Digital Sound Processing and Java

Documentation for the TarsosDSP[*]Audio Processing Library

Joren Six

IPEM[†], UGent
Sint-Pietersnieuwstraat 41, 9000 Ghent - Belgium
joren.six@ugent.be

May 19, 2015

The main goal of this text is to bridge the gap between a mathematical description of a digital signal process and a working implementation. The text starts with calculating sound buffers, then proceeds to illustrate audio output and explains the connection between the two. Then it proceeds with operations on sound. This text is meant to be accompanying releases of the TarsosDSP audio processing library: it should clarify the concepts used in the source code.

Contents

1 Introduction **3**
1.1 Developing TarsosDSP 4

2 Sampled Sound Using Java **5**
2.1 Audio buffers in Java 5
2.2 And Then There Was . . . Sound 8

3 Operations on Sound **10**
3.1 Audio Analysis - Sound Detection 11
3.2 Audio Effects - Echo 12
3.3 Synthesis - Sawtooth wave 13
3.4 Pitch Detection 13

[*]http://tarsos.0110.be/tag/TarsosDSP
[†]http://ipem.ugent.be

3.5 Time-Scale Modification in Time Domain 13
3.6 Percussion Detection 13
3.7 Filtering 13

1 Introduction

TarsosDSP is a Java library for audio processing. Its aim is to provide an easy-to-use interface to practical music processing algorithms implemented, as simply as possible, in pure Java and without any other external dependencies. This text serves two goals: it serves as a practical introduction into Music Information Retrieval (MIR) techniques for computer science students, and as documentation for the TarsosDSP library. The text assumes familiarity with object oriented programming constructs and a good knowledge of a programming language of the C-family, like Java. The concepts used should be transferable to other programming languages, or platforms as well.

TarsosDSP features implementation of the following algorithms. For each algorithm there is an example application[1] available.

- TarsosDSP was originally conceived as a library for **pitch estimation**, therefore it contains several pitch estimators: YIN [2], MPM [8], AMDF[10][2], and an estimator based on dynamic wavelets[6]. There are two YIN implementations, one remains within the comforts of the time domain, the other calculates convolution in frequency domain[3].
- Three **onset detectors** are provided. One described in [1], another is used by the BeatRoot system[3]. The algorithm described by [4] is also implemented.
- The WSOLA **time stretch algorithm**[13], which allows to alter the speed of an audio stream without altering the pitch is included. On moderate time stretch factors - 80%-120% of the original speed - only limited audible artifacts are noticeable.
- A **resampling** algorithm based on [12] and the related open source resample software package[4].
- A **pitch shifting** algorithm, which allows to change the pitch of audio without affecting speed, is formed by chaining the time-stretch algorithm with the resample algorithm.
- As examples of **audio effects**, TarsosDSP contains a delay and flanger effect. Both are implemented as minimalistic as possible.
- Several **IIR-filters** are included. A single pass and four stage low pass filter, a high pass filter, and a band pass filter.
- TarsosDSP also allows **audio synthesis** and includes generators for sine waves and noise. Also included is a Low Frequency Oscillator (LFO) to control the amplitude of the resulting audio.

[1] `http://tarsos.0110.be/tag/TarsosDSP`
[2] Partial implementation provided by Eder Souza
[3] The YIN FFT implementation was kindly contributed by Matthias Mauch
[4] Resample 1.8.1 can be found on the digital audio resampling home page `https://ccrma.stanford.edu/~jos/resample/`, maintained by Julius O. Smith

1.1 Developing TarsosDSP

If you are reading this text your are probably interested in the source code of TarsosDSP, either for extending the library or to use the source code as a reference. The project is hosted on GitHub[5]. Contributions to TarsosDSP are more than welcome, if you have a an algorithm to add or find a bug, do not hesitate to send me a message[6].

