depending on n_r, n_c and $levels\text{-}back$ a wide range of graphs can be generated, when $n_r = 1$ and $levels\text{-}back = n_c$, arbitrary directed graphs can be created with a maximum depth. In general choosing these parameters imposes the least constraints, so without specialist knowledge this is the best and most general choice. [10]. In our particular CGP approach, we have multiple inputs,

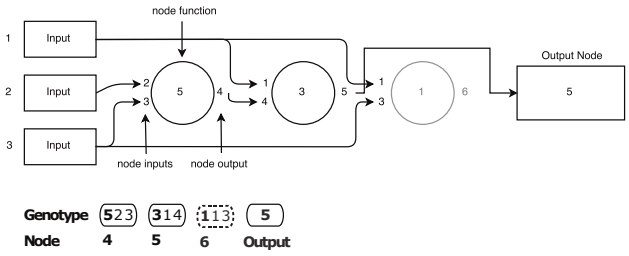


Figure 2. CGP graph with multiple inputs, one row, one output and its resulting genotype.

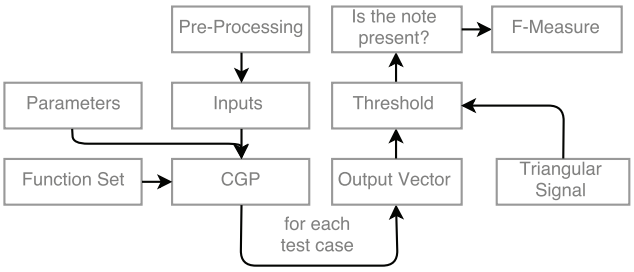


Figure 3. System architecture

only one row of graph nodes, one output (the result of the corresponding classifier), and $levels\text{-}back = n_c$ as depicted in Fig. 2.The resulting graph and genotype codification is also shown.

To perform the pitch detection using CGP, we developed a system where some important decisions and tasks were made besides the CGP. We defined what kind of inputs we used from the original piano audio signal, through a preprocessing task. We also had to develop a process to reach a binary output in order to perform our fitness function. The block diagram of our proposed system is much more than a simple CGP process and is depicted in Fig. 3. Each step of this system is described carefully throughout this section.

3.1. Preprocessig

The first task of the proposed system is the Preprocessing. For this, we used the piano samples (audio signals) from the MAPS database [11]. This is a huge dataset with multiple piano melodies in wave format. The piano sound signals are vectors in time with a sample rate of 44.100 samples per second.

For the preprocessing task, we split each piano sound signal into frames of 4096 samples width, corresponding to 96 milliseconds. To accomplish a good base for signal processing common tasks we transform the domain signals from the time domain to the frequency domain, using the Discrete Fourier Transform (DFT), shown in Equation 1. In order to do so, each frame is windowed (see Equation 2) using an hanning window (see Equation 3) to avoid spectral leakage. Then, the DFT is applied, obtaining signal frames in frequency domain.

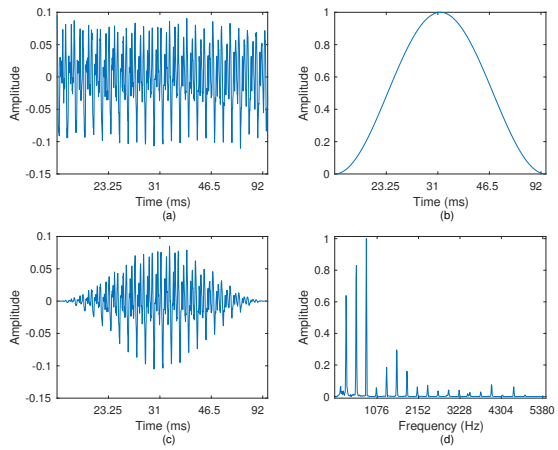


Figure 4. Preprocessing process: (a) input time signal piano note, (b) Hanning window, (c) resulting windowed signal, (d) frequency domain signal

$$X[k] = \sum_{n=0}^{N-1} x_w[n] e^{-j\left(\frac{2\pi}{N}\right)nk}, (k = 0, 1, ..., N-1). \quad (1)$$

$$x_w[n] = w[n].x[n]. \quad (2)$$

$$w[n] = 0.5\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right). \quad (3)$$

