

REAL TIME VOICE PATHOLOGY DETECTION USING AUTOCORRELATION
PITCH ESTIMATION AND SHORT TIME JITTER ESTIMATOR

by

Maria Astrinaki

A thesis submitted to the faculty of

University Of Crete

in partial fulfillment of the requirements for the degree of

Master of Science

Computer Science Department

University of Crete

June 2010

Copyright © 2010 Maria Astrinaki

All Rights Reserved

UNIVERSITY OF CRETE
SCHOOL OF SCIENCES AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

**REAL TIME VOICE PATHOLOGY DETECTION USING AUTOCORRELATION
PITCH ESTIMATION AND SHORT TIME JITTER ESTIMATOR**

thesis submitted by
Maria Astrinaki
in partial fulfillment of the requirements for the M.Sc. degree in Computer Science

Author:

Maria Astrinaki, Department of Computer Science

Committee Approval :

Yannis Stylianou, Associate Professor, Department of Computer Science, University of Crete, Supervisor

Athanasios Mouchtaris, Assistant Professor, Department of Computer Science, University of Crete

Yannis Tzitzikas, Assistant Professor, Department of Computer Science, University of Crete

Department Approval :

Panos Trahanias, Professor, Director of Postgraduate Studies, Department of Computer Science, University of Crete

Heraklion, July 2010

ABSTRACT

REAL TIME VOICE PATHOLOGY DETECTION USING AUTOCORRELATION PITCH ESTIMATION AND SHORT TIME JITTER ESTIMATOR

Maria Astrinaki

Computer Science Department

Master of Science

Voice is the result of the coordination of the whole pneumophonoarticulatory apparatus. Voice pathologies have become a social concern, as voice and speech play an important role in certain professions, and in the general population quality of life. The analysis of the voice allows the identification of the diseases of the vocal apparatus and currently is carried out from an expert doctor through methods based on the auditory analysis. In these last years emphasis has been placed in early pathology detection, for which classical perturbation measurements (jitter, shimmer, HNR, etc.) have been used. Going one step ahead the present work is aimed to implement a real time voice pathology detection system, combined with a Java interface.

ACKNOWLEDGMENTS

First, I thank my supervisor, professor Yannis Stylianou, for his continuous support in the M.Sc. program. Professor Stylianou was always there to listen and to give advice. He is responsible for involving me in research in the first place. He showed me different ways to approach a problem and the need to be persistent to accomplish any goal. I am grateful for his guidance, encouragement, motivation and trust during this work, and in general for his support all these years of collaboration.

I also thank professor Dutoit Thierry, that I had the opportunity to meet and collaborate with in the TCTS Lab of the Faculte Polytechnique de Mons, for his guidance, support and hospitality, but specifically I am indebted for giving me the opportunity to explore new ideas, to gain new perspective of cognitive science and computing and also to meet amazing people.

Thanks also to Nicola d'Alessandro, who taught me how to ask questions and express my ideas. Nicola has been a friend and mentor. He made me a better programmer, had confidence in me when I doubted myself, and brought out the good ideas in me. I am glad to have such people in my life.

I also thank my colleagues, past and present, with whom I shared my days at the Multimedia Informatics Laboratory, and made it a wonderful workplace and home for the past years. Special thanks should go to Yannis Agiomyrgiannakis, Miltiadis Vasilakis, Ioannis Pantazis, Andre Holzapfel, Maria Markaki, George Kafentzis, Christos Tzagarakis, Maria Koutsogiannaki,

George Tzedakis, Christina-Alexandra Lionoudaki and George Grekas.

Let me also say “thank you” to the following people : Sophie Karagiorgou, Elias Apostolopoulos, Elena Karamichali, Frixos Terzakis, Takis Gerovasilis, Katia Lampropoulou, Elias Tragos, Mariana Karmazi, Georgina Tryfou, Angelos Giannakopoulos, Krontiris Thanasis, Diamantis Antoniou, Lito Michala, Benoit Vryghem and Nikos Peteinarakis, who were always there to meet and talk, helping me at any time, and solving any unsolvable problems, having confidence in me and supporting me, and consuming incredible amounts of coffee. I thank you for your love and friendship that made me a better person and eased my way though life.

Last, but not least, I thank my family: my parents, Yannis Astrinakis, and Georgia Yperifanou, for giving me life in the first place, for educating me with aspects from both arts and sciences, for unconditional support and encouragement to pursue my interests, even when the interests went beyond boundaries of language, field and geography. My sister Nina, for sharing her experiences, for listening to my complaints and frustrations, and for believing in me. My brother Nico, for his unconditional love and bright big smile, that means the world for me.

Contents

Table of Contents	ix
List of Figures	xi
1 Introduction	1
1.1 What is Voice and Voice Dysphonia	1
1.2 Literature Overview	2
1.3 Real Time Approach	4
2 Background Information	7
2.1 Spectral Jitter Estimation Overview	7
2.2 What is Pure Data	10
2.3 What is Processing	12
2.4 What is OpenSoundControl	13
3 Description of the Real Time Voice Pathology Detection System	15
3.1 Main Programming Difficulties	15
3.1.1 Introduction to a new programming style	15
3.1.2 “Features” of Pure Data	16
3.2 Implementation of Pitch Detection in Pure Data	18
3.3 Externals and Patches Implemented for Autocorrelation Pitch Detec- tion in Pure Data	21
3.4 Implementation of SJE in Pure Data	24
3.4.1 Definition of computation frame size	24
3.4.2 Actual SJE Implementation	26
3.5 Externals and Patches Implemented for Spectral Jitter Estimator in Pure Data	28
3.6 Other Main Patches and Externals in Pure Data	31
3.7 Implementation of the Interface in Processing	34
4 Results	39
4.1 Synthetic signals	39
4.2 Actual signals	41

5	Conclusions	43
5.1	Conclusions	43
5.2	Summary of Contributions	44
5.3	Future Research	44
	Bibliography	47
A	User Manual	51
B	Source Code Documentation	55
C	Source Code	57
D	List of implemented Patches and Externals	59
D.1	Patches	59
D.2	Externals	60

List of Figures

2.1	Jitter Mathematical Model for different values of pitch deviation . . .	8
2.2	Jitter Enhancement Example	10
2.3	Flow graph of the SJE implementation	10
2.4	Simple Pure Data Patch Example	11
3.1	Simple Audio and Numerical Pure Data Objects	17
3.2	Oscillation Autocorrelation	18
3.3	Flow graph of Autocorrelation Method	19
3.4	Autocorrelation and Energy Patches	19
3.5	Flow graph of Autocorrelation Pitch Estimation	20
3.6	Autocorrelation Pitch Estimation Patch	21
3.7	Invert Clipping Object	23
3.8	Zero padding Object	24
3.9	Example of the Pure Data block object	25
3.10	Flow graph of the SJE implementation	27
3.11	SJE Patch	27
3.12	Linear Interpolation	28
3.13	Hanning Window	31
3.14	Pure Data Main Patch for Applying SJE to a Recorded File	32
3.15	Pure Data Main Patch for Applying SJE Directly to the Microphone's Input	33
3.16	Processing Environment	34
3.17	Processing Environment	35
3.18	SJE Bar Graph Interface	35
4.1	Pitch Estimation Error	40
4.2	Synthetic Jitter Estimations	40
4.3	Actual Signal Jitter Estimations	42

Chapter 1

Introduction

1.1 What is Voice and Voice Dysphonia

Voice is the result of a complex mechanism involving different organs of the respiratory system. More specifically, voice is the sound produced by the vibration of the vocal folds in the larynx, also called “voice box”. This sound, the larynx-fundamental tone, is then modified by the throat and mouth to produce speech.

Voice has a number of features:

1. Pitch: how high or low a voice is. During speech, pitch can vary in order to indicate meaning or emotion, also known as intonation.
2. Loudness: how loud or soft a voice is. During speech, loudness can vary in order to show emphasis and emotion.
3. Voice Quality: how clear a voice sounds. A disordered voice may sound strained, hoarse, breathy and rough.
4. Resonance: modification of voice as it passes through the throat, mouth and nose.

Any modification of this system, is likely to change one or more of these features, and as a result may cause a qualitative and/or quantitative alteration of the voice, which is defined as dysphonia. Dysphonia can be organic (Laryngitis, Neoplasm, Trauma, Endocrine, Hematological, and Iatrogenic) or functional (Psychogenic, Vocal misuse, Idiopathic), [1].

Although dysphonia is, in most cases, one of the main symptoms of mild laryngeal diseases, such as polyps or nodules, it is also the first symptom of neoplastic diseases such as laryngeal cancer. A major characteristic of these diseases is that they modify the structure of the “voice production system”; there are air flow turbulences in the vocal tract, mainly due to irregular vocal folds vibration and/or closure, resulting to spectral “noise”. In this case, is required a set of endoscopic analysis, by using videolaryngoscope (VLS) for a more accurate diagnosis.

However, dysphonia is often underestimated by patients, since it does not have severe symptoms. Hence, some delay in diagnosis is often found in case of neoplastic laryngeal diseases. As in all cases, early detection is the basic importance for pathology recovering.

1.2 Literature Overview

Within the medical environment, diverse techniques exist to assess the state of the voice of a patient. In order to make a diagnosis, clinicians and speech therapists usually combine different techniques to evaluate voice pathologies:

- Video endoscopy methods that use an image or video recording of the vocal folds, such as videokymography.
- Electroglottography (EGG) methods that use the measurement of the electrical resistance between two electrodes placed around the neck, through which the

vocal fold contact area can be estimated.

- Acoustic analysis methods that use recordings of the radiated speech signal to compute parameters related to pathological voice.

The visual inspection technique has the great advantage that is very precise, but on the other hand it is also very expensive and time consuming. The main disadvantage though, is that causes great discomfort to patients. Besides invasive measurement, EGG measurements are not robust to noise, sometimes are not possible to be measured, and patients feel uncomfortable in carrying the device. On the other hand, acoustic analysis is low-cost and non-invasive and provides objective measurements which allow the quantification of voice quality. Moreover these results can be further used for unsupervised classification of a voice as pathological or normal, [24], or even detect specific cases of dysphonia.

Therefore, there are many algorithmic approaches for the automatic signal analysis. A large number of works have focused on the automatic detection and classification of voice pathologies, by means of acoustic analysis, parametric and non-parametric feature extraction, automatic pattern recognition or statistical methods.

There are also Software tools, commercial and freely available, that allows efficient voice component manipulation. The most widely used are Multi Dimensional Voice Program (MDVP), [3], Praat, [4], Wavesurfer, [5], WinPitch, [6] and VOICEBOX [7].

Much focus has been centered on perturbation analysis measures such as jitter and shimmer, and on signal-to-noise ratios of voiced speech, which reflect the internal functioning of the voice. Noise is mainly the immediate effect of a pathological condition. There are two kinds of noise, that of additive noise, such as in cases of breathiness, and that of modulation noise, such as in cases of roughness.

In the current literature, voice quantification is based on Short-Term Jitter Estimations, [2]. Jitter is defined as the periodicity modulation of the voice signal. During sustained vowel phonation, periodicity modulations can be defined in the context of the glottal source signal as perturbations around a constant value of the glottal excitation pitch period. A high degree of jitter results in a voice with roughness that is usually perceived in recordings of pathological voices. Hence, a reliable estimation of jitter can be used to discriminate between healthy and dysphonic speakers. This method is extensively described in the Background Information Section.

1.3 Real Time Approach

Pathological voice recognition has been received a great attention from researchers in the last decade. Speech processing has proved to be an excellent tool for voice disorder detection. There has been great evolution in new algorithmic approaches, and continuous improvement off the existing. The persisting problem though, is that most of the research is focused on offline or non-real time approaches. Little or none have been made for the implementation of a real time system using acoustic analysis for voice pathology detection. There is significant lack of research in converting offline methods to real time, or proposing pure real time approaches.