TarsosDSP uses the Apache Aant[7] build system. The instructions below detail how you can build from source. When everything runs correctly you should be able to run all example applications and have the latest version of the TarsosDSP library for inclusion in your projects. Also the Javadoc documentation for the API should be available.

The libraries source code is separated in several folders:

- `src` contains the source files of TarsosDSP libraries.
  - `src/core` contains the main algorithms and core functionality
  - `src/examples` this directory contain example applications, most with Java Swing user interface.
  - `src/jvm` runtime specific audio input/output functions for the Java Virtual Machine can be found here.
  - `src/android` runtime specific audio input/output functions for the Android runtime (Dalvik, ART) can be found here.
  - `src/patcher` runtime specific audio input/output functions for patcher environments like Pure Data or Max MSP can be found here.
  - `src/test` contains unit tests for some of the DSP functionality.
- `doc` contains help documentation for DSP library
- `lib` contains libraries that are needed to run unit tests and to run TarosDSP in a patcher environment like MaxMSP or Pure Data.
- `build` contains ANT build files. Either to build Java documentation or runnable JAR-files for the example applications.

To make development with eclipse easy, first exclude the `src` folder as source folder and then include the subdirectories `src` as source folders.

[5] https://github.com/JorenSix/TarsosDSP
[6] mailto:joren.six@ugent.be
[7] http://ant.apache.org/

2 Sampled Sound Using Java

To process sound digitally some kind of conversion is needed from an analog to a digital sound signal. This conversion is done by an ADC: an analog to digital converter. An ADC has many intricate properties, making sure no information is lost during the conversion. For the principle of audio processing the most important ones are the *sampling rate* and *bit depth* [9].

The sampling rate measures samples per second, it is defined in Hertz (Hz). The *Nyquist-Shannon sampling theorem*[11] states that you need to sample at twice the maximum frequency of the information you want to convey. If lower sampling rates are used, part of the information is lost. Speech is contained in the frequency range from 30Hz to 3000Hz. Applying Nyquist-Shannon, a sampling rate of 6kHz should be enough. Some telephone systems use 8kHz, which is perfect for speech signals.

The human ear is capable of detecting sounds between about 20Hz and 20kHz, depending from person to person. Sampling musical signals at about twice the maximum hearing frequency makes sense. 44100Hz, 48000Hz are common sampling rates for musical information ($20kHz \times 2 < 44.1kHz$). [8]

The bit depth is the number of bits used to represent the value of a sample. Using signed integers of 16 bits is common practice. The following example shows how these concepts translate to the Java programming language. Bit depths of 24 bits or more, and sampling rates of 96kHz or 192kHz very rarely offer any benefits for music consumers[9]. There is, however, an advantage for music producers to use these formats: while mixing, mending and molding music a larger dynamic range (bit depth) and sample rate prevent quality loss.

2.1 Audio buffers in Java

In this section the theoretical concepts of sampling rate and bit depth are translated to practical programming constructs. This is done by using one of the most simple audio signals, sine wave. A sine wave is characterized by a frequency f and an amplitude a. Equation 1 shows a mathematical representation of this simple audio signal.

$$f(x) = a \times \sin(2\pi f) \tag{1}$$

The sine wave is depicted in Figure 1. The continuity of the signal, theoretically there are an infinite amount of points between $[0, 2\pi]$, makes it impossible to represent within a digital system. The signal needs to be sampled.

A sampled sine wave, useful for signal processing can be seen in Figure 2. Conceptually, sampling a signal means assigning each of the infinite amount of points to a finite number of bins in a grid. The sampling rate defines the horizontal granularity of the grid, the bit depth defines the vertical granularity. A sample rate of 44.1kHz and a bit depth of 16bits creates a fine enough grid for music.

