JOURNAL OF OBJECT TECHNOLOGY

Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2008**

Vol. 7, No. 8, November-December 2008

The μ-law CODEC

Douglas Lyon, Ph.D.

Abstract

This paper describes how to use a CODEC (COder-DECoder) to encode, decode, synthesize and play AU-format audio data. The AU format use a μ-law (pronounced mu-law) compression technique improving dynamic range over the linear encoding of audio. The μ-law CODEC dates from 1965, yet is still in common use today.

1 BACKGROUND

The standard μ-law format consists of logarithmically companded, 8 kHz sample rate, byte-quantized, voice-grade audio. The word *compandor* is a contraction of "compressor" and "expander" [BTL]. The sample time for an 8000 samples per second system is 1/8000 = 0.000125 second = 125 μs. Such a format generates an 8000 sample/second * 8 bits / sample = 64 kbps data stream and is common in telephony. Also, the peak bandwidth of such a compression format is

$$Bandwidth = \frac{Sampling\ Rate}{2} = 4khz\,. \qquad (0.1)$$

The International Telecommunication Union (ITU) formerly CCITT, has created a specification called G.711. There are two PCM (Pulse Code Modulation) algorithms defined within the G.711 standard, "A-Law" and μ-law;. In both the "A-Law" and μ-law format, the sample rate is 8 kHz. In a linear PCM system there are uniform voltage quantization steps [Bates].

A copy of the G.711 specification is available at http://www.itu.int/rec/T-REC-G.711/en. The μ-law encoding formula is given by:

$$y = F(x) = sign(x)V_{\max}\frac{\ln\left(1+\frac{\mu x}{V_{\max}}\right)}{\ln(1+\mu)}$$

where (0.2)

$$-V_{\max} \le x \le V_{\max}$$

The "A-law formula is given by:

$$y = \frac{A}{1+\ln A}\left(\frac{x}{V_{\max}}\right), \left|\frac{x}{V_{\max}}\right| < \frac{1}{A}$$

and (0.3).

$$y = \frac{sign(x)}{1+\ln A}\left(1+\ln\left(\frac{Ax}{V_{\max}}\right)\right), \frac{1}{A} \le \left|\frac{x}{V_{\max}}\right| \le 1$$

Typical values of the compression parameters used in (0.2) and (0.3) are:

$$\mu = 100 \text{ and } 255$$
$$A = 87.6 \quad (0.4)$$

A graph of (0.3), with $V_{\max} = 1$, and $\mu = 255$ is shown in Figure 1. The value of μ=255 is used for North American telephone transmission. The value of A=87.6 is used for European telephone transmission. The rationale for companding is that the human ear has an inability to differentiate between amplitudes of sound waves as the amplitude increases [Embree].

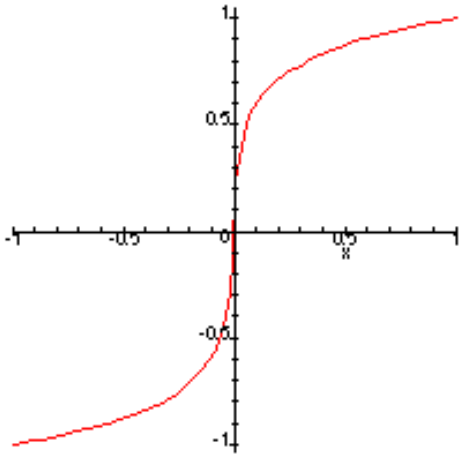


Figure 1. Graph of the μ-law transfer function, with μ=255.

2 SNR

One metric of PCM performance is the ratio of the signal power to quantization noise power (signal-to-quantizing distortion ratio). The basic idea behind "A-law" and μ-law companding is that a logarithmic curve may be used to improve the signal-to-quantizing distortion ratio at low signal levels.

Binary PCM has a number of quantization levels given by:

$$N_q = 2^{N_b} \tag{0.5}$$

where

$N_q =$ the number of quantization levels

$N_b =$ the number of bits per sample .