In this work we present the conversion of an originally designed offline method to a real time application combined with a Java Applet interface. The selected method was Spectral Jitter Estimation, SJE, [8], which is based on short-term measurements of jitter. SJE provides us with robust pathology detections, against the pitch period estimators. Pitch period estimators are based on the pseudo-periodicity in voiced speech, and that leads to ambiguity in the estimation of jitter. Although the SJE uses pitch period information, it was proved that this is not essential for the performance

of the algorithm.

The main goal of this work was the creation of a real time pathology detector, with a friendly user interface. The proposed system consists of three main parts, that are described in extend in the following chapters. The real time pitch estimator part, which provides us with pitch estimations using autocorrelation model for each frame, the actual real time SJE implementation, that uses the estimated pitch to estimate in real time current jitter values. The third part is the interface of the system which visualizes the jitter estimations. Instead of giving plane numbers or point plots as final results, there is a real time graphical visualization consisting of colored bars, that represent the current estimated jitter value.

The contents of this work are organized as follows. A general overview of the Spectral Jitter Estimator, and of tools that are used in this work, such Pure Data, Processing and OpenSoundControl, are provided in Chapter 2. In Chapter 3, the implementation and the extended description of the real time voice pathology detection system is presented. The programming difficulties, the actual implementations of the pitch detector, the SJE and the interface, are also shown. Externals and the code of the interface are presented in details. Finally, all the results and extracted conclusions can be found in Chapter 4 and 5, respectively. In the Appendices A, B, and C there are the user manual, the source code and the documentation.

Chapter 2

Background Information

2.1 Spectral Jitter Estimation Overview

As Vasilakis suggests, [25], jitter can be defined by a mathematical model that describes two periodic events. Jitter is defined as cycle-to-cycle perturbations of the glottal cycle lengths, which leads to a local aperiodicity. This kind of perturbations can be modeled and generated by considering two periodic events, which, when combined appropriately, may produce the observed perturbations. The local aperiodicity of jitter can be defined then, in relation to these two events, as the shift of one of the two with respect to the other. This shift can be measured to provide us with a quantitative value for jitter.

The mathematical jitter impulse train is :

$$g[n] = \sum_{k=-\infty}^{k=\infty} \delta[n - (2k)P] + \sum_{k=-\infty}^{k=\infty} \delta[n + \varepsilon - (2k + 1)P]$$

where P is the pitch period and $\varepsilon, \varepsilon \in [0, P]$, is the pitch deviation, both in samples. After Fourier transformation of this train we have :

$$G(\omega) = (1 + e^{-j\omega(P-\varepsilon)}) \sum_{k=-\infty}^{k=\infty} \frac{\omega_0}{2} \delta(\omega - k \frac{\omega_0}{2})$$

where $\omega_0 = \frac{2\pi}{P}$ is the fundamental frequency in rad. Then the square of the magnitude spectrum is computed, which leads to the log magnitude spectrum, which shows that there is the harmonic part of the log magnitude spectrum, as it is influenced by jitter, and the subharmonic part of the log magnitude spectrum, that appears because of the existence of jitter.

Harmonic part :

$$H(\varepsilon, l\omega_0) = 10\log_{10}\left(\frac{\omega_0^2}{2} + (1 + \cos[(P - \varepsilon)l\omega_0])\right), l \in N$$

Subharmonic part :

$$S(\varepsilon, (l + \frac{1}{2})\omega_0) = 10\log_{10}\left(\frac{\omega_0^2}{2} + (1 + \cos[(P - \varepsilon)(l + \frac{1}{2})\omega_0])\right), l \in N$$

In figure 2.1 are shown three examples of the two sub spectra, for values 0, 1 and 2 of ε . The circled intersections between the two parts, reveal each time the value of jitter.

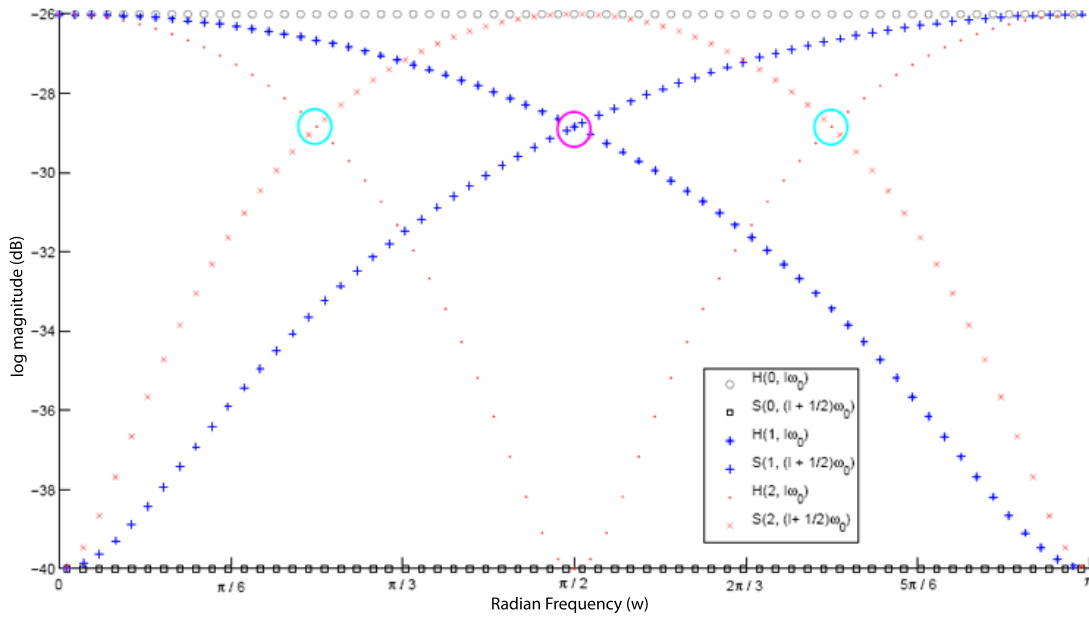


Figure 2.1 Log magnitude spectra of the harmonic and subharmonic parts of the mathematical jitter model.

Going further in the method, in order to compute short-time Spectral Jitter Estimations, we apply a Hanning window at each examined signal frame. The length of the window should be three or four times the period in order to have high enough resolution in the computed spectrum for the estimation. Fourier transformation is applied on the frame, and then the log magnitude spectrum of the Fourier transform is computed. By means of the local pitch period the magnitude spectrum is split into the harmonic and subharmonic spectra, as it was shown above, and the intersections between them are computed.

Often, there are resolution problems in the spectral magnitude, and this could provide “false” intersections. In order to avoid that, and all the computed intersections are valid, a threshold of 3 dB was defined. In other words, an intersection is classified as “valid” if the harmonic and subharmonic parts after its occurrence reach a difference in amplitude over the threshold. Otherwise this intersection is rejected. The valid intersections are further examined, by taking into account the prior knowledge of their expected locations. In more details, the highest possible value of jitter is the number of “valid intersections”. The spectrum is then divided in that number of equal segments, containing at least one existing intersection around near its center. If this is true then this jitter value is accepted. In the other case, it is rejected, the value is decreased by one, and this process is repeated until jitter reaches its limit, zero. In order to avoid pseudo-intersections that are considered valid due to the previous threshold, this process enhances the intersections by eliminating them supposed intersections, and grouping neighboring ones in clusters, as it is shown in Figure 2.2. In this example, SJE estimates jitter to $\varepsilon = 8$ samples, but after the enhancement jitters is estimated to $\varepsilon = 3$ samples.

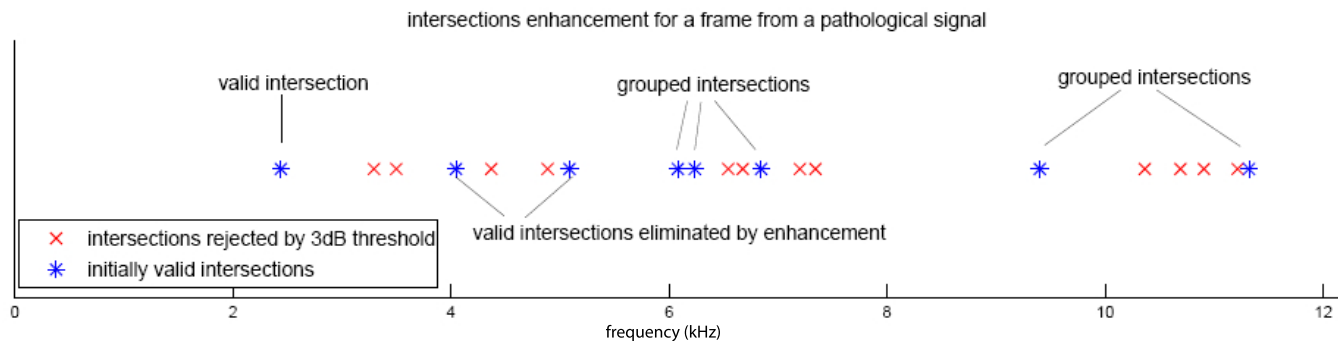


Figure 2.2 An example of intersections enhancement for a frame from a pathological signal.

The main idea of the SJE algorithm can be summarized in the data flow diagram presented below:

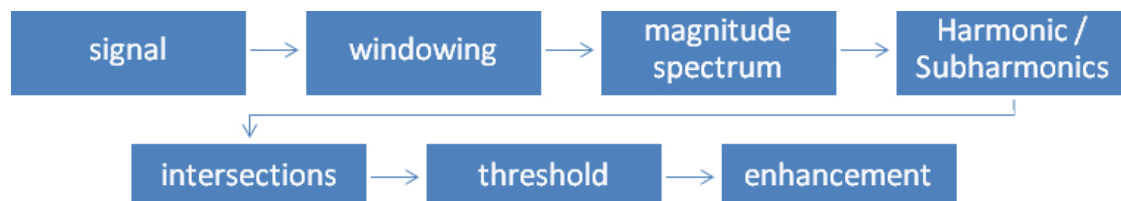


Figure 2.3 Flow graph of the signal manipulation information in order to implement SJE, and finally compute jitter.

2.2 What is Pure Data

Pure Data, also referred as PD, [9], is a real-time graphical programming environment for audio and graphical processing. It has been developed by Miller Smith Puckette at IRCAM in the 1990s for the creation of interactive computer music and multimedia works. It is open source software and was written for multiplatform from the beginning and therefore is portable. It runs at least on Win32, IRIX, LINUX and Mac-OSX platforms. PD is very similar in scope and design to Puckette's original Max program (developed while he was at IRCAM), and is to some degree interoperable with Max/MSP, [10], the commercial successor to the Max language.

As the PD community and developers grows rapidly past years contributing more and more library objects and patches, many collections of patches and externals (so-called “libraries”) are already available. This contribution resulted in the creation of Graphics Environment for Multimedia (GEM) external, and many other externals were designed to work with it, so it is possible to create and manipulate video, OpenGL graphics, images, etc. in real-time with seemingly endless possibilities for interactivity with audio, external sensors, etc. Additionally, PD is natively designed to enable live collaboration across networks or the internet, allowing musicians connected via LAN or even in different parts of the globe to interact in real time.