[8] For the more visually inclined, these concepts are also explained in *A Digital Media Primer for Geeks* by Monty of the Xiph foundation: `https://www.xiph.org/video/vid1.shtml`

[9] See e.g. `http://people.xiph.org/~xiphmont/demo/neil-young.html`

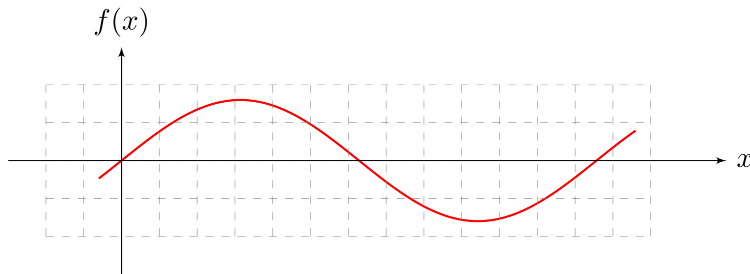

Figure 1: Continuous sine wave

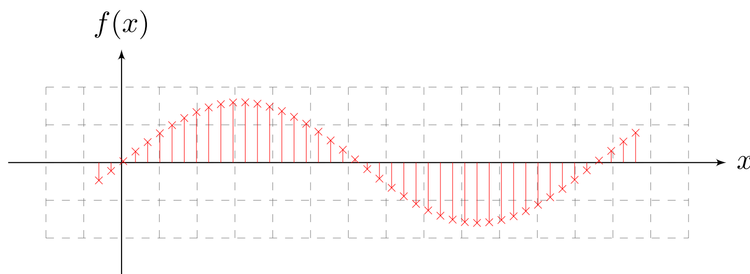

Figure 2: Sampled sine wave

The listing 6 shows how equation 1 translates to source code. The first four lines define the properties of the sampled wave, the frequency, amplitude, sample rate, and duration. For performance reasons $2\pi f$ is calculated before the loop starts, on line 5. Line six initializes the buffer, where the audio information is stored. The for loop (lines 7 - 10) constuct is used to calculate the value of every sample.

Listing 1: A sampled sine wave

```
1 double sampleRate = 44100.0;
  double frequency = 440.0;
  double amplitude = 0.8;
  double seconds = 2.0;
  double twoPiF = 2 * Math.PI * frequency;
6 float[] buffer = new float[(int) (seconds * sampleRate)];
  for (int sample = 0; sample < buffer.length; sample++) {
     double time = sample / sampleRate;
     buffer[sample] = (float) amplitude * Math.sin(twoPiF * time);
  }
```

After executing the code in Listing 6 the buffer contains a two seconds long pure tone of 440Hz, sampled at 44.1kHz. 440Hz is chosen because it is used as the general tuning

standard for musical pitch. On line 9, each sample is calculated using the `Math.sin` function and is converted to a `float` following the advice found in the Java Language Basics Tutorial[10]:

> *It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency.*

Pure sine tones are not commonly found in the wild. Most sounds in the real world have a rich harmonic spectrum. The following example shows a sound with one harmonic at 6 times the base frequency. The mathematical representation can be seen in equation 2. The example could be expanded with more harmonics. If you choose the harmonics carefully, a realistic sustained flute sound could be synthesized, only by adding more harmonics.