Typically, the DFT is computed using the Fast Fourier Transform algorithm (FFT), because it is faster, the difference in speed can be enormous. The resulting frequency domain signal used in our system is obtained using the Equation 1 where $x_w[n]$ is the time signal windowed and N is the number of samples in the vector. Recall that any signal cannot be uniquely represented for frequencies above $\frac{f_s}{2}$ (also known as the Nyquist frequency) where f_s is the sampling frequency of the sequence (also known as the Nyquist frequency). Above $\frac{f_s}{2}$ all signal energy is reflected back into the frequency range $-\frac{f_s}{2}$. Between $\frac{f_s}{2}$ and f_s, the reflection is in reverse order which gives rise to a DFT frequency domain period $[0; f_s]$.

Due to the DFT spectrum properties, the symmetry of the real part and the antisymmetry of the imaginary part relative to Nyquist frequency $\frac{f_s}{2}$, we may only use half of the resulting signal of the DFT, so only the first 2048 frequency bins were considered. The preprocessing process is illustrated in Fig. 4, from the input time signal of a piano note to generated CGP input in frequency domain.

3.2. Individual Encoding

Evolutionary Algorithms encode each possible solution (individual) to the problem as a set of genes. During the evolutionary process, the genes of each individual or possible solution to the problem are mutated, and possibly recombined, to create better and fitter individuals. At the end of each iteration (generation) all the individuals are evaluated and, according to their quality (fitness value) as a solution for the problem, some pass to next generation and some are discarded. Therefore, the individuals encoding plays a major role on achieving good results and high speed computing.

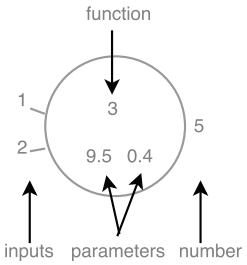


Figure 5. Node Genes(5): inputs, code function and real parameters

In general, when working with CGP, the genotype is composed by input nodes, function nodes and the output node. Our proposed algorithm contains 4 input nodes and one single output node. We used a single row CGP configuration with 100 function nodes. Each function node contains 5 genes (see Fig.5): two inputs, corresponding to the maximum function arity (the number of arguments or operands that the function takes), one integer corresponding to the function used from the function set and two function parameters. All function nodes return a vector. However, since our system is a classifier, the final output it must return a binary value. In order to transform the output vector into a binary value, our system uses a threshold after a comparison between the output and a base signal. This base signal is a triangular signal centered on frequency bin of the corresponding Fundamental Frequency of the note classifier. This threshold function has a constraint to ensure it is the last function before the binary output. This threshold is also evolved: the parameter θ mutates from an initial configurable value with an also configurable mutation probability.

3.2.1. Inputs. The algorithm inputs are obtained from the preprocessing system (see Subsection 3.1). Each piano sample is split into time frames and transformed into frequency domain using a DFT. By doing so, we get frames with 2048 samples with complex domain numbers, each one representing a time frame of a piano sample note with 96 milliseconds duration. From this vector $X[k]$ in frequency domain we may use two different representations of a complex number, Cartesian and polar:

1) $\Re\{X[k]\}$
2) $\Im\{X[k]\}$
3) $\measuredangle\{X[k]\}$
4) $||X[k]||$

We have a couple of vectors each one with 2 components, making 4 usable inputs. By having redundant infor-

mation, regarding the 4 inputs, we ensure the CGP system has a variety of representations of the same data, so it can be able to choose the one which best fits the problem.