The signal power varies from 0 to 1, inclusive, and is given by:

$$S \in [0...1]$$

The signal voltage, x, varies from:

$$-V_{\max} \le x \le V_{\max} \tag{0.6}.$$

For simplicity we assume that

$$V_{\max} = 1 \tag{0.7}$$

The uniform quantizer divides the signal voltage evenly among the number of quantization levels. The uniform quantizers' quantization error voltage is given by:

$$-\frac{1}{2^{N_b}} \le \varepsilon \le \frac{1}{2^{N_b}}$$

where (0.8)

$\varepsilon =$ quantization error voltage

The average quantization error is 0, but the root-mean-square (RMS) value of the quantization error is the mean square quantization noise power. The term "root-mean-square" refers to the square root of the mean of error voltage squared [Carlson and Gisser]. For a continuous probability distribution function, the expectation is taken by $E[X] = \int_{-\infty}^{\infty} x f_X(x)\,dx$ and the variance is taken by $\sigma_X^2(t) = E\left[|X(t)|^2\right]$. The variance expands to $\sigma_X^2(t) = \int_{-\infty}^{\infty} x^2 f_X(x)\,dx$. For a discrete random variable, $E[X] = mean = \frac{1}{N}\sum_{i=1}^{N} x_i$ and

$$\sigma_X^2(t) = \text{var}\,iance = E\left[|X(t)|^2\right] = \frac{1}{N}\sum_{i=1}^{N}\left(x_i - E[X]\right)^2 .$$

Thus, we compute the RMS error by integrating the square of the quantization error voltage over the range in (0.8) assuming a zero mean:

$$\sigma^2 = \frac{1}{\left(2/N_q\right)}\int_{-1/N_q}^{1/N_q} \varepsilon^2\, d\varepsilon = \frac{1}{3N_q^2} \tag{0.9}.$$

When the maximum signal voltage is constrained, as in (0.6) and (0.7) we compute the signal-to-quantization noise power as:

$$SNR_D = 10\log_{10}\left(3\times 2^{2N_b} S_x\right)$$
$$SNR_D = 10\log_{10} 3 + 20N_b \log_{10} 2S_x \qquad (0.10)$$

Where the signal power is given by S_x. If the signal power is equal to 1, then the range on the upper bound for the signal-to-quantization noise power is:

$$SNR_D \le 4.8 + 6N_b \qquad (0.11).$$

With the 8-bit PCM system and uniform quantization, the best we can hope for is a SNR of 52.8 dB. Note that the SNR (in dB) falls off linearly as a function of the power in (0.10). It can be shown that the companding equations of (0.2) and (0.3) will provide an improvement in the SNR when the signal power falls below -20 dB.

It must also be mentioned that for signal powers above -20 dB, companding degrades performance, relative to uniform quantization, assuming that the PDF (Probability Distribution Function) of a voice signal has a *Laplace* distribution of the form

$$p(x) = \frac{1}{2}\alpha\, \mathbf{e}^{(-\alpha\, x)}$$

[Carlson].

The compression parameter values given in (0.10) are based on an assumed PDF of an input signal. The PDF assumption is required for telephony applications. However, in the instance of audio files that are stored on static media (such as CD ROM, or web server hard drive) the computation of the PDF can be performed off-line. For such a system, the SNR is

$$SNR_D = \frac{S_x}{\sigma^2} = \frac{3N_q^2 S_x}{K_z}$$

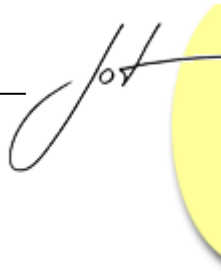
where (0.12)

$$K_z = 2\int_0^1 \frac{p(x)}{\left[y'\right]^2}$$

computing the derivative of (0.2) with respect to *x*, and substituting into (0.12) yields

$$K_Z = 2\int_0^1 \frac{p(x)\,(V_{max} + \mu\, x)^4 \ln(1+\mu)^4}{\mu^4\, V_{max}^4}\, dx \qquad (0.13)$$

Once the PDF, *p(x)*, is computed, the companding parameter can be precomputed using a criterion of optimality based on the bit rate budget. For example, a bit rate budget of 16 kbps might require a 4 bit sample with a 4 kHz sampling rate. Such a system can be used to stream audio via a low data-rate phone connection.