$$f(x) = 0.8 \times \sin(2\pi f) + 0.2 \times sin(2\pi 6f) \tag{2}$$

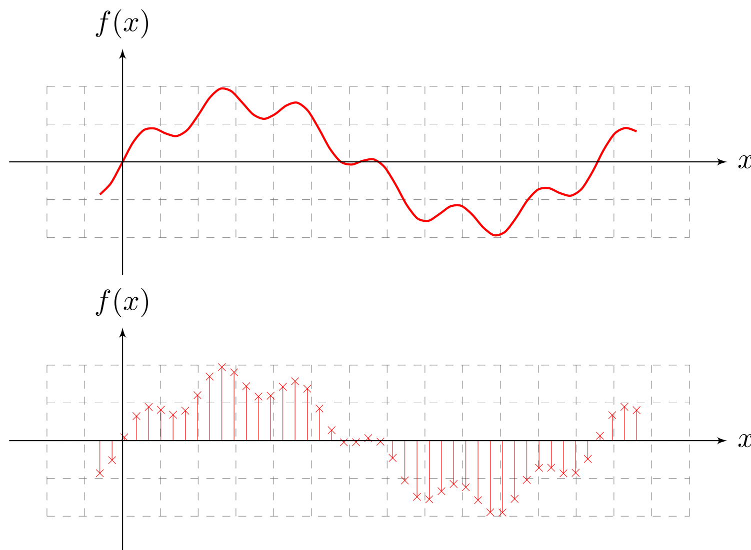

Figure 3: A continuous and discrete complex wave representing equation 2.

Figure 5 shows continuous and sampled versions of the wave represented by equation 2. Interesting to note is the visual superposition of the harmonic onto the wave with the base frequency. This superposition is the result of the addition in equation 2. Creating a buffer with this information:

[10] `http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html`

Listing 2: A complex wave buffer

```
double sampleRate = 44100.0;
double seconds = 2.0;

double f0 = 440.0;
double amplitude0 = 0.8;
double twoPiF0 = 2 * Math.PI * f0;

double f1 = 6 * f0;
double amplitude1 = 0.2;
double twoPiF1 = 2 * Math.PI * f1;

float[] buffer = new float[(int) (seconds * sampleRate)];
for (int sample = 0; sample < buffer.length; sample++) {
    double time = sample / sampleRate;
    double f0Component = amplitude0 * Math.sin(twoPiF0 * time);
    double f1Component = amplitude1 * Math.sin(twoPiF1 * time);
    buffer[sample] = (float) (f0Component + f1Component);
}
```

Line 17 of listing 2 shows how the two frequency components are being summed up, resulting in a buffer with the correct audio information. The next step is to convert the `float` buffer into pulse code modulated (PCM) bytes, which are understood by e.g. a sound card.

2.2 And Then There Was . . . Sound

A conversion step is needed before the sound can be heard. The `float` buffer contains numbers in the range $[-1.0, 1.0]$. Those need to be mapped to e.g. 16bit signed little endian PCM. This can be done by multiplying with $\lfloor(2^{16}-1)/2\rfloor = 32767$. Every sample is then 16 bits or two bytes, each sample is converted to two bytes. In Java lingo: the `byte` array needs to be twice the length of the `float` array.

Listing 3: Converting floats to bytes

```
final byte[] byteBuffer = new byte[buffer.length * 2];
int bufferIndex = 0;
for (int i = 0; i < byteBuffer.length; i++) {
    final int x = (int) (buffer[bufferIndex++] * 32767.0);
    byteBuffer[i] = (byte) x;
    i++;
    byteBuffer[i] = (byte) (x >>> 8);
}
```

To make the sound audible it can be written to a WAVE file. A WAVE file consists of a header followed by the sound data. The sound data is nothing more or less than the PCM format we calculated. The WAVE file header[11] format stems from the time that

[11] More information can be found on this webpage:
http://www-mmsp.ece.mcgill.ca/documents/audioformats/wave/wave.html

Microsoft and IBM were still best friends, it is defined in a joint specification[5]. Writing headers is a bit boring luckily there are a few utility classes available in the standard Java library in the `javax.sound.sampled` package which make this task effortless:

Listing 4: Writing a WAV-file

```
File out = new File("out.wav");
boolean bigEndian = false;
boolean signed = true;
int bits = 16;
int channels = 1;
AudioFormat format;
format = new AudioFormat(sampleRate, bits, channels, signed, bigEndian);
ByteArrayInputStream bais = new ByteArrayInputStream(byteBuffer);
AudioInputStream audioInputStream;
audioInputStream = new AudioInputStream(bais, format,buffer.length);
AudioSystem.write(audioInputStream, AudioFileFormat.Type.WAVE, out);
audioInputStream.close();
```

Once the WAVE file is stored to disc you can listen to it using about any media player. This is a bit of a drag so another option is to send the sound to the speakers directly. To get this working you need a, for the Java subsystem, correctly configured default sound card[12].