3.2.2. Function set. As depicted Figure 5, each function node has a gene that indicates which function of the function set should be used on that node. Table 1 shows our current function set or lookup table. Note that all the functions are prepared to receive one or two vectors and all of them return a vector, the maximum function arity is 2.

TABLE 1. FUNCTION SET LOOKUP TABLE

Index	Function	Description
1	SPAbs	Absolute value
2	SPBPGaussFilter	band pass Gaussian filter
3	SPConvolution	Convolution
4	SPCos	Cosine
5	SPDivide	poit to point Division
6	SPFFT	Absolute value of the DFT
7	SPGaussfilter	Gaussian filter
8	SPHighPassFilter	high pass filter
9	SPIFFT	Absolute value of Inverst DFT
10	SPLog	Natural logarithm
11	SPLog10	Common logarithm
12	SPLowPassFilter	low pass filter
13	SPMedFilter	Median filter
14	SPMod	Remainder after division
15	SPMulConst	Multiplication by constant
16	SPNormalizeMax	Normalization maximum
17	SPNormalizeSum	Normalization sum
18	SPPeaks	Find peaks
19	SPSin	Sine
20	SPSubtract	Subtraction
21	SPSum	Sum
22	SPSumConst	Sum with a constant
23	SPThreshold	tresholding
24	SPTimes	Multiplication

Our function set is basically composed by filtering operations on vectors and by arithmetic operations with constants and vectors. Each function may also have one or two real parameters that are encoded as parameter genes, these parameters also evolve during the training process.

3.2.3. Function parameters. As previously mentioned, the functions of the instruction set may have up to two parameters. In fact, most functions need real parameters to perform their tasks. In order to evolve those parameters as well, we encoded them in genotype as genes, so they can also mutate. We used 2 real parameters for each function node. This way, each function has its particular parameters with a particular meaning. Each parameter r_1, r_2 of each function has its own range. Those intervals are normalized into $[0, 1]$: all intervals are transformed from $[a, b]$ to a normalized one $[0, 1]$. By using this technique, the actual value of any parameter can bee seen as number between 0 and 1 or a percentage of the interval.

3.3. Mutation

In Cartesian Genetic Programming, mutation plays a major role on the evolution of the algorithm. Here, each gene might be subject to mutation according to a configurable parameter: the mutation probability. This parameter represents the probability of each gene to be mutated. For instance, $p = 0.01$ means that each gene will mutate with a 1% probability. In our case, different mutation processes are used according to the gene type and domain: if a function gene happens to be mutated, then a valid value must be chosen for selecting a new function in the function set lookup table; if a mutation occurs in a gene node input, then a valid value is the output of any previous node in the genotype or any system input; the valid values for the system output genes are the output of any node in the genotype or the address of a system input. All these mutations happen according to the uniform probability distribution function for integers. Two additional genes can also mutate: the real parameters used by the functions of the function set. These are important parameters used by those functions to perform specific tasks. According to each function, each parameter has a specific meaning and also has its own domain range where it can variate. In this case, we take any value in the normalized interval $[0, 1]$ and transform it into the real interval $[a, b]$. The mutation of the real genes (function parameters) is done using the normal distribution in order to address the entire range:

$$f(x) = \frac{e^{-(x-\mu)^2/(2\sigma^2)}}{\sigma\sqrt{2\pi}}, \tag{4}$$

where $f(x)$ represents the density function of x variable, with a normal distribution. This function is also represented as $N(\mu, \sigma)$, where μ is the mean and σ is the standard deviation. To perform the mutation of a function parameter, r_{old}, we generate a new random r_{mutate} using the normal distribution $N(\mu = r_{old}, \sigma)$, with σ being configurable in our system. This way, we ensure that when a mutation occurs in a real parameter, all the parameter interval is reachable, but with higher probability to mutate to closer values.

