Digital Sound Processing and Java

Documentation for the TarsosDSP[*]Audio Processing Library

Joren Six

University College Ghent, Faculty of Music
Hoogpoort 64, 9000 Ghent - Belgium
joren.six@hogent.be

November 27, 2012

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. Along the way the format of WAV-files are explained. 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.

In short this text should at least get you starting with audio DSP using Java.

Contents

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

1 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* [6].

The sampling rate measures samples per second, it is defined in Hertz (Hz). The *Nyquist-Shannon sampling theorem*[7] 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.

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 ($20kHz \times 2 < 44.1kHz$).

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.

1.1 Audio buffers in Java

$$f(x) = 0.8 \times \sin(2\pi f) \tag{1}$$

One of the most simple audio signals is a sine wave. This is also known as a pure tone. A pure tone is characterized by a frequency f and an amplitude. A sine wave (equation 1) is depicted in Figure 1.

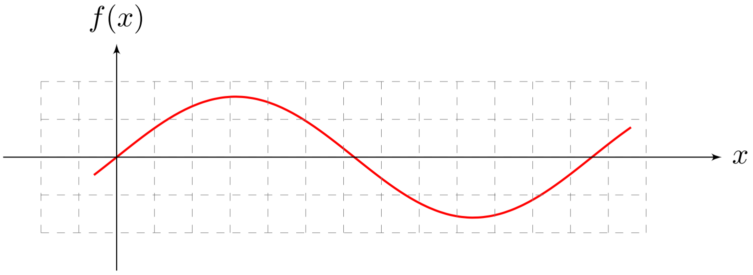

Figure 1: Continuous sine wave

To use the sine wave for signal processing it needs to be sampled. On Figure 2 the sampling rate defines the horizontal granularity, the bit depth defines the vertical

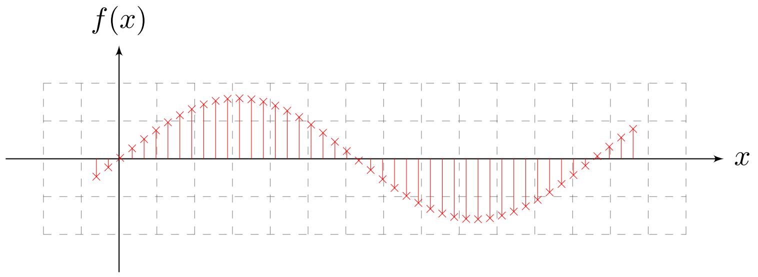

Figure 2: Sampled sine wave

granularity. How to create an array containing a sampled sine wave using Java can be seen in Listing 6.

Listing 1: A sampled sine wave

```
double sampleRate = 44100.0;
double frequency = 440.0;
double amplitude = 0.8;
double seconds = 2.0;
double twoPiF = 2 * Math.PI * frequency;
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. Each sample is calculated using the `Math.sin` function and is converted to a `float` following the advice found on `http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html`:

> *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 tones are not commonly found in the wild. A more realistic sound can be generated by using equation 2. This sound consists of a base frequency and a harmonic at 6 times the base frequency.

$$f(x) = 0.8 \times \sin(x) + 0.2 \times sin(6x) \qquad (2)$$

Creating a buffer with this information:

Listing 2: A complex wave buffer

```
double sampleRate = 44100.0;
```

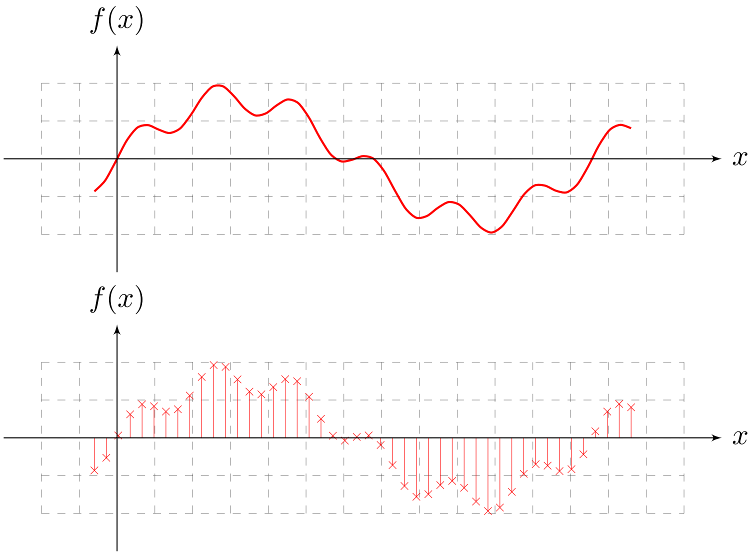

Figure 3: A continuous and discrete complex wave representing equation 2.

```
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);
}
```

1.2 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

```java
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[1] format stems from the time that Microsoft and IBM were still best friends, it is defined in a joint specification[3]. 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

```java
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[2].

Listing 5: Play a buffer

```java
SourceDataLine line;
DataLine.Info info;
info = new DataLine.Info(SourceDataLine.class, format);
line = (SourceDataLine) AudioSystem.getLine(info);
line.open(format);
line.start();
```

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

[2] 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

```
line.write(byteBuffer, 0, byteBuffer.length);
line.close();
```

With the basics covered we can continue with operations on sound.

2 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 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. The following examples are excerpts from the library and illustrate some of those basic ideas. You should be able to translate the concepts to another platform or environment.

2.1 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[3] a logarithmic unit is practical. To calculate the dBSPL level of a buffer b of length n use the following formula:

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

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).

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

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);
  }
});
```

2.2 Echo Effect

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.

2.3 Pitch Detection

TarsosDSP implements several pitch detection methods. YIN [2], the McLeod Pitch Method[5] and the Dynamic Wavelet pitch estimation algorithm [4]

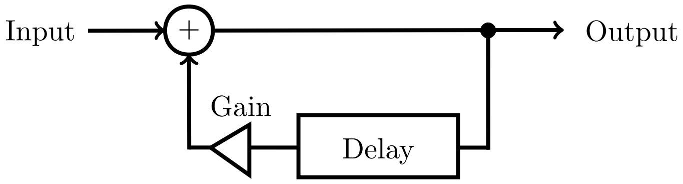

Figure 4: Block diagram representing a delay audio effect. The input is mixed with delayed and scaled output.

2.4 Time-Scale Modification in Time Domain

TarsosDSP contains an implementation of the time stretching algorithm described in [8], 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.

2.5 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].

2.6 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.

3 Utility functions

3.1 Write a WAV-file

3.2 Audio Playback

3.3 Interrupt a loop

3.4 Fourier Analysis

The FFT implementation used within TarsosDSP is by Piotr Wendykier and is included in his JTransforms library. JTransforms is the first, open source, multithreaded FFT library written in pure Java.

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

3.5 Audio Visualisation or: How I Learned to Stop Worrying and Love the DataLine Object

4 Pitch Detection

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

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

[5] Philip McLeod. *Fast, accurate pitch detection tools for music analysis.* PhD thesis, University of Otago. Department of Computer Science, 2009.

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

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

[8] Werner Verhelst and Marc Roelands. An overlap-add technique based on waveform similarity (wsola) for high quality time-scale modification of speech. In *proceedings of ICASSP-93*, pages 554–557, 1993.