Final Project of Speech and Audio Processing by Computer

Zeyu Liu

Professor Eom

ECE 6810

November 29, 2019

The George Washington University

Abstract

In this final project, I use MATLAB to read the TIMIT data file as a sound waveform. And then I quantize the waveform into 16-bits data, apply the u-law compression to the data and quantize the data again to convert it to an 8-bits data. After that, I implement the IMA ADPCM algorithm. By using the ADPCM encoder and ADPCM decoder, I get the new waveform. After that, I apply the u-law expansion to the new data. Finally, I obtain the reconstructed speech. Although the reconstructed speech keeps most of the original information, it has obviously noise.

Keywords: u-law, IMA ADPCM.

Introduction

In order to get a reconstructed speech, there are some steps to do in the *Figure 1*.

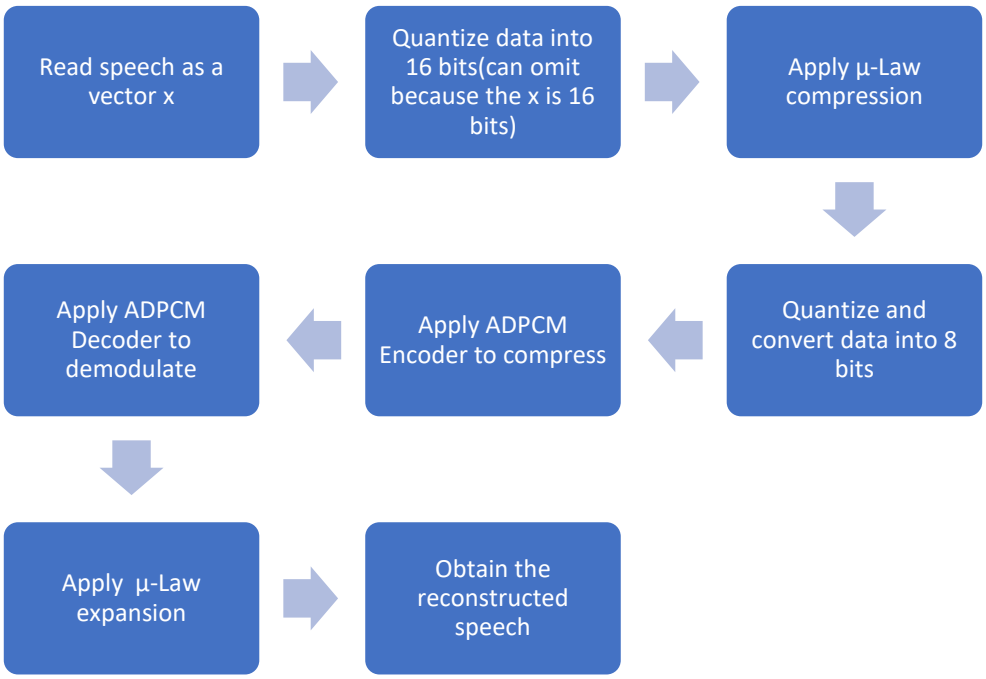

Figure 1. The steps of get a reconstructed speech.

Furthermore, we need to know some concepts of quantization, u-Law and IMA ADPCM (Interactive Multimedia Adaptive differential pulse-code modulation).

The first concept is quantization.

The quantization is the process of discretizing the amplitude of signal. According to the relationship between the input and output of the quantizer in the quantization process, there are two ways to obtain the quantization signal. One is uniform quantization; another one is non-uniform quantization. In the uniform quantization, the SNR (signal to noise ratio) of the small signal is small, while the SNR in the large signal is large. A better signal usually has a larger SNR. So uniform quantization is disadvantageous to the small signal. In order to improve the SNR of small signal, the quantization level could be further subdivided. At this time, the SNR of large signal is also improved. However, it will also increase the bit rate, and it will be required a

wider frequency bandwidth for transmission. In the non-uniform quantization, using the quantify characteristics of compression is an effective method to improve the SNR of small signal. The basic idea is to compress the large signal and enlarge the small signal before using uniform quantization. Because the amplitude of small signal is enlarged greatly, the SNR of the small signal is improved greatly too. This process is referred to as "compression quantization". The essence of compression quantization is to "compress big and compensate small", which makes the small signal's SNR consistent in the whole dynamic range. Comparing with the compression method, there are expansion method which characteristics are just opposite. At present, we commonly used A-Law or μ-law algorithm to compress and expand the signal. In MATLAB, it is called "compand".

The second concept is the μ-Law algorithm. Because the letter μ in Greek is pronounced "mu", the term mu-law comes from μ law; in addition, because of the same pronunciation, the term is sometimes written as u-law. In communication area, we learn about that the quality of communication can be affected by inconstant SNR. In order to keep the constant SNR, the μ-Law and A-Law are introduced. It is a "companding" algorithm, the main purpose of the algorithm is to increase SNR. In this project, we use μ-Law algorithm.

The μ-Law compressor equation:

$$F(\mathrm{x}) = \mathrm{sgn}(\mathrm{x})\frac{\ln(1+\mu|x|)}{\ln(1+\mu)} \quad -1 \le x \le 1.$$

The μ-Law expansion equation:

$$F^{-1}(\mathrm{y}) = \mathrm{sgn}(\mathrm{y})\frac{1}{\mu}\left((1+\mu)^{|y|}-1\right) \quad -1 \le y \le 1.$$

The μ-Law is used in North America and Japan where μ=255. The A-Law is widely used in Europe. μ-Law algorithm. (2019, October 26). Retrieved November 29, 2019, from Wikipedia: https://en.wikipedia.org/wiki/M-law_algorithm

However, in MATLAB, the formula for the μ-law compressor is:

$$F(\mathrm{x}) = \frac{V \ln(1+\mu|x|/V)}{\ln(1+\mu)} \mathrm{sgn}(\mathrm{x})$$

Where μ is the μ-law parameter of the compressor, V is the peak magnitude of x, and sgn is the signum function, the range of this function is −1 to 1. The input can have any shape or frame status. This block processes each vector element independently. So, we can compare the two compressors in the figures below.

