FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Pitch Shifting of Audio Signals in Real Time Using STFT on a Digital Signal Processor
Student:	Jan Onderka
Supervisor:	Dr.-Ing. Martin Novotný
Study Programme:	Informatics
Study Branch:	Computer engineering
Department:	Department of Digital Design
Validity:	Until the end of summer semester 2018/19

Instructions

Develop a program that will shift the spectrum of a stereo audio signal. Program will be implemented in ADSP BF548 EZ-KIT design kit. Specifically, the signals on the stereo input of the kit shall be transformed into higher or lower tones (e.g. by one, two or three octaves) and the transformed signals shall be outputted on its stereo output. The left and right channel shall be processed independently. Output signals should have a low distortion and a low latency with respect to the input. If possible, compare your solution with other known solutions with respect to distortion, latency, etc.

References

Will be provided by the supervisor.

doc. Ing. Hana Kubátová, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 31, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Pitch Shifting of Audio Signals in Real Time Using STFT on a Digital Signal Processor

Jan Onderka

Department of Digital Design
Supervisor: Dr.-Ing. Martin Novotný

May 15, 2018

Acknowledgements

I would like to thank Dr.-Ing. Martin Novotný for supevising my bachelor thesis, prof. Ing. Roman Čmejla, CSc. for insights into the phase vocoder algorithm and Ing. Radek Sedláček, Ph.D. for lending me a digital signal processor kit.

I would also like to thank my family for their support during my studies and work on the bachelor's thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 15, 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Onderka, Jan. *Pitch Shifting of Audio Signals in Real Time Using STFT on a Digital Signal Processor*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Cílem této práce je implementace pitch shifteru, tedy měniče výšky zvuku, stereofonního zvukového signálu v reálném čase s požadavkem na nízké zkreslení a nízkou latenci. Jsou zváženy různé algoritmy a je vybrán algoritmus Ocean založený na posunu košů krátkodobé Fourierovy transformace. Je opravena chybná rovnice v popisu algoritmu. Algoritmus je implementován v reálném čase na přípravku digitálního signálového procesoru ADSP BF548 EZ-KIT a jeho výstupy jsou porovnány s dalšími pitch shiftery. Je zjištěno, že implementace funguje téměř stejně dobře jako komerční pitch shifter. Je prezentován závěr, že další radikální zlepšení algoritmů pro pitch shifting založených na krátkodobé Fourierově transformaci není možné, jelikož se již přibližují principiálním omezením transformace.

Klíčová slova Pitch shifting, číslicové zpracování signálu, reálný čas, nízká latence, Blackfin, BF548, krátkodobá Fourierova transformace, STFT, windowing, váhovací okno, Overlap and Add, OLA, Constant Overlap and Add, COLA, banka filtrů, SOLA, Synchronous Overlap and add, PSOLA, Pitch Synchronous Overlap and Add, algoritmus Rollers, phase vocoder, algoritmus Ocean

Abstract

The goal of this work is to implement a pitch shifter of a stereophonic audio signal in real time with a requirement of low distortion and low latency. Basic principles of audio processing are explained. Various pitch shifting algorithms are described and considered, resulting in selection of the Ocean algorithm based on Short time Fourier transform bin shifting. An erroneous equation in the algorithm description is fixed. The algorithm is implemented in real-time on a digital signal processor kit ADSP BF548 EZ-KIT and its outputs are compared to other pitch shifters. The implementation is found to perform almost as well as a commercial pitch shifter. It is concluded that further drastic improvements of Short time Fourier transform based pitch shifters are impossible as they are approaching the fundamental limits of the transform.

Keywords Pitch shifting, digital signal processing, real time, low latency, Blackfin, BF548, Short time Fourier transform, STFT, windowing, window function, Overlap and Add, OLA, Constant Overlap and Add, COLA, filter bank, SOLA, Synchronous Overlap and add, PSOLA, Pitch Synchronous Overlap and Add, Rollers algorithm, phase vocoder, Ocean algorithm

Contents

List of Figures

List of Tables

Introduction

Changing the pitch of audio signals without changing the speed of their playback is one of the more complex areas of digital signal processing. Since humans perceive sound as harmonically related (frequencies being consonant with other frequencies that are their multiples and dissonant with frequencies that are not), it is not possible to simply perform a linear shift of all audio frequencies; this results in changing the harmonic relationships of frequencies and produces audible dissonances that worsen with larger shift.

This results in need of non-linear processing techniques which are usually computationally expensive. Such computational costs are critical when performing pitch shifting in real time, where the amortised time of processing of the incoming audio sample must be smaller than sample duration. It is also important to consider latency, i.e. the time delay between the input and the pitch-shifted signal. Delays between otherwise similar signals may cause unintended distortion of their sum and/or confuse the listener.

Fortunately, the quickly decreasing costs of computing have opened a path to inexpensive digital processing units powerful enough to perform pitch shifting and related transforms in real time with low distortion and latency. While the underlying theoretical algorithms are usually academically published and well-known, as well as implemented and used in digital audio workstation software for personal computers, I have found no non-commercial real-time implementation of a general pitch shifting algorithm on an embedded processor.

There is, however, a large number of pitch shifting pedals for guitar players which use embedded hardware to pitch shift. Many of them rely on heuristics (tracking) to detect guitar notes and chords and synthesise a new signal. This results in a number of disadvantages: the tracking can fail and produce different notes/pitches, false negatives or false positives; the output signal may have a different frequency content (timbre) than the equivalent pitch-shifted signal; the pedal will probably produce wildly varying results for signals containing something else than the intended instrument, e.g. different musical instruments, human voice, other naturally occuring or synthesised sounds and

their combinations. Some of the pitch shifting pedals use general pitch shifting algorithms, but the exact used algorithms and implementations are closely guarded secrets of their manufacturers.

Knowing this, I have decided to develop a real-time general audio pitch shifter implementation on an inexpensive digital processing unit. After considering various hardware options and approaches, noted in the Analysis section, I have decided to base my solution on an Analog Series Blackfin 5xx series digital signal processor (DSP). For development, I have used a BF548 EZ-KIT evaluation kit that contains a high-end Blackfin BF548 processor and an AD1980 audio codec for signal input and output. While the BF548 processor is fairly expensive, the Blackfin DSP family consists of closely-related processors varying mostly in their memory sizes and peripherials; a low-cost processor and audio codec might be selected for mass-production of a potential product.

Chapter 1

Goals

The main goal of my work is to analyse existing pitch shifting algorithms, select an appropriate one and create a program for the selected BF548 EZ-KIT that will create a high-quality pitch-shifted version of the incoming stereophonic audio signal in real time. Upon a closer inspection, this neatly decomposes into four separate concerns.

1.1 Pitch shifted sound quality

Since there are no constraints on the type of incoming audio signal (e.g. vocals or a specific type of instrument), my program should be able to accurately pitch shift all of them. While doing this perfectly is an impossible task in real-time processing, I shall select an algorithm that does not unacceptably degrade for a worst-case input, maintaining a good quality for all cases.

1.2 Pitch shifting latency

The most obvious problem that separates real-time pitch shifting algorithms from their non-time-constrained counterparts is that of latency. Empirical research came to a conclusion that latencies of up to 20-30 milliseconds of audio delay in the whole signal chain are acceptable for listeners [1]; I shall therefore select an algorithm that allows processing with a latency lower than that.

1.3 Stereophonic field preservation

Humans are very discerning about the stereophonic field created when a similar but slightly different signal (latency, phase, amplitude) reaches each ear, creating a sense of sound directionality. As this can be critical in musical context, I shall select an algorithm that preserves this stereophonic field.

1.4 Implementation feasibility

The selected algorithm must be feasible to implement on the BF548 EZ-KIT. This means it cannot use more computing and memory resources than are actually available.

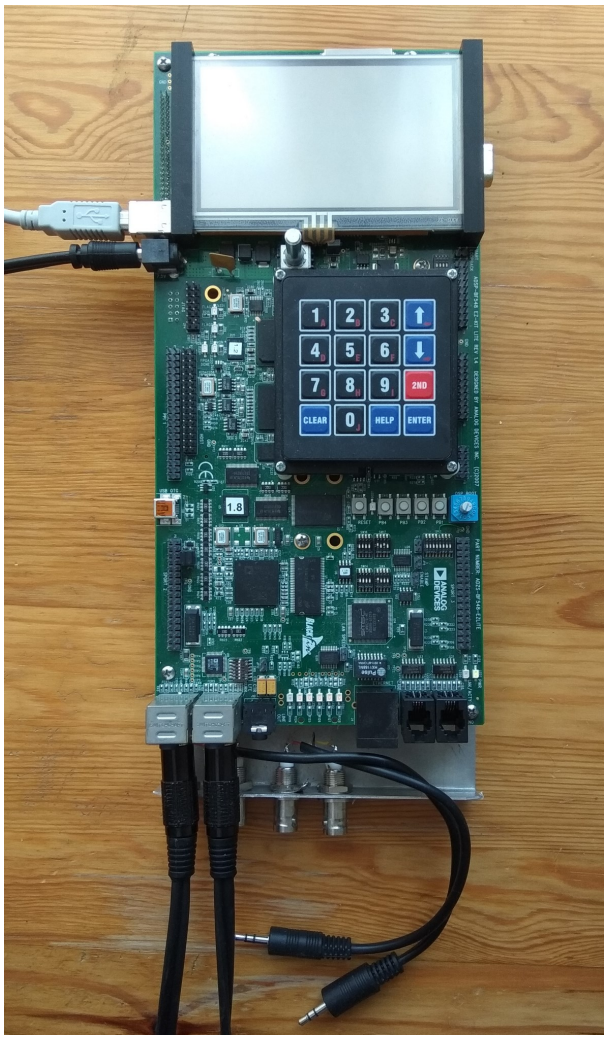


Figure 1.1: Analog Devices BF548 EZ-KIT
The kit on which a real-time pitch shifting algorithm is to be implemented. Its functions, connections and peripherials useful for implementation shall be discussed later in chapter 7.

Chapter 2

Principles of audio processing taken into design considerations

Human sound perception is not completely understood despite the extensive research undertaken. General concepts have been empirically identified and successfully used for different areas of audio processing, however [2, p. 209]. I shall go over various considerations that have to be taken in mind for digital signal processing of audio data.

2.1 Physical aspects of sound

"Sound is a waveform consisting of density variations in an elastic medium, propagating away from the source. The propagation medium can be air, water, or a solid material." ([3, p. 55])

What allows modern audio signal processing is the fact that while sound is a density variation waveform, it is easily converted into and from audio signals. By an audio signal, a variation of electrical potential is conventionally meant. Various electroacoustic transducers, e.g. microphones and speakers, can be used for the conversion [3, p. 55, 68]. As the task is just to process an incoming stereophonic audio signal, outputting another one, a further discussion of the various transducers is not necessary. A special interest remains, however, in the manner of how human brain processes sound; this specifies the desired behaviour and limits of the pitch shifting designs.

2.2 Frequency range of human hearing

While the exact hearing range varies with each human and changes with age, the commonly stated frequency range of human hearing is from 20 Hz to 20 kHz [3, p. 57]. It should be noted that the perceived loudness of tones decreases at the ends of frequency range, facilitating a smooth falloff of perceived tones

rather than an abrupt cutoff [2, p. 211]. The high end of the frequency range is particularly important for consideration. Due to the Nyquist-Shannon sampling theorem, one is able perfectly reconstruct a digitally sampled signal with the highest frequency B if and only if the sampling frequency f_s satisfies the equation [4, p. 36]

$$f_s > 2B$$

. This means all frequencies in human hearing range can be digitally sampled with sampling rate $f_s > 40$ kHz. However, frequencies above f_s present in the original signal will cause aliasing if they are not attenuated below the sensitivity of the system. This necessitates a low-pass filter added before sampling [4, p. 34-36]. Such a filter is also needed for a conversion back to analog signal for a similar reason. As an ideal low-pass filter is noncausal and cannot be implemented in real time due to its frequency window being unbounded in time [4, p. 83], the sample rate chosen has to be higher, typically 44.1 Hz or 48 kHz.

2.3 Human pitch perception

Various human pitch perception models and theories have been proposed, with experiments on the topic being seemingly inconclusive and often contrary [2, p. 228].

Despite that, the basic understanding of human pitch perception, put together by Hermann von Helmholtz in his influential *On the Sensations of Tone as a Physiological Basis for the Theory of Music*[5], is suitable for most purposes including this one. To summarise his basic findings, pitched instruments basically produce a signal consisting of a sum of sinusoids with different frequencies: a lowest frequency called the first or fundamental frequency and its multiples, called harmonics or overtones. Upon sound wave perception by the auditory system in ear, the human brain tries to find a harmonic spectrum of the incoming frequencies. If it finds such a spectrum, it responds by producing a sensation of pitch based on the corresponding fundamental frequency and a timbre corresponding to the relationship between the fundamental frequency and harmonics [5][6, p. 15-21, 35]. Modern pitch perception models usually roughly concur with this interpretation of pitch perception, differing in some less important phenomena not explained by this basic model [2, p. 216-229].

It should be noted that while pitch is a subjective psychoacoustical attribute of sound, in conventional cases, its progression is perceived approximately logarithmically. That is, if a human hears a sinusoidal 100 Hz and then a sinusoidal 400 Hz tone, the human will classify the interval between them as a double of an interval between a sinusoidal 100 Hz and a sinusoidal 200 Hz tone. This forms the basis of contemporary Western music that almost exclusively uses 12-tone equal temperament (12-TET), which defines the smallest interval between two frequencies corresponding to pitches, a semitone,

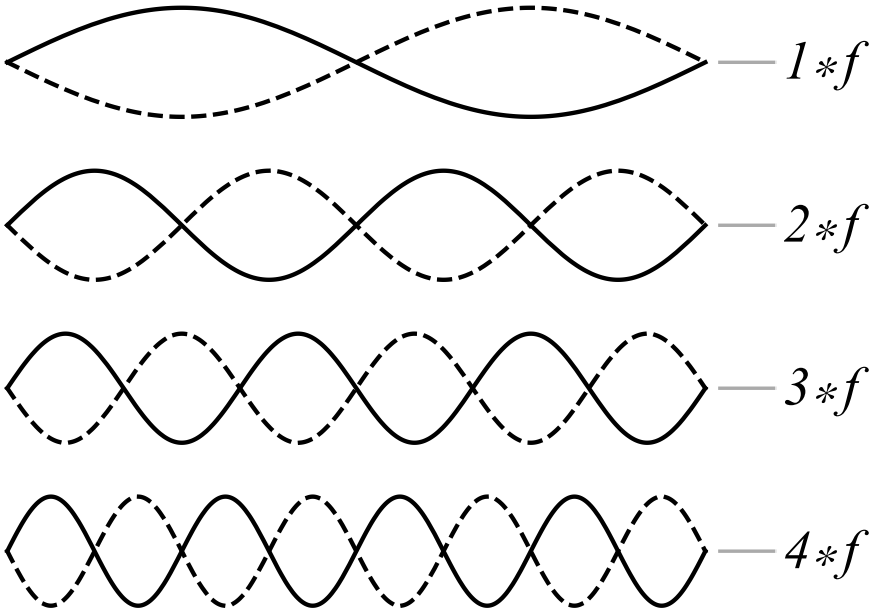


Figure 2.1: First four harmonics
Four harmonically related sinusoids. Considering the first wave to be the fundamental, the lower waves are its second harmonic, third harmonic and fourth harmonic respectively. They all have a multiplicative relationship both to the fundamental and other harmonics.
The basic sinusoids are represented by solid lines. Dashed lines represent their inverted amplitude counterparts, which can also be made by moving the basic sinusoids phase by $-\pi/2$.
Inspired by [5, p. 46 fig. 17].

to have a ratio of $\sqrt[12]{2}$. To arrrive at a definition of pitch shifting consistent with the contemporary Western music, I shall also treat pitch as a logarithm of frequency.

The multiplicative relationship between frequencies forms a basis of a strict pitch shifter requirement. In order not to change the perception of pitch difference (i. e. the multiplicative relationship between frequencies), all of the frequencies in the audio signal can only be multiplied by the same multiplier. In other words, if F_in is the input frequency, F_{out} is the output frequency and the following equation holds true for all F_{in} frequencies in the input audio signal:

$$F_{out} = k * F_{in},$$

then the output signal is a perfectly pitch shifted input signal by the multiplier k.