Listing 5: Play a buffer

```
SourceDataLine line;
DataLine.Info info;
info = new DataLine.Info(SourceDataLine.class, format);
line = (SourceDataLine) AudioSystem.getLine(info);
line.open(format);
line.start();
line.write(byteBuffer, 0, byteBuffer.length);
line.close();
```

Within TarsosDSP conversion from bytes to samples is done automatically, but it remains important to know what happens behind the scenes e.g. to look for performance bottlenecks. Once the basic conversions are covered, the next step is to perform operations on sound.

[12]By default the Java Runtime provided by Oracle or Sun does not play nice with PulseAudio on Linux. To alliviate this problem see the tutorial here: http://tarsos.0110.be/artikels/lees/PulseAudio_Support_for_Sun_Java_6_on_Ubuntu

3 Operations on Sound

Operations on sound are commonly done in blocks. Operations on individual samples are most of the time not efficient nor practical. E.g. if you would want to estimate the frequency of audio you would need at least one complete period of the signal, surely more than one sample.

For most standard operations, like filtering or an echo, a block size of 1024[13] samples is common. With the default sample rate of $44.1kHz$ this means that each operation is done on a block of audio of $1024/44.1kHz = 23.21ms$. This block size provides a practical trade-off between computational performance, update speed and usability. Depending on the operation, larger block sizes might add a too large delay, smaller ones might not provide enough audio-information to be able to perform the wanted operation.

Chaining operations is also common. An architecture that allows a chain of arbitrary operations on audio blocks of arbitrary size results in a flexible processing pipeline. These concepts are implemented in the TarsosDSP audio library. Figure 4 shows the processing pipeline. Currently, only single channel audio is allowed, which helps to makes the processing chain extremely straightforward[14]. A schematic representation can be found in Figure 4. The source of the audio is a file, a microphone, or an empty stream. The `AudioDispatcher` chops incoming audio in blocks of a requested number of samples, with a defined overlap. Subsequently the blocks of audio are scaled to a `float` in the range `[-1,1]`, as mentioned in Section 2.1. The blocks are encapsulated in an `AudioEvent` object which contains a pointer to the audio, the start time in seconds, and has some auxiliary methods, e.g. to calculate the energy of the audio block.

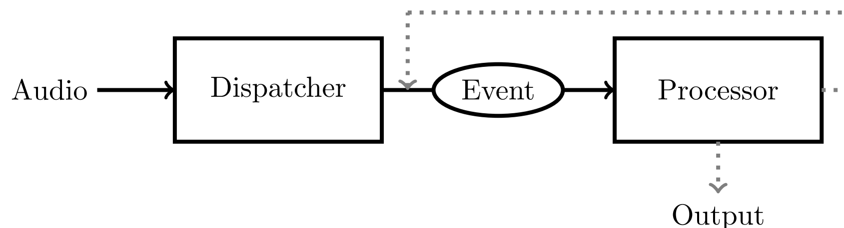

Figure 4: A schematic representation of the TarsosDSP processing pipeline. The incoming audio (left) is divided into blocks. Each block is encapsulated in an `Event` object by the `Dispatcher`. The event objects flow through one or more `Processor` blocks, which may or may not alter the audio and can generate output (e.g. pitch estimations). Dotted lines represent optional flows.

The `AudioDispatcher` sends the `AudioEvent` through a series of `AudioProcessor` objects which execute an operation on audio. The core of the algorithms are contained

[13] The size of a block is often a power of two, mainly because this eases implementation of several algorithms, either in hardware or software. E.g. the FFT algorithm is easy to implement with blocks the size of a power of two. In TarsosDSP the FFT algorithm allows other block sizes as well.

