

Supplement to Chapter 4 of *The Science of Digital Media* – Digital Audio Representation

Worksheet – Digital Audio > Non-Linear Companding and μ -Law Encoding¹

Modeling Environment: MATLAB

Introduction:

μ -law encoding is an example of a non-linear companding (compression and expansion) method used in telephone communication. Rather than quantizing audio samples with equal-sized quantization levels, non-linear companding quantizes lower amplitude values in more detail than higher amplitude ones. This method works well for telephone communication because it reduces the signal-to-noise ratio in the area where it matters most – in low amplitude values, which are common in human speech and are particularly subject to noise distortion. The equation for non-linear compression by μ -law encoding is

$$m(x) = \text{sign}(x) \left(\frac{\ln(1 + \mu|x|)}{\ln(1 + \mu)} \right)$$

where $-1 \leq x \leq 1$, and $\text{sign}(x)$ is -1 if x is negative and 1 otherwise. μ is 255 when samples are being quantized to 8 bits. (In some sources, you will see this equation using \log_2 rather than the natural log. The definitions are equivalent.)

Decompression operates by the inverse equation:

$$d(x) = \text{sign}(x) \left(\frac{(\mu + 1)^{|x|} - 1}{\mu} \right)$$

Using MATLAB, let's see what effect the compression equation has on audio samples.

Create the μ -law function with $\mu = 255$. Call it f .

```
f = inline('sign(x)*((log(1+255*abs(x)))/log(256))', 'x');
```

Plot the μ -law function over the domain interval $x=[-1 \ 1]$.

```
fplot(f, [-1 1]);
```

The graph should look something like this:

¹This material is based on work supported by the National Science Foundation under Grant No. DUE-0340969. This worksheet was written by Dr. Jennifer Burg (burg@wfu.edu).

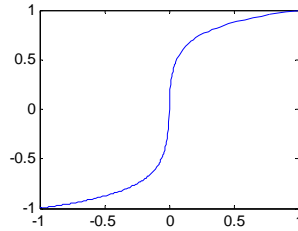


Figure 1

Let's think about what this graph represents and how it relates to the digital encoding of audio samples. Imagine that you are trying to digitize the analog audio wave like the one pictured below.

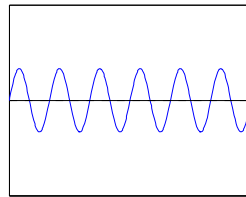


Figure 2

In non-linear companding, 16-bit digital audio samples are compressed to 8-bit values at the transmission end, and then decompressed back to 16-bit values when they are received. We want to see how much error is introduced by this compression.

The logarithmic function above is the right "shape," but it doesn't have the right units. Let's think about how you can apply the function so that it maps 16-bit samples to 8-bit ones.

With n bits, 2^n different quantization levels ranging from -2^{n-1} to $2^{n-1}-1$ can be represented. If 16 bits are initially used for each audio sample, then we can represent $2^{16}=65,536$ different quantization levels: $-32,768$ to $32,767$). After reducing the bit depth to 8 bits, we can represent values between -2^7 and 2^7-1 (that is, -128 to 127).

The input values for function f range between -1 and 1 . We can scale 16-bit values to the interval $[-1, 1)$ by dividing by $32,768$. Output will then also be in the interval $[-1, 1)$. We can scale these to 8-bit values by multiplying by 127 . Thus, to find out what b maps to, we can use evaluate $f(b/32768) * 128$

Alternatively, we can change the function so that it takes values between $-32,768$ and $32,767$ as input and yields integer values between -128 and 127 as output.

In MATLAB, to evaluate the function at a given $x=n$, use:
 $q = inline('floor(128*(sign(x)*((log(1+255*abs(x)/32768)))/log(256))))', 'x');$

$q(n)$

Replace n by your 16-bit sample value.

Now, using q , you can answer the following questions.

Question 1: Which of the following statements is true?