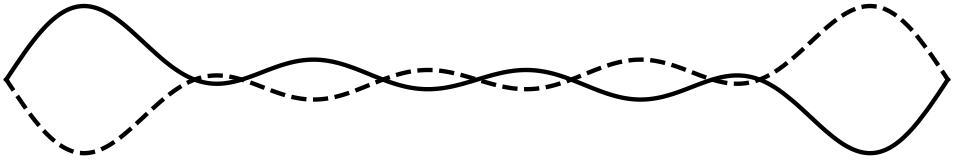


Figure 2.2: Sum of first four harmonics
Four harmonically related sinusoids from Figure 2.1 added together, with their amplitude divided by 4 to provide a comparison to a single sine wave.

2.4 Signal sinusoids' amplitude and phase consideration

While a pitch shifting process would seem to be straightforward from the preceding equation, a real-world audio signal poses a number of complications. Consider the classic sinusoid equation:

$$y(t) = A * sin(2\pi ft + \phi)$$

Amplitude A, frequency f and phase ϕ are held constant in this equation. Note that this does not provide an exact mapping to real-world signals as these are created by (or at least can be thought of as) composing various products of sinusoidal waves with varying amplitude modifiers.

Humans are very sensitive not only to the frequency content of the signal, but the amplitudes of various frequency components as well. This means it is important to preserve the amplitudes of the pitch shifted composing waves. While humans are not very sensitive to phase (von Helmholtz's research even points to absolute insensivity [5, p. 127]), it is important for ϕ to be held constant, that is, to maintain constant phase of the pitch shifted sinusoids where the input sinusoids' phase is held constant: otherwise, the equation does not result in $y(t)$ forming a single, perfect sinusoid.

2.5 Gabor uncertainity principle

Due to the Gabor uncertanity, a fundamental property of signals related to the Heisenberg uncertainity principle, it is not possible to exactly localise a signal both in frequency and in time. For a general signal, the standard deviations of time and frequency estimates o_t and o_f must follow this equation [7]

$$\sigma_t \sigma_f \geq \frac{1}{4\pi} \approx 0.08 \text{ cycles} = 80 \text{ ms} * \text{Hz}$$

This means no pitch shifting algorithm using localisation of signal in real time can achieve perfect output for a general signal. It is, however, possible to balance the time and frequency resolution. If it is possible to make predictions

about the signal, the output can be also improved. A pitch shifting algorithm without the need to localise the signal could overcome this limitation; however, no such general algorithm is known.

Chapter 3

Spectrum separation and recombination techniques

During my studies of known pitch shifting algorithms giving usable results in real time, I have realised that most of them, barring the Synchronous Overlap and Add (SOLA) family of algorithms (which shall will be discussed in subsection 4.1.1 and subsection 4.1.2), have a common pipeline:

1. Separate the signal into spectral components centered on different frequencies.
2. Frequency shift them to a multiple of the center frequency (with emphasis on preserving constant phase as discussed in section 2.4).
3. Recombine them to get the output signal.

I have found it wise to discuss the common spectrum separation and recombination techniques separately before explaining the algorithms built on top of them.

3.1 Discrete Fourier transform (DFT)

The Discrete Fourier transform is the fundamental tool that enables thinking about a discrete signal in terms of sinusoidal waves which describe it. It is a sequence of complex numbers $X(b)$ defined as [4, p. 53]

$$X(b) = \sum_{n=0}^{N-1} x(s) e^{-j2\pi bs/N}.$$

Let me enlighten the readers about the useful properties and intuitive meaning of this incredibly important transformation tool.

First of all, the input signal sequence $x(s)$ is a sequence of N real or complex elements $x(0), x(1), \ldots, x(N-1)$ (though real-valued signals are

typical in the real world). Its Discrete Fourier transform, the sequence $X(b)$, is a complex-valued sequence of elements $X(0), X(1), \ldots, X(N-1)$.

The most important properties of DFT are its linearity, which can be expressed as

$$c_1 x_1(s) + c_2 x_2(s) \xrightarrow{\text{results in}} c_1 X_1(b) + c_2 X_2(b)$$

, where c_1 and c_2 are constant, and its invertibility using the Inverse Discrete Fourier transform (IDFT)

$$x(s) \xrightarrow{\text{DFT}} X(b) \xrightarrow{\text{IDFT}} x(s),$$

where the Inverse Discrete Fourier transform is defined as

$$x(s) = \frac{1}{N} \sum_{n=0}^{N-1} X(b) e^{j2\pi bs/N}$$

Note that the only difference from the DFT is the division by N (will be explained later) and a missing minus sign in the exponent [4, p. 75].

The invertibility, assured by the proven Fourier invertibility theorem, means that signals can be represented in its Discrete Fourier transform form without loss of data. They can also be manipulated in this form (called the frequency domain for a time domain signal, that is, in discrete processing, a sequence where the elements are positioned linearly according to the time of their capture) and then converted back.

3.1.1 Explanation of Discrete Fourier transform for real-valued time-domain signals

For better understanding of the Fourier transform, particularly as it applies to real-valued time-domain signals, consider the Euler's formula:

$$e^{jx} = \cos x + j \sin x$$

Using the Euler's formula, the Fourier transform can be represented by an equivalent equation to the standard exponential form:

$$X(b) = \sum_{s=0}^{N-1} x(s) \left[\cos\left(-2\pi \frac{b}{N} s \right) + j \sin\left(-2\pi \frac{b}{N} s \right) \right]$$

Since cosine is an even function and sine is an odd function, the signs of their arguments can be propagated to arrive to the rectangular form of discrete Fourier transform [4, p. 54]:

$$X(b) = \sum_{s=0}^{N-1} x(s) \left[\cos\left(2\pi \frac{b}{N} s \right) - j \sin\left(2\pi \frac{b}{N} s \right) \right]$$

It is now a bit clearer what is going on. I shall now consider $x(t)$ to be real-valued and split the $X(b)$ into two real sequences, $X(b)_{real}$ and $X(b)_{imag}$, representing the real and imaginary parts respectively. Due to the trigonometric functions being real-valued and $x(s)$ now considered also real-valued, only the cosine part will yield real outputs and only the sine part will yield imaginary outputs:

$$X(b)_{real} = \sum_{s=0}^{N-1} x(s) \cos\left(2\pi \frac{b}{N} s\right)$$

$$X(b)_{imag} = j \sum_{s=0}^{N-1} -x(s) \sin\left(2\pi \frac{b}{N} s\right)$$

In the next step, I shall force the inner computations into separate sequences and name them M_{real} and M_{imag}.

$$X(b)_{real} = \sum_{s=0}^{N-1} M_{real}(b, s)$$

$$X(b)_{imag} = j \sum_{s=0}^{N-1} M_{imag}(b, s)$$

$$M_{real}(b, s) = x(s) \cos(2\pi \frac{b}{N} s) = x(s) \sin\left(2\pi \frac{b}{N} s + \frac{\pi}{2}\right)$$

$$M_{imag}(b, s) = -x(s) \sin(2\pi \frac{b}{N} s) = x(s) \sin\left(2\pi \frac{b}{N} + \pi\right)$$

Let me refresh the reader's memory with the function of a sinusoid:

$$y(t) = A * sin(2\pi f t + \phi)$$

It is apparent that the sequences M_{real} and M_{imag} can be rephrased like this: they multiply the input signal with argument s by a sinusoid with unit amplitude, frequency b/N samples^{-1} and phase $\pi/2$ or π, respectively, at time s samples.

The reader is advised to think deeply about this rephrasing when thinking about the sums in the equations for $X(f)_{real}$ and $X(f)_{imag}$:

$$X(b)_{real} = \sum_{s=0}^{N-1} x(s) \sin\left(2\pi \frac{b}{N} s + \frac{\pi}{2}\right)$$

$$X(b)_{imag} = j \sum_{s=0}^{N-1} x(s) \sin\left(2\pi \frac{b}{N} s + \pi\right)$$

This translates to the result being the sum of the multiplication of the whole input signal by a sinusoid with unit amplitude, frequency b/N samples^{-1} and phase $\pi/2$ or π for $X(f)_{real}$ and $X(f)_{imag}$, respectively.

What does this all mean? If a real-valued time-domain signal is sampled N times, sample value at sample s being stored as $x(s)$, $x(s)$ can be described by a sum of $2N$ sinusoids: the complex value of $X(b)$, also called frequency bin b, describes the "similarity" of the signal to sinusoids with amplitude $Re(X(b))$ for the real part and $Im(X(b))$ for the imaginary part, frequency b/N samples^{-1} and phase $\pi/2$ for the real part and π for the imaginary part.

There are a few interesting points to note. If $x(s)$ is a real sinusoid with amplitude A_o and frequency $f = b/N$ samples^{-1} (lesser than $f = 1/2$), the resulting frequency bin complex amplitude will be $A_oN/2$. This is the reason for the scale factor in the Inverse Discrete Fourier transform: the scale factor is needed to remove this scaling. The reason why the scale factor is $1/N$ and not $1/\frac{N}{2}$) is due to the considered sinusoids being real, not complex, sinusoids [4, p. 70].

In case of a sinusoid present in the input signal whose frequency does not precisely match frequency of some bin, its content will be split between all bins, with the bins with closer frequency generally getting more of its content due to spectral leakage. Unfortunately, the discussion about spectral leakage is beyond the scope of this bachelor thesis, as are other Discrete Fourier transform and filtering concepts. I recommend consulting [4] as a starting point if the reader wants to understand these concepts in greater depth.

When sampling in time domain, the samples are usually taken with a constant frequency f_s called the sampling frequency. It should be no wonder that the frequency of the bins expressed in samples per second is

$$f_b = \frac{f_s b}{N}\text{Hz}.$$

Note that the bins are always equally spaced f_s/N Hz from each other and N samples are needed to compute the Discrete Fourier transform. This means that despite many advantages of DFT, it will always separate the spectrum using these linearly spaced bins and need the same number of samples to compute any bin regardless of its frequency.

3.2 Fast Fourier transform (FFT)

The Fast Fourier transform is an algorithm used for computing the Discrete Fourier transform quickly. Since DFT consists of N frequency bins, every one of them summing N input samples, the algorithmic complexity of DFT is $O(N^2)$. The Fast Fourier transform can reduce that to $O(N\ log\ N)$ by using the fact many arithmetic operations used to compute DFT are redundant. It does this by breaking the DFT down into smaller DFTs and then combining those. Fast Fourier transform has a variation, Inverse Fast Fourier transform, that performs the same thing for Inverse Discrete Fourier transform [4, p. 127-159].

While the Fast Fourier transform algorithm can be adapted for every N, the original algorithm could only process power-of-two [4, p. 128]. This property is retained in many implementations since it results in the fastest FFT processing.

3.3 Bandpass filter bank

Another way to perform spectral separation is by filtering. Filtering is essentially processing of a time-domain signal that results in results in signal spectral content change [4, p. 161]. A bandpass filter attenuates frequencies below and above the one frequency band it passes, making it ideal for spectral separation. Since spectral separation usually involves more frequency bands of interest (pitch shifting is an extreme example, since all frequencies below the Nyquist frequency are of interest), more filters are needed. A filter bank is just an array of parallel bandpass filters, each of which extracts a different signal.

Note that a bandpass filter bank has algorithmic complexity of $O(NM)$, where N is filter length and M is the number of filters (if all of the filters have the same length). Unfortunately, unlike the special case of DFT, which can be sped up using its special properties, it is generally not possible to speed up general filter bank processing drastically. The biggest avenue for speedup is the fact the filters are parallel, making them ideal for implementation on massively parallel hardware.

3.4 Short time Fourier transform (STFT)

The Short time Fourier transform is is a technique for shifting a long or a long-running signal into frequency domain using short frames [8, p. 395].

Essentially, from the engineering standpoint, the Short time Fourier transform is just a way to select the N samples that will be processed using DFT in a fair way. The classic algorithm using FFT essentially works in this way: [8, p. 397-398]

1. Select N most recent samples (the current "frame").
2. Multiply them by a window function.
3. Transform them into frequency domain using FFT of size N.
4. The frequency domain data can now be used.
5. Wait for N/O samples to arrive, O being a number specifying frame overlap.

Note that the Short time Fourier transform can be also thought about as a filter bank [4, p. 711]. It has to be noted that while it has much lower computing requirements than a typical filter bank thanks to the FFT algorithm,

it suffers from the equal spacing of bins and need for N samples to compute every bin.

3.4.1 Window function

A window function is a function that is zero-valued outside a chosen interval. This makes them important for computing Short time Fourier transform: no more than N samples can be transformed by DFT of size N. A rectangular window is not a fine choice in most circumstances due to spectral leakage concerns beyond the scope of this thesis [4, p. 83], but these concerns may be somewhat intuitively explained by considering the fact that the window rapidly changes from multiplication by 1 to multiplication by 0. This means abrupt changes that affect most of the bins.

Various smoother windowing methods have been proposed, the most popular for algorithms performing resynthesis being the von Hann window, its symmetric form being

$$w_s(n) = \frac{1}{2}\left[1 - \cos\left(\frac{2\pi n}{N-1}\right)\right] = \sin^2\left(\frac{\pi n}{N-1}\right)$$

.

A periodic form of any symmetric window form can be computed by adding 1 to N and discarding the last sample. This way, the periodic form can be given as [4, p. 85]

$$w_p(n) = \frac{1}{2}\left[1 - \cos\left(\frac{2\pi n}{N}\right)\right] = \sin^2\left(\frac{\pi n}{N}\right)$$

.

The von Hann window (mistakenly called Hanning in many sources) is nice for resynthesis because of its small spectral leakage and its ends smoothly tapering to zero. The periodic form is of particular interest since the windows add to nice constants when they are overlapped by a fraction of N. This will become important in subsection 3.5.1.

3.4.2 Spectrogram

It should be noted that while this thesis mainly concerns itself with usage of STFT for usage in pitch shifting algorithms, it is also used for making spectrograms, that is, visualisations of the Discrete Fourier transform of signals. This is very useful when studying audio signals, as one can visualise what is heard. I shall make use of spectrograms when testing and interpreting the results of various pitch shifter implementations.

For reference and to provide a demonstration of STFT, I have made two spectrograms of the same signal containing first four harmonics of a sinusoid added together. The lowest harmonic has a frequency of 3520 Hz, the frequency of musical note A_7 in the $A_4 = 440$ Hz Western equal-tempered scale. The

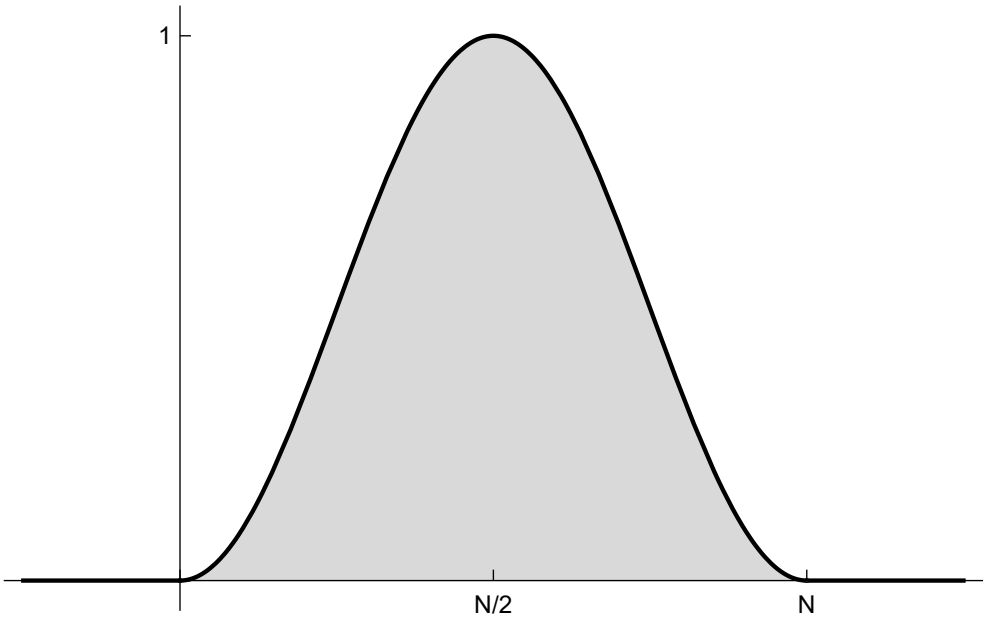


Figure 3.1: Periodic von Hann window
Note that the window is exactly one in $N/2$ and zero in intervals $(-\inf, 0]$ and $[N, \inf)$.

signal contains 0.2 seconds of silence, 0.6 seconds of the harmonics and, again, 0.2 seconds of silence. The sample rate is 48 kHz, resulting in frequencies up to the Nyquist frequency 24 kHz useful for visualisation in spectrograms. In both spectrograms, von Hann window and overlap $O = 4$ are used.