Using the code below to obtain the standard μ-Law compressor and the TIMIT data file μ-Law compressor:

```
% standard u-Law compression principle
xu = linspace(-1,1);
yu = xu;
u1 = compand(yu,255,max(abs(yu)),'mu/compressor');
u2 = compand(yu,63,max(abs(yu)),'mu/compressor');
u3 = compand(yu,7,max(abs(yu)),'mu/compressor');

subplot(1,1,1),plot(xu,u1,xu,u2,xu,u3,xu,yu);
legend('u = 255','u = 63','u = 7','y = [-
1,1]','location','nw');
title('standard u-Law compression with different u');
grid on
pause;
% u-Law compression for the range of y1
xu = linspace(-0.1,0.1);
yu = xu;
u1 = compand(yu,255,max(abs(y1)),'mu/compressor');
u2 = compand(yu,63,max(abs(y1)),'mu/compressor');
u3 = compand(yu,7,max(abs(y1)),'mu/compressor');

subplot(1,1,1),plot(xu,u1,xu,u2,xu,u3,xu,yu);
legend('u = 255','u = 63','u = 7','y = [-
0.1,0.1]','location','nw');
title('u-Law compression with different u for range of
y1');
grid on
```

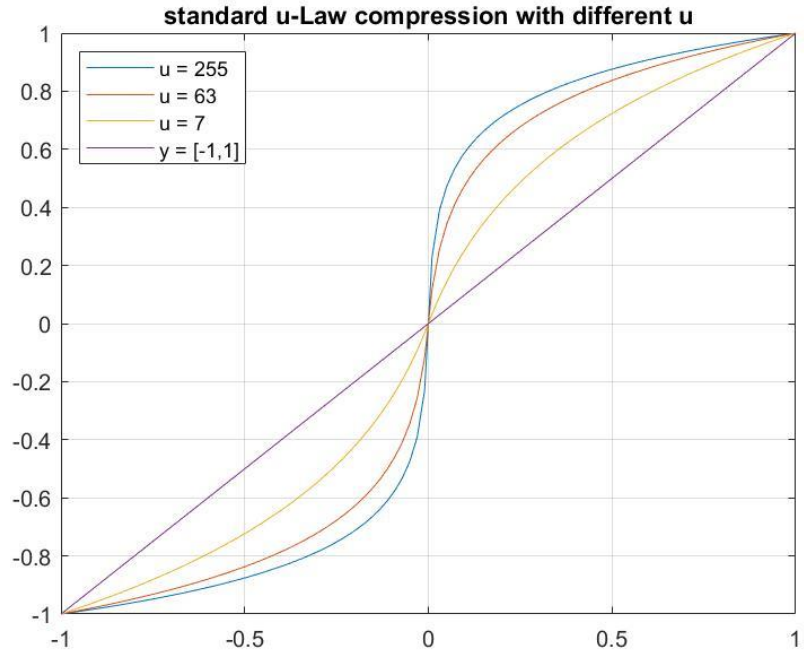

Figure 2. Plot standard μ-Law compression with different u

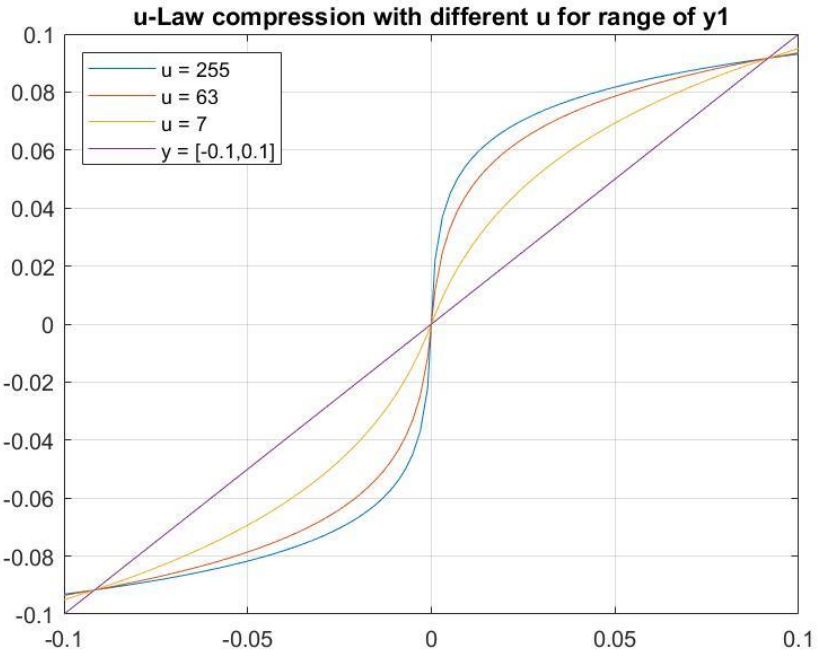

Figure 3. Plot μ-Law compression with different μ for range of y1

There are different range in *Figure 2* and *Figure 3* because of the peak magnitude, but the shape is similar. Because the closer to zero, the greater the slope. The amplitude of small signal is enlarged greatly then large signal, the SNR of the small signal is improved greatly too.