3 THE ULAWCODEC CLASS

The *UlawCodec* class performs the μ-law CODEC function, in addition to providing file save and open services. Java can, in principle, process any file format.

Class Summary

```
public class UlawCodec implements Runnable {
public UlawCodec()
public UlawCodec(String name)
public UlawCodec(short linearArrayOfShort[])
public UlawCodec(double linearArrayOfDouble[])
public UlawCodec(byte ulawArrayOfByte[])
public void readAUFile(String fileName)
public void readAUFile()
public void writeAUFileString fileName)
public void writeAUFile()
public void playSync
public void playAsync
public byte [] getUlawData()
public void setUlawData(byte ulawArrayOfByte[])
public double[] getDoubleArray()
public int getLength()
public double getDuration()
public void reverseUlaw()
public static void main(String argc[])
}
```

4 CLASS USAGE

The *UlawCodec* has several constructors, each has, as it main goal, to construct a μ-law encoded byte array in a private storage area. The only way to obtain access to this storage area is via the *getUlawData* and *setUlawData* methods. This is due, in part, to a series of parallel data structures that must maintain their consistency. For example, when you invoke the *getDoubleArray* method, a check is performed to see if the internal *DoubleArray* is null. If the array is null, it is set using computations involving the *ulawData*. The consistency maintenance mechanism is invisible to the programmer.

Suppose the following variables are pre-defined:

```
UlawCodec ulc;
String fileName;
byte ulawArrayOfByte[];
short linearArrayOfShort[];
Double linearArrayOfDouble[];
int length;
double timeInSeconds;
String args[];
```

To read in a Sun AU file, using a standard file open dialog box:

```
ulc = new UlawCodec();
```

To read in a Sun AU file, using a file name:

```
ulc = new UlawCodec(fileName);
```

To construct a UlawCodec instance from a 16 bit linear data array:

```
ulc = new UlawCodec(linearArrayOfShort);
```

To construct a UlawCodec instance from a linear double array:

```
ulc = new UlawCodec(linearArrayOfDouble);
```

To overwrite the internal data and read in a new AU file, given a file name:

```
ulc.readAUFile(fileName);
```

To prompt the user for a read file name and overwrite the internal data:

```
ulc.readAUFile();
```

To write the internal data as new AU file, given a file name:

```
ulc.writeAUFile(fileName);
```

To prompt the user for a file name, then write a Sun AU file:

```
ulc.writeAUFile();
```

To play synchronously, returning only after the sound is played:

```
ulc.playSync();
```

To play asynchronously, returning right away and playing the sound in the background:

```
ulc.playAsync();
```

To get the raw companded byte data:

```
ulawArrayOfByte = ulc.getUlawData();
```

To set the raw companded byte data:

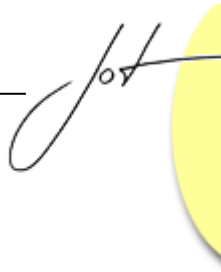
```
ulc.setUlawData(ulawArrayOfByte);
```

To get the data as an array of linear doubles:

```
linearArrayOfDouble = ulc.getDoubleArray();
```

To get the number of samples:

```
length = ulc.getLength();
```



To get the play time in seconds:

```
timeInSeconds = ulc.getDuration();
```

To reverse the U-law data, forcing recomputation of the *DoubleArray*:

```
ulc.reverseUlaw();
```

To test the read play and write methods:

```
UlawCODEC.main(args);
```

5 READING AND WRITING μ-LAW

The following example, excerpted from the *UlawCodec.java* file, shows how the main method is implemented:

```
1. public static void main(String argc[]){
2.  UlawCodec ulc = new UlawCodec();
3.  ulc.playSync();
4.  ulc.writeAUFile();
5. }
```

Line 2 shows the default constructor for the CODEC. The default constructor opens the standard file dialog box in order for the user to select a AU file. Line 3 plays the sound and does not return until the sound has completed playing. The *writeAUFile* opens a dialog box and the user must type in a file name to save the AU file.