The spectrogram in Figure 3.2 is made by a high frequency resolution STFT with $N = 2048$. Note the long duration of spurious frequency content during transition (smearing) and a slight negative time offset explained by the need to get 2048 samples before starting the computation of spectrogram. In real time, this would actually translate into latency.

On the other hand, the spectrogram in Figure 3.3 is made by a low frequency resolution STFT with $N = 256$. There is almost no frequency smearing during the transitions, but the exact frequency of the various components is not well known. The frequency bins all have the same size. Note that since human hearing hears frequencies logarithmically scaled, the most important frequencies are below 4000 Hz, which is handled here only by a few bins!

3.5 Overlap-Add (OLA)

Since common audio signals change over short amounts of time, the majority of pitch shifting algorithms processes signal in small chunks (frames). Analysing and resynthesising frames with no overlap results in significant discontinuities

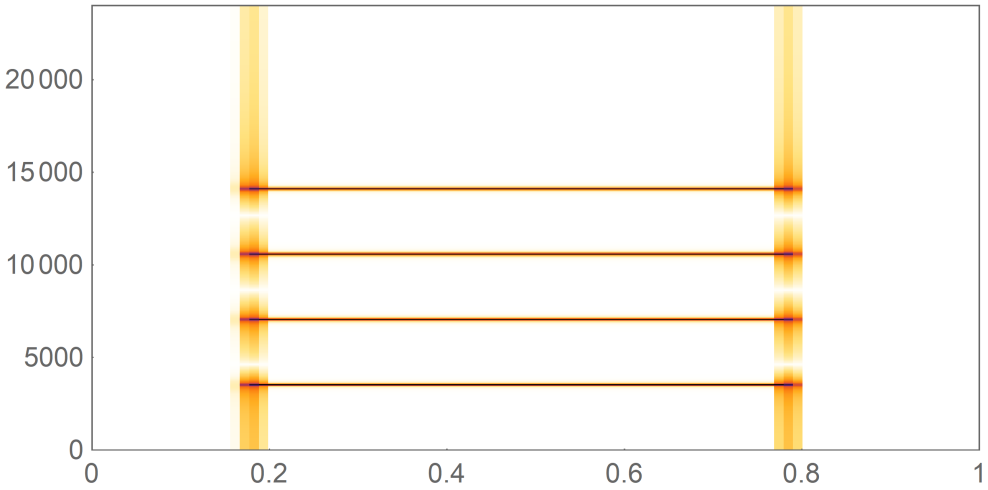


Figure 3.2: High frequency resolution spectrogram of signal transition

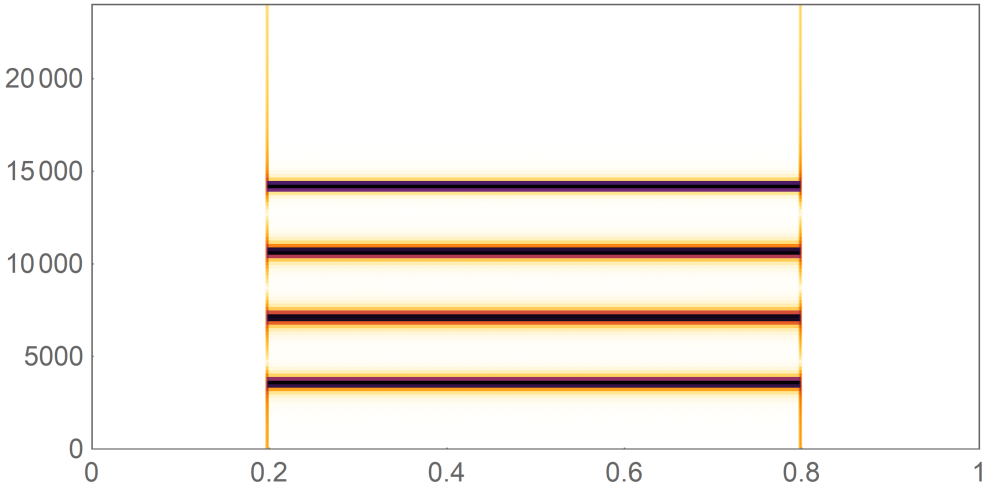


Figure 3.3: Low frequency resolution spectrogram of signal transition

at the frame edges where one resynthesised frame is immediately swapped for another one which does not perfectly match it at the edge.

To solve this problem, the Overlap-Add technique was created: after overlapping frames are selected in the same manner as with the STFT algorithm described in section 3.4 and transformed (not necesarilly using DFT and IDFT), the resynthesised frame samples are added to an output buffer with samples belonging to different frames that were taken from the same input stample being added to the same output sample. The output sample from output buffer can be used only after all overlapping frames this sample belongs to have been added.

3.5.1 Constant Overlap-Add criterion (COLA)

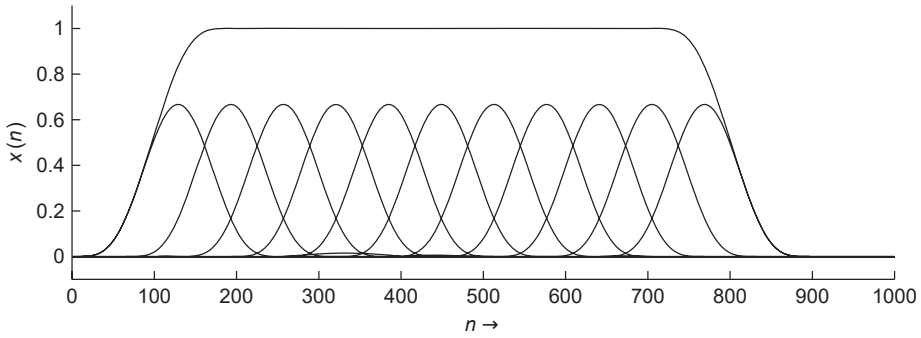


Figure 3.4: Sum of the square of von Hann windows
Von Hann windows with $66,\overline{6}\%$ ($O = 3$) overlap, and their Overlap-Add output, satisfying the Constant Overlap-Add property at the center. Used from [8, p. 233]

The Constant Overlap-Add criteron is important for preserving the amplitude of the signal processed by Overlap-Add algorithm. Essentially, the sum of windows of overlapping frames must be equal to 1. This assures that if the windows do not change during frequency processing, the resulting resynthesised signal will retain its original amplitude without artifacts.

Chapter 4

Analysis of real-time pitch shifting algorithms

Roughly speaking, there are two distinct types of pitch shifting algorithms: those that use a Short time Fourier transform (STFT) to convert the current frame to frequency domain, modify its bins and resynthesize them by converting back to time domain, and those that do not (time domain algorithms). I shall consider both categories separately.

4.1 Time domain algorithms

It is apparent that time domain algorithm have a much greater freedom in their functioning, as they are not defined by the presence of a STFT transform and a resynthesis, but rather the absence thereof.

4.1.1 Synchronous Overlap and Add (SOLA)

It should be noted that the classic SOLA algorithm is basically a time stretching algorithm (changing the duration of the signal without changing its pitch), not a pitch shifting one (changing the pitch of the signal without changing its duration). However, it is easily modified for pitch shifting by changing the sample rate of the signal before or after SOLA; this changes both the duration and the pitch of the signal and therefore can turn a time stretching algorithm into a pitch shifting one and vice versa [8, p. 201].

The basic algorithm functions as follows: [8, p. 191, 192]

1. The input signal is split into overlapping blocks.
2. The blocks are repositioned using the scaling factor.
3. Cross-correlation (similarity as a function of displacement) of the succeeding blocks is computed for the possible overlap interval.

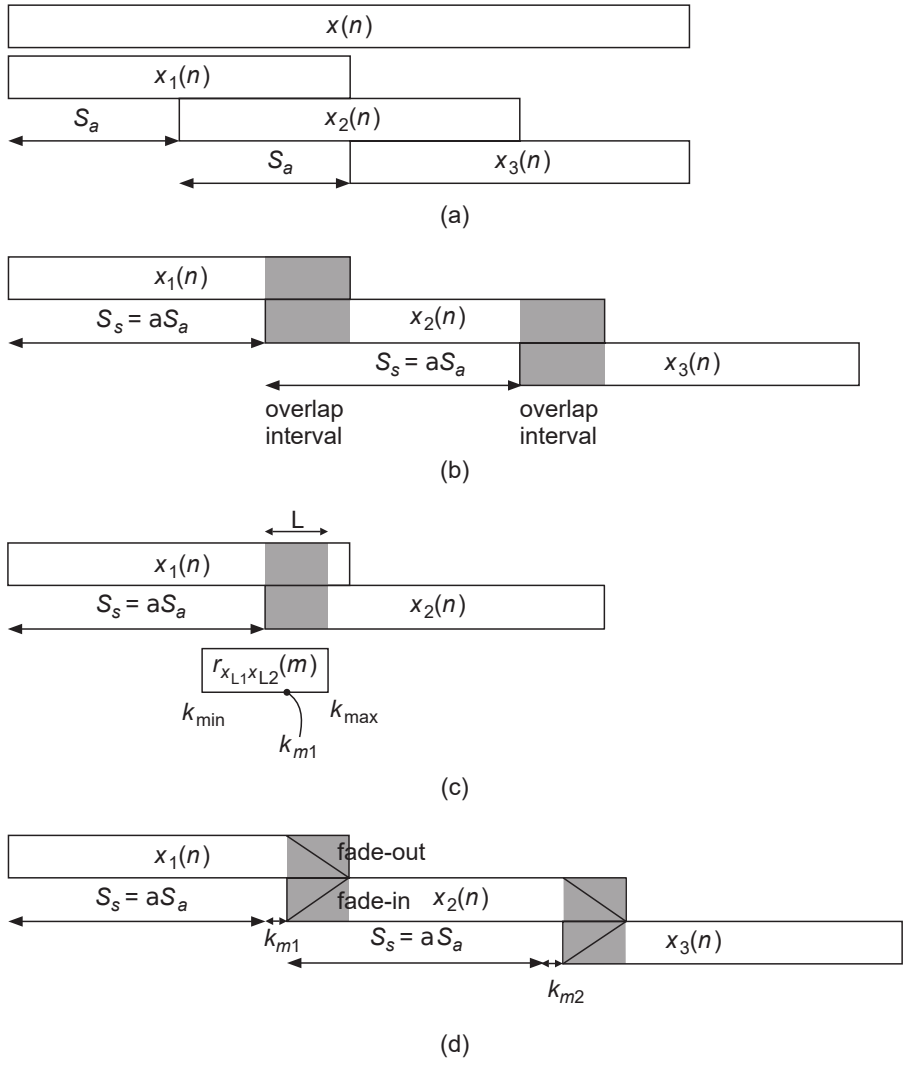


Figure 4.1: Principle of Synchronous Overlap and Add
a) Signal splitting, b) block repositioning, c) cross-corelation and discrete-time lag computation, d) fading and overlap-adding of blocks. Used from [8, p. 192]

4. The discrete-time lag is selected as the displacement which has the highest cross-corelation.

5. The repositioning is changed using the discrete-time lag, resulting in the succeeding blocks having the maximum similarity.

6. The amplitudes of the succeeding blocks are changed during the overlap, the ending one fading out and the starting one fading in, supplying a window function that assures the satisfaction of the Constant Overlap and Add criterion.

The SOLA algorithm has become a cornerstone of its time-scaling and pitch shifting algorithm family. The derived algorithms usually try to solve its shortcomings, namely its extreme computational requirements for computing successive convolutions of two long windows and its reliance on time-domain similarity which can sacrifice the frequency-domain similarity, resulting in noticeable artifacts.

4.1.2 Time Domain Pitch Synchronous Overlap and Add (TD-PSOLA)

The PSOLA algorithm, also known as TD-PSOLA to differentiate it from Frequency domain Pitch Synchronous Overlap and Add (FD-PSOLA), is a variation of SOLA used mainly for pitch shifting of voices and monophonic instruments. It is based on a hypothesis that the input signal can be characterized by its pitch, a consideration that cannot be made for general signals. In case of signals with different competing pitches, the algorithm can (and presumably will) fail, producing results with extreme artifacts.

The TD-PSOLA analysis algorithm performs these steps: [8, p. 194]

1. The pitch period is determined by analysis and pitch marks are placed at a pitch-synchronous rate during the periodic parts of sound and constant rate during the silent ones.

2. Segments centered at pitch mark are extracted using a von Hann window, with the length of two pitch periods to try to satisfy the Constant Overlap-Add criterion.

The TD-PSOLA synthesis algorithm performs these steps (this is done for every synthesis mark): [8, p. 194]

1. A synthesis pitch mark is placed according to the pitch multiplier and the best corresponding analysis segment is chosen for them.

2. The segment is overlap-added.

3. The next synthesis pitch mark is computed to be centered one pitch period after the current one.

Various filters can be inserted between the analysis and synthesis part, which can be done to approximate the vocal tract. This is interesting for pitch shifting of exclusively vocal audio.

While this algorithm does not require extensive computing resources, it can only reliably process monophonic signals and suffers from artifacts that increase with pitch multiplier becoming vastly different from 1. While various workarounds and improvements that reduce audible artifacts exist, the monophonicity is an inherent feature of the algorithm [8, p. 194, 196, 197].

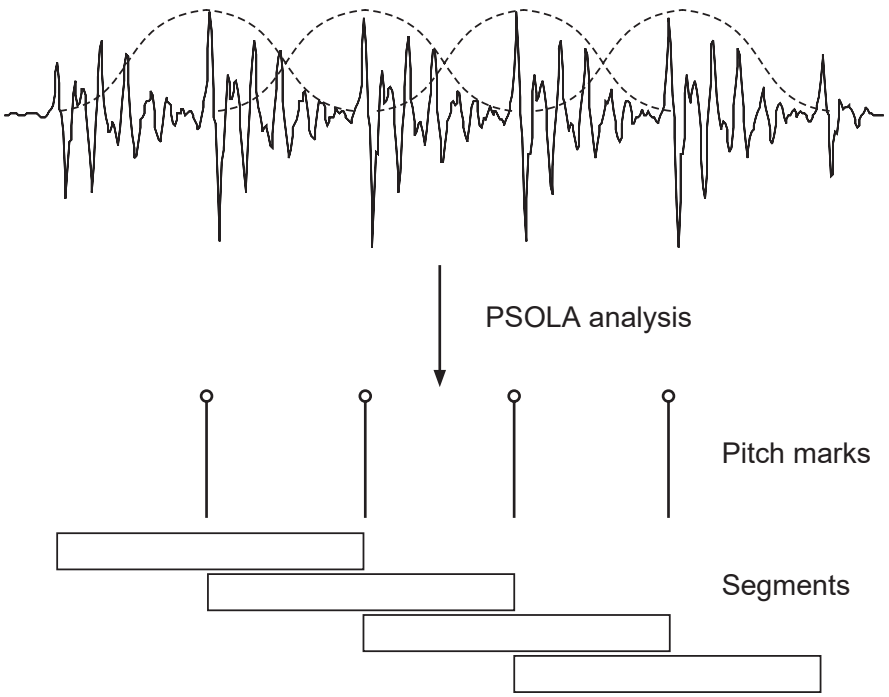


Figure 4.2: Principle of Pitch Synchronous Overlap and Add analysis
Pitch marks are placed according to the detected pitch and segments around them are extracted. Used from [8, p. 195]

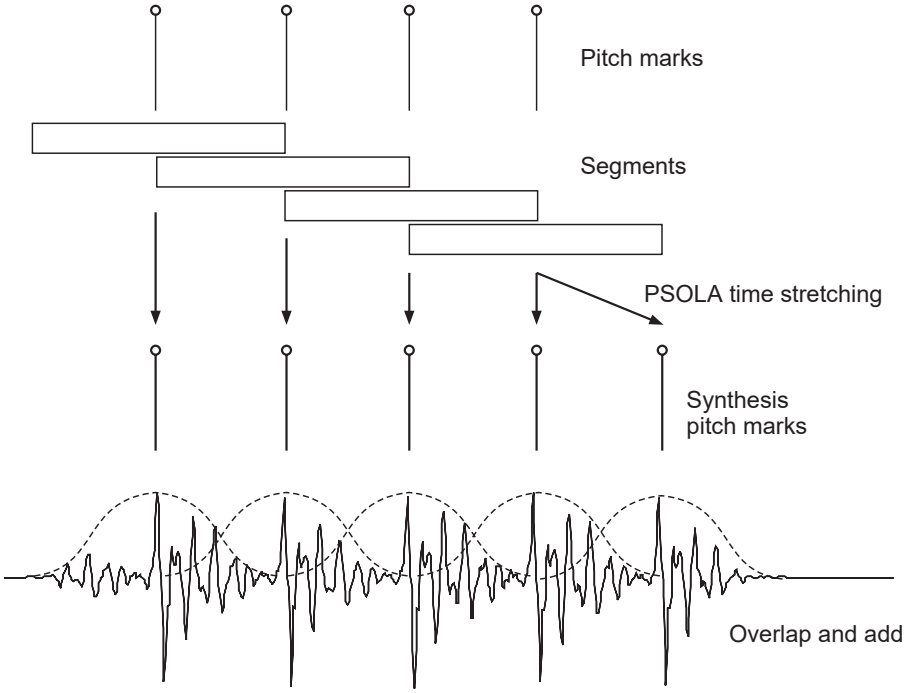


Figure 4.3: Principle of Pitch Synchronous Overlap and Add synthesis Extracted segments are resynthesised at synthesis pitch marks. Note that when time stretching for pitch shifting, some segments will be discarded or used more than once. Used from [8, p. 195]

4.1.3 Rollers algorithm

The novel Rollers algorithm uses a large number of Infinite Impulse Response (IIR) filters that are frequency shifted and then added together [9]. This novel approach approximates a frequency scaling (pitch shifting) operation with extremely low latency (around 5 milliseconds) and a low amount of artifacts, notably detuning (which can be regulated by increasing the size of the filter bank), tremolo, frequency notches and resonance. Most ”identifying marks” of classic techniques, such as phasiness, transient duplications and stereo field loss are absent, however. The most problematic property of the algorithm is its computational cost: as each filter must be processed in parallel, even a normal personal computer struggles with this task (or at least did in 2008, when the paper was released), prompting the authors to consider moving to parallel hardware such as general-purpose graphics processing units [9, ch. 7].