Now, in order to verify the above statement, we use the μ-Law compression in *Figure 3* for some samples. The test code and figures are here:

```
% Additional work for analysis the u-Law
compression
[x,Fs] = audioread('LDC93S1.wav');
N = length(x);
M = 320; % number of segments
Ns = floor(N/M); % number of samples per segments
for i = 1:50:M
    % start index = (i-1)*Ns+1, end index = i*Ns
    xs = x((i-1)*Ns+1:i*Ns);
    subplot(2,1,1),plot(xs);
    title('original data in different segments');
    pause;
    xs = uencode(xs,16);
    xs = udecode(xs,16);
    x2 =
compand(xs,255,max(abs(b)),'mu/compressor');
    subplot(2,1,2),plot(x2);
    title('u-Law compression in different
segments');
    pause;
end
```

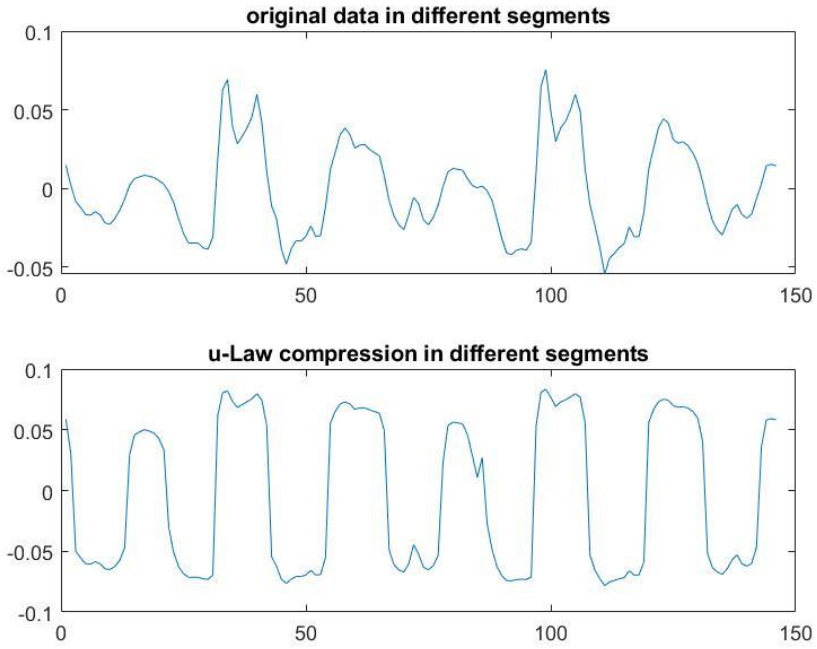

Figure 4. μ-Law compression sample 1

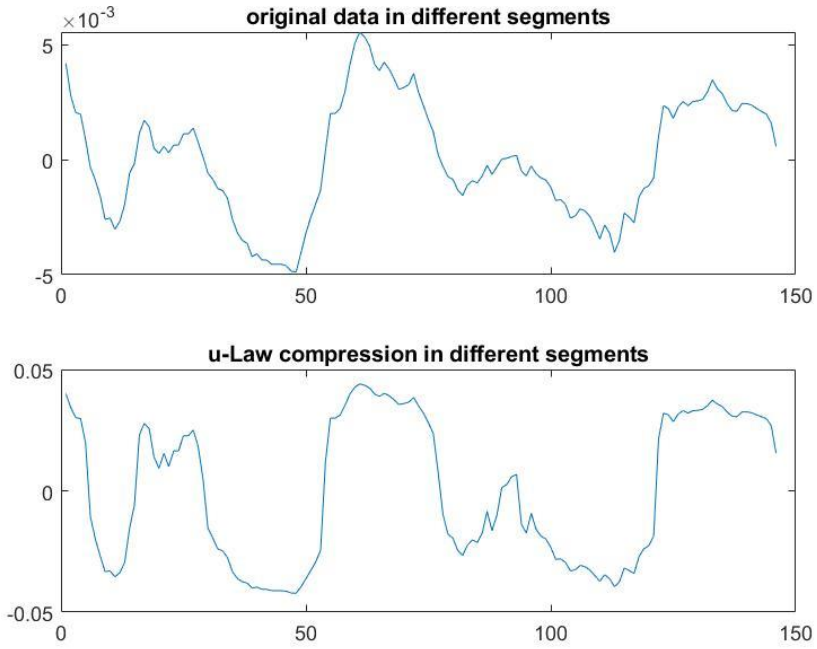

Figure 5. μ-Law compression sample 2

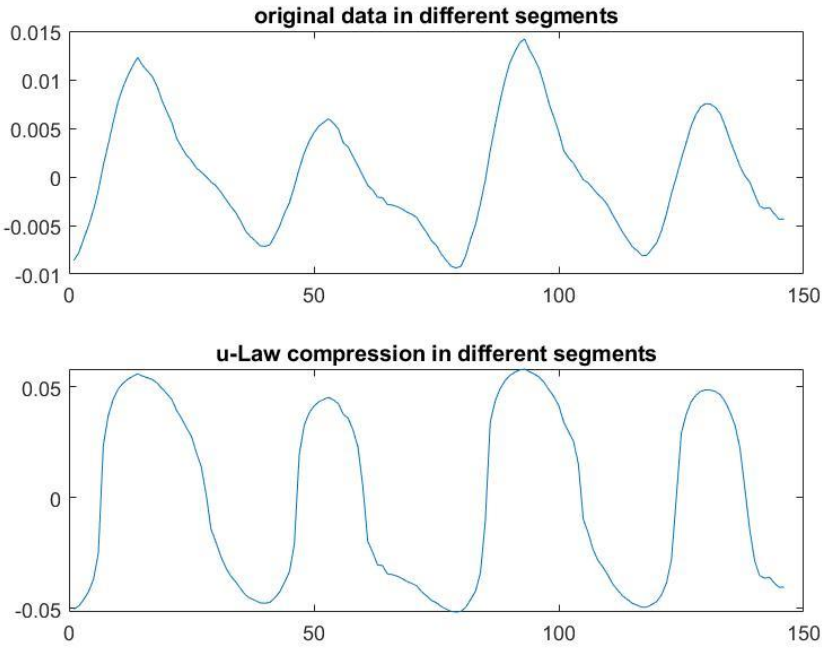

Figure 6. μ-Law compression sample 3

In *Figure 4*, *Figure 5*, and *Figure 6*, we can find that both the small signal and the large signal are amplified, and the amplification factor of the small signal is larger than that of the large signal. In addition, the small signal waveform changes more, while the large signal waveform changes less. This result confirms our point. You can find more samples in the "Figures" file.