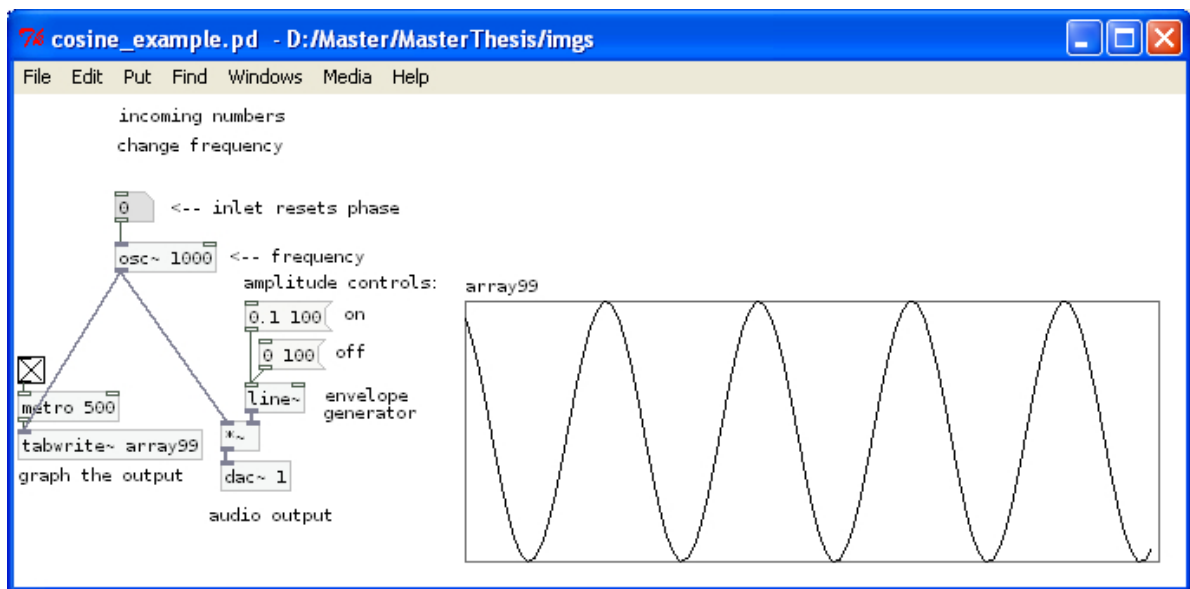


Figure 2.4 A simple Pure Data Patch Example implementing a cosine oscillation.

Both PD and Max are representative examples of dataflow programming languages. In such languages, functions or “objects” are linked or “patched” together in a graphical environment which models the flow of the control and audio. It is easy to extend PD with third-party plug-ins, so-called “externals”, [11]. This makes the program easily extensible through a public API and encourages developers to add

their own control and audio routines, mainly using the C programming language. However, PD is a programming language in its own right. Modular, reusable units of code written natively in PD, called “patches” or “abstractions”, are used as standalone programs and freely shared among the PD user community. Many useful information, abstractions and answers can be found in the Pure Data forum, [12].

Therefore, Pure Data is used as a rapid prototype-software for DSP-algorithm, as a music instrument and universal multimedia software and is also used in the academic field for teaching and exploring computer music, voice analysis and new media art.

2.3 What is Processing

Processing was initiated by Benjamin Fry, [13], and Casey Reas, [14]. In Fall 2001, formerly of the Aesthetics and Computation Group, [15], at the MIT Media Lab, [16]. We can see it as the follow up of Design by Numbers, [17], created by John Maeda, [18], a programming environment strongly oriented at beginners. It is an open source project free to download and free to use, available for GNU/Linux, Mac OS X, and Windows, with already a large community around it.

As the Processing web site mentions: “Processing is an open source programming language and environment for people who want to program images, animation, and interactions. It is used by students, artists, designers, researchers, and hobbyists for learning, prototyping, and production. It is created to teach fundamentals of computer programming within a visual context and to serve as a software sketchbook and professional production tool. Processing is developed by artists and designers as an alternative to proprietary software tools in the same domain”, [19].

One of the stated aims of Processing is to act as a tool to get non-programmers

started with programming, through the instant gratification of visual feedback. The language builds on the graphical capabilities of the Java programming language, simplifying features and creating a few new ones. Those picking it up for the first time will find lots of very well written and archived code samples, a large amount of information on the internet and answers to question on the Processing forum, [20]. The people behind Processing, [21], have made huge efforts in helping the community to grow and learn. Even if Processing was aimed at beginners, its versatility and ease of development still is relevant to anyone wanting to write creative software pieces.

2.4 What is OpenSoundControl

Open Sound Control, referred to as OSC, [22], is a protocol for communication among computers, sound synthesizers, and other multimedia devices that is optimized for modern networking technology. Bringing the benefits of modern networking technology to the world of electronic musical instruments, OSC's advantages include interoperability, accuracy, flexibility, and enhanced organization and documentation.

This simple yet powerful protocol provides everything needed for real-time control of sound and other media processing while remaining flexible and easy to implement.

OSC Features:

1. Open-ended, dynamic, URL-style symbolic naming scheme.
2. Symbolic and high-resolution numeric argument data.
3. Pattern matching language to specify multiple recipients of a single message.
4. High resolution time tags.
5. "Bundles" of messages whose effects must occur simultaneously.

6. Query system to dynamically find out the capabilities of an OSC server and get documentation

There are dozens of implementations of OSC, including real-time sound and media processing environments, web interactivity tools, software synthesizers, a large variety programming languages, and hardware devices for sensor measurement. OSC has achieved wide use in fields including computer-based new interfaces for musical expression, wide-area and local-area networked distributed music systems, inter-process communication, and even within a single application.

OSC was originally developed, and continues to be a subject of ongoing research at UC Berkeley Center for New Music and Audio Technology (CNMAT), [23].

Chapter 3

Description of the Real Time Voice Pathology Detection System

3.1 Main Programming Difficulties

3.1.1 Introduction to a new programming style

At the moment that the implementation of the system started we faced many difficulties. The first, and at the same time the biggest one, was the nature of a data flow language, and especially the real time data flow programming. Here, a program is modeled, conceptually, as a directed graph of the data flowing between operations. There are some common features with the functional languages, but main concepts are implied, or even absent. The priority of a computation depends on the position of the “object”, left or right, higher or lower in the “graph”. Concepts such as “if”, “for” do not have a direct implementation, which is not always clear to use, and in many cases give ambiguous results. This programming style is not widely used, and to change from one style to the other, and at the same time resulting in writing successful, working patches is time demanding.

3.1.2 “Features” of Pure Data

Even though Pure Data provides many, small, simple objects, in order to implement a so demanding system, such as SJE, the use of externals was essential. At that point several small, but serious problems appear, all in relation to some “features” of Pure Data.

A great problem with this environment is the absence of a debugger. There is no other way to debug a patch other than “by hand”, just by using “print” commands across the process. This, by no means, is an acceptable debugging technique, but it can work in a small scale patch/external. When the patch/external size and number increases, and there are dependences between them, the problem is more than obvious, and a debugging tool is absolutely necessary.

Moreover, we came across with synchronization problems. Pure Data consists of two main parts, the data or numerical part, which is asynchronous, and the audio part, which is synchronous. Generally in the numerical part the input in each DSP cycle is one single number, whereas in the audio part, the input is a block of numbers which is further manipulated as a concrete set. Comparing the two aspects, the audio thread is certainly much faster but more difficult to manipulate, and the dataflow part is easier but a lot slower.

Figure 3.1 presents the three types of Pure Data Objects. The first object has two data inlets and one data outlet, and implements the addition of these two inlets. The second object also implements the addition of its two inlets, but in this case the inlets are audio, block of numbers. This means that the outlet contains a block of number with the same size as the inlet, and each of its elements is the result sample by sample addition. If we give a closer look we can see that the audio I/O are bold, whereas the numerical is not. The third presented object, is a cosine wave oscillator; it has one audio inlet, one numerical inlet and one audio outlet. The numerical inlet

is only used for the initialization of the frequency. In order to change the frequency, the audio inlet has to be used. In the outlet a cosine wave oscillation with the given frequency is produced.

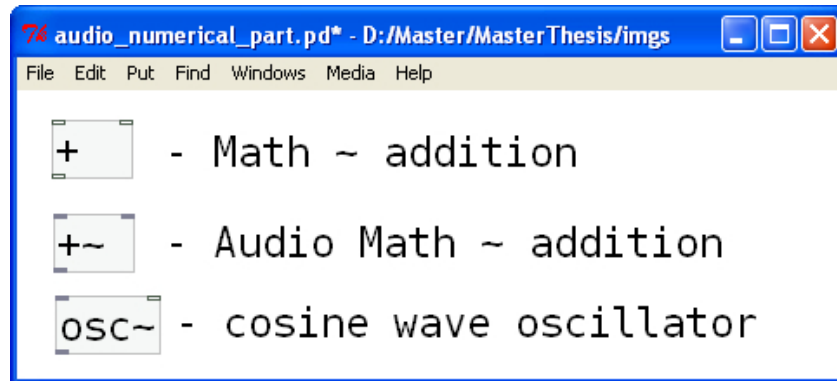


Figure 3.1 Mathematical Addition Object, Audio Addition Object, Cosine Wave Oscillator Object.

The synchronization problem lays on the fact that these two threads are not synchronous. This means that we cannot combine dataflow objects with audio objects, and even if we do, we have to be very conservative and careful. To be more accurate, since PD is mainly applied in music, this synchronization is not really a problem, because the ear cannot detect it. Though in pathology detection it is essential everything to be synchronized, in order to have a reliable results.

To solve this problem there were two alternatives; either the whole platform should be tweaked from the source code, re-synchronize the two threads, and then re-compile it, or implement all the necessary existing data objects as new externals that build corresponding audio objects. The first approach was rejected, because of the lack of knowledge for this kind of system manipulation as well as the inability to debug it. The second approach was achievable, reliable and efficient but also time consuming. Each of these externals that were created will be described extensively in the implementation section below.

3.2 Implementation of Pitch Detection in Pure Data

In order to have a real time pitch estimator for our system we decided to use the autocorrelation pitch estimation algorithm. The main reason for this selection was that for this estimator there are no dependencies in past or future windows for the final pitch computation, and also is relatively impervious to noise. Even if autocorrelation is computationally extremely expensive, by using adaptive computation techniques can be expedited and run in real time. The main disadvantage is that it is very sensitive to sampling rate. This is due to the fact that fundamental frequency is calculated directly from a shift in samples; the lower the sampling rate is, the lower the pitch resolution is.

From the mathematical point of view, we have that for a given signal $f(t)$, the continuous autocorrelation $R_{ff}(\tau)$ is defined as the continuous cross-correlation integral of $f(t)$ with itself, at lag τ .

$$R_{ff}(\tau) = \bar{f}(-\tau) * f(\tau) = \int_{-\infty}^{\infty} f(t + \tau) \bar{f}(t) dt = \int_{-\infty}^{\infty} f(t) \bar{f}(t - \tau) dt$$

where \bar{f} represents the complex conjugate and $*$ represents convolution. For a real function, $\bar{f} = f$. The discrete autocorrelation R at lag j for a discrete signal x_n is

$$R_{xx}(j) = \sum_n x_n \bar{x}_{n-j}$$

In Figure 3.2 is given the autocorrelation of an oscillation.



Figure 3.2 Oscillation Autocorrelation.

The main idea of autocorrelation is presented in figure 3.3.

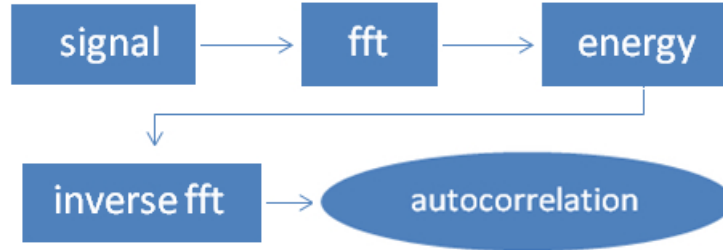


Figure 3.3 Flow graph of the Autocorrelation Method.

Now for the autocorrelation patch implementation we take as input the current signal block, which then becomes the input for the Fast Fourier Transform, `fft`, object. After `fft` is applied, we have the real and imaginary part, which gives us the energy of the input. If then we apply inverse `fft` on the energy we compute the autocorrelation. This processes is what patches `autocorr~` and `nrg~` implement and it is shown in Figure 3.4.