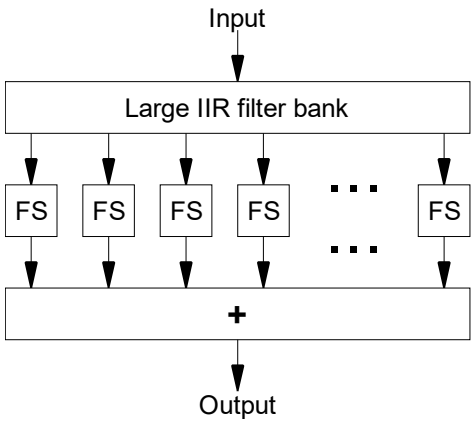


Figure 4.4: The principle of Rollers algorithm

4.2 Frequency domain algorithms

All of the following frequency domain algorithms except the Frequency Domain Pitch Synchronous Overlap and Add (FD-PSOLA) share a common step in their pipeline: before processing an incoming frame of audio samples, it is transformed into frequency domain using the Short time Fourier transform (STFT). The frequency domain algorithms work with the resulting frequency bins. After the pitch shift algorithm is done processing, an inverse transform is used. Due to the invertibility of the Fourier transform, doing this without any pitch shifting done leaves the resulting time domain data unchanged (with the exception of latency) from the input frame.

These algorithms are popular since the transformation into and out of frequency domain can be peformed by the Fast Fourier transform (FFT) algorithm, drastically decreasing their computational cost.

4.2.1 Standard phase vocoder

The phase vocoder is a classic algorithm proposed by J. L. Flanagan and R. M. Golden in [10] that tries to preserve both frequency bin (vertical) and time frame (horizontal) coherence of signals when performing a time-scale modification (TSM). While there are different implementations, the STFT one using FFT is the most well-known, requiring much less computing power than the other (filter bank and Gaboret) approaches. The idea of pitch shifting using the phase vocoder is to calculate the instanteneous frequency for each bin and to integrate the corresponding phase increment so that the signal can be reconstructed as a weighted sum of cosines of the phases [8, p. 250, 251].

Unfortunately, the standard phase vocoder introduces significant transient smearing and phasiness (or phase dispersion [8, p. 254, 255]) artifacts, especially when the pitch multiplier drastically differs from 1 [11, ch. 1].

4.2.2 Improved phase vocoder

A highly successful improvement of the phase vocoder, known either as an improved phase vocoder or a phase-locked vocoder, works with a hypothesis that the spectral peaks of sound do not vary drastically between frames and that their phase changes linearly in frequency. Thus, it finds spectral peaks, connects them to previous ones, propagates them (linearly in frequency) and rotates all frequency bins assigned to each peak equally. This, in theory, makes sure that the different frequencies propagated from one source do not change their phase relationship, which would result in phasiness [8, p. 250, 251].

Unfortunately, the improved phase vocoder is susceptible to analysis errors, even with constant-frequency sinusoids in case of small window size, and to stereophonic field loss [11, ch. 5].

4.2.3 Frequency Domain Pitch Synchronous Overlap and Add (FD-PSOLA)

This algorithm works similarly to the TD-PSOLA described in subsection 4.1.2. The only change is that the optional filtering done between the analysis and synthesis phase is performed by transforming the pitch epoch window into frequency domain and filtering there [12, p. 2]. This results in a decrease of computing complexity for expensive filters. Since the task is to implement a general pitch shifter, not a vocal-oriented one, where this can be used to emulate a vocal tract and get better results, this algorithm is not suitable.

4.2.4 Ocean algorithm

The Ocean algorithm is a novel algorithm for pitch shifting [11] which performs no analysis of the frequency spectrum, only shifting the frequency bins by the multiplier k in this simple manner (a is the input bin, b is the output bin and m is synthesis window size divided by analysis window size):

$$b = \lfloor mka + 0.5 \rfloor$$

.

The phase of the bin also has to be modified. This introduces a modulation effect, however, due to the the fact that the frequency bin shift actually also shifts the window, which can cause edge artifacts. The authors deal with this problem by introducing a synthesis window, which causes the output to no longer satisfy the Constant Overlap-Add property. This is solved by demodulating the result.

The full algorithm explanation in the original paper is quite lacking and confusing, but I have eventually figured out the probably intended sequence of steps performed for every STFT frame. Considering that N_a is the analysis size (equal to the frame size N), N_s is the synthesis size and O is the frame overlap.

1. Select N most recent samples.
2. Multiply them by an analysis window function (e. g. a von Hann window).
3. Transform the result into frequency domain using Fast Fourier transform on the N_a samples.
4. Shift the frequency bins and modify their phase, multiplying them by $m = N_s/N_a$)
5. Transform the result back into time domain using Inverse Fast Fourier transfom on N_s bins.
6. Multiply the start of the resulting sequence by a synthesis window function with N_a window size, discarding the other $N_s - N_a samples$.
7. Add the result to output buffers.
8. Multiply the output buffer which will not have anything more added to it by the demodulation modifier (real array with size $N_{analysis}$), starting at $N_{analysis}/O$ samples of the demodulation modifier.

Unfortunately, while doing away with most classic pitch shifting artifacts, this algorithm introduces detuning artifacts, worsening with decreasing N. It also introduces transient duplications for $k > 1$. On the other hand, as the algorithm contains no bin analysis (e.g. finding the bin with maximum amplitude and shifting according to that), it does not affect stereophonic field [11, ch. 3].

For the core transformation, as well as the aforementioned bin shift

$$b = \lfloor mka + 0.5 \rfloor$$

, a phase shift is needed since the bin phases would not line up between frames. The authors use a phase shift formula [11]

$$\omega_b = \omega_b e^{-j\frac{2\pi p(b-ma)}{mON}}.$$

Unfortunately, it has come to my attention during implementation that this formula is wrong. I shall derive the correct formula using the Fourier shifting theorem, which states that if one decides to start sampling $x(n)$ starting at sample k instead of 0, the resulting DFT of those time-shifted sample values is [4, p. 72]

$$X_{shifted}(r) = e^{j2\pi kr/N} X(r).$$

This applies to fixing the phase of nonzero frames because they are being sampled starting from $k = N\frac{p}{O}$. This results in

$$X_{shifted}(r) = e^{j2\pi \frac{p}{O} r} X(r).$$

As we have acquired analysis bins, we need to compute the phase shift relative to analysis size. Let me insert $\frac{b}{m}$ and a for r, which gives the equations

$$X_{shifted}(\frac{b}{m}) = e^{j2\pi\frac{p}{O}\frac{b}{m}}X(b),$$

$$X_{shifted}(a) = e^{j2\pi\frac{p}{O}a}X(a).$$

Now then, for the correct phase shift of the resulting bin b, one needs to undo the phase shift done by analysis bin a and apply the phase shift for analysis bin $\frac{b}{m}$, which results in

$$\omega_b = \omega_b e^{j2\pi\frac{p}{O}\frac{b}{m}} - e^{j2\pi\frac{p}{O}a} = \omega_b e^{j2\pi\frac{p}{O}(\frac{b}{m}-a)} = \omega_b e^{j\frac{2\pi p(b-ma)}{mO}}$$

.

It can be seen that I have arrived to a different phase shift equation. I have verified that my changed phase shift equation works as it is supposed to in my implementation. The original phase shift equation fails severely. The errors in the original paper might be explained by the authors not checking the correspondence of their mathematical equations to the variables they used in code.

As the computation of amplitude demodulation in the algorithm is not very well explained, I have devised an alternative solution consisting of applying the frequency domain transformation (bin and phase shift) to unity signal. Since this signal is windowed by analysis and synthesis window, I would expect the overlapped-and-added sequence of O frames to result in a signal that would take the exact form of the amplitude modulation of the signal. By dividing by this modulation, the real signal could be demodulated. Unfortunately, the implementation using this idea (discussed in more detail in subsection 7.4.2) still suffers from moderate amplitude modulation for certain pitch multipliers as detailed in subsection 8.6.2.

Chapter 5

Selection of a suitable pitch shifting algorithm for BF548 real-time processing

While the conventional algorithms such as PSOLA and improved phase vocoder are well-known, I have not chosen them mainly because of their dependence on signal analysis, performing well for monophonic signal but introducing artifacts for heavily polyphonic music. As my task is to create a truly universal pitch shifter, such behaviour is not acceptable. It should also be noted that separate analysis of the signal for the left and right channel of the stereophonic signal is likely to introduce differences between their processing that alter or destruct the perception of stereophonic field for signals that have one, while analysis of mixed signal is likely to result in extreme artifacts if both channels feature vastly different sounds (which can happen easily, for example, if two guitars playing separate parts are mixed in separate channels).

Only the Juillerat and Müller's Rollers algorithm and the Juillerat and Hirsbrunner's Ocean algorithm do not seem to suffer from such problems, while their authors note other problems, notably detuning errors, which do not result in a high quality loss in no circumstances, the worst case being slight audible detuning of signals with few harmonics, such as pipes.

The Rollers algorithm provides superior latency for acceptable-quality signals on the scale of 5 milliseconds, decreasing with higher frequencies due to the ability to fine-tune the filters to balance time-frequency localization for the particular frequency. It is, however, extremely demanding on computing power, constructing a large number of bandpass filter banks and frequency shifting those. The authors have even planned to use a graphics processing unit to run the algorithm on in the future as even the personal computer available to them could not handle high-quality filters! After some preliminary computations of feasibility of implementation of this algorithm on a digital signal processor, I

have decided against it as success could not be guaranteed. I would expect, however, that some current field-programmable gate arrays (FPGA) could be up to task as the algorithm is massively parallelisable.

After considering the drawbacks of the other algorithms, I have chosen to implement the novel Ocean algorithm which contains no analysis-dependent processing, instead of relying on shifting the raw input bins, and filtering done by switching into frequency domain by the means of Short time Fourier transform, which is quite fast on modern processors using the Fast Fourier Transform algorithm.

Chapter 6

Analog Devices Blackfin platform, tools and libraries

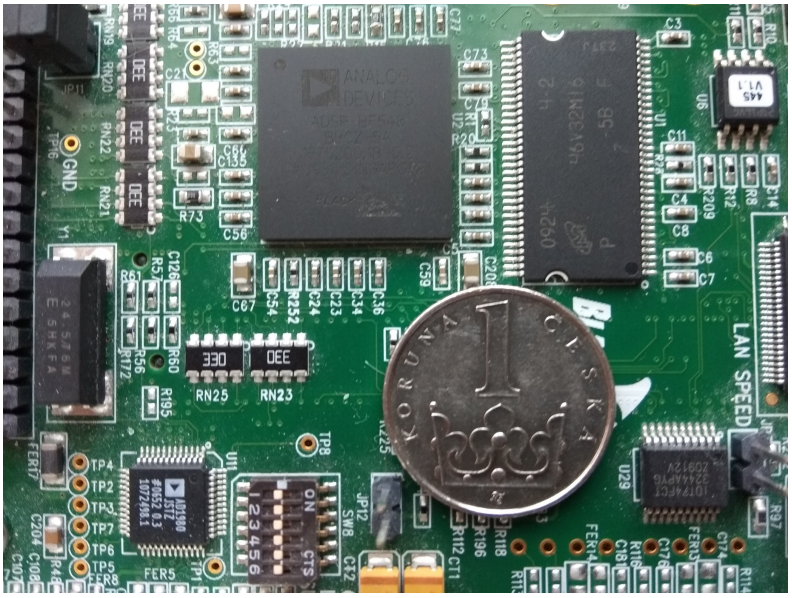


Figure 6.1: Blackfin BF548 digital signal processor
The used processor, mounted on the BF548 EZ-KIT, is shown in the upper middle of the image. The AD1980 audio codec is also shown in lower left. A 1 CZK coin is used for size comparison.

In this chapter, I shall discuss the Blackfin platform and tools and libraries supplied by Analog Devices for it. The core of the kit the algorithm is implemented on is a BF548 digital signal processor, one of the most powerful ones in the Blackfin processor family.

6.1 Blackfin processor architecture

Blackfin is a RISC processor architecture optimised for digital processing needs. This is made apparent not only by its dual fixed-point arithmetic units, but also by support of advanced digital signal processing functions such as built-in integer saturation arithmetic support and zero-overhead loops. the high processor speeds in hundreds of MHz. Most Blackfin processors features a three-level cache. The L1 cache is accessed in one clock cycle [13, p. 6-3] and the L2 cache is accessed in multiple clock cycles with optional cache.

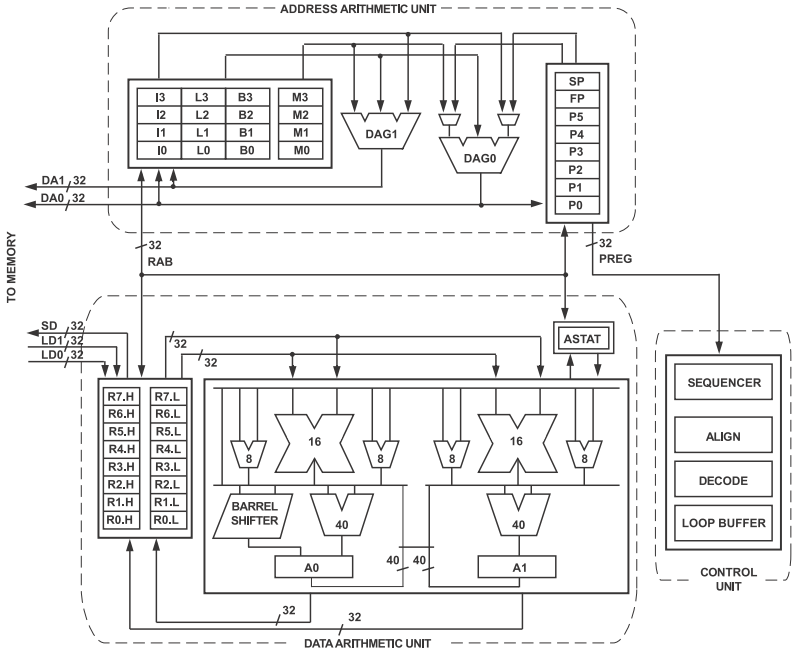


Figure 6.2: Blackfin core architecture
Note the two 16-bit multipliers speeding up the multiplications tremendously when performing 16-bit multiplications.