The third concept is the IMA ADPCM.

IMA ADPCM originated from the ADPCM algorithm by the Interactive Multimedia Association and it is used in entertainment multimedia applications. It's a very simple codec algorithm with a fixed compression rate 4:1 and a reasonable SNR. ADPCM is a lossy compression mechanism.

IMA ADPCM compresses data recorded at various sampling rates. Sound is encoded as a succession of 4-bit or 3-bit data packets. Each data packet represents the difference between the current sampled signal value and the previous value. The compression ratio obtained is relatively

modest. The ADPCM algorithm takes advantage of the high correlation between consecutive speech samples, which enables future sample values to be predicted. Instead of encoding the speech sample, ADPCM encodes the difference between a predicted sample and the speech sample. This method provides more efficient compression with a reduction in the number of bits per sample yet preserves the overall quality of the speech signal.

Recommended Practices for Enhancing Digital Audio Compatibility in Multimedia Systems. (1992, October 21). Retrieved November 29, 2019, from Columbia University: http://www.cs.columbia.edu/~hgs/audio/dvi/IMA_ADPCM.pdf

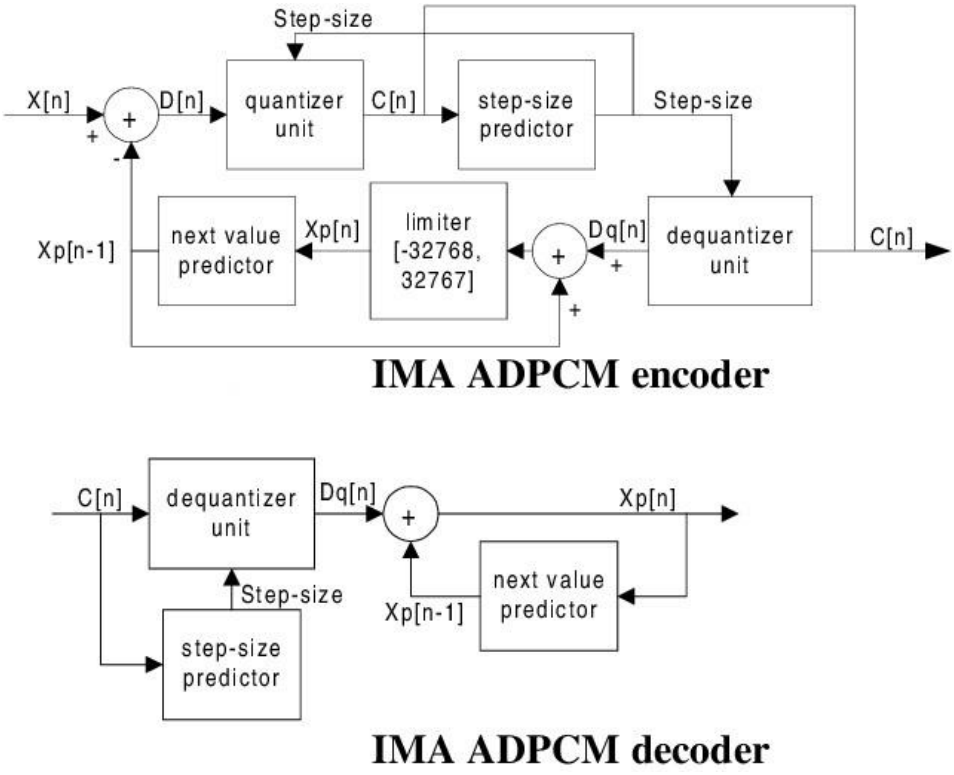

Figure 7. IMA ADPCM encoder and IMA ADPCM decoder.

From Figure7, we can see that in order to implement this algorithm we need three parameters, which is predictor, step index and step. It uses self-adaptive mechanism to change the size of the quantization order, that is mean use small quantization steps (step size) to encode small differences and use large quantization to encode large difference values. Moreover, it uses

previous sample values to estimate the next input sample value. So, the difference between the actual value and the predicted value is minimum.