- A. $q(255) - q(0) > q(32767) - q(32058)$ OR
- B. $q(32767) - q(32058) > q(255) - q(0)$?

Question 2: Another way of asking question 1 is this – which quantization interval gets mapped to a wider range of values: the portion of the x-axis between 0 and 255 or the portion of the x-axis between 32,058 and 32,767?

Question 3: Based on your answers to questions 1 and 2, using μ -law encoding, which samples are quantized in a more fine-grained manner – the low amplitude input values, or the high amplitude ones?

Another way to view this is to plot the function on a smaller scale, starting at the low amplitude end. We'll begin with $x = 0$ to 1000 and go up from there. Try this:

```
fplot(q, [0 1000 0 50]);
```

Then plot for $x = 1$ to 2000. To keep the plot on the same scale, set the y-axis to values between 50 and 100.

```
fplot(q, [1001 2000 50 100]);
```

Question 4: How does the graph of the function change as you move from low amplitudes to high amplitudes?

Another way to requantize from 16-bit samples to 8-bit samples is to divide by 256 and round. This would be an example of linear quantization, as opposed to μ -law encoding's non-linear method. Let's compare the two methods graphically.

We'll call the new function d .

```
d = inline('round(x/256)', 'x');  
fplot(d, [1001 2000 0 50]);
```

Compare the graph for function d to the one for function q . You might want to look at this graph more closely, as you did with the μ -law function, by zooming in on 1000 unit sections.

Question 5: What's the difference between the graphs for functions d and q ?

Question 6: Assuming that d and q represent two different ways of compressing 16-bit samples to 8-bit samples, explain what these graphs show you about relative precision with which d and q measure samples at different amplitudes?

Question 7: For function q , how many different integer input values from the domain $-32,768$ to $32,767$ map to an output of 1? (Reminder: In MATLAB, to evaluate the function q at value n , use $q(n)$).

Question 8: For function q , how many different integer input values from the domain $-32,768$ to $32,767$ map to an output of 127?

Question 9: What can you conclude from questions 7 and 8?

Question 10: In question 7, you identified m input values that map to 1 with the function q . When you decompress these with the inverse function, what do all of these m values map back to (i.e., what is their value in 16 bits after decompression)?

Hint: Use the inverse function to map the values back to values between $-32,768$ and $32,767$.

Question 11: What is the percentage error for an original 16-bit sample with a value of 3? What is the percentage error for an original 16-bit sample with a value of 6? What is the percentage error for an original 16-bit sample with a value of 8? To answer this for value b , evaluate $v(q(b))$. What you are doing is compressing the 16-bit sample using q . Then you're decompressing it using v . The error is the difference between the original input value, which is b , and the value you get back after decompression. Find the ratio of this error relative to b , as a percentage.

Question 12: If you compressed a 16-bit sample with values of 1 through 127 by dividing by 256 and rounding, what would be the percentage error for each value? (Assume that to decompress, multiply by 256.)

Question 13: What can you conclude from questions 10, 11, and 12?

Question 14: The highest value that compresses to 66 is 2212. What does this map back to? What is the error?

Question 15: Consider the largest of the m values that map to 66, which is 2212. If you use the linear method of compressing and decompressing, what is the error?

Question 16: What can you conclude from questions 14 and 15?

Conclusions:

Non-linear companding methods such as μ -law encoding take 16-bit digital audio samples and compress them to eight bits. 16-bit samples offer a dynamic range of 96 dB. 8-bit samples offer a dynamic range of 48 dB. You might expect that in compressing 16-bit samples to eight bits, we would be sacrificing half the dynamic range in a digital audio file, but this is not the case. Upon decompression with non-linear companding, the low amplitude samples return to values very close to what they were originally, so we haven't lost the ability to represent low amplitude signals clearly. The result is that μ -law encoding effectively yields a dynamic range of about 72 dB using only eight bits.