6.1.1 Blackfin BF548

The Blackfin BF548 is a top-of-the-line Blackfin processor, supporting a host of peripherials. Its performance and data cache sizes are the most interesting to us, however. The BF548 is operates at maximum clock frequency of 533 MHz [14, p. 3] and has two 16 KB banks of L1 cache [13, p. 3-7]. The L2 cache has a capacity of 128 KB [13, p. 3-8].

6.2 VisualDSP++ integrated development environment

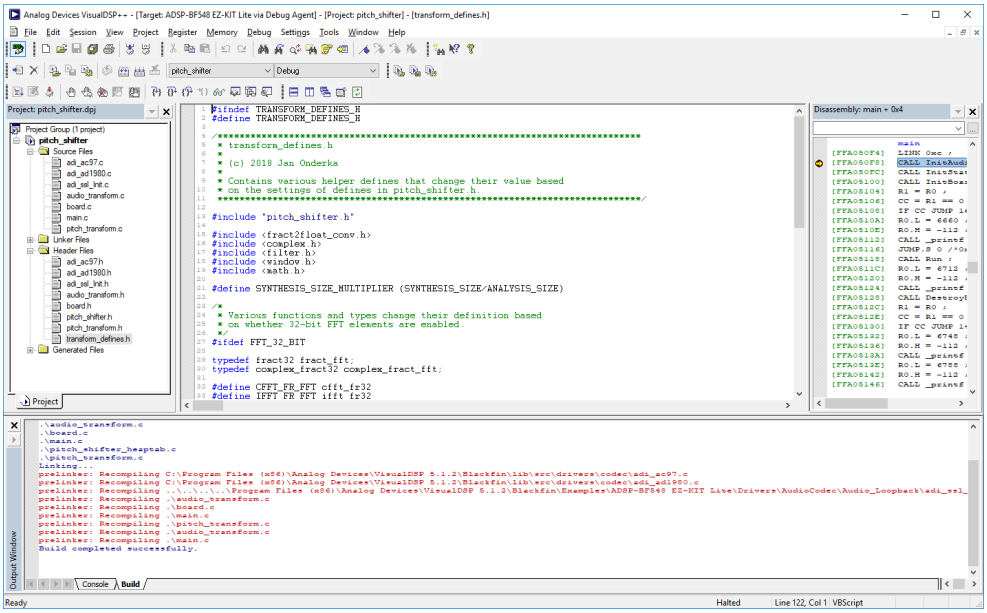


Figure 6.3: A typical VisualDSP++ session
The right pane in particular is interesting as it allows for viewing code disassembly, important especially for speed optimisation.

The kit ships with an evaluation licence for the Analog Devices VisualDSP++ integrated development environment (IDE). The evaluation licence is unrestricted for 90 days. After this period, connections to kit/processor simulators and emulators are disabled and the available code space memory is restricted to 60 kB [15, p. 1-11 and 1-12]. As the completed code uses approximately 80 kB of code space and this is not easily optimised, an unrestricted licence is a must.

VisualDSP++, even in the 5.1.2 version I am using, can be best described as a relic of the last century (according to the IDE splash screen, the first version of it was actually released in 1995). While it provides most of the functionality expected from an IDE, it suffers from notable instability, often crashing to desktop in situations where the debugged processor is disconnected, reconnected, reset by hardware or performs an invalid memory access. The last point makes debugging invalid memory accesses essentially impossible, forcing a thorough code inspection.

I would like to mention the Expert Linker utility in particular. According to Analog Devices:

> The Expert Linker is a graphical tool that simplifies complex tasks such as memory-mapping manipulation, code and data placement, overlay and shared memory creation, and C stack/heap adjustment. This tool complements the existing VisualDSP++ LDF format by providing a visualization capability enabling new users to take immediate advantage of the powerful LDF format flexibility. ([16, p. 4-1])

Unfortunately, I have found working with the Expert Linker to be a terrible experience that defies my understanding. Needless to say, I was not able to ascertain any useful information by using it.

I would definitely consider buying a licence to the CrossCore IDE, a successor to VisualDSP++ based on the Eclipse IDE [17], if I planned to work with the Blackfin family of processors professionally. That said, while I would not say that VisualDSP++ is a particularly good IDE, it is usable in the real world.

6.3 Used Analog Devices libraries

The VisualDSP++ installation, in addition to the core IDE and compilers, contains libraries for their processor lines and examples tailored to the various evaluation kits. The libraries are notably in their original source code form as opposed to many the practices of many vendors who only ship compiled binaries. This is very welcome as it allows for inspection of the source code in order to figure out usage and behaviour quirks.

6.3.1 Device Drivers and System Services

Analog Devices have implemented a "device driver mode" to provide concise, effective and easy-to-use interfaces for commonly used functionalities on embedded systems [18, p. 1-1], providing universality in Blackfin family. Unfortunately, I have found the driver interfaces to be very abstract, leaving me to figure out their proper functionality and usage using the hardware reference [19] and driver implementation source code.

This was hampered by the fact that the Analog Devices references and manuals are quite complex, owing to the general complexity of the Blackfin family of processors. For example, the BF54x hardware reference, while extremely useful and definitely recommended for anyone developing on this platform, is 2376 pages long [19]. This has impeded my progress, but I have acquired an understanding of the platform, especially the fundamental and Direct memory access (DMA) capabilities of the processor.

6.3.2 Digital signal processing library

A digital signal processing (DSP) library is also supplied for Blackfin by Analog Devices. It is mostliy written in ANSI C with the more computation-intensive routines are implemented in assembly language, speeding up the computation beyond optimisation achievable by the C compiler. This frees signal processing developers from the need to write hand-crafted assembly.

For pitch-shifting purposes using Ocean algorithm, fast implementations of Fast Fourier transform (FFT) and Inverse Fast Fourier transform (IFFT) are particularly important. In the Blackfin DSP library, the fastest power-of-two versions exist as the `cfft` [20, p. 4-98] and `ifft` [20, p. 4-189] families of functions and are augmented with support of dynamic scaling: unlike the basic FFT and IFFT which essentially scale the resulting values by N and $1/N$, respectively, dynamic scaling divides the input at any FFT stage if and only if the largest absolute input value is greater than or equal to 0.25 [20, 4-99]. This solves overflow problems and largely solves underflow problems. The functions require an array of precomputed constants called the twiddle factor to be input. These can be computed beforehand by the `twidfftrad2` family of functions [20, p. 4-242] and used for different N sizes provided they are all powers of 2.

FFT/IFFT functions that are available accept their input in form of complex numbers of type `complex_fract16` or `complex_fract32`, comprised of two values (real and imaginary) with fractional fixed-point types `fract16` and `fract32` for `complex_fract16` and `complex_fract32`, respectively. These types and provided arithmetic functions manipulating them are perfect for digital signal processing, manipulating the signal as if it was between 1 and -1 (excluding 1 due to the two's complement notation) and clipping it if it goes outside that range.

Also provided are functions for computing various well-known symmetric windows. For pitch-shifting purposes, the gen vonhann family of functions which generate a symmetric von Hann window is particularly interesting [20, p. 4-185].

Chapter 7

Ocean algorithm pitch shifter implementation on BF548 EZ-KIT

In this chapter, I shall go over the pitch shifter implementation.

I have implemented the pitch shifter as a VisualDSP++ project, writing it in ANSI C99 with emphasis on processing speed. I was worried whether I would need to write critical code sections in Blackfin assembly language, but the VisualDSP++ compiler proved to be good at aggressively optimising costly operations (mostly buffer copies combined with element-by-element multiplications).

For greater maintainability and readability facilitated by high cohesion and low coupling, my implementation decomposes codec handling and the Ocean algorithm into 5 separate components:

- Common definitions: preprocessor definitions that change qualitative and quantitative characteristics of the pitch shifter.
- Peripherial handling: initialisation and destruction of codec driver,setting up codec driver audio buffer processing and handling filled buffer interrupts.
- Pitch transformation: handling the transformation into frequency domain, shifting bins including modifying their phases and transformation back into time domain.
- Audio transformation: extracting frames from audio buffers, windowing, setting up and running pitch transformation on frames and performing amplitude demodulation.

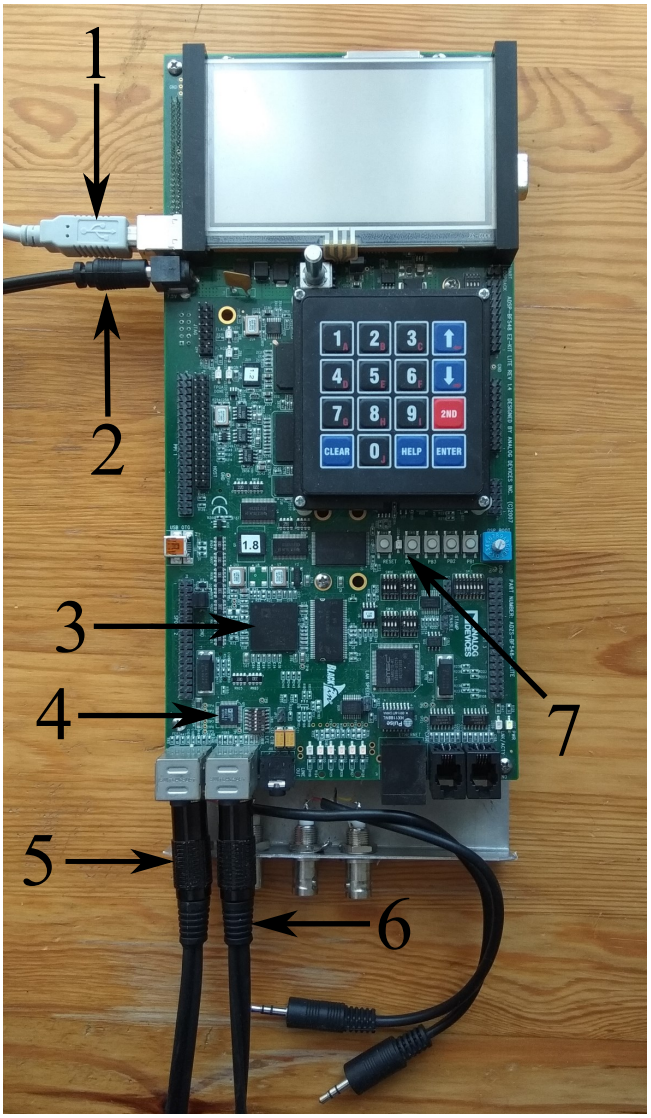


Figure 7.1: Used features of Analog Devices BF548 EZ-KIT
Only a few kit features and connections are used:
1) USB-B debugging connection to the host computer,
2) 7.5 V power supply connection required for powering the kit,
3) Blackfin BF548 digital signal processor,
4) AD1980 codec used for input/output audio signal conversion,
5) Line In stereo audio input using a 3.5 mm audio jack,
6) Surround stereo audio output using a 3.5 mm audio jack,
7) Pushbutton used to terminate the running program.
Note that the metal board under the BF548 EZ-KIT containing adapters from 3.5 mm audio jacks to RCA connectors was installed by the university staff. I have not used it.

- Main processing routines: initialising board and audio transformation components and synchronisation of audio transformations to filled buffer interrupts.

I shall describe each one of them in greater detail in the following sections.

7.1 Common definitions

I have put the basic preprocessor definitions that impact the pitch shifter characteristics (quality, latency, computational speed) into file `pitch_shifter.h`. I shall list the various definitions here:

- `FRAME_OVERLAP`: Frame overlap O defined as number of overlapped frames contributing to one sample.
- `ANALYSIS_SIZE`: Analysis window size N_a, must be a power of two. 512 is a reasonable minimum.
- `SYNTHESIS_SIZE`: Synthesis window size N_s, must be a power of two. 512 is a reasonable minimum.
- `FFT_32_BIT`: If defined, the Fast Fourier transform will be performed with 32-bit scalars, if not, it will be performed with 16-bit scalars. Massive performance impact.
- `LINE_IN_AS_INPUT`: If defined, codec Line In is used as the input source. If not, codec Mic In is used as the input source (in stereophonic configuration).
- `TERMINATE_BUTTON`: Flag of pin on which the program termination button resides.

There are also definitions that configure the placement of important arrays in memory: this enables their quick change if the current settings deplete the memory of L1 or L2 cache.

Note that the pitch multiplier is not controlled by a preprocessor definition.

Complementing this header is `transform_defines.h`. In this file, I have put definitions dependent on `pitch_shifter.h` that should not be tweaked in most situations.

7.2 Peripherial handling

In `board.h` and `board.c`, I handle everything related to the board peripherials. This includes initialising and destroying used Blackfin system services (see subsection 6.3.1) and the codec driver.

Initialisation is handled by function `InitBoard()` and destruction by function `DestroyBoard()`. Function `IsTerminateButtonPressed()` can be used to check whether the terminate button is being pressed. Global buffers `g_InData` and `g_OutData` are provided for use by transformation routines and external functions `InputBufferFinished()` and `OutputBufferFinished()` (declared, but not defined by this component) called by the codec driver interrupt provide a fast way to notify the main loop that the codec has processed a buffer.

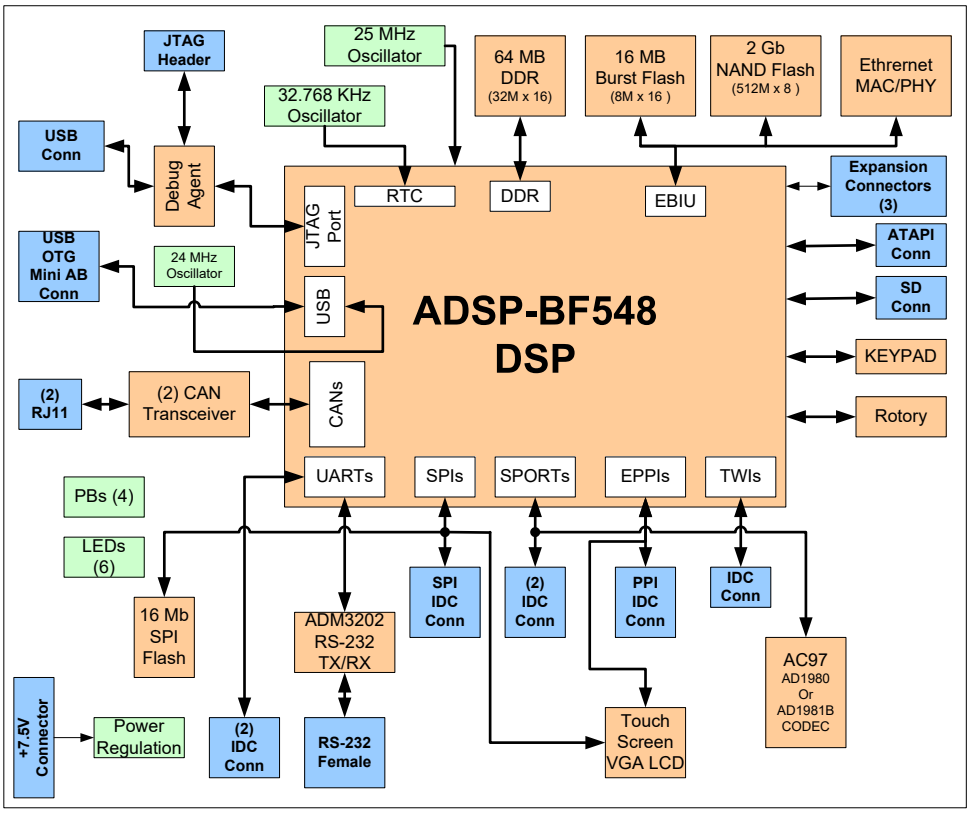


Figure 7.2: Analog Devices BF548 EZ-KIT board architecture

The AD1980 codec (or AD1981B, with different board pinout but same for our purposes) is connected to SPORT0 serial bus and handled by the Blackfin AD1980 driver. The program termination pushbutton is connected to processor pin PB11 and handled by the Blackfin Flag driver. Used from [15, p. 2-18].