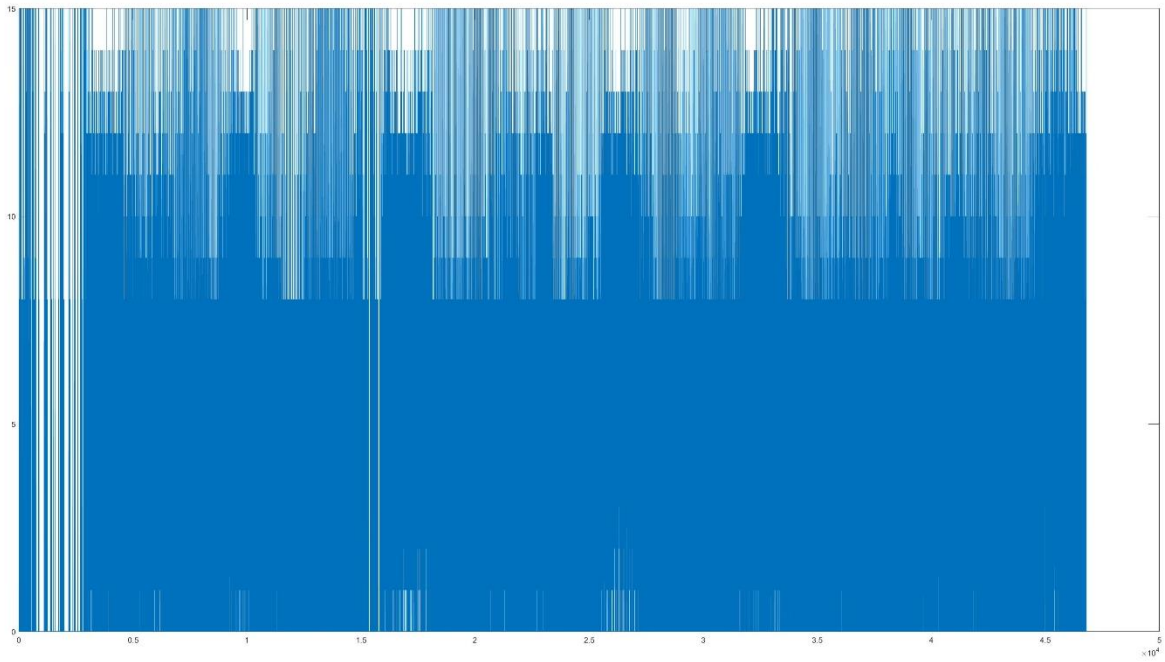

Figure 8. IMA ADPCM encoder encodes the input data into only 16 levels (4-bit format). The y-axis is from 0 to 16, the x-axis is the speech signal.

By using IMA ADPCM encoder, the 16-bit data samples encoded as 4-bit differences result in 4:1 compression format. Like the figure 8. Hence the data can be transferred faster.

Method

To complete the project, I use MATLAB to answer all the questions in the final project.

1. Read TIMIT data file into a vector x.

   By using MATLAB function “audioread”, we can read the TIMIT speech as a vector x.

```
[x,Fs] = audioread('LDC93S1.wav');
N = length(x);
plot(x);
pause;
Fs = 16000;
```

2. Apply u-Law compression to the data x that is quantized in 16 bits, and convert to an 8-bit data y.

   To solve question, need three steps in *Figure 1*.

   Step 1: Quantize data into 16 bits.

   Using uencode() and udecode() functions in MATLAB. Because the uencode() do a uniform quantization and encoding, the and udecode() only do uniform decoding. After using the two functions, we can get the uniform quantized data. However, in this speech file, it tells us the data that is already quantized in 16 bits. So, the result is almost same if we do not do the 16 bits quantization before. The waveform is almost same, but the sound is different. Because the quantization map floating-point scalars in [-1, 1] to uint8 (unsigned) integers, and produce a staircase plot. Here is the code:

```
% uencode(x,16) Uniform quantization and encoding of
the input into 16-bits. In this file, uencode and
udecode 16 step may omit.
a = uencode(x,16);
plot(a);
pause;
% udecode(x,16) Uniform decoding of the input.
b = udecode(a,16);
plot(b);
pause;
```

Step 2: Apply μ-Law compression.

The function “compand” can use mu-law or A-law compressor or expander to the data.

OUT = compand(IN, PARAM, V, METHOD) computes mu-law or A-law compressor or expander computation with the computation method given in METHOD. PARAM provides the mu or A value, usually use 255. V provides the input signal peak magnitude.

METHOD = ‘mu/compressor’ mu-law compressor.

METHOD = ‘mu/expander’ mu-law expander.

METHOD = ‘A/compressor’ A-law compressor.

METHOD = ‘A/expander’ A-law expander.

Here is the code:

```
% compand() Source code mu-law compressor or expander.
c = compand(b,255,max(abs(b)),'mu/compressor');
plot(c);
pause;
```

Step 3: Quantize and convert data into 8 bits.

I used the function “uencode” and “udecode” again which are explained before. And convert the data into 8 bits. The data need quantization again, because u-law compress function is uneven. It will change the y-axis interval size. But when we use ADPCM, we need the same interval of the y-axis.

```
a1 = uencode(c,8);
plot(a1);
pause;
y = udecode(a1,8);
plot(y);
pause
```