6 THE OSCILLATOR CLASS

Several instances of the *Oscillator* class may be made to create banks of Oscillators. The Oscillator class makes use of double precision data arrays and can make very low-distortion waveforms.

7 CLASS SUMMARY

```
public class Oscillator {
public Oscillator(double frequency, int length)
public double[] getSineWave()
public double[] getSquareWave()
public double[] getSawWave()
public double[] getTriangleWave()
public double getDuration()
public int getSampleRate()
public double getFrequency()
public void setModulationIndex(double I)
public void setModulationFrequency(double fm)
```

```
public double[] getFM()
public double[] getAM()
}
```

Class Usage

The Oscillator class has a number of private properties that are accessed via get and set methods. An Oscillator instance is created for a fixed carrier frequency and number of samples. All waveforms vary from -1 to 1. Suppose the following variables are predefined:

```
double frequency = 440;
double length = 2000; // the total number of samples
Oscillator osc;
double audioData[];
double timeInSeconds;
int sampleRate;
double indexOfModulation;
```

Then to make an instance of an Oscillator:

```
osc = new Oscillator(frequency, length);
```

To get sine, square, saw tooth and triangle waves:

```
audioData = osc.getSineWave();
audioData = osc.getSquareWave();
audioData = osc.getSawWave();
audioData = osc.getTriangleWave();
```

To get the time the wave form will last, in seconds:

```
timeInSeconds = osc.getDuration();
```

To get the sample rate, in Hz:

```
sampleRate = osc.getSampleRate()
```

To get the frequency, in Hz:

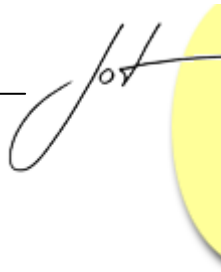
```
frequency = osc.getFrequency()
```

To set the index of modulation of the FM oscillator:

```
osc.setModulationIndex(indexOfModulation);
```

To set the modulation frequency of both the AM and FM oscillators:

```
osc.setModulationFrequency(frequency);
audioData = osc.getFM()
audioData = osc.getAM()
```



8 CLASS EXAMPLES

The *AudioFrame* class is able to generate a series of waveforms using the Oscillator class and the *UlawCodec* class.

```
public class AudioFrame extends ClosableFrame {
     private UlawCodec ulc;
     private Oscillator osc =
        new Oscillator(440,4000);
...
public class AudioFrame extends ClosableFrame {
     private UlawCodec ulc;
     private Oscillator osc =
        new Oscillator(440,4000);
...
public void play() {
        ulc.playSync();
}
public void sineWave() {
     ulc = new UlawCodec(
        osc.getSineWave());
     play();
}
public void squareWave() {
     ulc = new UlawCodec(
        osc.getSquareWave());
     play();
}
public void sawWave() {
     ulc = new UlawCodec(
        osc.getSawWave());
     play();
}
public void triangleWave() {
     ulc = new UlawCodec(
        osc.getTriangleWave());
     play();
}
public void am() {
```

```
        osc.setModulationIndex(0.5d);
        osc.setModulationFrequency(200d);
        ulc = new UlawCodec(
            osc.getAM());
        play();
    }

    public void fm() {
        osc.setModulationIndex(0.5d);
        osc.setModulationFrequency(200d);
        ulc = new UlawCodec(
            osc.getFM());
        play();
    }
```

9 CLASS IMPLEMENTATION

An *Oscillator* is able to generate repeated waveforms by constructing a single cycle of the waveform into a wavetable. The wavetable is copied repeatedly into an array of double data known, internally as *audioData*.