7.2.1 Audio codec interfacing

The BF548 EZ-KIT uses an AD1980 codec connected to the SPORT0 interface of the BF548 processor as seen in Figure 7.2 [15, p. 1-25]. The AD1980 codec

implements the AC'97 codec developed by Intel, with support for a proprietary mode with several more options. It has analog-to-digital converter total harmonic distortion of 78 dB and resolution of 16 bits (resulting in a signal-to-quantization-noise ratio of 96.33 dB) [21, p. 3], which provides sufficient quality but only 3 bits of headroom. Its standard sample rate of 48 kHz is completely fine for our purposes.

Supplied Analog Devices AD1980 codec driver can be used for interfacing with the codec. During testing (as detailed in section 8.1), I have found that when receiving from Line In, the input samples were attenuated by almost exactly 7.5 dB, resulting in a 7.5 dB output attenuation. I have therefore increased the recording gain by 7.5 dB compared to the default codec value. This essentially solved the problem, but the signal to noise ratio surely worsened. I did not conduct a more in-depth investigation. It should be undertaken before using the AD1980 codec as-is in a final product (which I would not recommend in any case due to its not very impressive characteristics).

I also found a bug concerning the codec during implementation. If the standard input/output (I/O) is used after codec initialization, it somehow knocks the codec interfacing out of synchronization about one third of the time. The codec then periodically outputs 1-3 invalid samples with a period corresponding to the codec driver internal circular buffer size (presumably old values previously written there). As with the previous problem, I did not investigate the root cause of the problem further.

I have not found the root cause of this behaviour, but would hazard a guess that the codec driver and standard I/O driver compete for interrupt priority. This theory is probably incomplete, however, since that would explain the codec driver outputting a wrong value occasionally when using standard I/O, but not the continuation of this behaviour afterwards. I intend to investigate this behaviour further when I have the time. For the time being, I have refrained from using standard I/O after initialising and before destroying the codec driver. This solved my problem nicely.

While the communication with the codec can be delegated to the Analog Devices implementation, I needed to figure out how exactly should I set up the input and output buffers so that the processing of the analysis window could begin immediately after all of the needed samples were received from the codec and have the maximum time available for processing while respecting the output sample deadline.

7.2.2 Input and output circular buffer design

There are three basic controllable constants in the Ocean algorithm that can be tuned to get the optimal balance of processing speed, latency and quality: size of the analysis window N_a, window overlap O and size of synthesis window N_s. The window overlap is the number of analysis windows processed during one size of the analysis window. Naturally, it follows that the sample length

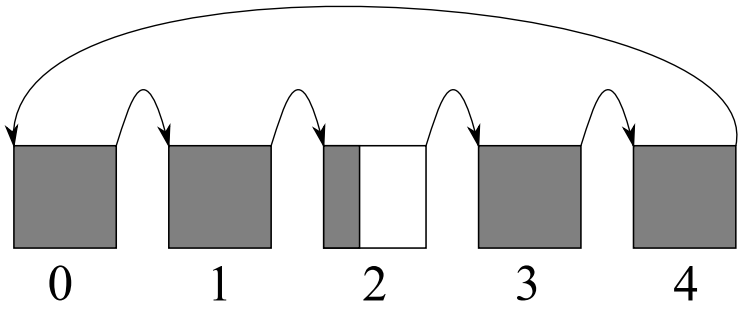


Figure 7.3: Chain of buffers as seen by codec driver
Buffer 2 is being processed by the codec driver. Latest buffer completion interrupt was fired for buffer 1. Grey rectangle areas represent relatively new sample data while the white area represents sample data taken in last buffer pass and not yet replaced.

between starts of processing of succeeding windows can be written as:

$$s_a = \frac{N_a}{O}$$

Because of the real-time requirement, the program needs to be done with the processing of the current frame before starting processing the next one; therefore, processing can take at most the time corresponding to s_a incoming samples at 48 kHz sample rate. This means that every s_a samples, an interrupt should be received which signifies that a new frame should be processed.

This requirement can be implemented by submitting buffers with s_a sample size to the Blackfin codec driver which feeds them with the incoming audio data in a circular fashion, firing an interrupt after each buffer has been filled and continuing with the next one. The codec driver essentially treats the buffers as a chain as seen in Figure 7.3.

For estabilishment of the number of the buffers needed, I have considered that a frame processing routine first copies the latest N_a samples to a special processing array. As this requires O buffers and the buffer after the last processed is currently being refilled by the codec driver, $O + 1$ buffers are required for the whole array, which puts the start of the latest N_a samples at the start of buffer two positions after the latest buffer filled by the codec driver. To speed up the frame processing routine, the best solution to put the buffers inside a continuous array. This is being shown by Figure 7.4.

I have applied the same idea is applied to the output buffers. Since the synthesis window has the same size as the analysis window, the input and output buffers map perfectly 1:1.

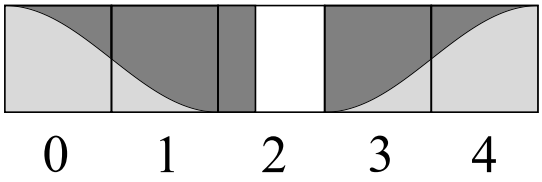


Figure 7.4: Array of buffers as seen by processing algorithm
While buffer 2 is being filled, buffer 3, 4, 0 and 2 can be copied for processing. The light grey Hann window shows how the samples will be windowed before Fast Fourier transform. Windowing is actually performed after copying the samples out to a separate buffer.

7.3 Pitch transformation

I have implemented the heart of the Ocean algorithm (described in subsection 4.2.4) in files `pitch_transform.h` and `pitch_transform.c`. Function `InitPitchTransform()` initialises the pitch transform, taking the phase multiplier as its sole argument. Function `PitchTransform()` pitch transforms an input transformation array, invalidating it and storing the result in an output transformation array. It takes the current frame number as its sole argument.

7.3.1 Transformation arrays

For the purposes of pitch transformation processing, two separate arrays of size N_s are declared, `g_FrameInData` and `g_FrameOutData`. They are declared globally so that the caller can fill the input and retrieve the output.

7.3.2 Core pitch transformation implementation

Bin and phase shift buffers containing precomputed values are initialised using the equations from subsection 4.2.4 by calling `InitPitchTransform()` with a pitch multiplier as its sole argument. The core function `PitchTransform()` is implemented as follows:

1. `g_FrameInData` is transformed into frequency domain using Fast Fourier transform and stored into `g_FrameOutData`.

2. `g_FrameInData` bins are zeroed.

3. Each `g_FrameOutData` bin $a < N_a/2$ is phase shifted by a precomputed value selected depending on the current frame number and added to `g_FrameInData` bin with precomputed number $b = \lfloor mka + 0.5 \rfloor$.

4. The bins $a > N_a/2$ are filled according to the $a < N_a/2$ bins so the FFT result is still real-valued (not doing that can result in precision artifacts). The bin $N/2$ is copied over.

5. `g_FrameInData` is transformed back into time domain using Inverse Fast Fourier transform and stored into `g_FrameOutData`.

7.3.3 Fast Fourier transform implementation considerations

I have used the Blackfin DSP library FFT implementation, described in subsection 6.3.2, for both the transformation into frequency domain using the Fast Fourier transform (FFT) and the transformation back to time domain using the Inverse Fourier transform (IFFT). As touched on there, FFT scaling its result by N poses a huge problem for 16-bit numbers: with a window size on the upper reasonable limit, $N = 4096$, $\log_2 N = 12$ bits of the signal are irretreivably lost. This results in extreme artifacts.

This problem is alleviated by using dynamic FFT scaling, also explained in subsection 6.3.2, where the FFT result is scaled based on its maximum value, retaining most of the information in majority of cases, but degrading during sudden changes of signal amplitude. It can also be eliminated entirely by using 32-bit numbers for internal processing, which gives an ideal amount of headroom at the price of a big performance hit (32-bit FFTs are approximately 5 times slower than their FFT counterpart for N around 1024 due to the slower multiplications explained by section 6.1).

7.4 Audio frame processing

In `audio_transform.h` and `audio_transform.c`, I have implemented audio frame processing required by the algorithm. It neatly solves analysis windowing, synthesis windowing and amplitude demodulation, leaving only the frequency domain manipulation to be done in the core transformation implementation.

Function `InitAudioTransform()` initialises the audio transformation (including calling `InitPitchTransform()` and initialising amplitude demodulation array). It takes pitch multiplier as its sole argument.

Function `AudioTransform()` is called with the index of the newly finished buffer. The function first zeroes the output buffer that was just finished. It then processes each channel separately in this manner:

1. The newly available processing frame consisting of N_a samples is moved from the input buffer into `g_FrameInData` while being windowed with an analysis Hann window.

2. Pitch processing routine `PitchTransform()` is called with the current frame number.

3. The result is windowed with a synthesis Hann window.

4. The result is added into the output buffer array starting with the buffer that is going to be output next.

It then applies amplitude demodulation to the output buffer that is just going to be processed and increments the frame number (resetting back to zero on frame overlap).

7.4.1 Windowing implementation

The Ocean algorithm calls for an analysis window and a synthesis window, both with size N_a. This retains the latency of analysis windowing and removes artifacts stemming from the phase shifts affecting the analysis window. In order not to introduce spurious amplitude pulsing artifacts, the windows need to satisfy the Constant Overlap-Add requirement. This requires usage of periodic versions of classic symmetrical windows. Following the authors of the Ocean algorithm, I have chosen to use the von Hann window, described in subsection 3.4.1. The tried and proven von Hann window seems not to be a cause of concern. To refrain from polluting the implementation with unneeded algorithms, I have chosen to have the window generated for me by the Blackfin DSP library, which includes a family of functions for generation of the symmetrical form as described in subsection 6.3.2. I have thus increased the size of the array holding my window to $N_a + 1$, using only the first N_a entries. The resulting window is used both for analysis and synthesis windowing.

7.4.2 Amplitude demodulation computation

I have used the idea of using unity signal to compute the amplitude modulation as detailed in subsection 4.2.4. The whole idea consists of using a sequence which is always 1 as a signal, audio tranforming it for O frames (without performing the amplitude demodulation, of course) and putting the resulting Overlap-Add output in an array of size N_a (since the whole cycle of analysis windows completes in N_a samples). This decreased the amplitude modulation, but did not solve it completely, as it is still present for some pitch multipliers. This is discussed in subsection 8.6.2.

I have used floating-point numbers to store the demodulation. By recomputing the demodulation buffer and putting the multiplicative inverses of the original unity signal modulation result in it, I was able to perform the demodulation just with one scalar emulated floating-point multiplication for each sample channe. This could be done reasonably fast even on the fixed-point Blackfin achitecture.

7.5 Main processing routines

I have put the main processing routines in a conventionally named source file `main.c`. Functions `InputBufferFinished()` and `OutputBufferFinished()`, declared in `board.h` and called by the codec interrupt, are also defined here.

`InputBufferFinished()` first checks if the audio transformation of last finished input buffer is completed. If not, it increments a counter of unprocessed buffers (it is a surefire sign that the hardware is not fast enough to perform the audio transform with current configuration). The function then sets the index of the input buffer to a global variable and also sets a flag that a new input bufffer is available for processing.

`OutputBufferFinished()` just sets the index of the output buffer to a global variable.

The main loop inside the `Run()` function continues looping until the terminate button (located on the kit) is pressed. In the loop body, it checks the new input buffer flag. If it is set, it waits for the input and output buffer synchronisation. After they are synchronised, it calls `AudioTransform()`. After the transform is completed, it clears the input buffer flag.

The `main()` function is the main program entry point. It initialises the audio transformation and board components and calls the `Run()` function. After it returns, `main()` destroys the board and prints statistics taken while running and enters an infinite loop which does nothing (serving as a stopgap preventing the `main()` function from ever returning, since returning from `main()` does not make sense in embedded software).

Chapter 8

Achieved performance and testing

In order to test the implementation, I have devised a testbench using a cheap but acclaimed professional external audio interface Focusrite Scarlett 2i2. Since the pitch shifter would be used in conjunction with audio interfaces in the real world and these can reproduce the exact same passband as the AD1980 codec (20 Hz to 20 kHz), I have found no need for measurements using an oscilloscope.

8.1 Testbench setup

Cable setup is detailed in Figure 8.1. The correct levels were set by setting the input style to line input and manipulating the master volume knob and the left and right input channel gain so the audio interface output levels would not clip the AD1980 and the AD1980 output levels would not clip the audio interface inputs. As mentioned in subsection 7.2.1, I have found out that with default settings, processed signal was attenuated by almost exactly by 7.5 dB compared to passthrough signal. I have therefore increased AD1980 recording gain by 7.5 dB, which essentially solved the problem. In the end, the input channel gain knobs were set to approximately 10½ o'clock and master output gain knob to approximately 2 o'clock. Setting both channel gain knobs to the exact same value proved to be a problem since there are no setting labels.

The audio interface bit depth was selected to be 24 bits and sample rate to 48 kHz. The bit depth of played test sounds differed, but was never lower than 16 bits.Their sample rate was 44100 Hz. It is exceedingly unlikely that these settings could introduce noticeable artifacts.

As for the software side, the completed implementation was loaded into the BF548 processor via the VisualDSP++ studio. Audacity audio editor and recorder was then used to play the original data via the audio interface outputs

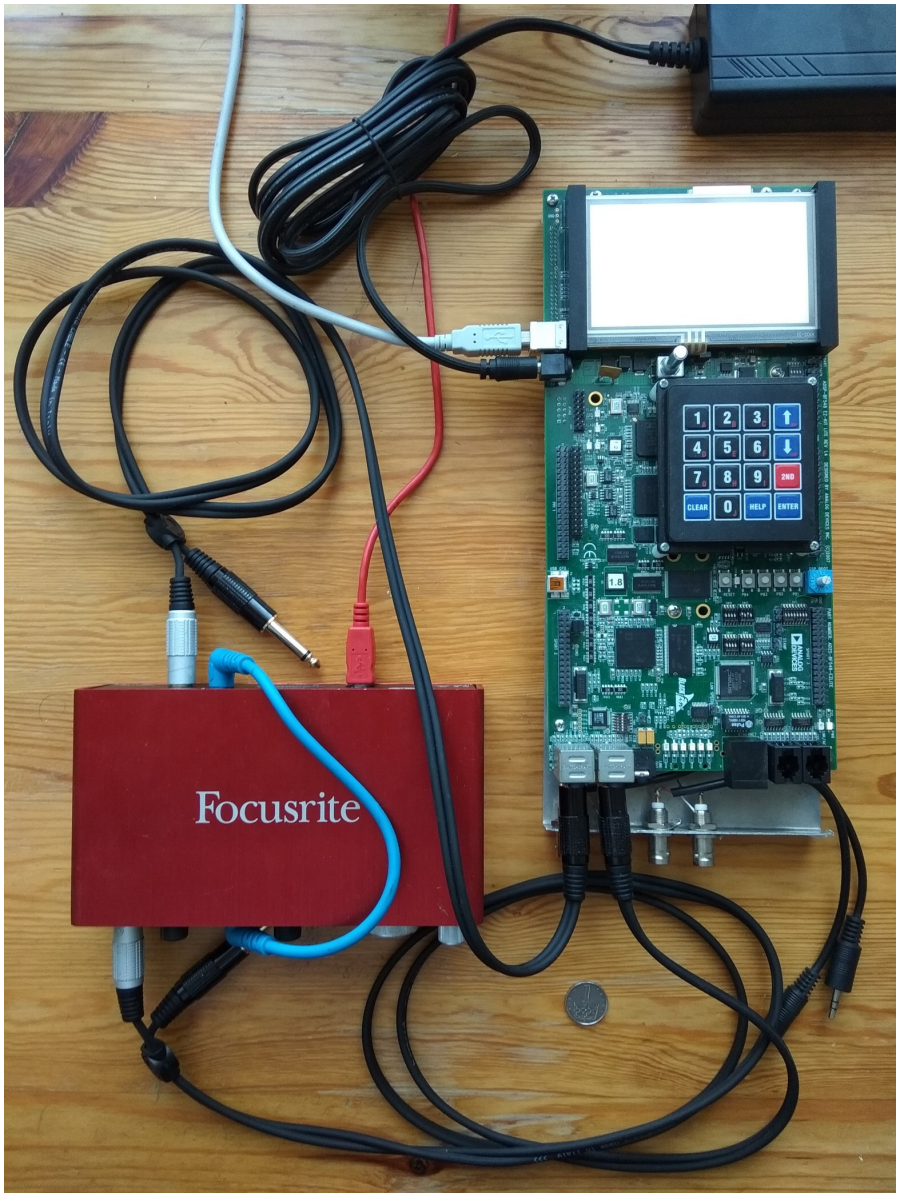


Figure 8.1: Implementation testbench
Left output of the audio interface is connected to one input channel of AD1980 Line In. The same processed channel is fed back from AD1980 Surround Out to the left input of the audio interface. The right channel of the audio interface is looped back to act as a control, allowing comparison between it and the processed channel. The feedback cable was replaced by the previously unused second channel jacks during the testing of stereophonic functionality.

while recording its inputs.