3. Implement the IMA ADPCM algorithm.

The algorithm is function adpcm_encoder and adpcm_decoder. Here is the code:

```
function adpcm_y = adpcm_encoder(raw_y)

% This m-file is based on the app note: AN643, Adaptive differential pulse
% code modulation using PICmicro microcontrollers, Microchip Technology
% Inc. The app note is avaialbe from www.microchip.com
% Example:  Y = wavread('test.wav');
%           y = adpcm_encoder(Y);
%           YY = adpcm_decode(y);

IndexTable = [-1, -1, -1, -1, 2, 4, 6, 8, -1, -1, -1, -1, 2, 4, 6, 8];

StepSizeTable = [7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 19, 21, 23, 25, 28, 31, 34, 37, 41, 45, 50, 55, 60, 66, 73,
80, 88, 97, 107, 118, 130, 143, 157, 173, 190, 209, 230, 253, 279, 307, 337, 371, 408, 449, 494, 544, 598, 658,
724, 796, 876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066, 2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428,
4871, 5358, 5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899, 15289, 16818, 18500, 20350, 22385,
24623, 27086, 29794, 32767];

prevsample = 0;
previndex = 1;

Ns = length(raw_y);
n = 1;

raw_y = 32767 * raw_y;          % 16-bit operation

while (n <= Ns)
    predsample = prevsample;
    index = previndex;
    step = StepSizeTable(index);

    diff = raw_y(n) - predsample;
    if (diff >= 0)
        code = 0;
    else
        code = 8;
        diff = -diff;
    end

    tempstep = step;
    if (diff >= tempstep)
        code = bitor(code, 4);
        diff = diff - tempstep;
    end
    tempstep = bitshift(tempstep, -1);
    if (diff >= tempstep)
        code = bitor(code, 2);
        diff = diff - tempstep;
    end
    tempstep = bitshift(tempstep, -1);
    if (diff >= tempstep)
        code = bitor(code, 1);
    end

    diffq = bitshift(step, -3);
    if (bitand(code, 4))
        diffq = diffq + step;
    end
    if (bitand(code, 2))
        diffq = diffq + bitshift(step, -1);
    end
    if (bitand(code, 1))
        diffq = diffq + bitshift(step, -2);
    end

    if (bitand(code, 8))
        predsample = predsample - diffq;
    else
        predsample = predsample + diffq;
    end

    if (predsample > 32767)
        predsample = 32767;
    elseif (predsample < -32768)
        predsample = -32768;
    end

    index = index + IndexTable(code+1);

    if (index < 1)
        index = 1;
    end
    if (index > 89)
        index = 89;
    end

    prevsample = predsample;
    previndex = index;

    adpcm_y(n) = bitand(code, 15);
    %adpcm_y(n) = code;
    n = n + 1;
end
```

```
function raw_y = adpcm_decoder(adpcm_y)

% This m-file is based on the app note: AN643, Adaptive differential pulse
% code modulation using PICmicro microcontrollers, Microchip Technology
% Inc. The app note is avaialbe from www.microchip.com
% Example:  Y = wavread('test.wav');
%           y = adpcm_encoder(Y);
%           YY = adpcm_decode(y);
IndexTable = [-1, -1, -1, -1, 2, 4, 6, 8, -1, -1, -1, -1, 2, 4, 6, 8];

StepSizeTable = [7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 19, 21, 23, 25, 28, 31, 34, 37, 41, 45, 50, 55, 60, 66, 73,
80, 88, 97, 107, 118, 130, 143, 157, 173, 190, 209, 230, 253, 279, 307, 337, 371, 408, 449, 494, 544, 598, 658,
724, 796, 876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066, 2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428,
4871, 5358, 5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899, 15289, 16818, 18500, 20350, 22385,
24623, 27086, 29794, 32767];

prevsample = 0;
previndex = 1;

Ns = length(adpcm_y);
n = 1;

while (n <= Ns)
    predsample = prevsample;
    index = previndex;
    step = StepSizeTable(index);
    code = adpcm_y(n);

    diffq = bitshift(step, -3);
    if (bitand(code, 4))
        diffq = diffq + step;
    end
    if (bitand(code, 2))
        diffq = diffq + bitshift(step, -1);
    end
    if (bitand(code, 1))
        diffq = diffq + bitshift(step, -2);
    end

    if (bitand(code, 8))
        predsample = predsample - diffq;
    else
        predsample = predsample + diffq;
    end

    if (predsample > 32767)
        predsample = 32767;
    elseif (predsample < -32768)
        predsample = -32768;
    end

    index = index + IndexTable(code+1);

    if (index < 1)
        index = 1;
    end
    if (index > 89)
        index = 89;
    end

    prevsample = predsample;
    previndex = index;

    raw_y(n) = predsample / 32767;
    n = n + 1;
end
```

Figure 9. The code of ADPCM encoder and ADPCM decoder.

4. Apply the ADPCM Encoder to compressed data y, and obtain encoded data z.

   Using the ADPCM encoder code in part 3.

```
z = adpcm_encoder(y);
plot(z);
pause;
```

5. Apply the ADPCM Decoder to the encoded data z and obtain decoded data y'.

   Using the ADPCM decoder code in part 3.

```
y1 = adpcm_decoder(z);
plot(y1);
pause;
```

6. Apply u-Law expansion to the decoded data y' and obtain the reconstructed speech x'.

7. Play and plot the reconstructed speech x'.

   I will show the part 6 and 7 results together by using MATLAB code.

```
x1 = compand(y1,255,max(abs(y1)),'mu/expander');
plot(x1);
pause;
sound(x1,Fs);
pause;
% wirte a file to the record result sound.
audiowrite('reconstructedx1.wav',x1,Fs);
```

   Using the compand() function to expand the data, and write the reconstructed audio in a new file. Then we can compare the original signal and the reconstructed speech.

Results

1. **Read TIMIT data file into a vector x.**

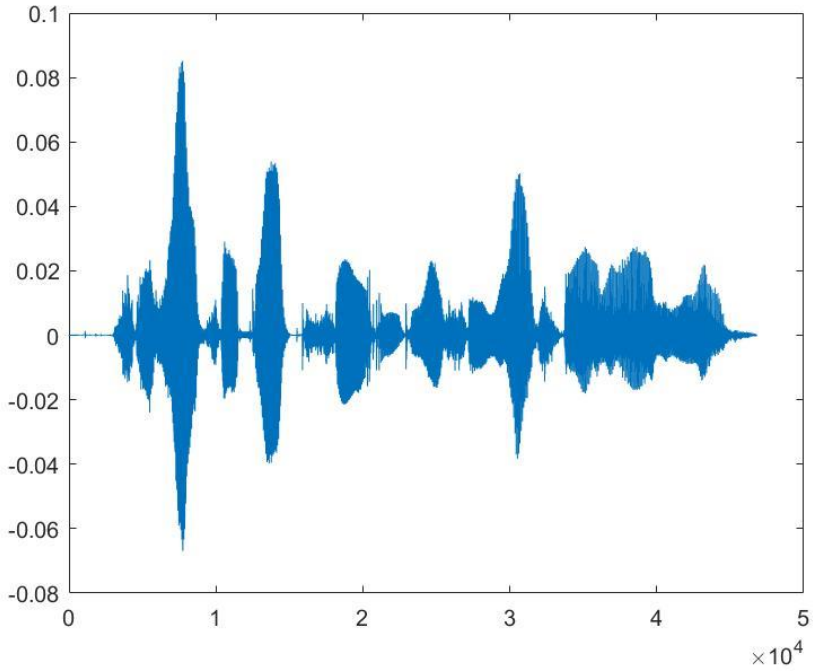

Figure 10. The TIMIT original speech.