The *Oscillator* class is implemented with a series of private class variables:

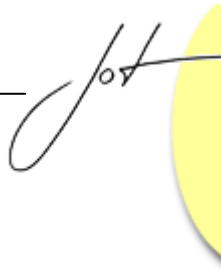
```
1.
2.   import futils.utils.Computation;

3.   public class Oscillator {
4.   private double audioData[];
5.   private double waveTable[];
```

Lines 4 and 5 show that the audioData waveTables are unallocated until the constructor is invoked. Line 6 shows the *sampleRate*. The constructor could be overloaded to take other sample rates [Lyon].

```
6.   private int sampleRate = 8000;
```

Line 7 shows the frequency, in Hz. For a sine wave, this is the number of wave table cycles that must be clocked out, per second. Lambda is the number of seconds in the period of one cycle. Line 9 shows the number of samples in a single cycle of the wave table, if this were computed with precision. Keep in mind that the length of a wave table is always an integer and the samplesPerCycle must be converted as a result.

```
7.   private double frequency;
8.   private double lambda;
9.   private double samplesPerCycle;
```



Oscillator construction requires that the carrier frequency and number of cycles be known. Line 3 show the memory allocation for the audioData.

```
1.   public Oscillator(double frequency_, int length) {
2.   frequency = frequency_;
3.   audioData = new double[length];
```

Once the period of the waveform is computed, on line 5, we are able to compute the number of samples in a cycle of the wave table. This is the samplesPerCycle variable, cast into an integer. Given the integral approximation, the computation for the actual frequency at which the wave table is clocked out is:

$$f_{actual} = sampleRate / waveTable.length \qquad (0.14)$$

With the wave table length being:

$$waveTable.length = round(sampleRate / f) \qquad (0.15)$$

To compute the error in the digital oscillator instance, we subtract the frequency that we wanted from the rate cycles are clocked out of the wave table.

$$f_e = f - f_{actual}$$

For example, for a frequency of 440 Hz, *waveTable.length* = 18 *sampleRate* = 8000 *audioData.length* = 4000 and the actual frequency = 444.444 Hz. Exact frequencies may be had when the frequency desired is an exact multiple of the *sampleRate*. For example, 400 will be reproduced with precision because 8000/400 = a *waveTable.length* of 20.

```
4.   //the period of the wave form is
5.   lambda = 1/frequency;
6.   //The number of samples per period is
7.   samplesPerCycle = sampleRate * lambda;

8.    delta_freq = 1/samplesPerCycle;
9.    waveTable =
10.     new double[(int) samplesPerCycle];

11.  }
```

Building the WaveTable

The *AudioDataFromTable* method in the *Oscillator* class is used to turn a single cycle of the *WaveTable* into a long array of audio data. A constraint on the *audioData* array is that it must have an absolute value that is strictly less than 1 (due to the companding formulas).

```
1. private double[] AudioDataFromTable() {
2.   int k = 0;
3.   for (int i = 0; i < audioData.length; i++) {
```

Line 4 builds the audioData from the waveTable. While the indexes, *i* and *k* both begin at 0, lines 6 and 7 reset *k*, while index *i* increments on.

```
4.       audioData[i] = waveTable[k];
5.       k++;
6.       if (k >= waveTable.length)
7.            k = 0;
8.   }
9.   System.out.println("\nlambda="+lambda+
10.      "\nfrequency = "+frequency+
11.      "\nwaveTable.length = "+waveTable.length+
12.      "\nsampleRate = "+sampleRate+
13.      "\naudioData.length = "+audioData.length+
14.      "\nactual frequency = "+actualFrequency());
15.  return audioData;
16.  }
```

In the *getSineWave* method, the *waveTable* is computed for a single cycle. To make sure that the absolute value of the amplitude of the sine wave is always less than 1, 0.98, on line 20, first multiplies it.

```
17.  public double[] getSineWave() {
18.  for (int i=0; i<waveTable.length; i++)
19.      waveTable[i] =
20.           0.98*Math.sin(twopi * i/waveTable.length);
21.  return AudioDataFromTable();
22.  }
```