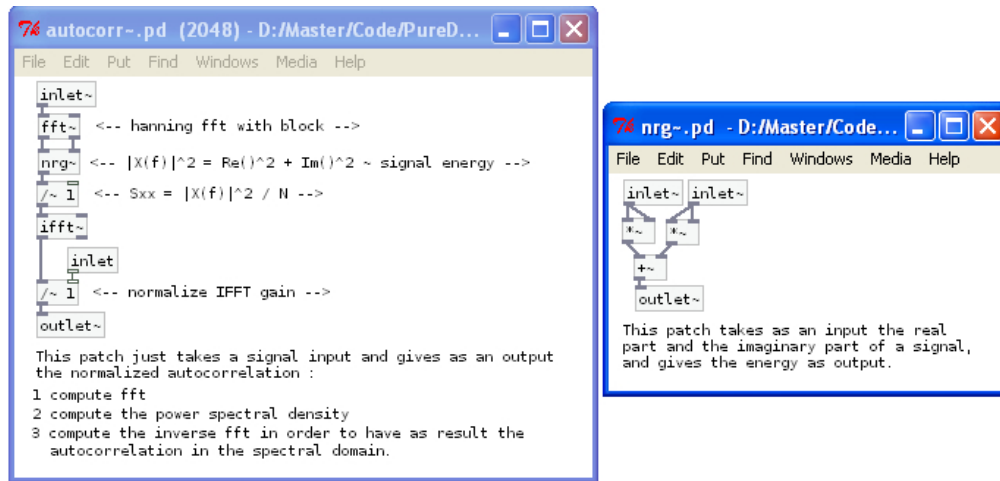


Figure 3.4 The first patch implements the Autocorrelation model, whereas the second patch implements the energy computation.

Since the basic autocorrelation model is implemented, the next step is to use it combined with some improvements in order to have a pitch estimation. This means that the initial signal will be centered, from that we will keep only the useful infor-

mation, and this information will be the input to the autocorrelation patch. From the autocorrelation signal will be extracted a frequency, which corresponds to pitch.

This could be easily represented in a graph diagram, figure 3.5.



Figure 3.5 Flow graph of the signal manipulation information in order to extract pitch, by using autocorrelation model.

The process is very simple, even if the description seems a little complicated. At the beginning we estimate the maximum and minimum value of the input, and then we subtract its mean value in order to center it. The centered signal, its initial maximum and minimum value become input arguments for the object “invclip~”, which implements inverted clipping of a signal block. With the usage of a clipping factor of 80% the desired output remains the same as the input if its sample value is greater than the 80% of its maximum, or less than 80% of its minimum, otherwise is set to zero; this object, among with the following externals are described in details in the next section. In other words, we keep only the samples with high energy from the initial input. Then we use the “zeropad~” object, which in this case sets to zero all the samples from the middle of the signal block until the end. The utility of that is that from the beginning we use two times bigger the block size we need, so after the autocorrelation computation, where the useful data size is half of the initial we will get the initial desired size. In order to avoid data loss due to the “virtual” extension of the block, we also use a “virtual” overlap of 50%. In this way, all the input is processed with the right frame size, without any data loss. Using this zero-padded signal, we apply the autocorrelation model, and we convert its output to frequency. This estimated frequency is the estimated pitch that we described above.

In the actual pitch extraction patch, figure 3.6, the computation steps become straightforward.

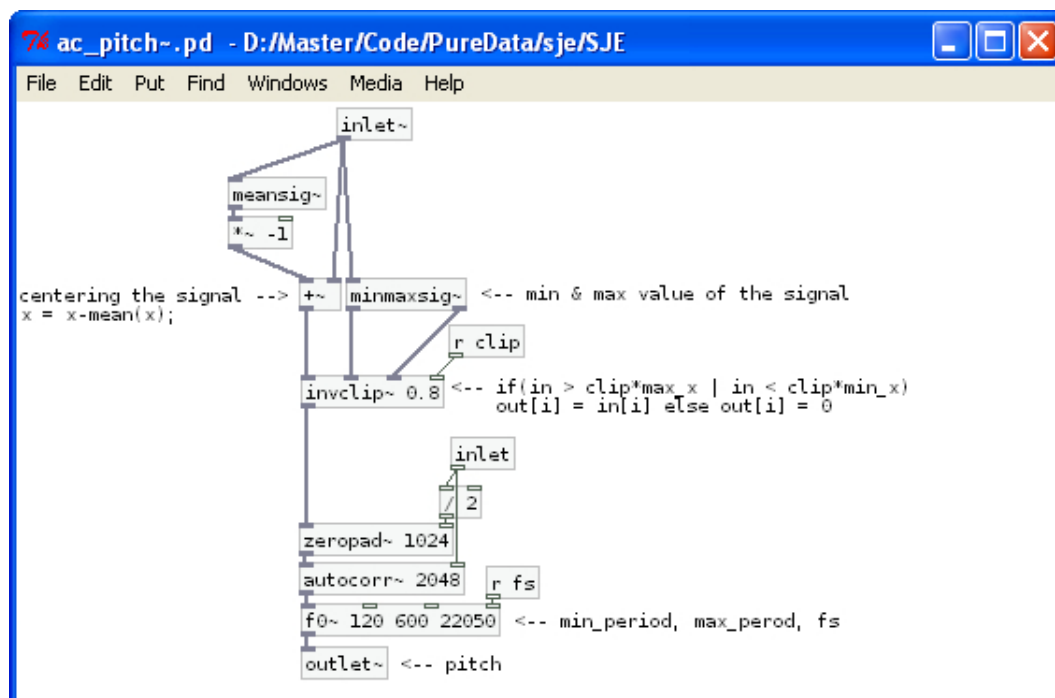


Figure 3.6 Actual Pure Data Patch for the implementation of Autocorrelation Pitch Estimation. The patch takes as input a signal and gives the estimated pitch as output.

3.3 Externals and Patches Implemented for Autocorrelation Pitch Detection in Pure Data

For the implementation of this segment of the system, there was need for the creation of some externals and some patches.

Patches:

1. `autocorr~`: This patch just takes a signal input and gives as an output the normalized autocorrelation, which was described in section 3.2.

2. `nrg~`: This patch takes as an input the real part and the imaginary part of a signal, and gives the energy as output.

The energy E_s of a continuous-time signal $x(t)$ is defined as

$$E_s = \langle x(t), x(t) \rangle = \int_{-\infty}^{\infty} |x(t)|^2 dt$$

Similarly, the spectral energy density of signal $x(t)$ is

$$E_s(f) = |X(f)|^2$$

where $X(f)$ is the Fourier transform of $x(t)$.

Externals:

1. `meansig~`: computes the mean value of a signal block

This object takes as input a signal and gives to the output a signal block containing the mean value. Its mathematical definition is :

$$\bar{x} = \frac{1}{n} \sum_0^n x_i$$

2. `minmaxsig~`: computes the minimum and maximum value of a signal block

This object takes as input a signal and gives to the left output a signal block containing the minimum value, and to the right output a signal block containing the minimum value.

3. `invclip~`: inverted clipping of a signal block

This object takes as main inputs a signal, the minimum and maximum of the input and as secondary input a clipping factor. Wherever the input signal is greater than “clipping factor”% of its maximum value, or less than “clipping factor”% of its minimum value, the output is a signal block equal to the input, everywhere else is set to zero. The mathematical approach of `invclip` is :

$$clipping_{A,m}(x) = \begin{cases} x(m), & |x(m)| > A \\ 0, & otherwise \end{cases}$$

where $m = 0, 1, 2, \dots, N$, with N the size of the window.

In figure 3.7 we give a graphical example of a signal before and after inverted clipping.

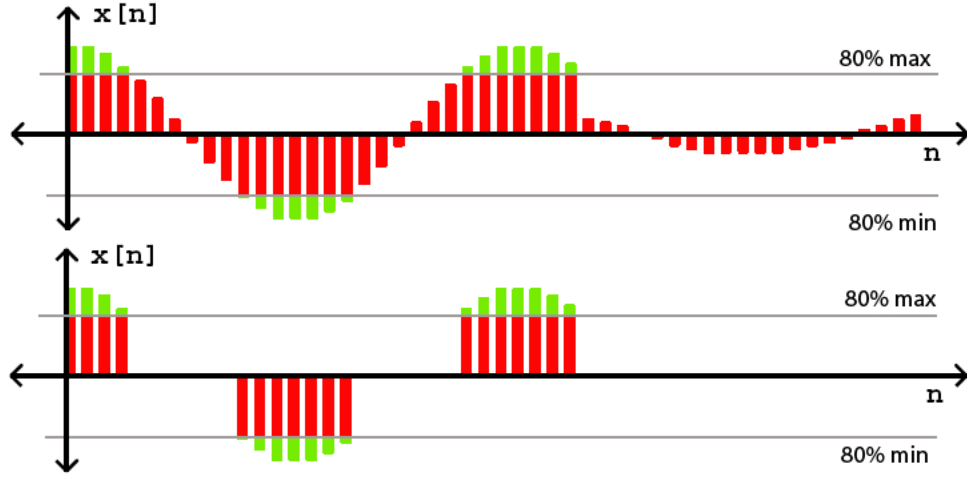


Figure 3.7 A signal of 50 samples before using invclip, (part a), and the same signal after applying “invclip”, (part b).

4. zeropad~: zero-padding to a signal block

In order to have zero-padding in pure data, we were obligated to use two times the normal size of the wanted window. So after the zero padding the valid values will have the right length. This object takes as main input a signal and as secondary input an integer. The output is a signal block equal to the input from zero indexes until the integer input index. The remaining of the output is set to zero.

The mathematical definition : Zero padding consists of extending a signal, or spectrum with zeros. It maps a length N signal to a length $M > N$ signal.

$$\text{zeropad}_{M,m}(x) = \begin{cases} x(m), & |m| < N/2 \\ 0, & \text{otherwise} \end{cases}$$

where $m = 0, \pm 1, \pm 2, \dots, \pm M_h$, with $M_h = (M - 1)/2$ for M odd, and $M/2 - 1$ for M even.

A graphical example of a signal before and after zero padding is presented in figure 3.8.

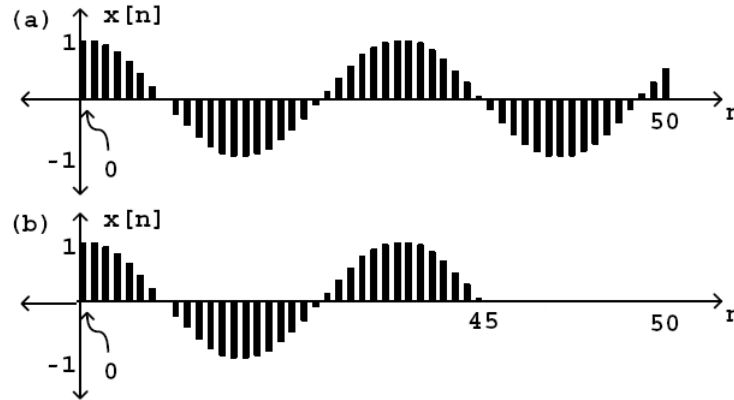


Figure 3.8 A signal of 50 samples before using zero padding, (part a), and the same signal after applying “zeropad 45”, (part b) .

5. $f0\sim$: computing $f0$ of an autocorrelation signal block

This object takes as main input the autocorrelation signal, as secondary inputs the minimum period, the maximum period and the sampling frequency, and computes the $f0$. The output is a signal block containing the same value.

3.4 Implementation of SJE in Pure Data

3.4.1 Definition of computation frame size

Since the mathematical model of Spectral Jitter Estimation has been described in section 2.1, the next step is to proceed in its real time implementation.

From the beginning a great issue was set; which window size will be used for the estimations. Although the proposed frame size for the jitter estimation is three or four times the frame pitch, in Pure Data this is not possible. The reason why we cannot obtain this, is because of the design of the $block\sim$ object and the $switch\sim$ object. These

objects are responsible to set the block size, overlap and up/down-sampling ratio for the window. PD's default block size is 64 samples. The `inlet` and `outlet` objects re-block signals to adjust for differences between parent and sub-patch, but only power of two adjustments are possible. So for "normal" audio computations, all blocks should also be a power of two in size. Which cancels the initial approach of the algorithm. In figure 3.9 a simple example of `block~` object is given.