As previously described in Section 3.2, the output of the system is obtained by using a threshold value in order to have a binary output. This threshold mutates independently from the genotype genes, with a different configurable probability.

3.4. Evolutionary Strategy

Our evolutionary strategy is a variant of a simple evolutionary algorithm known as $1+\lambda$ [12], which is widely used for CGP: the new offspring is obtained promoting the fittest individual and generate λ new individuals trough mutation. Also, an offspring can replace a parent when it has the same fitness as its parent and there is no other population member with a better fitness (see Algorithm 2). According to Goldman [13], an empirical value for λ is 4.

Algorithm 2 Algorithm $((\mu + \lambda)\, EA)$

1: $t \leftarrow 0$;
2: Initialize P_0 with μ individuals chosen uniformly at random;
3: **while** a stop condition is fulfilled. **do**
4: **for** i = 1 to λ **do**
5: choose $x_i \in P_t$ uniformly at random;
6: mutate each gene x_i with probability p;
7: **end for**
8: Create the new population P_{t+1} by choosing the best μ individuals out of $P_t \cup \{x_1, \ldots, x_\lambda\}$;
9: $t \leftarrow t + 1$;
10: **end while**

During the evolutionary process, there is a reasonable percentage of inactive genes. Such inactive genes have a neutral effect on the genotype fitness [14]. However, the influence of neutrality in CGP has been investigated in detail by Vassilev and Miller [15] and was shown shown to be extremely beneficial to the efficiency of the evolutionary process. For better computing performance, we also took in account the similarity between individuals: when an individual has the same active genes than the offspring parent, there is no need to compute its fitness.

3.5. Fitness Function

The main goal of the proposed system is to evolve a classifier for each piano note. The output of each classifier is binary: when the corresponding note is detected the output is 1, otherwise is 0. During the evolutionary process, we used as inputs an amount of piano samples with the desired pitch (fundamental frequency) and the same amount of piano samples with different pitches. Thus, for each classifier, we used 50 true positive piano samples and 50 piano samples with different pitches. During the evolutionary process, the evaluation of each individual (classifier) is done using F-measure, eq. (5)

$$F_{measure} = 2 \times \frac{recall \times precision}{recall + precision}, \tag{5}$$

where:

$$precision = \frac{tp}{tp + fp}, \; recall = \frac{tp}{tp + fn}. \tag{6}$$

One of our system peculiar characteristics is the binaritazion process, since the CGP output is a signal vector processed and filtered. In order to accomplish a binary output, where 1 means the presence of the corresponding pitch in the analyzed frame and 0 means the opposite, we used a comparison process between the CGP ouput vector normalized in amplitude $O_{CGP}(n)$ and a base signal with the frequency corresponding to the pitch of the estimator, $B_{F0}(n)$. The first step is the normalization of the output vector in amplitude, this way all the elements of the vector are between 0 and 1. The base signal is obtained with a triangular mask on frequency domain around the F0 of the estimator. We used a triangular mask with 3 configurable points in both in size and amplitude. We then generate the following scalar:

$$x = \sum_{n=0}^{N} O_{CGP}(n) * B_{F0}(n), \tag{7}$$

where x measures the interception between the two discrete time signals. If we approximate these signals to continuous time we could see x as the intersected area between the two signals. Finally, we used a threshold function to accomplish the binary result:

$$T(x) = \begin{cases} 1, & \text{if } x > \theta \\ 0, & \text{if } x <= \theta \end{cases}, \tag{8}$$

where θ is the threshold value. Since both signals are normalized the max value for x is:

$$x = \sum_{n=0}^{N} B_{F0}(n). \tag{9}$$

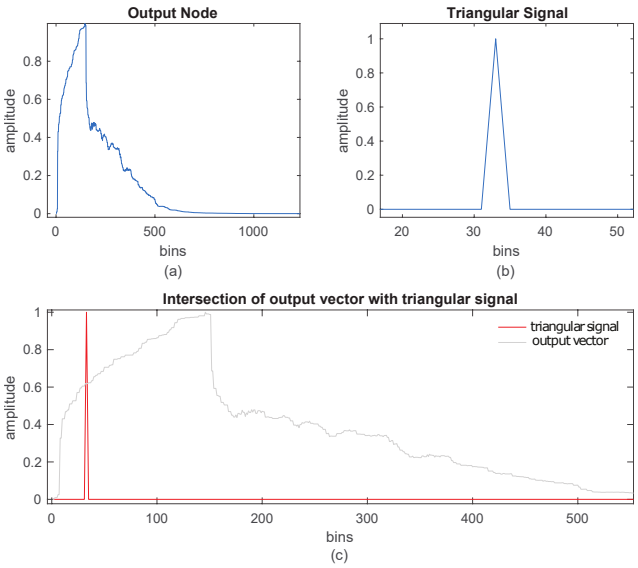


Figure 6. (a) CGP output signal, (b) base triangular signal (c) computing intersection for threshold.

All the fitness process including the sum and the thresholding is illustrated in Fig. 6 with tree graphs.

4. Experiments and Results

Since this is our first approach to the problem of Pitch Estimation of piano notes using Cartesian Genetic Programming, our main goal is to show the applicability of CGP on this problem. We evolved one classifier for each piano note. Each piano key is represented by the corresponding MIDI note number, being 60 the MIDI note number corresponding to the C4 musical note (the middle C). The piano sounds used for training and testing are those from

the MAPS database [11]. Table 2 shows the values of the configurable parameters for our system. The evolutionary process consisted of 30 runs with 5000 generations each, using 50 positive and 50 negative cases. The classifiers were evaluated using F-measure (Equation 5).

TABLE 2. EXPERIMENTS PARAMETERS

Parameter	Value
Frame Size	4096
Fitness Threshold	0.5
Positive Test Cases	50
Negative Test Cases	50
Outputs	1
Rows	1
Columns	100
Levels Back	100
Population Size	4
Mutation Probability	5%
Runs	30
Generations	5000

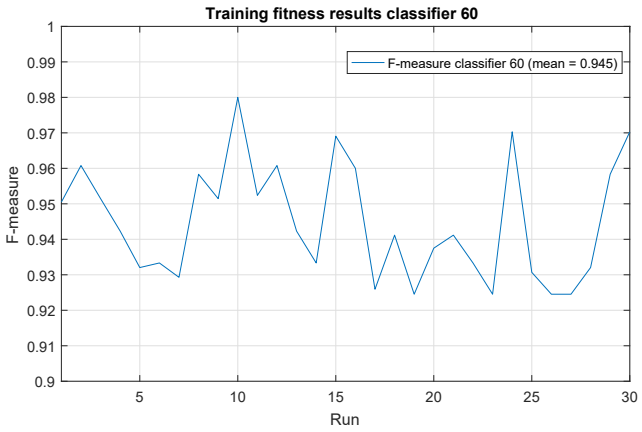


Figure 7. Classifier 60 fitness values for 30 runs using F-measure

Figure 7 shows the individual results of each run (i.e. evolved pitch estimator) for classifying the note 60. The best generated classifier was chosen as the pitch estimator for note 60. Figure 8 depicts the resulting program of this classifier. It shows the functions used in each node as well as the inputs of each node. For instance: the node 101 contains the Gaussian filter function and the function arguments (inputs) are the outputs of the nodes 36 and 87. The resulting program is a set of mathematical functions over vectors - our phenotype.

After the training process, each classifier was tested with a different test set. Each test set consisted in 144 negative notes (48×3) and 5 positive notes, comprising a total of 149 piano sound samples. Table 3 shows our results. As a first approach, these results are very encouraging, since for almost notes we achieved a classifier with F-Measure values greater than 70%.