For the reader's consideration, I have saved the testing results converted to 16-bit bit depth and 44100 Hz sample rate in Waveform Audio File format (`.wav`) to the microSD card submitted with this thesis. See Appendix B for more information.

8.2 Performance

I have selected the most interesting pitch shifter configurations to pitch shift the sounds with, detailed in Table 8.1. Before testing them with audio data, I needed to set their overlap factor. Since I wanted to use the highest power-of-two overlap factor possible (it should have had a positive impact on the resulting quality), I have experimentally recorded the maximum overlap where all input frames can be processed for each configuration. I have verified this by using the unprocessed buffer functionality described in section 7.5. This overlap was then used for further testing of the configuration.

Notice that the Blackfin BF548 is powerful enough for use of a very generous overlap $O = 16$ for every configuration with Fast Fourier Transform (FFT) scalar 16 bit depth and synthesis size up to 2048. The first bottleneck is due to the fact that Blackfin only has 16-bit multipliers as mentioned in section 6.1. The second is due to the fact that with synthesis size over 2048, the FFT data arrays no longer fit inside the 32 kB L1 cache, forcing at least one to be put into L2 which takes much longer to fetch from and write back.

Since analysis sizes bigger than 2048 result in considerable latency and synthesis sizes over 4096 do not result in much noticeable improvement according to the authors of the Ocean algorithm [11, ch. 3], I did not consider these. As the authors have concluded that overlap factor $O \geq 8$ is necessary for high-quality results, I have deemed the performance of the implementation

Table 8.1: Settings of configurations used for testing and maximum overlap factor

Name	**FFT bits**	**Analysis size**	**Synthesis size**	**Max overlap**
LQ	16	512	512	**32**
LQE	16	512	2048	**16**
MQE	16	1024	2048	**16**
MQEE	16	1024	4096	**4**
HQ	16	2048	2048	**16**
HQE	16	2048	4096	**4**
HQ32	32	2048	2048	**4**
EHX	Professional guitar pitch shifter			
AUD	Audacity audio editor non-real-time pitch shifter effect			

satisfactory. The overlap factor most probably could be raised to $O = 8$ for every tested configuration after a further aggressive manual optimisation.

8.3 Test sounds

I have used five different sounds for thorough testing, detailed in table Table 8.2. With the exception of `piano_stereo`, all of them are monoaural and were rerecorded with the left audio channel pitch shifted and the right channel looped back.

As well as various configurations of my implementation, two other pitch shifters are tested. The first one of them is a commercial guitar pitch shifting pedal Electro-Harmonix Epitome. It supports pitch shifting by multipliers $\frac{1}{2}$ and 2 (in musical terms, octave down and octave up) and is set up in the testbench the exact same way as my pitch shifter. The other pitch shifter tested for comparison is a built-in non-real-time pitch shifting algorithm in the Audacity audio editor.

I have used five different multipliers to pitch shift the testing sounds: $\frac{1}{2}$ and 2, since they are the only multipliers supported by the commercial pitch shifter, $\frac{3}{2}$ for a fractional harmonic relationship, $\frac{11}{13}$ for a quite disharmonic relationship and 1 for a simple identity passthrough with the only desired effect being system latency. (The passthrough is only recorded once for every analysis size.)

8.4 Sound latency

Using the `sine` sound, I was able to ascertain the sound latency by measuring the duration between the start of the sinusoid in the right loopback channel and the start of the pitch shifted sinusoid in the left channel. I arrived to results described in Table 8.3.While there were some transient artifacts in the pitch shifted sounds, I have determined the latency to be largely (to 2 ms precision) dependent only on the analysis size and overlap. This is in line with my expectations: each sample is processed only after every frame containing it is processed. A frame is processed only after all of its (analysis size) samples have been processed. The processing takes analysis size divided by frame overlap samples to complete; therefore, the latency is a sum of those two.

I have also analysed the EHX pitch shifter and determined its latency to be roughly between analysis window sizes 1024 and 2048.

The feeling I have when confronted with this latency is that the latency for $N_a = 512$ is almost imperceptible, $N_a = 1024$ is somewhat perceptible and $N_a = 2048$ is very perceptible.

Table 8.2: Description of sounds used for testing

Sound name	**Length**	**Description**
bass	26 s	All notes on a four-string bass guitar played in an ascending sequence. The bass guitar is notably difficult to pitch shift well due to the pure harmonic relationships of its low-frequency notes.
piano	5 s	Recording of a grand piano playing. Grand piano has a rich frequency content due to the resonances of various woods.
piano_stereo	5 s	The same recording, but this time, both audio channels are pitch shifted, as opposed to only the left one, the right one being looped back. The sound contains an interesting and varying stereophonic field, making it ideal for a verification of stereophonic field preservation.
sine	1.44 s	0.5 seconds of silence, followed a 440 Hz sinusoidal wave for 0.44 seconds, followed by another 0.5 seconds of silence. The sine wave can be used to analyse the behaviour of a pitch shifter on transients and its ability to accurately pitch shift a single wave.
song	13 s	A song excerpt containing a guitar and a male singing voice, both being fairly conventional targets for pitch shifting.
speech	15 s	An old speech excerpt containing a single male voice and a speaking chorus. A large amount of recording noise is present, interesting for an analysis of how the pitch shifter transforms noise.

Table 8.3: Latency as a function of analysis size and overlap

Analysis size	**Overlap**	**Worst case latency**
512	32	13 ms
512	16	14 ms
1024	16	28 ms
1024	4	34 ms
2048	16	52 ms
2048	4	67 ms
EHX pitch shifter		36 ms

8.5 Stereophonic field preservation

As explained by the authors of the Ocean algorithm, no stereophonic field loss can occur when processing a stereophonic signal with Ocean algorithm. I have verified this with the `piano_stereo` test sound. The stereophonic field was always perfectly preserved. The commercial pitch shifter does not support stereophonic processing (very notably, as it also contains two other sound effects which are stereophonic, the pitch shifter being the only monophonic effect). If confronted with a stereophonic signal, it sums it to mono before processing. This obviously results in a total loss of a stereo field.

8.6 Sound quality

The general pitch shifting quality of the algorithm is comparable to other pitch shifters. Some artifacts were found and shall be discussed.

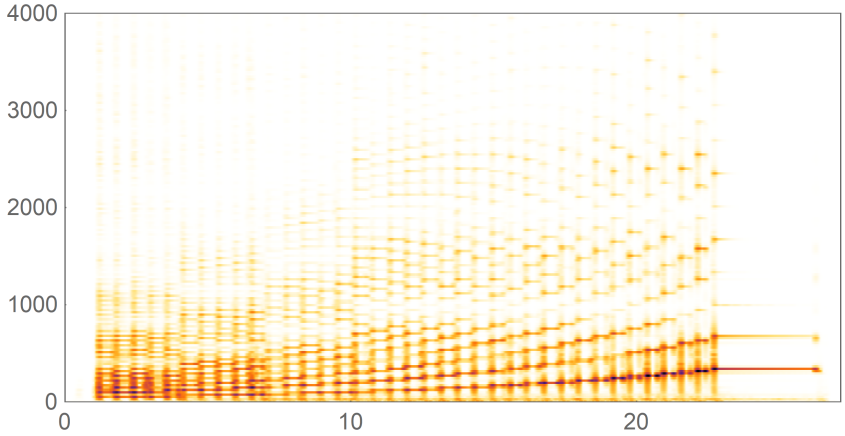
8.6.1 Detuning and popping artifacts

The Ocean algorithm suffers from known detuning artifacts that worsen with lower frequencies. The most notable comparison can be made when shifting the bass guitar sound by an octave down. All of the pitch shifters essentially fail here, not being able to transform the lower octave of the bass guitar without major detuning artifacts. Once the bass guitar gets out of the lower octave, the quality considerably increases (since the frequency is now twice as big, which means twice as many bins are available for one note). Except for the low quality configurations with analysis size 512, seen in Figure 8.2 b), the detuning artifacts are not particularly jarring from the second octave onwards.

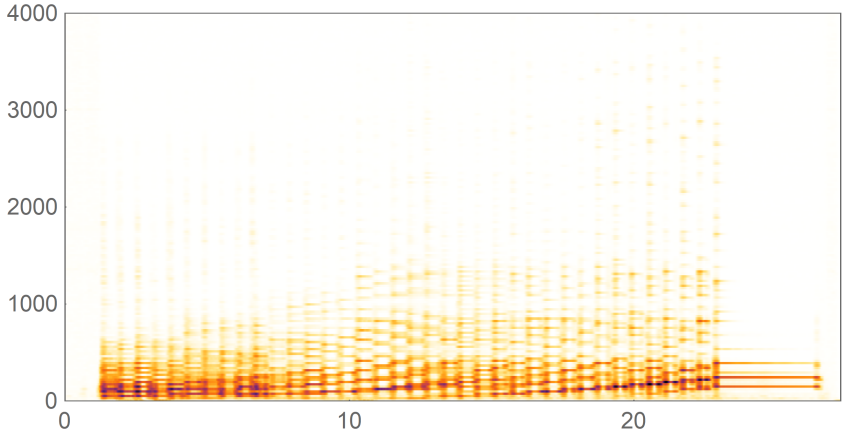
The configurations with enhanced synthesis sizes suffer from popping artifacts here. As seen in Figure 8.2 c), the enhanced synthesis size results in a larger number of similar frequencies appearing (consider the end of the spectrogram, where one note is held for an increased amount of time). These frequencies interact with each other in a way that reminds me of a popping sound. The Audacity implementation also suffers from occasional popping artifacts here, although as seen in Figure 8.3 d), this is not due to the interaction of similar frequencies.

The high quality setting of my pitch shifter implementation seen in Figure 8.3 e) is very similar to the commercial pitch shifter in Figure 8.3 f). I would not be surprised if the commercial shifter used a variation of an Ocean algorithm. The commercial pitch shifter has a quirk: it adds some harmonics to the pitch shifted signal. I am sure that I had the dry (passthrough) knob on the commercial pitch shifter set to negative infinity when recording. This makes me think the commercial pitch shifter always surreptiously mixes a bit of the dry signal and signal pitch shifted shifted an octave up (pitch multiplier 2) into the signal pitch shifted an octave down (pitch multiplier $\frac{1}{2}$) in order

a) Original, non-shifted signal



b) Low quality ($N_a = 512, N_s = 512, O = 32$)



c) Medium quality with enhanced synthesis size ($N_a = 1024, N_s = 2048, O = 16$)

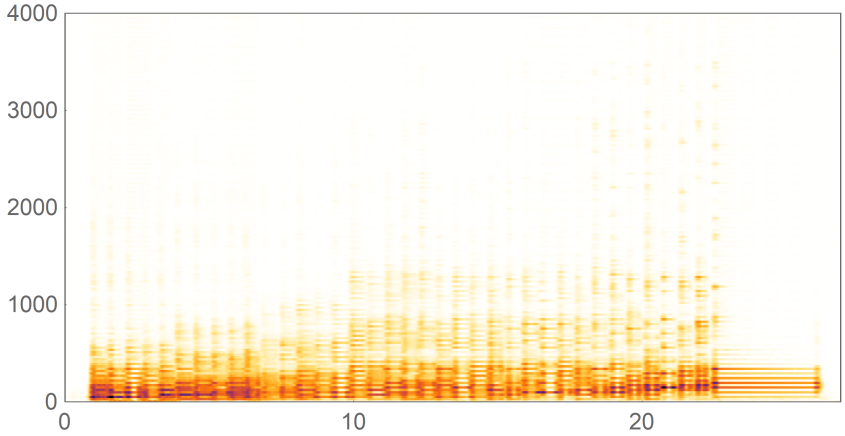
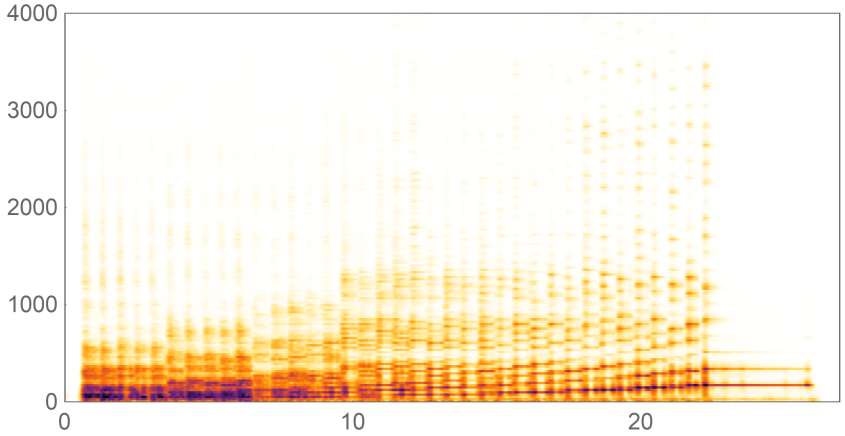
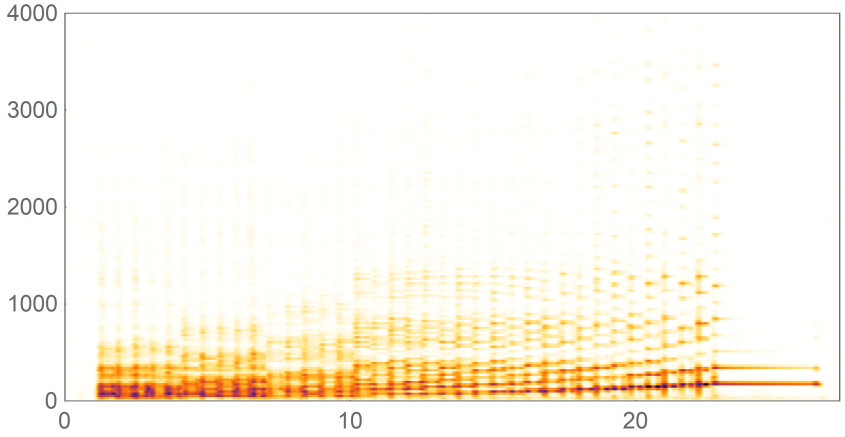


Figure 8.2: Spectrograms of bass guitar pitch shifted with multiplier $\frac{1}{2}$ The spectrograms are made with Hann windows, window size $N = 16384$ and overlap $O = 32$. This guarantees high frequency precision.

d) Audacity non-real-time pitch shifter



e) High quality ($N_a = 2048, N_s = 2048, O = 16$)



f) Commercial pitch shifter

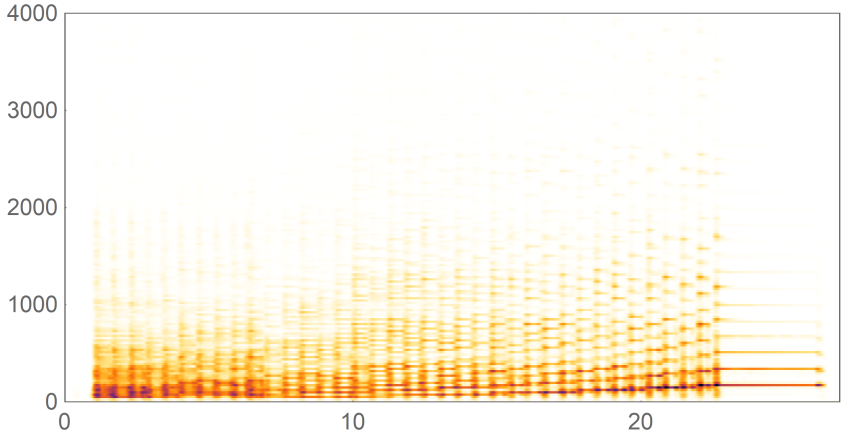


Figure 8.3: Spectrograms of bass guitar pitch shifted with multiplier $\frac{1}{2}$, continued

The spectrograms are made with with Hann windows, window size $N = 16384$ and overlap $O = 32$. This guarantees high frequency precision.

to strenghten the harmonic relationships so that the listener does not notice minor detuning errors. This, however, results in an organ-sounding sound, which is understandable considering that organ sounds are made by mixing sounds of different pipes with frequencies related by small ratios.