To build a wave table for a saw wave, we set the initial voltage to -1, then compute a change in voltage, *dv* using the length of the wave table, *L* so that $V_0 = -1, dv = \frac{2}{L}$. then, after L-1 intervals, the final voltage will reach a value of *1-dv*. The following code implements the *getSawWave* method:

```
1.   public double[] getSawWave() {
```

In line 2, the initial voltage is set to a value that is a little higher than 1.0. This is due to the constraint on the CODEC's input. Also, in line 4, the check is i<waveTable.length rather than i <= waveTable.length. Thus the saw wave will end at (v-dv), rather than at 1.0 volts.

```
2.   double v = -0.99;
3.   double dv = 2.0 / (double) waveTable.length;
4.   for (int i=0; i<waveTable.length; i++){
5.       waveTable[i] = v;
```

```
6.      v += dv;
7.   }
8.   System.out.println("Sawwave ends at:"+(double)(v-dv));
9.   return  AudioDataFromTable();
10.  }
```

The saw wave output is shown in Figure 2.

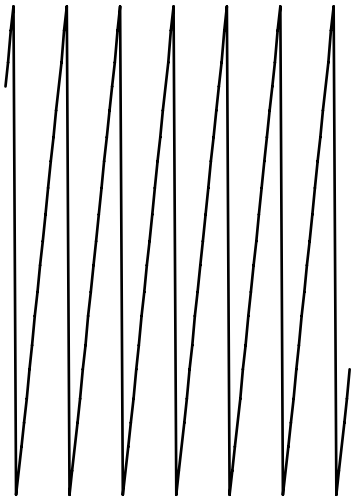


Figure 2. The saw wave output

Examples

To play a tone, use:

```
private static void playTone(int f, int dur) {
        Oscillator osc = new Oscillator(f, dur);
        UlawCodec ulc = new UlawCodec(osc.getSineWave());
        ulc.play();
    }
```

The ULaw class contains a play method that enables the playing of u-law compressed audio directly to an output stream:

```
public void play() {
        stop();
     if (ulawData == null) return;
        AudioData audioData =
                new AudioData(ulawData);
        audioDataStream = new AudioDataStream(audioData);
     AudioPlayer.player.start(audioDataStream);
    }
```

10 CONCLUSION

CODECs play a central role in multi-media programming. The Internet has become a hotbed of voice over IP activity. Isn't it fascinating that a 1965 standard for voice CODECs is still in common use today [BTL]

REFERENCES

[BTL] *Transmission Systems for Communications*, by Members of the Technical Staff, Bell Telephone Laboratories, Western Electric Company, Inc., Technical Publications, Winston-Salem, NC, 1965.

[Carlson and Gisser] *Electrical Engineering*, by A. Bruce Carlson and David G. Gisser, Addison Wesley, Reading MA, 1981.

[Carlson] *Communication Systems*, by A. Bruce Carlson, McGraw Hill, 1986.

[Embree] *C Language Algorithms for Digitial Signal Processing*, by Paul M. Embree and Bruce Kimble, Prentice Hall, EnglewoodCliffs, NJ, 1991.

[Lyon] *Java Digital Signal Processing*, Douglas A. Lyon and H. Rao, M&T Books, November 1997.

About the author

Douglas A. Lyon (M'89-SM'00) received the Ph.D., M.S. and B.S. degrees in computer and systems engineering from Rensselaer Polytechnic Institute (1991, 1985 and 1983). Dr. Lyon has worked at AT&T Bell Laboratories at Murray Hill, NJ and the Jet Propulsion Laboratory at the California Institute of Technology, Pasadena, CA. He is currently the Chairman of the Computer Engineering Department at Fairfield University, in Fairfield CT, a senior member of the IEEE and President of DocJava, Inc., a consulting firm in Connecticut. Dr. Lyon has authored or co-authored three books (Java, Digital Signal Processing, Image Processing in Java and Java for Programmers). He has authored over 30 journal publications. Email: lyon@docjava.com. Web: http://www.DocJava.com.