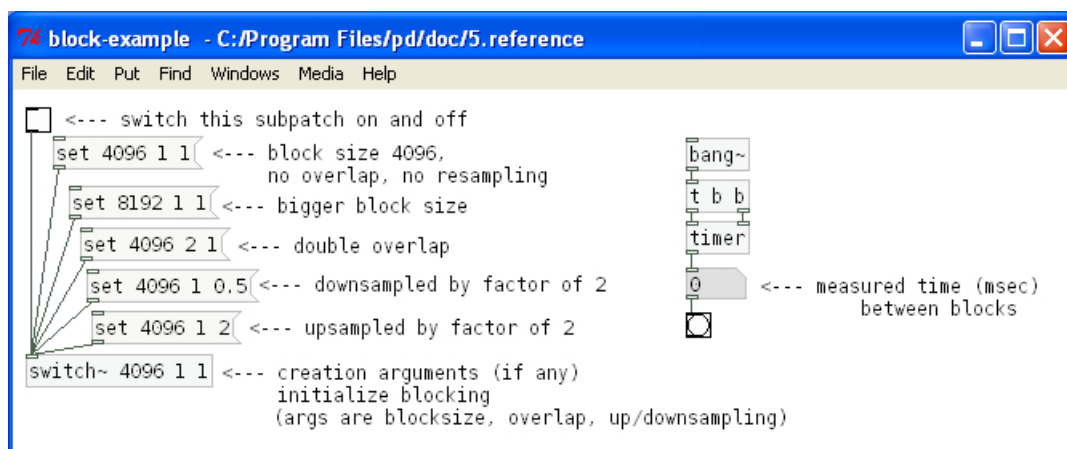


Figure 3.9 A simple example of the usage of the `block` object in sub patch.

Considering that the block size in Pure Data has to be always a power of two is one of the reasons that the suggested SJE window length cannot be implemented. The other one is that even if this limitation did not exist, due to the nature and design of the object, the idea of constantly changing window length results to data loss.

The only solution to that was to define a constant frame size. After extended algorithmic testing, it was shown that a block size of 1024 samples is as accurate as a block size of three times the pitch samples for the voice pathology detection.

In the following section, where the implementation of SJE will be presented, we will see that the arguments of the `block` object set the window length at 2048, the overlap at 50% and there is no over or down sampling factor. This is a virtual block

size. In fact, we use a window of 2048 samples, which half of it is zero padded, so the valid information is in the first 1024 samples. Then we use an overlap of half the block size so that there will not be any data loss. This is done because of the autocorrelation properties. In different case, for example if the window size was 1024 samples without any overlap, after the application of autocorrelation method the useful information would be only 512 samples.

3.4.2 Actual SJE Implementation

As long as we have concluded in the size of the computation block, the next step is the quality check of the incoming frame; whether is “silence”, “voice” or “unvoiced”. After that we continue with the full implementation of the mathematical jitter model as it was previously described.

To begin with, the received frame is checked if it a “silenced” frame or not. In case that it is silence, then it is set to zero, which results to infinite pitch, which then results to zero jitter. In the opposite case, the frame is checked if it is “voiced” or “unvoiced”. Respectively here, in case of “unvoiced” frames, the input is set to zero, which results to infinite pitch, which then results to zero jitter. Provided that the frame is neither “silence” nor “unvoiced”, the SJE estimation starts. The frame input is redirected to two main routes. From one route the pitch is extracted, as it was shown above, and from the other route we apply a hanning window, compute its Fourier transform, estimate the energy, and finally convert it to decibels so that the log magnitude spectrum will be obtained. When these two routes are computed, are passed as input arguments at the `harmonic_jitter_peaks ~` object, which computes the harmonic and subharmonic peaks of a signal block. Then, between these peaks we estimate their linear interpolation in order to estimate their intersections. This linear interpolation is made with the `lerp~` object and the estimation of the number of

intersections is achieved with the `intersections~` object which contains the thresholding limitation of three decibels. The final step is the enhancement of the just estimated intersections, which is computed by the object `intersections_enhancement~`.

In figure 3.10, we see a simple flow graph of the above described process, which is easily converted to a complete Pure Data patch, as it is shown in Figure 3.11.

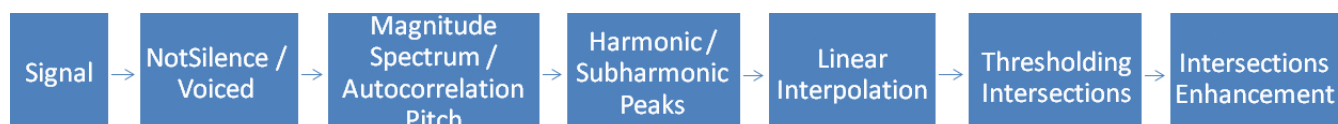


Figure 3.10 Flow graph of the signal manipulation information of SJE.

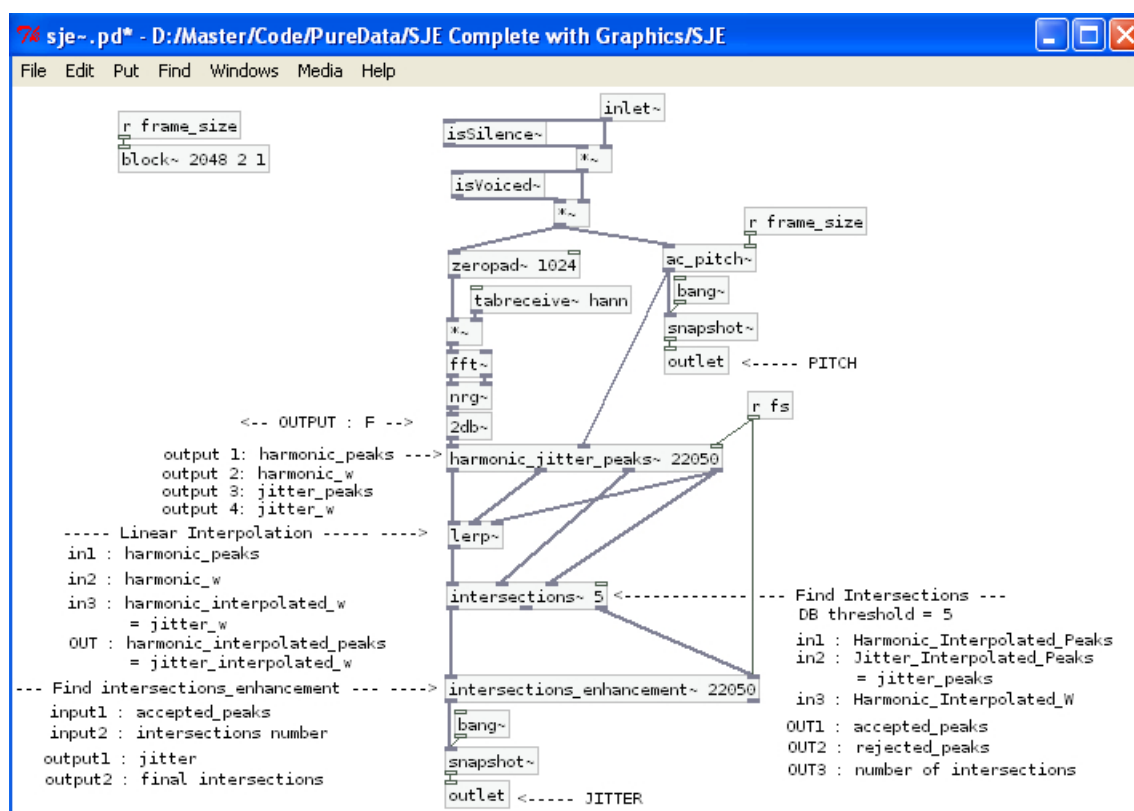


Figure 3.11 Actual Pure Data Patch for the implementation of SJE. The patch takes as input a signal and gives the estimated SJE as output.

3.5 Externals and Patches Implemented for Spectral Jitter Estimator in Pure Data

For the implementation of this segment of the system, there was need for the creation of some externals and some patches.

Externals:

1. `harmonic_jitter_peaks~`: computes the harmonic peaks of a signal block

This object takes as main input a signal in decibels, as second input its pitch, and gives to the output the harmonic peaks as a signal block.

2. `lerp~`: computes the linear interpolation as signal block

This object takes as input a signal containing the Y values, as second input a signal containing the X values, and as third a signal containing in-between values for the computation. The output gives a signal block containing the linear interpolation result.

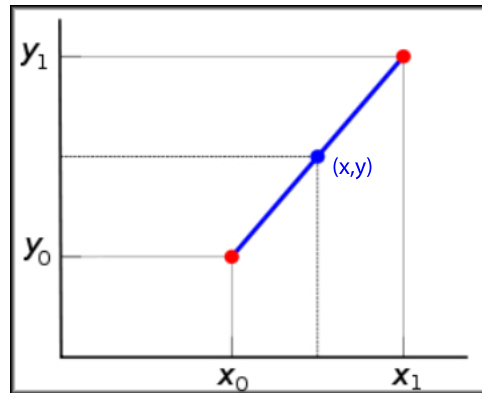


Figure 3.12 Given the two red points, the blue line is the linear interpolate between the points, and the value y at x may be found by linear interpolation..

$$y = y_0 + (x - x_0) \frac{y_1 - y_0}{x_1 - x_0}$$

3. intersections~: intersections between two signal blocks

This object takes as signal inputs the harmonic and jitter peaks. Computes its intersections, and if the deference between the two is bigger than the given threshold, then it is considered as a valid intersection, and is given as signal block output. If the difference is less than five decibels, then it is considered as a invalid intersection, and is also given as second signal block output. As third signal block output is considered total numbers of intersections.

4. intersections_ enhancement~: enhancement of intersections as a signal block

This object takes as signal inputs the accepted peaks and the intersections number that were previously estimated and gives as signal outputs the jitter and the final intersections.

5. sum~: This object takes as input a signal and gives the sum as a signal output.

$$sum = \sum_0^{N-1} x(n)$$

where N is the block size and $n = 0, 1, 2, \dots, N - 1$

6. subBlock~: This object takes as an input a signal, and two integers. The values of the input that have indexes between the input numbers are given to the output as a signal block. The rest values are set to zero.

$$sumBlock = \sum_A^B x(n)$$

where $0 < A < B < N$, N is the block size and $n = A, A + 1, \dots, B$

Patches:

1. std~: This patch takes as an input a signal, computes the standard deviation, and redirects it in the output.

Standard deviation of the sample, denoted by s_n and defined as:

$$s_n = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2}$$

2. 2db~: This patch takes as in input a signal and converts it to decibels.

When referring to measurements of power or intensity, a ratio can be expressed in decibels by evaluating ten times the base-10 logarithm of the ratio of the measured quantity to the reference level. Thus, if L represents the ratio of a power value P_1 to another power value P_0 , then LdB represents that ratio expressed in decibels.

$$L_{dB} = 10 \log_{10} \left(\frac{P_1}{P_0} \right)$$

When referring to measurements of field amplitude, as we do here, it is usual to consider the ratio of the squares of A_1 , measured amplitude, and A_0 , reference amplitude. This is because in most applications power is proportional to the square of amplitude, and it is desirable for the two decibel formulations to give the same result in such typical cases.

$$L_{dB} = 10 \log_{10} \left(\frac{A_1^2}{A_0^2} \right) = 20 \log_{10} \left(\frac{A_1}{A_0} \right)$$

3. moses~: This patch implements the “moses” Pure Data object, but not in the dataflow part of the environment, but in the audio part. Moses takes numbers and outputs them at left if they’re less than a control value and at right if they’re greater or equal to it.
4. hanning~: This patch fills the hanning array with a bit of audio oscillator (sin). When the bang is hit, the patch reset phase, update the frequency (if needed) and start the writing in the table. The synchronicity of these three operations makes the content of the array a Hanning window.