For comparing our results with other techniques and algorithms we generated the graph depicted in Figure 9, where

```
Node 1 = input (1)
Node 2 = input (2)
Node 3 = input (3)
Node 4 = input (4)
Node 5 = SPConvolution ( 1 , 1 )
Node 6 = SPFFT ( 1 , 4 )
Node 7 = SPConvolution ( 4 , 4 )
Node 8 = SPTimes ( 5 , 7 )
Node 10 = SPSum ( 2 , 1 )
Node 11 = SPIFFT ( 10 , 4 )
Node 12 = SPPeaks ( 8 , 11 )
Node 13 = SPPeaks ( 11 , 10 )
Node 14 = SPSum ( 3 , 10 )
Node 15 = SPSubtract ( 1 , 13 )
Node 16 = SPAbs ( 13 , 12 )
Node 17 = SPLog10 ( 16 , 10 )
Node 18 = SPThreshold ( 5 , 16 )
Node 19 = SPCos ( 17 , 18 )
Node 20 = SPLog ( 8 , 2 )
Node 23 = SPNormalizeSum ( 6 , 14 )
Node 24 = SPAbs ( 13 , 13 )
Node 33 = SPDivide ( 12 , 23 )
Node 36 = SPHighPassFilter ( 19 , 20 )
Node 87 = SPSum ( 15 , 33 )
Node 101 = SPGaussfilter ( 36 , 87 )
Node 104 = SPIFFT ( 24 , 101 )
```

Figure 8. Evolved classifier code for pitch 60

TABLE 3. TEST RESULTS FOR 19 CLASSIFIERS

classifier	tp	tn	fp	fn	f-measure
48	5	135	9	0	0.53
50	5	140	4	0	0.71
52	5	138	6	0	0.63
53	5	142	2	0	0.83
55	5	138	6	0	0.63
57	5	139	5	0	0.67
60	5	142	2	0	0.83
61	4	142	2	1	0.73
62	4	144	0	1	0.88
63	4	144	0	1	0.88
64	5	138	6	0	0.63
65	5	141	3	0	0.77
66	5	139	5	0	0.67
67	5	141	3	0	0.77
68	5	141	3	0	0.77
69	5	141	3	0	0.77
70	5	142	2	0	0.83
71	5	142	2	0	0.83
72	5	142	2	0	0.83

besides F-mesure we can see the error rate in percentage for the data-set test with 96ms frames. According to Emiya [16] the three main monophonic pitch estimatiors are: Parametric F0 estimator, the Nonparametric F0 estimator and the YIN estimator [17] and those estimators have mean error rates of 2.4%, 3.0% and 11.0% respectively. Our Cartesian Genetic Programing approach to F0 estimation reaches the mean error rate of 2.5%, a very encouraging result.

5. Conclusions and Future Work

With our work and experiments, we have shown the feasibility of the proposed approach. The results are encouraging and it can bee considered a good starting point. The

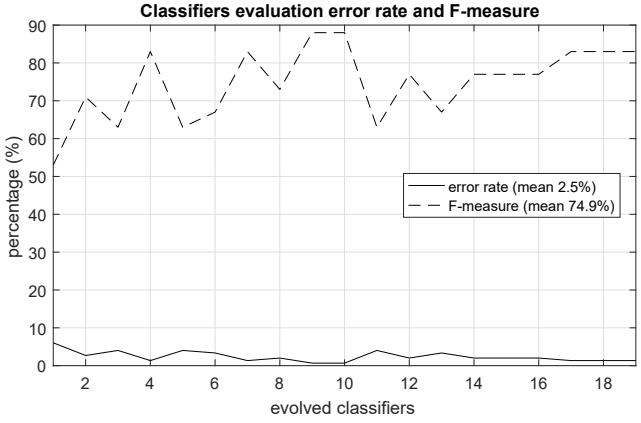


Figure 9. Graph with 19 classifiers' evaluation results in error rate and F-measure

results accomplished with the CGP technique are in line or even better than the most popular algorithms for pitch recognition on piano notes.

