Sound and Java

joren.six@hogent.be

March 28, 2011

Abstract

The main goal of this text is to bridge the gap between a mathematical description of a digital signal process and a working implementation. This text is a work in progress and for the moment it starts with calculating sound buffers by hand, proceeds with audio output and explains the connection between the two. Along the way it also explains how a WAV-file looks.

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

Contents

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* [1].

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

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

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.

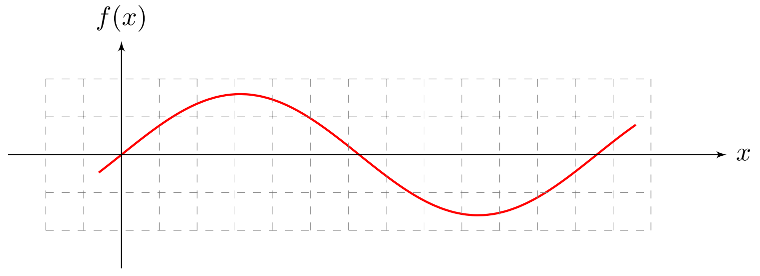

Figure 1: Continuous sine wave

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

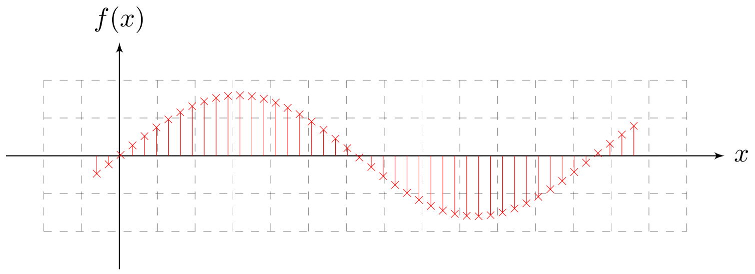

Figure 2: Sampled sine wave

vertical granularity. How to create an array containing a sampled sine wave using Java can be seen in Listing 1.

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 1 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.

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

Creating a buffer with this information:

Listing 2: A complex wave buffer

```
double sampleRate = 44100.0;
```

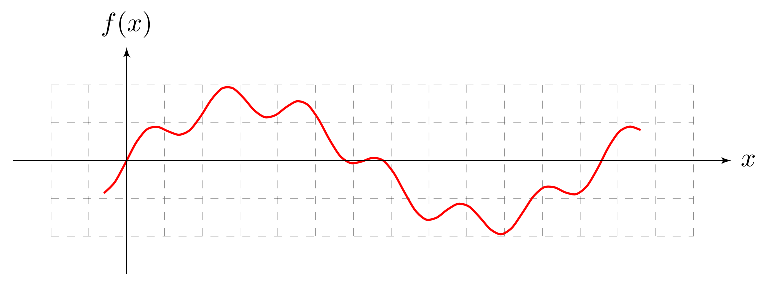

Figure 3: Continuous complex wave

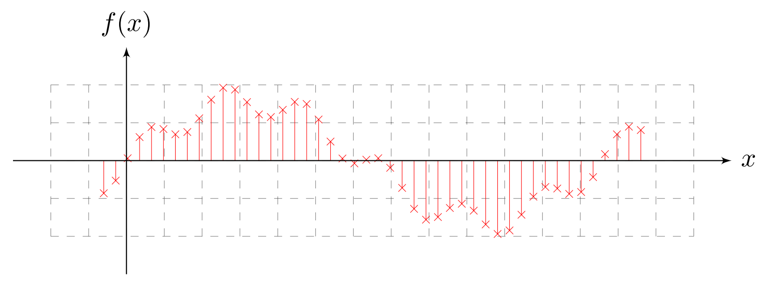

Figure 4: Sampled complex wave

```
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 heared. 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 WAV file. A WAV file consists of a header followed by the sound data. In Java the header can be written using classes in `javax.sound.sampled`:

Listing 4: Writing a WAV-file

```java
File out = new File(         );
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();
```

Another option is to send it to the speakers directly.

Listing 5: Play a buffer

```java
SourceDataLine line;
DataLine.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();
```

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

1.3 Operations on Sound

Operations on sound are commonly done in blocks. Operations on individual samples are not efficient nor practical. E.g. if you would want to estimate the frequency of a signal you would need a couple of periods, surely more than one sample.

Chaining operations is also common. An architecture that allows a chain of arbitrary operations on audio results in a flexible . This will be shown in the following examples.

1.3.1 Silence Detection

See the source code on `https://github.com/JorenSix/TarsosDSP` ;).

2 References

References

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