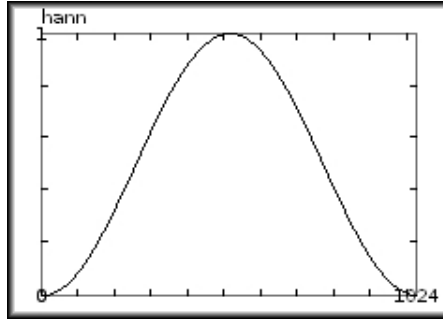


Figure 3.13 Hanning Window.

The Hann function, is a discrete probability mass function given by

$$w(n) = 0.5 \left(1 - \cos \left(\frac{2\pi n}{N-1} \right) \right) \text{ or } w(n) = \sin^2 \left(\frac{\pi n}{N-1} \right)$$

5. `isSilence~`: This path estimates if the input is a silence part or not. If the input is silence then the output is zero, else if the input is not silence then the output is one.
6. `isVoiced~`: This patch estimates if the input is a voiced part or not. If the input is unvoiced then the output is zero, else if the input is voiced then the output is one.

3.6 Other Main Patches and Externals in Pure Data

The main goal for the system was to be totally implemented by native Pure Data objects and patches. Unfortunately this was not possible; therefore we limited the usage of externals, only where it was absolutely necessary, and focused in making the best use of the possibilities provided by Pure Data. Indeed this resulted in small, reusable, stand alone patches. Most of them were presented in sections 3.3 and 3.5. Below we present additional implementations.

1. ReadWave: This patch takes as first input the name of the wav and stores it in the array voice. As output gives the length of the wav file.
2. recordWav: This patch takes as input a signal and records it into a wav file that the user has previously selected through the interface.
3. sendOSCdata: This patch takes as input a number and with the usage of OSC sends it to the interface application for the visualization.
4. receiveOSCdata: This patch receives an OSC message from the interface.

These implementation were used in the main patches that run the Spectral Jitter Estimator and communicate with the interface. There are two versions of the main patch. The first one is made for loading and testing existing files, “_main_file.pd”, (see in figure 3.14).

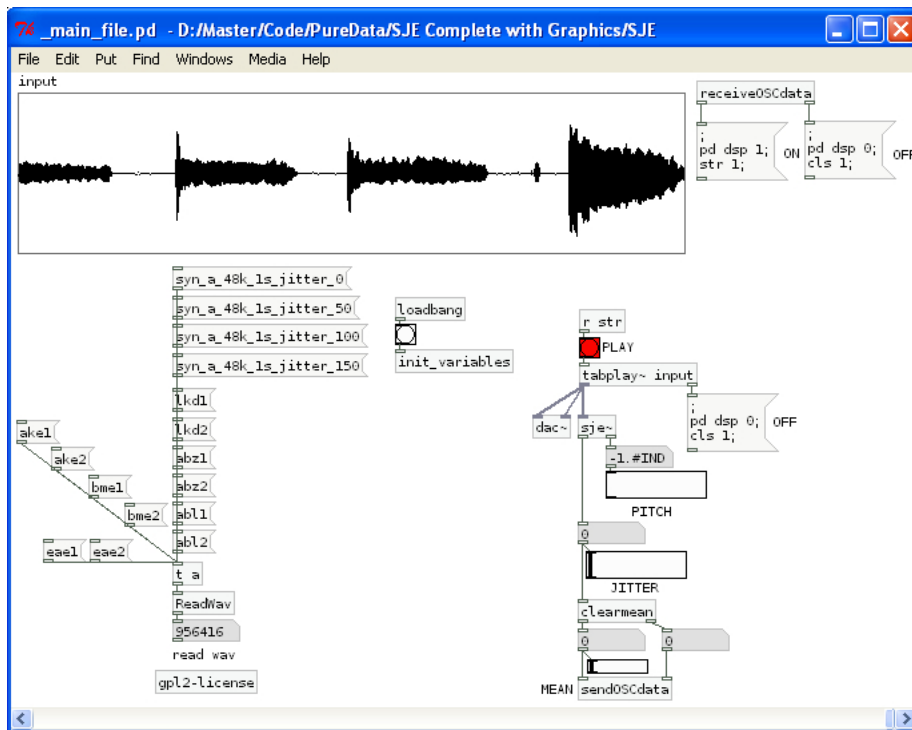


Figure 3.14 Pure Data Main Patch for Applying SJE to a Recorded File.

The moment the file opens, the object `init_` variables sets a first variable initialization, regarding the frame size, the overlap between frames, the “silence” threshold, the “voiced” threshold, etc. Then, by clicking on one of the recommended files, the corresponding wav file is loaded. If the DSP is on, either from the PD user or the interface, the SJE computation starts, and send the estimation back to the interface.

The second one, “_ main_ mic.pd”, (see figure 3.15), is made for testing voice input directly received from the microphone. The functionality is exactly the same as described above. The only deference is that now there is no file to be loaded. Once the DSP is on, the computation of SJE is applied to the microphone input. Here there is also an additional feature; the input recording option. By hitting the “Save” button a file name is defined, and by hitting the “Rec”, the recording starts until “Stop” is selected.

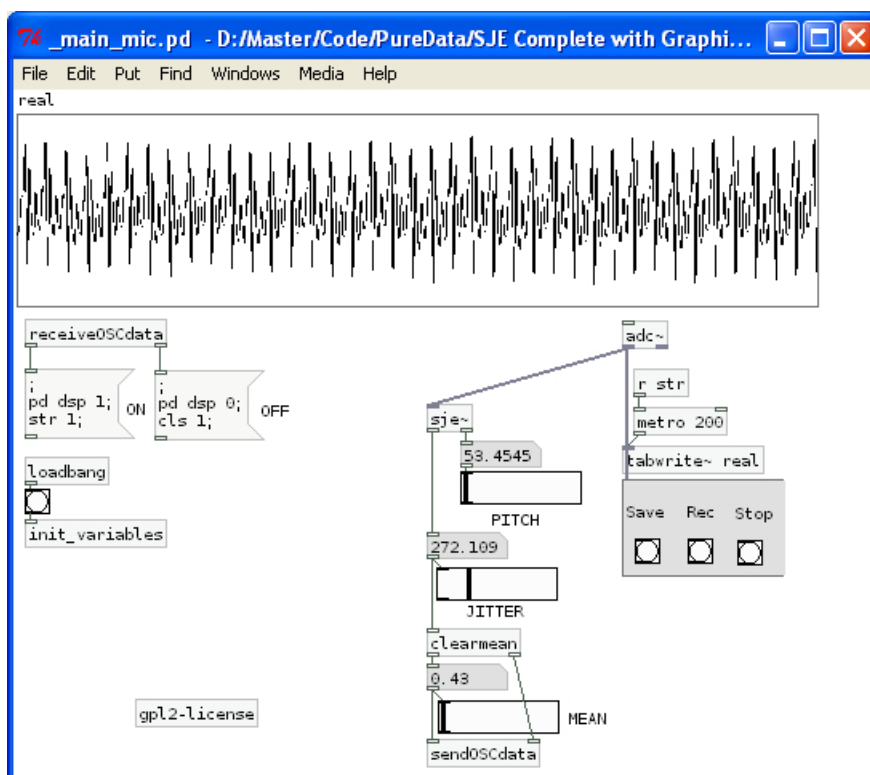


Figure 3.15 Pure Data Main Patch for Applying SJE Directly to the Microphone’s Input.

3.7 Implementation of the Interface in Processing

When spectral jitter is finally extracted it has to be presented in order the quality of voice to be evaluated. The visualization of the results is by itself a complicated task. In many cases, even for scientist or people familiar with numbers, the plane numerical representations or simple linear plots of the extracted information are not always easy to understand and assess; not to mention how difficult it is for people that are not accustomed to use point graphs and thresholds. Additionally a big problem with Pure Data is that the environment is not so friendly for the user to see and understand the graphs. In figure 3.16 we can see the poor visualization possibilities of Pure Data.

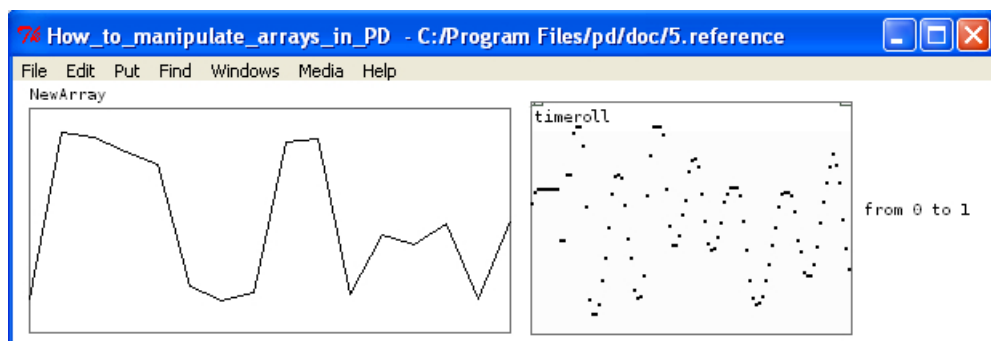


Figure 3.16 Examples of visualization possibilities of Pure Data, a simple array and the time roll object.

Hence, interface was implemented with Processing, figure 3.17, a Java based environment which was described in section 2.3.

The goal of this section of our system is to present the extracted estimations, in a way, that anyone can understand. For that purpose we chose that the data visualization should be with a color bar graph. Using ten estimated values, if only 45% or less of these values are above the pathological threshold then a green bar is “drawn” representing this percentage value. If 45% to 50% of these values are above the pathological threshold then we have a gray bar, since in this percentage area the

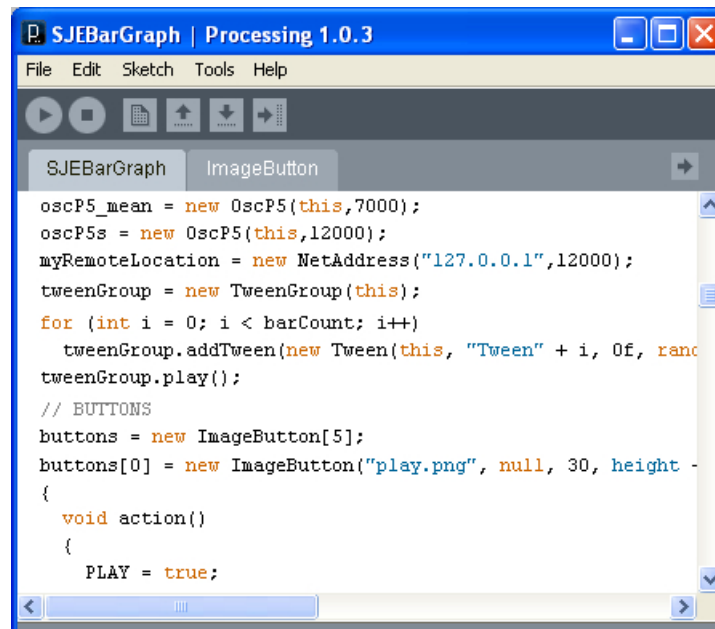


Figure 3.17 Processing Environment and part of the SJE Bar Graph Interface source code.

estimations are ambiguous. Finally, if the percentage of the pathological estimations is over 50% the bar is in red color. Under each bar appears the time of occurrence.

In figure 3.18 the SJE Bar Graph Interface is presented.



Figure 3.18 SJE Bar Graph Interface. The top first part of the graph, visualizes Local Jitter, whereas the second part of the graph, visualizes Updated Jitter.

The above presented interface is a very basic and simple approach. It consists of only two files; the “ImageButton.pde”, which is the class manipulating the button images and mouse events, and the “SJEBarGraph.pde”, which is the main file creating the graph.