Planning ahead, we aim to test and tune all the CGP parameters in order to obtain even better results. We are also considering taking into account additional inputs for our system, such as the generated Cepstrum of each audio frame. Another important aspect that we are planning to take into account is the use of the harmonic information during the comparison process.

References

[1] S. Harding, J. Leitner, and J. Schmidhuber, "Cartesian genetic programming for image processing," in *Genetic Programming Theory and Practice X*. Springer, 2013, pp. 31–44.

[2] C. Yeh, A. Roebel, and X. Rodet, "Multiple fundamental frequency estimation and polyphony inference of polyphonic music signals," *Trans. Audio, Speech and Lang. Proc.*, vol. 18, no. 6, pp. 1116–1126, Aug. 2010. [Online]. Available: http://dx.doi.org/10.1109/TASL.2009.2030006

[3] A. P. Klapuri, "Multiple fundamental frequency estimation based on harmonicity and spectral smoothness," *IEEE Transactions on Speech and Audio Processing*, vol. 11, no. 6, pp. 804–816, Nov 2003.

[4] G. Reis, F. Fernandéz de Vega, and A. Ferreira, "Audio analysis and synthesis-automatic transcription of polyphonic piano music using genetic algorithms, adaptive spectral envelope modeling, and dynamic noise level estimation," *IEEE Transactions on Audio Speech and LanguageProcessing*, vol. 20, no. 8, p. 2313, 2012.

[5] M. Marolt, "A connectionist approach to automatic transcription of polyphonic piano music," *IEEE Transactions on Multimedia*, vol. 6, no. 3, pp. 439–449, June 2004.

[6] M. Mueller and F. Wiering, Eds., *An efficient temporally-constrained probabilistic model for multiple-instrument music transcription*. Malaga, Spain: ISMIR, October 2015.

[7] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992, vol. 1.

[8] ——, "Genetic programming ii: Automatic discovery of reusable subprograms," *Cambridge, MA, USA*, 1994.

[9] J. F. Miller and P. Thomson, "Cartesian genetic programming," in *Genetic Programming*. Springer, 2000, pp. 121–132.

[10] J. F. Miller, "Gecco 2013 tutorial: cartesian genetic programming," in *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*. ACM, 2013, pp. 715–740.

[11] V. Emiya, N. Bertin, B. David, and R. Badeau, "Maps-a piano database for multipitch estimation and automatic transcription of music," 2010.

[12] M. Eigen, *Ingo Rechenberg Evolutionsstrategie Optimierung technischer Systeme nach Prinzipien der biologishen Evolution*. mit einem Nachwort von Manfred Eigen, Friedrich Frommann Verlag, Struttgart-Bad Cannstatt, 1973.

[13] B. W. Goldman and W. F. Punch, "Analysis of cartesian genetic programmings evolutionary mechanisms," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 3, pp. 359–373, 2015.

[14] J. F. Miller and S. L. Smith, "Redundancy and computational efficiency in cartesian genetic programming," *IEEE Trans. Evolutionary Computation*, vol. 10, no. 2, pp. 167–174, 2006.

[15] V. K. Vassilev and J. F. Miller, "The advantages of landscape neutrality in digital circuit evolution," in *Evolvable systems: from biology to hardware*. Springer, 2000, pp. 252–263.

[16] V. Emiya, B. David, and R. Badeau, "A parametric method for pitch estimation of piano tones," in *2007 IEEE International Conference on Acoustics, Speech and Signal Processing-ICASSP'07*, vol. 1. IEEE, 2007, pp. I–249.

[17] A. De Cheveigné and H. Kawahara, "Yin, a fundamental frequency estimator for speech and music," *The Journal of the Acoustical Society of America*, vol. 111, no. 4, pp. 1917–1930, 2002.