[14] Actually multichannel audio is accepted as well, but it is automatically downmixed to one channel before it is send through the processing pipeline

in these `AudioProcessor` Note that the size of a block of audio can change during the processing flow. This architecture supports three types of operations on sound:

Analysis
: During sound analysis a certain feature of audio is extracted. The sound itself is not altered. Examples of analysis operations are estimating pitch, calculating loudness (see section 3.1), and detecting onsets. Some analysis operations need only one block of audio, like estimating pitch. Other operations compare the current block of audio with a few previous blocks, this is done in some onset detection methods. A third kind of audio analysis operations can not operate in real-time, since they need information on previous and future blocks of audio. For example, reliable beat tracking can only be done by post-processing onsets.

Audio Effects
: An operation is called an effect when it uses an audio stream as input and alters it in a certain way. Examples of audio effects include flanger, filters, pitch shifting, phaser, echo (see section 3.2), and chorus. Time stretching, altering the speed of audio without affecting its pitch, is a special kind of audio effect since it changes the length of the audio (block), while the others do not.

Synthesis
: When audio is generated using only a mathematical construct, or prerecorded samples, an audio synthesis operation takes place. The main difference with an effect is that it is not dependent on audio input. Generating random noise is one of the more simple synthesis operations.

The following examples are excerpts from the library and illustrate how each operation type is implemented within TarsosDSP.

3.1 Audio Analysis - Sound Detection

This section describes how to implement a program that reacts to the presence of sound: when the sound level reaches a certain threshold the algorithm sends a notification. This functionality can e.g. be used to implement a burglar alarm system.

A sound detection algorithm has to calculate the energy of the signal. Audio signal energy is commonly expressed in decibel sound pressure level (dBSPL), a logarithmic unit. Since the human ear has a large dynamic range[15] a logarithmic unit is practical. To calculate the dBSPL level of a buffer b of length n use the following formula:

$$20 \log_{10} \frac{\sqrt{\sum_{i=0}^{n} b[i]}}{n}$$

[15]The ratio of the quietest sound the ear can hear and the loudest the ear can bear is about 10^{12}.

For this applications the size of the buffer does not matter that much. The size should span some time to make the measurement more meaningful but if the buffer is too large the response time of the algorithm suffers, e.g. a buffer of 10240 samples gives a minimal delay of $232ms$ at $44100Hz$. For this application buffers from 512 to 4192 samples make sense (causing a delay from 12 to 95ms).

The flow of the program is straightforward. Each block of audio is analyzed and a decibel value is calculated. If the value reaches a certain threshold sound is present, otherwise silence is assumed. With the TarsosDSP library this can be implemented as follows:

Listing 6: Detecting sound or silence

```
// create a new dispatcher
AudioDispatcher dispatcher;
dispatcher= AudioDispatcher.fromDefaultMicrophone(1024, 0);
dispatcher.addAudioProcessor(new AudioProcessor() {
  float threshold = -70;//dB
  @Override
  public boolean process(AudioEvent audioEvent) {
    float[] buffer = audioEvent.getFloatBuffer();
    double level = soundPressureLevel(buffer);
    if(level > threshold){
      System.out.println("Sound detected.");
    }
    return true;
  }

  @Override
  public void processingFinished() {}

  /**
   * Returns the dBSPL for a buffer.
   */
  private double soundPressureLevel(final float[] buffer) {
    double power = 0.0D;
    for (float element : buffer) {
      power += element * element;
    }
    double value = Math.pow(power, 0.5)/ buffer.length;;
    return 20.0 * Math.log10(value);
  }
});
```

3.2 Audio Effects - Echo

As an example of a simple audio processing operation an echo effect, a delay, is implemented. The idea of this section is, next to showing how an echo effect works, to explain audio manipulation by processing blocks.