For the development, the use of two libraries was necessary:

1. Ijeomamotionlib, [30]

As it is described in the ijeomamotionlib web site, “Motionlib is a library for doing time/frame based animations using tweens. Tweens can be put into timelines (by using key frames), groups and sequences. Tweens can be used to animate objects and parameters”. It also supports start, end, pause, resume events for all its motion objects. Tween is short for between, and used to describe an animation that occurs by interpolating the state of an animation between two defined states. This differs from traditional cell animation, where every frame is defined.

2. oscP5

A protocol for communication among computers, sound synthesizers, and other multimedia devices, section 2.4.

In other words Ijeomamotionlib is handling the graphics, whereas oscP5 is responsible for the communication and data exchange between Processing and Pure Data.

As we can see in figure 3.18 the interface consist of three main parts, the Local Jitter Graph, the Updated Jitter Graph and the Function Buttons, which all are based on Ijeomamotionlib.

In the Local Jitter Graph section, each bar is one incoming value, provided from Pure Data through oscP5 each 300 milliseconds. Each of these incoming values rep-

resents the percentage of ten Jitter values computed in Pure Data, that are above the pre determined threshold, [2]. If the Jitter value is greater than 124, is considered pathological; if the Jitter value is less than 124, then it is considered normal.

Respectively, in the Updated Jitter Graph section, each bar is one incoming value, provided from Pure Data through oscP5 each 300 milliseconds; which represents the updated mean of these computed percentages.

The colorization of the bars in both graphs consists of three colors. Green, if the incoming percentage is in between zero and 45% , gray if the percentage is in between 45% and 50% , and red if percentage is in between 50% and 100% .

In the last part, the “player like” option function buttons are placed. It works exactly like a common player. By pressing the “play” button, the computation starts, by pressing the “stop” button, the computation stops. The “play” and “stop” buttons when pressed send OSC messages to Pure Data in order to enable the computation. By pressing the “pause” button, the computation pauses. Just like a usual player. By pressing the “record” button, instantly a text file with name the current date and time is created, which keeps as log the time in milliseconds, the current jitter value and the updated jitter value. If the “pause” button is pressed while the “recording”, the recording pauses also. There is no log kept in the file, until “play” is pressed again. Then the log continues to be kept from this moment and after. The “stop” button closes also the log file, and stops the recording. To terminate the application the user much press the “X” button on the top right side of the window, just like any other application.

Chapter 4

Results

In this chapter we present the results of the performed experiments. In the first section the results of the real time system with synthetic signals as input are shown. In the second section we compare the real time implementation results, with the results of the initial off line implementation, on actual speech recordings.

4.1 Synthetic signals

For the initial validation of the real time system we used synthetic signals. Specifically, we used synthetic jittered speech signals from a sustained phonation recording of the vowel /a/ with average fundamental frequency of 125 Hz. The duration of the created signals was set to one second, the sampling frequencies was 48 kHz and the value of ε , the pitch deviation in samples, varied from 0 sample to 10 % of the pitch period.

Testing on these synthetic jitter files showed that the real time autocorrelation pitch estimation is very close to the estimations provided from Wavesurfer. As it is shown in figure 4.1, for the first fifteen synthetic files the maximum difference of the real time estimation approach is 1.2. The expected mean value is 125, whereas the

estimated is 126.2, in the worst case.

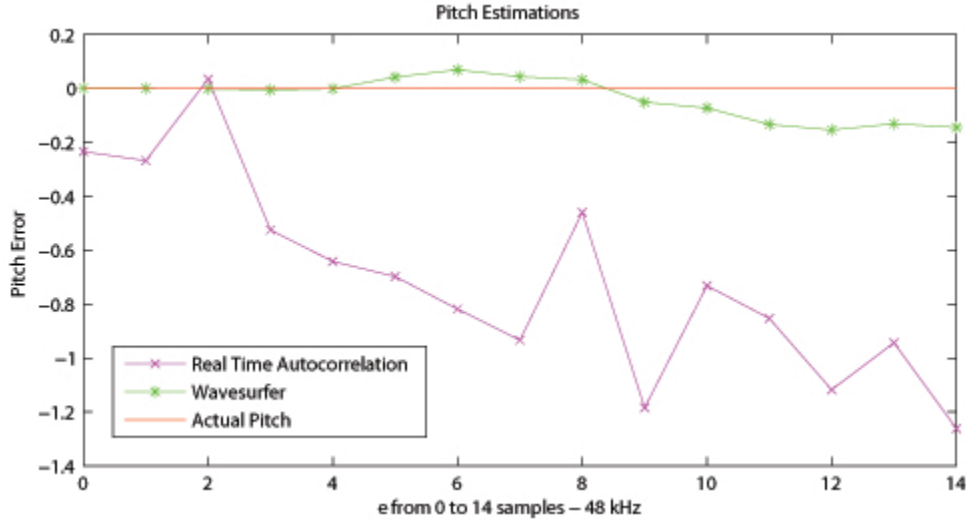


Figure 4.1 The pitch estimation error for the synthetic jitter signals.

In figure 4.2, we present the estimations provided by the initial offline SJE implementation with MATLAB as well as the estimations of the real time implementation from Pure Data.

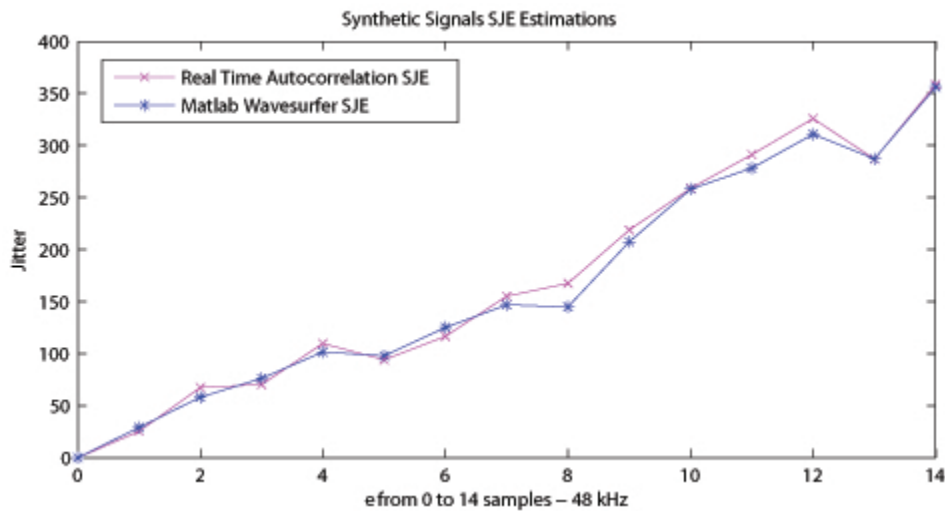


Figure 4.2 The short time jitter estimations of synthetic signals for both real time and off-line implementations.

The error presented, is due to different accuracies in number representations be-

tween MATLAB and Pure Data. An other factor responsible for that is slight differences in the windowing accuracy. The results were the same using either a frame size of 1024 or 2048 samples.

4.2 Actual signals

To further evaluate our system, actual voice signals were used for jitter estimation. The used database consists of 31 different patients. For each patient there are recorded files before and 3 months after treatment.

Patients consisted of:

- men and women
- smokers and non smokers
- ages from 28 to 81

All suffered from benign laryngeal lesions, such as vocal fold polyps, Reinke's edema, and leukoplakia. All patients underwent microlaryngoscopy and surgical treatment.

The sampling frequency used for the recordings was 22050 Hz.

The estimated jitter values for ten different recordings, for both offline and on-line approaches are presented in figure 4.3.

As it is shown, in eight out of the ten cases the differences between offline and online approaches are very small. But there are two specific cases where the difference is considerably big. In the first case this is due to the fact that the recordings contain noise and the real time approach is sensitive to that, whereas for the off-line approach we hand labeled the recordings before the jitter extraction. In the second case there is much more difference between the two estimations due to the signal

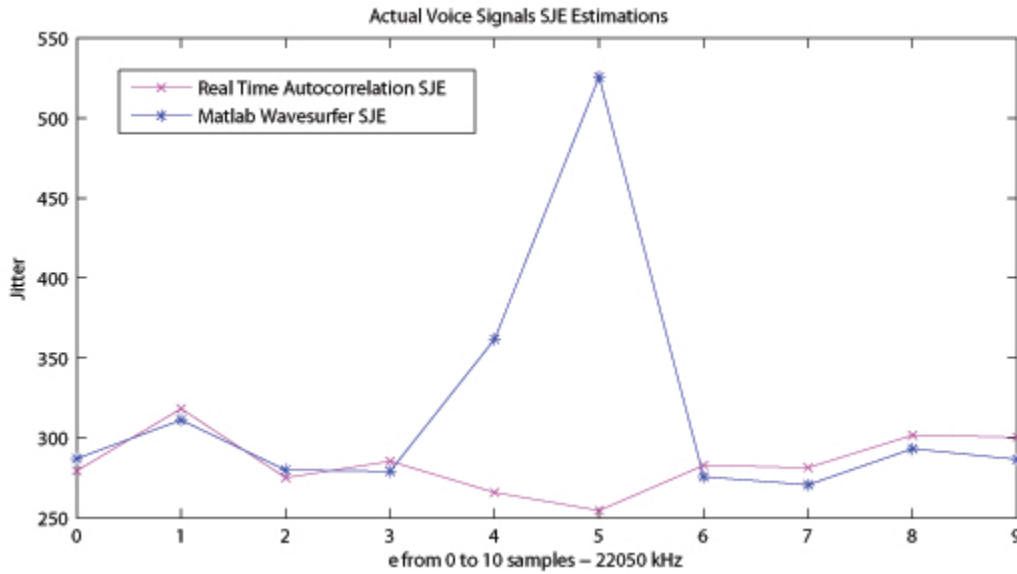


Figure 4.3 The short time jitter estimations of actual signals for both real time and off-line implementations.

quality which is really bad, (quite noisy because of the patient’s bad voice quality). The real time implementation “labels” most of the incoming frames as noise, so they are rejected from the computation, hence this estimation is much smaller than the off-line implementation.

When the recording does not contain noise, and when the quality of the patient’s signal is not extreme, such as in cases 4 and 5, the real time SJE estimation is as reliable as the off-line SJE estimation.

Chapter 5

Conclusions

5.1 Conclusions

This work has presented the construction of a real time voice pathology detection system, based on simple algorithms, such as autocorrelation pitch estimation and Spectral Jitter Estimator, combined with a simple, yet friendly and easy user interface. No digital signal processor board, also referred as DSP board, [26], was used, so there are no hardware dependencies. Hence, it is portable, platform independent and can be used from anyone in any system. That gives the opportunity of using a first real time voice quality estimation tool, without the usage of complicated, expensive equipment. Furthermore it is based on the acoustical analysis of the voice signal, a non-invasive method, so it is a very comfortable way for patient examination.

From another point of view it gives the opportunity not only for actual pathology detection, but also for constant voice monitoring. It can be also applicable to people working in jobs that require essential vocal use, such as singers, teachers, lecturers, newscasters, etc, to show voice abuse . When one speaks for an extensive period, the vocal folds do not vibrate as they should, so at the beginning there are “normal”

results, but at some point pathological results appear, false pathology detections. This by no means, implies that there is an actual pathology, but it is an alert, that the speaker needs to rest his/hers “vocal box”, since chronic or habitual or vast voice mis-usage could result in a variety of laryngeal pathologies.

5.2 Summary of Contributions

This research is a first step to the research field of real time pathology detection. It has shown that this task is not only achievable but also gives reliable results. It is very encouraging and opens the way to new approaches. The existing algorithmic solutions that have been until now proposed, could be re-implemented in real time. With these methods combined, another similar yet improved and more concrete system could be build. Other perspective can be given to solution approaches for this problem.