2. **Apply u-Law compression to the data x that is quantized in 16 bits, and convert to an 8-bit data y.**

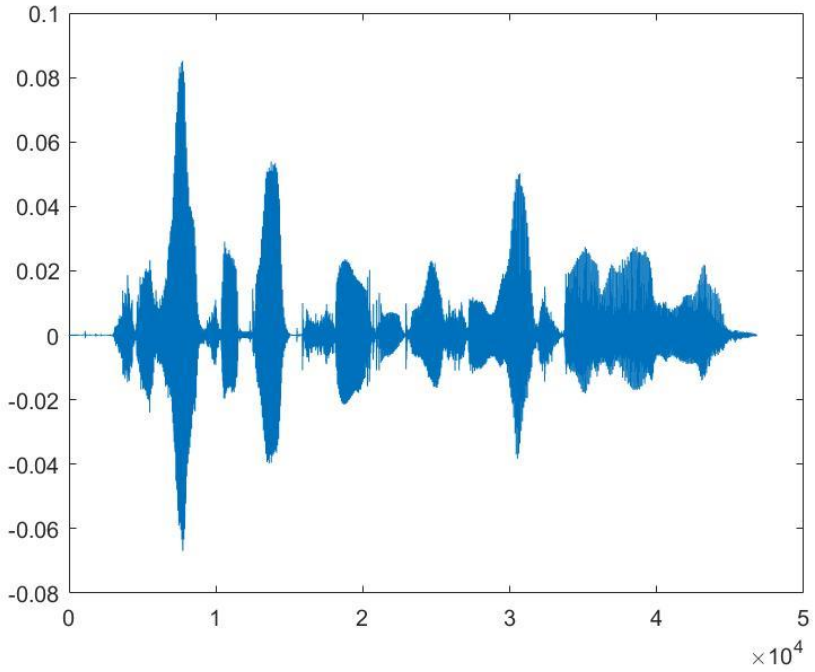

Figure 11. Convert the data into 16 bits.

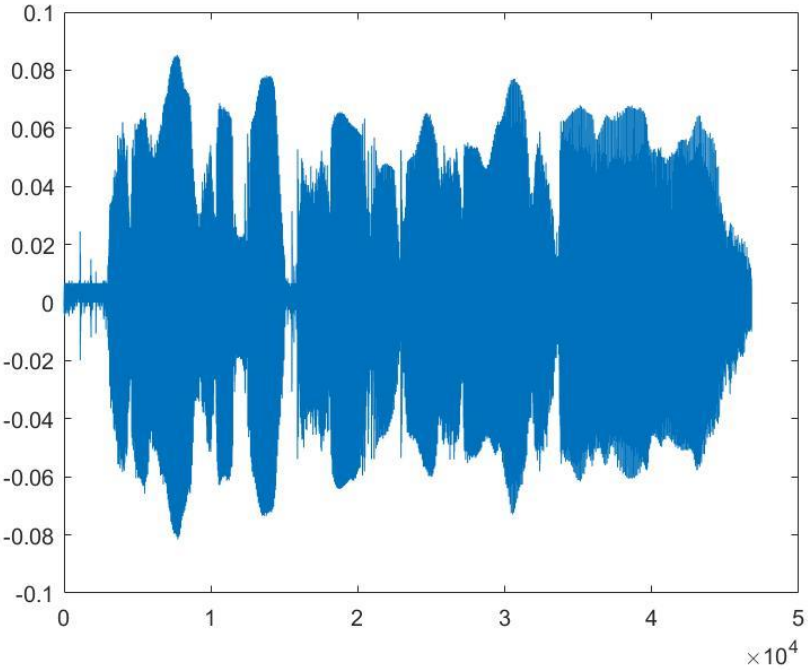

Figure 12. Apply u-Law compression to the data.

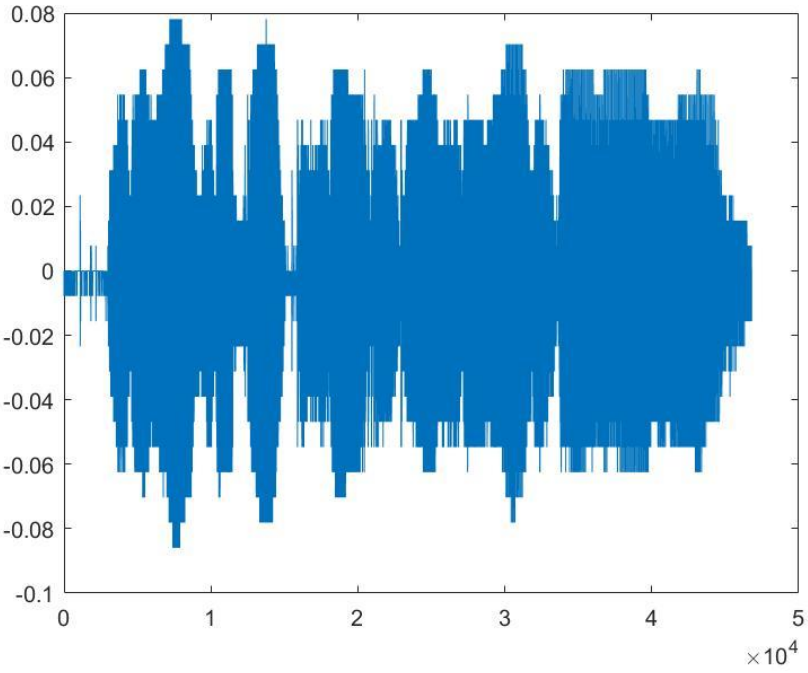

Figure 13. Convert the data into 8 bits.

