Chapter 5

PROGRAMMING EXERCISE

Mu-Law Encoding in C++

Mu-law encoding is a non-linear companding method that can be used to reduce the bit depth of a digital audio signal in a way that preserves the dynamic range of samples at low amplitudes. The word “companding” is used because this method works by compressing the bit depth of an audio signal and then expanding it again after the signal has been transmitted. Mu-law encoding is non-linear because it uses more bits to quantize low amplitude samples as opposed to high 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 mu-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. \(\mu\) 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:

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

Here is a scenario in which mu-law encoding could be used. A 16-bit digital audio signal – for example, a telephone signal – is to be transmitted across a network. It is reduced to 8-bits by mu-law encoding and then transmitted. At the receiving end, it is decoded back to a 16-bit signal. Because the amount of error for low amplitude samples is less than it would have been in a linear encoding, the effect is that a dynamic range of 72 dB is achieved with mu-law encoding (as opposed to 48 dB that you would expect from an 8-bit audio signal).

It’s possible to implement mu-law encoding by applying the equations above directly. However, the same effect can be achieved with a bit manipulation method that resembles the way floating point numbers are represented. Here’s a sketch of the algorithm:
Step 1. Assuming the 16-bit sample value is in sign-and-magnitude format, save the sign bit.
Step 2. Clip the magnitude to 32,635. (The purpose of the clipping is to avoid overflow when the bias is added in the next step. Note that the values for 16-bit samples range from $-2^{15}$ to $2^{15}-1$. The largest magnitude is 32,767.)
Step 3. Add a bias of 132 to the magnitude. (The purpose of adding the bias is to ensure that a 1-bit will appear somewhere in the exponent region, as determined in Step 4 below.)

Algorithm 1 mu-law encoding

Step 4. Let the exponent region be defined as the eight bits to the right of the sign bit. Let $p$ be the position of the leftmost 1-bit in the exponent region, counting positions from 0 to 7 from right to left.
Step 5. Let the mantissa region be defined as the 4 bits to the right of position $p$.
Step 6. Encode eight bits by making the leftmost bit the sign bit (saved in Step 1), the next three bits the binary encoding of $p$, and the last four bits the mantissa region identified in Step 5.

Let’s try this on an example. The original sample is 0010 1110 1001 1011. (We insert a blank space after every fourth bit for readability.)

<table>
<thead>
<tr>
<th>Step</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>Sign bit is 0.</td>
</tr>
<tr>
<td>Step 2</td>
<td>No clipping necessary.</td>
</tr>
<tr>
<td>Step 3</td>
<td>0010 1110 1001 1011 + 1000 0100 = 0010 1111 0001 1111 (base 2)</td>
</tr>
<tr>
<td>Step 4</td>
<td>Exponent region underlined and in boldface.</td>
</tr>
<tr>
<td></td>
<td>0010 1111 0001 1111</td>
</tr>
<tr>
<td></td>
<td>Leftmost 1-bit in exponent region is in position 6. That’s 110 base 2.</td>
</tr>
<tr>
<td>Step 5</td>
<td>Mantissa region is underlined and in boldface.</td>
</tr>
<tr>
<td></td>
<td>0010 1111 0001 1111</td>
</tr>
<tr>
<td>Step 6</td>
<td>Encode eight bits as</td>
</tr>
<tr>
<td></td>
<td>0110 0111</td>
</tr>
</tbody>
</table>

Thus, 0110 0111 is the 8-bit μ-law encoding of 0010 1110 1001 1011. (In fact, this value would probably be put in one’s or two’s complement format after encoding, but that step is not important to your understanding of the concept of μ-law encoding.)

This procedure effectively encodes the samples in a logarithmic manner. Neighboring quantization values at high amplitudes correspond to larger differences in amplitudes, as compared to neighboring values at low amplitudes. This lowers the signal-to-quantization noise ratio at lower amplitudes, where the noise is otherwise most pronounced.

Decoding reverses this process. The steps are given in Algorithm 2.

Algorithm 2 decoding

Step 1. Save the sign bit (the most significant bit of the eight bits given).
Step 2. Save the exponent region (the three bits to the right of the sign bit) as a 3-bit field with value $e$. 
**Step 3.** Save the mantissa region (the remaining four bits) as a four-bit field called $m$, which we view as a binary field.

**Step 4.**

\[
\begin{align*}
  a &= 3 + e \\
  b &= (1000\ 0100 \text{ left shifted by } e \text{ bits}) - 1000\ 0100 \\
\end{align*}
\]

decoded sample = $(m \text{ left shifted by } a \text{ bits}) + b$

**Step 5.** Affix the sign bit as the leftmost bit.

---

**Algorithm 2 mu-law decoding**

For our example, the steps work as follows:

**Step 1.** Sign bit is 0.

**Step 2.** Exponent region is $110_2 = 6_{10}$.

**Step 3.** Mantissa region is $0111$.

**Step 4.**

\[
\begin{align*}
  a &= 3 + 6 = 9 \\
  b &= (1000\ 0100 \text{ left shifted 6 bits}) - 1000\ 0100 = 10\ 0000\ 0111\ 1100 \\
\end{align*}
\]

decoded sample = $(0111 \text{ left shifted by 9 bits}) + b = 0\ 1110\ 0000\ 0000 + 10\ 0000\ 0111\ 1100 = 10\ 1110\ 0111\ 1100$

**Step 5.** With sign bit affixed, decoded sample value is $0010\ 1110\ 0111\ 1100$

The decoded sample is $(0010\ 1110\ 0111\ 1100_2 = 11,900_{10})$, while the original sample value was $(001011010011011_{2} = 11,931_{10})$. If you try this algorithm on a sample of smaller magnitude, you'll see that the percent error is much smaller compared to using truncation to reduce the bit depth. Using the algorithm, a sample value of $193_{10}$ is decoded as $196_{10}$, with an error of 1.5%. Using truncation to achieve eight bits, $193_{10}$ ($0000000011000001_{2}$) is encoded as 0, for a 100% error. Mu-law encoding distributes the error in a way that matches human hearing better than linear quantization methods. It increases the SQNR in low amplitude signals, which is where the noise is most easily perceived.

Your assignment is to implement mu-law encoding by both methods: applying the equations, and then using bit manipulations. Compare the results.