Also this work revealed many weak sides of Pure Data. Even if at the beginning it seems like a disadvantage or a problem, it should not be considered so. Since the problem is spotted, a solution could be given. It gives new aspects for the improvement of existing tools. New extensions could be proposed for the implementation of a debugger, the synchronicity solution and the frame efficiency clarification.

5.3 Future Research

Voice processing in Pure Data is a very challenging topic. Especially pitch synchronous analysis of voice is a very demanding field, which, in many cases, pushes Pure Data to its own limits. For that reason, implementation of more complex objects that deal with all the synchronization and frame efficiency problems, and extend the flexibility of already existing objects, is demanding. A possible alternative solu-

tion for the synchronization problems could be the OpenCorn Kernel, [27], which was implemented at the Polytechnic of Mons, [28].

Great contribution could be given by a web-based system for the remote acquisition and automatic analysis of vocal signals. That would give the possibility to users to submit vocal signals through a simple web-interface and have them analyzed in real-time. This could provide first-level information on possible voice alterations. It is also a great opportunity of large mass screening of voice-related pathologies. Additionally there is the possibility to build a controlled database of voice signals, useful for statistical analysis. The main guidelines to build the controlled database are noise reduction during voice collection and sample labeling through human intervention, e.g. based on patients conditions. Such a database could allow to correlate pathological conditions at different levels with various kind of voice signals.

Apart from the nature of the platform, many things could follow this research. A first next step could be the implementation of YIN pitch estimator, [29], which is also based on the autocorrelation method but with a number of modifications that are combined to prevent errors. Moreover, other proposed algorithms for voice pathology detection could be additionally implemented, tested and compared. The only presupposition is that there will have little or even better, no dependencies on future or past frames. Generally, this entirely real time idea, if the synchronization problem is solved, could not only be applied in speech dysphonia, but also in speaker or speech recognition.

Additionally the creation of a debugger, could solve many problems and save a lot of time for the developers, conducting to the accelerated improvement of existing objects and externals. This could be a feature very encouraging for new and more people to start involving in pure signal and voice/speech processing rather in just music or sound that is the case until today. This will result in the evolution of the

research in the field of real time implementations.

Bibliography

- [1] Wikipedia, “Dyshponia”, <http://en.wikipedia.org/wiki/Dysphonia> (Accessed April 15, 2009).
- [2] Miltiadis Vasilakis, Yannis Stylianos “Voice Pathology Detection Based on Short-Term Jitter Estimations in Running Speech”, *Folia Phoniatica et Logopaedica*, Volume 61, Issue 3, June 2009, pp. 153-170.
- [3] KayPENTAX, “Multi Dimensional Voice Program”, <http://www.kayelemetrics.com/> (Accessed November 10, 2009).
- [4] Praat, “Praat: doing Phonetics by Computer”, <http://www.fon.hum.uva.nl/praat/> (Accessed November 10, 2009).
- [5] Wavesurfer, “TMH KTH”, <http://www.speech.kth.se/wavesurfer/> (Accessed October 1, 2009).
- [6] WinPitch, “WinPitch”, <http://www.winpitch.com/> (Accessed November 15, 2009).
- [7] VOICEBOX, “VOICEBOX: Speech Processing Toolbox for MATLAB”, <http://www.ee.ic.ac.uk/hp/staff/dmb/voicebox/voicebox.html> (Accessed June 15, 2009).

-
- [8] Miltiadis Vasilakis, Yannis Stylianou “Spectral jitter modeling and estimation”, Biomedical Signal Processing and Control, Volume 4, Issue 3, July 2009, pp. 183-193, Special Issue on New Trends in Voice Pathology Detection and Classification M & A of Vocal Emissions.
- [9] Pure Data, “PD Community Site”, <http://www.puredata.org> (Accessed February 20, 2008).
- [10] Max/MSP, “Cycling 74”, <http://www.cycling74.com/> (Accessed February 20, 2008).
- [11] Externals, “HOWTO write an External for puredata”, <http://pdstatic.iem.at/externals-HOWTO/> (Accessed March 15, 2008).
- [12] Pure Data Forum, “Pure Data Forum”, <http://puredata.hurlleur.com> (Accessed February 20, 2008).
- [13] Ben Fry, “Ben Fry”, <http://benfry.com/> (Accessed April 20, 2009).
- [14] Casey Reas, “REAS.com / C.E.B. Reas”, <http://reas.com/> (Accessed April 20, 2009).
- [15] Aesthetic + Computation Group, “MIT Meadia Laboratory”, <http://acg.media.mit.edu/> (Accessed July 20, 2009).
- [16] Wikipedia, “MIT Media Lab”, http://en.wikipedia.org/wiki/MIT_Media_Lab (Accessed August 15, 2009).
- [17] Design by Number, “DBN”, <http://dbn.media.mit.edu/> (Accessed November 05, 2009).

-
- [18] John Maeda, “John Maeda”, <http://plw.media.mit.edu/people/maeda/> (Accessed December 20, 2009).
- [19] Processing, “Processing.org”, <http://processing.org/> (Accessed Mai 20, 2009).
- [20] Processing Forum, “Processing Discourse”, http://processing.org/discourse/yabb_beta/YaBB.cgi (Accessed Mai 20, 2009).
- [21] People Behind Processing, “People”, <http://processing.org/people.html> (Accessed Mai 20, 2009).
- [22] OpenSoundControl, “OSC”, <http://opensoundcontrol.org/> (Accessed April 10, 2009).
- [23] CNMAT, “UC Berkeley Center for New Music and Audio Technology”, <http://cnmat.berkeley.edu> (Accessed February 20, 2008).
- [24] T. Dubuisson, T. Dutoit, B. Gosselin, M. Remacle, “On the Use of the Correlation between Acoustic Descriptors for the Normal/Pathological Voices Discrimination”, EURASIP Journal on Advances in Signal Processing, Analysis and Signal Processing of Oesophageal and Pathological Voices, 2009.
- [25] Miltiadis Vasilakis, “Spectral Based Short-Time Features for Voice Quality Assessment”, xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx Master Thesis, 2009.
- [26] DSP, “Digital Signal Processor”, http://en.wikipedia.org/wiki/Digital_signal_processor (Accessed February 25, 2008).
- [27] OpenCorn, “OpenCorn DevBlog”, <http://dev.opencorn.org> (Accessed April 20, 2009).

- [28] FPMs, “Faculte Polytechnique de Mons”, <http://www.fpms.ac.be/FPMsHome/en/Accueil> (Accessed March 10, 2009).
- [29] xxxxxxxxxxxxxxxxxxxxxxxxx, “xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx”, http://recherche.ircam.fr/equipes/pcm/cheveign/pss/2002_JASA_YIN.pdf (Accessed February 20, 2010).
- [30] jeoma, “motionlib”, <http://ekeneijeoma.com/processing/motionlib/> (Accessed March 10, 2010).

Appendix A

User Manual

In order to run the application, there are some steps to be made. First of all the two platforms, Pure Data and Processing have to be installed. After that, for the visualization, and the exchange of data between the two platforms, some libraries have to be added.

1. Ijeomamotionlib, [30]
2. oscP5

The installation instructions are well described in the official sites of ijeomamotionlib and oscP5.

When this is finished the application is ready to be used.

In order to open the graphical interface either execute the “SJEBaRGraph.exe”, which directly launches the applet, or open the “SJEBaRGraph.pde” file. This will open the source code in the “Processing Environment”, as it is shown in figure . In this case press the play button, which is placed in the top left corner. This will execute the code and the applet will launch, figure 3.18.

The “player like” option function buttons are placed in the bottom of the application. It works exactly like a common player. By pressing the “play” button, the computation starts, by pressing the “stop” button, the computation stops. The “play” and “stop” buttons when pressed send OSC messages to Pure Data in order to enable the computation. By pressing the “pause” button, the computation pauses. Just like a usual player. By pressing the “record” button, instantly a text file with name the current date and time is created, which keeps as log the time in milliseconds, the current jitter value and the updated jitter value. If the “pause” button is pressed while the “recording”, the recording pauses also. There is no log kept in the file, until “play” is pressed again. Then the log continues to be kept from this moment and after. The “stop” button closes also the log file, and stops the recording. To terminate the application the user much press the “X” button on the top right side of the window, just like any other application. In order though, to have valid visualized results, the “main” Pure Data patch has to be first opened.

In case of testing an existing file with the application, double click to open the Pure Data file “_ main_ file.pd”, figure 3.14. Click on the file name to load one of the recommended files. Then minimize the patch and continue with the interface. To give some further information, the moment “play” is pressed in the interface, or “on” message is clicked in the patch, the user can see the produced numbers in the Pure Data window, and at the same time, the visualization if the applet is active. If “stop” is pressed in the interface, or “off” message is clicked in the patch, the computation stops.

Second, in case of testing voice input from the microphone with the application, double click to open the Pure Data file “_ main_ mic.pd”, figure 3.15. The functionality is exactly the same, as it was above described. Press “play” in the interface or “on” in the patch to activate the input, “stop” in the interface or “off” in the patch to

deactivate it. Here has been also added an option for recording also the input, which has not been added into the interface. “Save” button is used to create a wav file, which name is defined by the user, “Rec” is to start the recording, and “Stop” to stop it. The “Stop” button does not stop the computation, only the recording thread is deactivated.

It is important to note that in the case that the platform is not windows, the externals have to be recompiled, so that Pure Data will have the required libraries.

Appendix B

Source Code Documentation

Each separated file, either external or patch, is fully commented on its functionality and dependencies. The documentation lies in each C file for the externals, and in each Pure Data file for the patches.

Appendix C

Source Code

The source code is provided at the website: www.csd.uoc.gr/~astrin, as well as in the attached CD.

Appendix D

List of implemented Patches and Externals

D.1 Patches

1. `autocorr~`: This patch just takes a signal input and gives as an output the normalized autocorrelation.
2. `nrg~`: This patch takes as an input the real part and the imaginary part of a signal, and gives the energy as output.
3. `std~`: This patch takes as an input a signal, computes the standard deviation, and redirects it in the output.
4. `moses~`: This patch implements the `moses` Pure Data object, but not in the dataflow part of the environment, but in the audio part. `Moses` takes numbers and outputs them at left if they're less than a control value and at right if they're greater or equal to it.
5. `2db~`: This patch takes as in input a signal and converts it to decibels.

6. `hanning~`: This patch implements a hanning window.
7. `isSilence~`: This patch estimates if the input is a silence part or not. If the input is silence then the output is zero, else if the input is not silence then the output is one.
8. `isVoiced~`: This patch estimates if the input is a voiced part or not. If the input is unvoiced then the output is zero, else if the input is voiced then the output is one.

D.2 Externals

1. `meansig~`: computes the mean value of a signal block.
2. `minmaxsig~`: computes the minimum and maximum value of a signal block.
3. `invclip~`: inverted clipping of a signal block.
4. `zeropad~`: zero-padding to a signal block.
5. `f0~`: computing f0 of an autocorrelation signal block.
6. `harmonic_jitter_peaks~`: computes the harmonic peaks of a signal block.
7. `lerp~`: computes the linear interpolation as signal block.
8. `intersections~`: intersections between two signal blocks.
9. `intersections_enhancement~`: enhancement of intersections as a signal block.
10. `sum~`: This object takes as input a signal and gives the sum as a signal output.

-
11. `subBlock~`: This object takes as an input a signal, and two integers. The values of the input that have indexes between the input numbers are given to the output as a signal block. The rest values are set to zero.
 12. `ReadWave`: This patch takes as first input the name of the wav and stores it in the array `voice`. As output gives the length of the wav file.
 13. `recordWav`: This patch takes as input a signal and records it into a wav file that the user has previously selected through the interface.
 14. `sendOSCdata`: This patch takes as input a number and with the usage of OSC sends it to the interface application for the visualization.
 15. `receiveOSCdata`: This patch receives an OSC message from the interface.