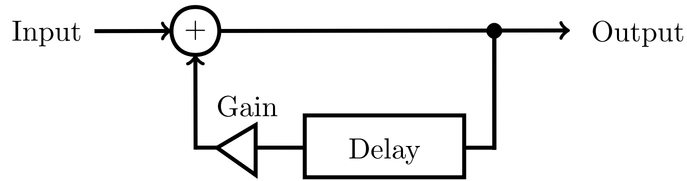

Figure 5: Block diagram representing a delay audio effect. The input is mixed with delayed and scaled output.

3.3 Synthesis - Sawtooth wave

3.4 Pitch Detection

TarsosDSP implements several pitch detection methods. YIN [2], the McLeod Pitch Method[7] and the Dynamic Wavelet pitch estimation algorithm [6]

3.5 Time-Scale Modification in Time Domain

TarsosDSP contains an implementation of the time stretching algorithm described in [13], it can playback audio quicker or slower without affecting changing pitch. Slow playback is e.g. very practical to transcribe the melody of a song.

3.6 Percussion Detection

The onset detector implementation is based on a VAMP plugin example by Chris Cannam at Queen Mary University, London. The method is described in [1].

3.7 Filtering

In the `be.hogent.tarsos.dsp.filters` package several frequency filters can be found. With a high pass filter, audio with frequencies above a certain threshold are kept. A low pass filter does the reverse, audio with frequencies below a threshold is kept. Together they can create a band pass filter which can e.g. be constructed to focus on the melodic range of a song and ignore the rest.

This document is a work in progress, for more information see the source code on `https://github.com/JorenSix/TarsosDSP` ;).

References

[1] Dan Barry, Derry Fitzgerald, Eugene Coyle, and Bob Lawlor. Drum Source Separation using Percussive Feature Detection and Spectral Modulation. In *Proceedings of the Irish Signals and Systems Conference (ISSC 2005)*, 2005.

[2] Alain de Cheveigné and Kawahara Hideki. YIN, a Fundamental Frequency Estimator for Speech and Music. *The Journal of the Acoustical Society of America*, 111(4):1917–1930, 2002.

[3] Simon Dixon. Automatic Extraction of Tempo and Beat From Expressive Performances. *Journal of New Music Research (JNMR)*, 30(1):39–58, 2001.

[4] Chris Duxbury, Juan Pablo Bello, Mike Davies, and Mark Sandler. Complex domain onset detection for musical signals. In *In Proc. Digital Audio Effects Workshop (DAFx)*, 2003.

[5] IBM and Microsoft. *Multimedia Programming Interface and Data Specifications 1.0.* Microsoft Press, 1991.

[6] Eric Larson and Ross Maddox. Real-Time Time-Domain Pitch Tracking Using Wavelets. 2005.

[7] Philip McLeod. *Fast, Accurate Pitch Detection Tools for Music Analysis.* PhD thesis, University of Otago. Department of Computer Science, 2009.

[8] Phillip McLeod and Geoff Wyvill. A Smarter Way to Find Pitch. In *Proceedings of the International Computer Music Conference (ICMC 2005)*, 2005.

[9] Ken C. Pohlmann. *Principles of Digital Audio / Ken C. Pohlmann.* Sams, Indianapolis :, 2nd. ed. edition, 1989.

[10] M. J. Ross, H. L. Shaffer, A. Cohen, R. Freudberg, and H. J. Manley. Average Magnitude Difference Function Pitch Extractor. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 22(5):353–362, October 1974.

[11] C. E. Shannon. Communication in the Presence of Noise. *Proceedings of the IRE*, 37(1):10–21, January 1949.

[12] Julius O. Smith and Phil Gosset. A Flexible Sampling-Rate Conversion Method. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP 1984)*, volume 2, 1984.

[13] Werner Verhelst and Marc Roelands. An Overlap-Add Technique Based on Waveform Similarity (WSOLA) for High Quality Time-Scale Modification of Speech. In *IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP 1993)*, pages 554–557, 1993.