8.6.2 Amplitude modulation and oscillating noise artifacts

My implementation of the pitch shifter using demodulation by transformed unity signal demonstrates presence of some amplitude modulation artifacts, as is visible on the shifted sinusoid in Figure 8.4. Using pitch multipliers 2 and $\frac{3}{2}$, the sine wave is clearly demodulated.

It seems this also has an effect on the background noise, as demonstrated in Figure 8.5. Note that the amplitude modulation frequency is directly dependent on the analysis window overlap O. This is understandable because more analysis windows are added together using a higher overlap O and since the modulation is created in ridges between the window peaks, more ridges mean a higher modulation frequency.

While I did expect that increasing the FFT scalar precision from 16 bits to 32 bits would have a positive effect on lessening this artifact, I have run another test with the exact same settings as the 32-bit high quality configuration ($N_a = 2048, N_s = 2048, O = 16$) with only the precision changed to 16 bits and I have found no difference, either audible or visible on a spectrograph. It seems that increasing FFT scalar precision is not needed when using dynamic FFT scaling as the quality of the samples has already been limited to 16 bits by codec.

The reason why the amplitude modulation still persists even after demodulation (although it is lessened to somewhat passing levels) still eludes me. Perhaps the whole idea of computing amplitude demodulation cannot result in exact demodulation (it could be said that the authors of the Ocean algorithm point this out, although the used language is vague and unclear [11, ch. 3]), perhaps my implementation computes the demodulation levels in a wrong way (this theory would be suported by the authors of the Ocean algorithm computing demodulation exactly for pitch multiplier $\frac{3}{2}$ [11, ch. 3]). The only consolation is that the modulation artifacts do not result in the destruction of the sound, feeling more like an added effect.

8.7 Ocean algorithm and implementation evaluation

I am pleasantly surprised by the results I have achieved with my implementation of the Ocean algorithm. While there are amplitude modulation artifacts still present and popping artifacts appear for configurations where the synthesis size is larger than the analysis size, it is fully comparable to a free non-real-time pitch shifting impementation and even a commercially available pitch shifter. The performance of the chosen Blackfin BF548 even is adequate for stereophonic processing, a feature that is absent on the commercial pitch shifter.

I am somewhat confused by the fact that increasing synthesis size did not appear to provide noticeable benefits claimed by the Ocean algorithm authors. When pitch shifting, it results in separation of a single frequency into multiple ones instead of a better exact frequency approximation. I am not sure if I implemented synthesis windowing correctly, but I think that the only reasonable interpretation of synthesis windowing while retaining the analysis window latency is to use a synthesis window of the same size as the analysis window.

It should be said that the implemented algorithm does not provide any advancement over the compared pitch shifters. It is apparent to me that pitch shifting using the Fast Fourier transform has already reached its peak: the transform fundamentally does not allow lowering of its latency without demodulation artifacts. I am convinced that general filter banks are the most reasonable form of overcoming this limitation.

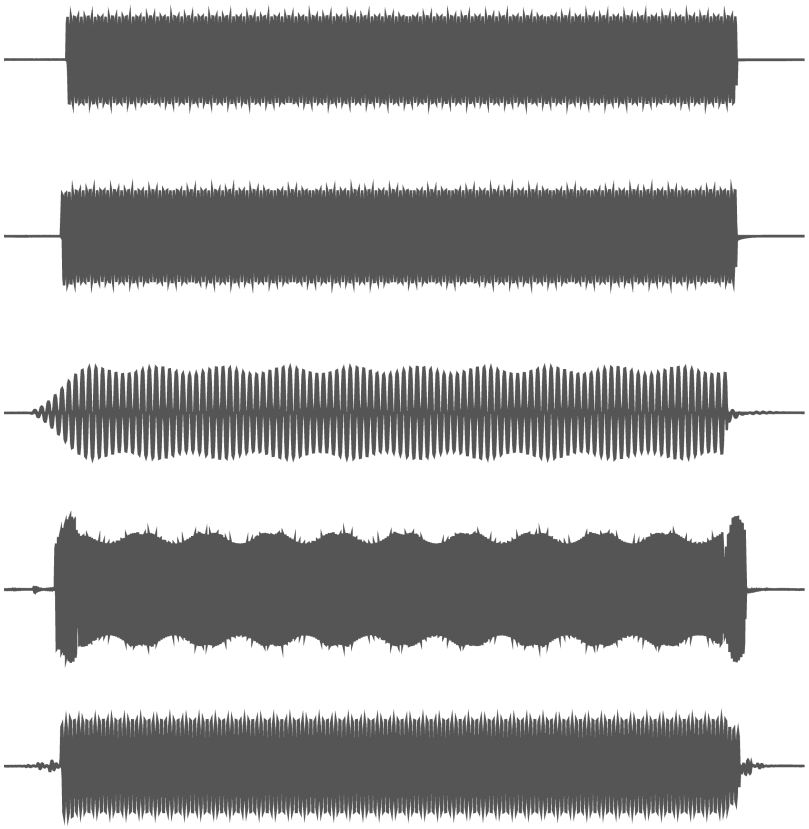


Figure 8.4: Pitch shifted sinusoid
The top waveform is the original, the waveforms below it are shifted with multiplier 1, $\frac{1}{2}$, $\frac{3}{2}$ and $\frac{11}{13}$, respectively. High quality shifting with $N_a = 2048, N_s = 2048, O = 16$ used.

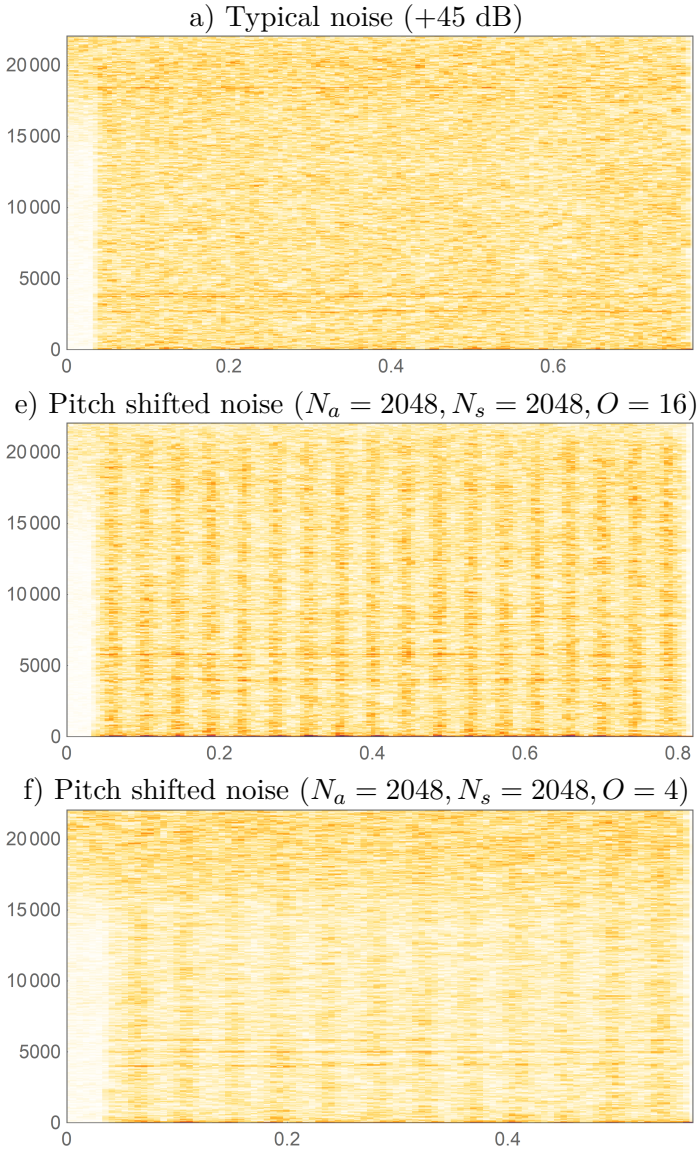


Figure 8.5: Noise pitch shifted with multiplier $\frac{3}{2}$ spectrograms
The spectrograms are made with with Hann windows, window sizewindow size $N = 1024$ and overlap $O = 4$. This guarantees high time precision.

Conclusion

My task was to implement a stereophonic audio pitch shifter on a digital signal processor BF548 EZ-KIT. To prepare for this task, I have researched known pitch shifting algorithms, undestanding of which requires some knowledge about principles of human hearing, music and sound processing. I have provided an introduction to these topics. After that, I have detailed various common techniques pitch shifting algorithms build on in chapter 3. I have analysed both the common and some novel algorithms, discussing their benefits and drawbacks. The Ocean algorithm seemed to be the best choice to me, having drawbacks but no critical flaws.

I have discussed the chosen platform and the tools and libraries it provides. Using them, I have implemented a pitch shifter that uses the Ocean algorithm. I have neatly decomposed the functionalities required and successfully implemented the algorithm despite unforeseen problems. For example, the audio codec on the kit does not function well when the program is also writing to the standard output, which is somewhat hard to debug when using the standard output to print debugging statements concerning this problem. I have found an error in the most important part of the Ocean algorithm, the phase shift equation, forcing me to derive its correct version by myself. I have enhanced the computation of amplitude demodulation in the Ocean algorithm,computing the values automatically for each multiplier, although I was not able to completely eliminate amplitude modulation artifacts.

I have tested the algorithm in and found the implementation to perform roughly as well as a commercial pitch shifter and a freeware non-realtime pitch shifter excluding the amplitude modulation artifacts. I have not been able to replicate the Ocean algorithm authors' claims that the synthesis size bigger than the analysis size helps with eliminating artifacts, with my experience being contrary.

While the amplitude modulation and popping artifacts might be resolvable by modifying the implementation or the used algorithm details not exactly specified in the original paper, have been convinced that improving the Ocean

algorithm latency or further reducing detuning artifacts without one having a negative effect on the other is essentially impossible as it is limited by the inherent properties the Discrete Fourier transform. The future work on pitch shifting should therefore focus on different algorithms. The Rollers algorithm seems to be a fine choice since it can overcome these limitations by using different filter banks, with the filters perhaps placed in a logarithmic relationship. Since the costs of processing fall rapidly, it could probably even be implemented on an embedded device.

Bibliography

[1] Lago, N.; Kon, F. The Quest for Low Latency. In *Proceedings of the International Computer Music Conference*, volume 30, International Computer Music Association, 2004, pp. 33–36.

[2] Gold, B.; Morgan, N.; et al. *Speech and Audio Signal Processing: Processing and Perception of Speech and Music*. New York: Wiley, second edition, 2011, ISBN 978-1-118-14289-9.

[3] Tashev, I. J. *Sound capture and processing: practical approaches*. New York: Wiley, 2009, ISBN 978-0-470-31983-3.

[4] Lyons, R. *Understanding Digital Signal Processing (3rd Edition)*. New Jersey: Prentice Hall, 2011, ISBN 013-7-02741-9.

[5] von Helmholtz, H. *On the sensations of tone as a physiological basis for the theory of music*. London, New York: Longmans, Green, and Co., third edition, 1895, no ISBN issued.

[6] Campbell, M.; Greated, C.; et al. *Musical Instruments: History, Technology, and Performance of Instruments of Western Music*. Oxford: Oxford University Press, 2004, ISBN 978-0-198-16504-0.

[7] Hall, M. What is the Gabor uncertainty principle? In *Agile* blog*, [online], 2004-01-15, retrieved 2018-04-20. Available from: `https://agilescientific.com/blog/2014/1/15/what-is-the-gabor-uncertainty-principle.html`

[8] Zolzer, U. *DAFX: Digital Audio Effects*. New York: Wiley Publishing, second edition, 2011, ISBN 978-0-470-66599-2.

[9] Juillerat, N.; Schubiger-Banz, S.; et al. Low latency audio pitch shifting in the time domain. In *2008 International Conference on Audio, Language and Image Processing*, July 2008, pp. 29–35, doi:10.1109/ICALIP.2008.4590019.

[10] Flanagan, J. L.; Golden, R. M. Phase vocoder. *The Bell System Technical Journal*, volume 45, no. 9, Nov 1966: pp. 1493–1509, ISSN 0005-8580, doi:10.1002/j.1538-7305.1966.tb01706.x.

[11] Juillerat, N.; Hirsbrunner, B. Low latency audio pitch shifting in the frequency domain. In *2010 International Conference on Audio, Language and Image Processing*, Nov 2010, pp. 16–24, doi:10.1109/ICALIP.2010.5685027.

[12] Hua, K. Reunderstand PSOLA. In *Kanru Hua's website*, [online], Dec 2015, retrieved 2018-05-14. Available from: `http://khua5.web.engr.illinois.edu/writings/reunderstand-psola.pdf`

[13] Analog Devices, Inc. *Blackfin® Processor Programming Reference*. Feb 2013, rev. 2.2, retrieved 2018-05-11. Available from: `http://www.analog.com/media/en/dsp-documentation/processor-manuals/Blackfin_pgr_rev2.2.pdf`

[14] Analog Devices, Inc. *ADSP-BF54x Blackfin Embedded Processors Data Sheet*. 2014, rev. E, retrieved 2018-05-14. Available from: `http://www.analog.com/media/en/technical-documentation/data-sheets/ADSP-BF542_BF544_BF547_BF548_BF549.pdf`

[15] Analog Devices, Inc. *ADSP-BF548 EZ-KIT® Lite Evaluation System Manual*. Jul 2012, rev. 1.4, retrieved 2018-05-11. Available from: `http://www.analog.com/media/en/dsp-documentation/evaluation-kit-manuals/ADSP-BF548_ezkit_man_rev.1.4.pdf`

[16] Analog Devices, Inc. *VisualDSP++® 5.0 Linker and Utilities Manual*. Jan 2011, rev. 3.5, retrieved 2018-05-11. Available from: `http://www.analog.com/media/en/dsp-documentation/software-manuals/50_lnkr_mn_rev_3.5.pdf`

[17] Analog Devices, Inc. *Getting Started with CrossCore® Embedded Studio 1.1.x*. Mar 2015, rev. 1, retrieved 2018-05-11. Available from: `http://www.analog.com/media/en/technical-documentation/application-notes/EE372v01.pdf`

[18] Analog Devices, Inc. *Device Drivers and System Services Manual for Blackfin® Processors*. Jan 2011, rev. 4.3, retrieved 2018-05-11. Available from: `http://www.analog.com/media/en/dsp-documentation/software-manuals/50_ddss_mn_rev_4.3.pdf`

[19] Analog Devices, Inc. *ADSP-BF54x Blackfin® Processor Hardware Reference*. Feb 2013, rev. 1.2, retrieved 2018-05-11. Available from: `http://www.analog.com/media/en/dsp-documentation/processor-manuals/ADSP-BF54x_hwr_rev1.2.pdf`

[20] Analog Devices, Inc. *VisualDSP++® 5.0 C/C++ Compiler and Library Manual for Blackfin® Processors*. Jan 2011, rev. 5.4, retrieved 2018-05-11. Available from: `http://www.analog.com/media/en/dsp-documentation/software-manuals/50_bf_cc_rtl_mn_rev_5.4.pdf`

[21] Analog Devices, Inc. *AD1980 AC '97 SoundMAX® Codec Data Sheet*. 2002, rev. 0, retrieved 2018-05-11. Available from: `http://www.analog.com/media/en/technical-documentation/data-sheets/AD1980.pdf`

Appendix A

Glossary

COLA Constant Overlap-Add criterion

DFT Discrete Fourier transform

FD-PSOLA Frequency-domain Pitch Sychronous Overlap-Add method

FFT Fast Fourier transform

STFT Short time Fourier transform

OLA Overlap-Add method

PSOLA Pitch Sychronous Overlap-Add method

TD-PSOLA Time-domain Pitch Sychronous Overlap-Add method

SOLA Synchronous Overlap-Add method

Appendix B

Contents of the enclosed microSD card

- `README.txt` ..... Readme detailing the various directories
- `blackfin` ..... Pitch shifter implementation
- `testing`
  - `sounds` ..... Testing sounds and their pitch shifted counterparts
    - `bass`
    - `piano`
    - `piano_stereo`
    - `sine`
    - `song`
    - `speech`
  - `stats` ..... Statistics of various pitch shifter configurations