

Supplement to Chapter 5 of *The Science of Digital Media – Digital Audio Processing*

Programming Assignment – Digital Audio > μ -Law Encoding¹

Objectives:

1. To develop an understanding of the bitwise implementation of μ -Law encoding by designing a program to perform the compression.
2. To see the complexity advantages of this implementation in comparison to the logarithmic definition.

Introduction:

The foundation of μ -law encoding (also known as nonlinear companding) rests upon the non-uniformity of human perception of sound. Namely, the human ear is more sensitive to sounds of low amplitudes than high amplitudes. μ -law encoding takes advantage of this characteristic in order to compress a 16-bit audio signal in 8 bits per sample. The formula for μ -law encoding is given below, where x is the sample value to be compressed and the result is a compressed, 8-bit encoding:

$$y = \frac{\text{sign}(x) * \ln(1 + \mu|x|)}{\ln(1 + \mu)}$$

Once the compressed signal has been transmitted, it must then be expanded back to its original form from the 8-bit encoding. To do this, the following formula is used:

$$z = \text{sign}(y) * \frac{1}{\mu} \left[(\mu + 1)^{|y|} - 1 \right]$$

It should be noted that μ -law compression is lossy; applying the first equation then the second results in a signal that is slightly different than the original. However, the use of the natural logarithm in the compression results in a compression that quantizes low-amplitude samples with more precision (more quantization levels) than high-amplitude samples. For a more complete discussion of how these formulas implement μ -law encoding, see Section 5.8.2 in the text.

While these formulas can be used to calculate the μ -law encoding of a signal, the logarithmic operations are highly inefficient for a processor. Bitwise operations are much less complex, and as a result a bit-oriented implementation of μ -law encoding can be used in place of the above

¹This material is based on work supported by the National Science Foundation under Grant No. DUE-0340969. This programming assignment was written by Todd Martin.

definition. They produce nearly the same results, but the bitwise algorithm involves less work for the processor. You will implement a version of μ -law encoding that uses bit operations to perform the same compression that is described in the equations above.

Instructions:

The Assignment

The assignment is to implement μ -law encoding using bit operations. It will be necessary to use a programming language that allows individual bits to be manipulated within allocated memory – e.g., C or C++. The program should have a menu format, in which the user is first prompted to input a *raw* audio file (with a bit depth of 16) along with the sampling rate and signal length. (Audio processing programs such as Adobe Audition allow files to be saved in raw format.) Using this information, calculate the number of samples and read in these samples, 16 bits at a time, from the specified file. Store the samples in an array of *short* integers, which are 16 bits in most systems.

The user should now be given a menu of operations that can be performed on this audio signal. These should include (1) compressing using μ -law and storing in a file and (2) decompressing using μ -law decompression and storing in a file. Proper constraint checking should be implemented in order to be sure that a compressed signal is created before option (2) can be selected.

To implement the μ -law encoding, you should create a *MuLaw* class (assuming your selected language is object-oriented) and create methods *compress()* and *decompress()* along with *store_compressed()* and *store_uncompressed()*. The class should also have a constructor that takes an array of *short* integers (assuming that a short is 16 bits) representing the uncompressed samples and deep-copies it into a member array. The *compress()* method should implement μ -law encoding, storing the results in an array of *unsigned chars*, which are 8 bits each. The *decompress()* method should do the opposite, reversing the μ -law encoding and storing the result in the *short* array member. The respective *store_X()* methods should query the user for an output file name and write the data to the file.

In order to implement the compression, you should examine the description of the bitwise version of μ -law encoding in Section 5.8.3 of *The Science of Digital Media*. The algorithm requires knowledge of the bit shift and bitwise AND and OR operators in C++ or your chosen programming language. In this implementation, you should perform the following steps for *each* of the samples in the sample array (in a loop):

1. Store the sign of the current 16-bit sample (stored in a *short*). This can be done with the following C++ statement:

```
int sign = (sig_uncompressed[i] < 0) ? 0 : 0x80;
```
2. Replace any negative sample values with their absolute values.

3. Create a *long* integer called *adj* and use `static_cast<long>(sig[i])` to convert the current 16-bit *short* sample value to a *long*. This is necessary because the next step could create a value larger than 32,767 – the largest possible value that can be stored in a *short*.
4. Add a bias of `132L` (using the *L* to store as a literal) to *adj*. Now crop *adj* to 32,767.
5. Now use this value to create the exponent and mantissa of the μ -law encoding. These value can be created in C++ as follows:

```
unsigned char exponent = nBits[(adj >> 7) & 0xFF] - 1;  
unsigned char mantissa = (adj >> (exponent + 3)) & 0x0F;
```
6. These exponent and mantissa value can be combined to form the 8-bit μ -law encoding of the 16-bit sample. This value should be stored in an array of *unsigned chars*, which are 8 bits in length. The following C++ statement can be used to create the encoding:

```
sig_compressed[i] = ~(sign | (exponent << 4) | mantissa);
```

These operations should be included in the `compress()` method of the *MuLaw* class. The `decompress()` method should reverse these bitwise operations – taking each 8-bit μ -Law encoding and storing a 16-bit sample value in the decompressed *short* array. (Note that in this implementation the decompression *overwrites* the original 16-bit signal, because there is only one decompressed array member.)

To test the program, you can open an audio processing program like Adobe Audition and open the file in which the compressed or decompressed signal is stored (as a *.raw* file). The program should ask you what the *.raw* file represents – a 16-bit PCM encoding or an 8-bit μ -law encoding. Choose the appropriate representation so that the program recognizes the signal bits correctly.

Before Writing the Program

To run your program, you will need input files, so create these first or ask your instructor if he or she plans to create them for you. The audio files should be stored as *.raw* files – an operation that can be done in audio processing programs like Adobe Audition. This program is *not* designed for any format that has a header, so the *.raw* file format is important.

Ideas for Further Experimentation and Analysis

- Examine your algorithm with regard to complexity. Compare this to the implementation that you could have done by coding the formulas for μ -law encoding given on the first page. Why is it advantageous to implement the compression using bitwise operations?