3. **Apply the ADPCM Encoder to compressed data y, and obtain encoded data z.**

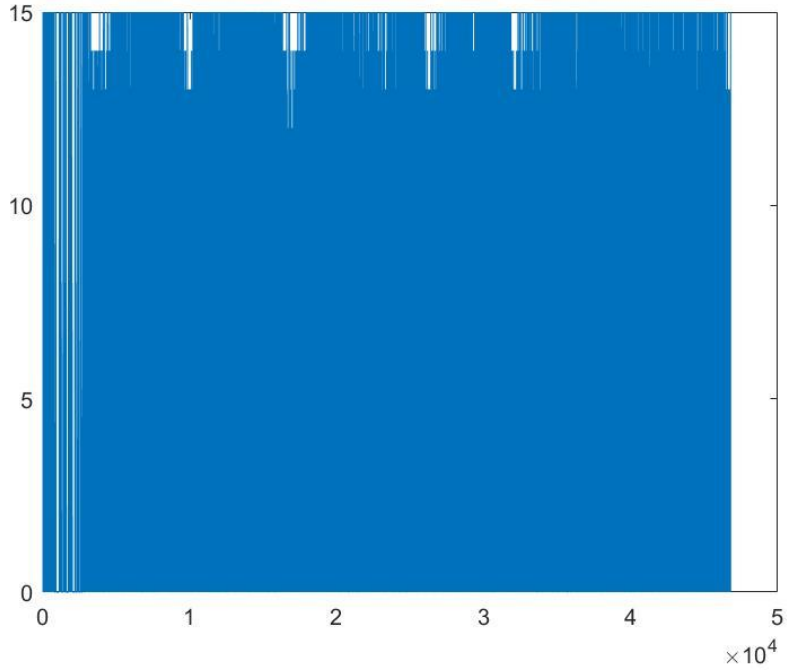

Figure 14. Apply the ADPCM Encoder to compressed data

4. **Apply the ADPCM Decoder to the encoded data z and obtain decoded data y'.**

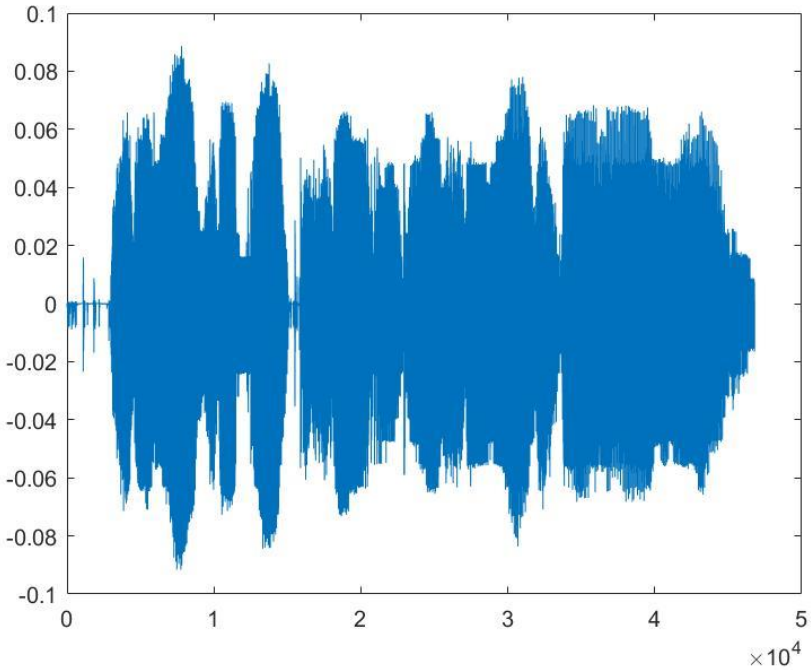

Figure 15. Apply the ADPCM Decoder to the encoded data

5. **& 6. Apply u-Law expansion to the decoded data y' and obtain the reconstructed speech x'. Play and plot the reconstructed speech x'.**

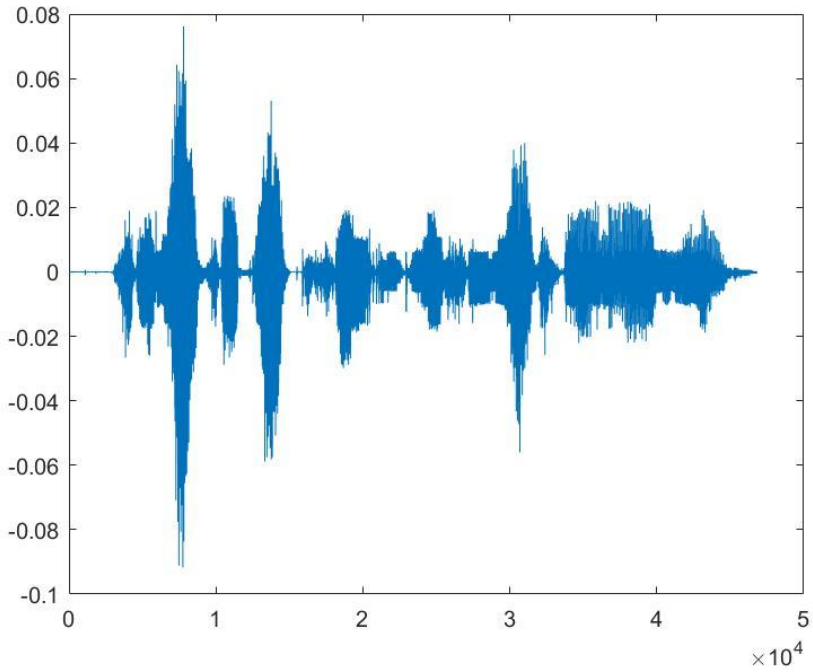

Figure 16. The reconstructed speech.

Result Analysis

From the result of the reconstructed speech, comparing with the original speech, it is similar with the original waveform, but they are not the same. The amplitude of the two audios has a little different.

The sound in reconstructed speech has some noise. Although I can hear every word in it, however, it is not clearly like the original speech. The noise comes mainly from two aspects.

1. The quantization noise is caused by the quantization of amplitude and time. During the conversion from 16 bits to 8 bits, the uniformly quantized data is converted from a floating point to an integer. Although we used the non-uniform μ-Law algorithm, the SNR is improved. However, the quantization distortion still exists. The quantization noise is like the Figure 17.

2. ADPCM is a lossy compression algorithm. The noise will be generated after using it.

To solve the noise problem, we need to find better quantization and compression algorithms.

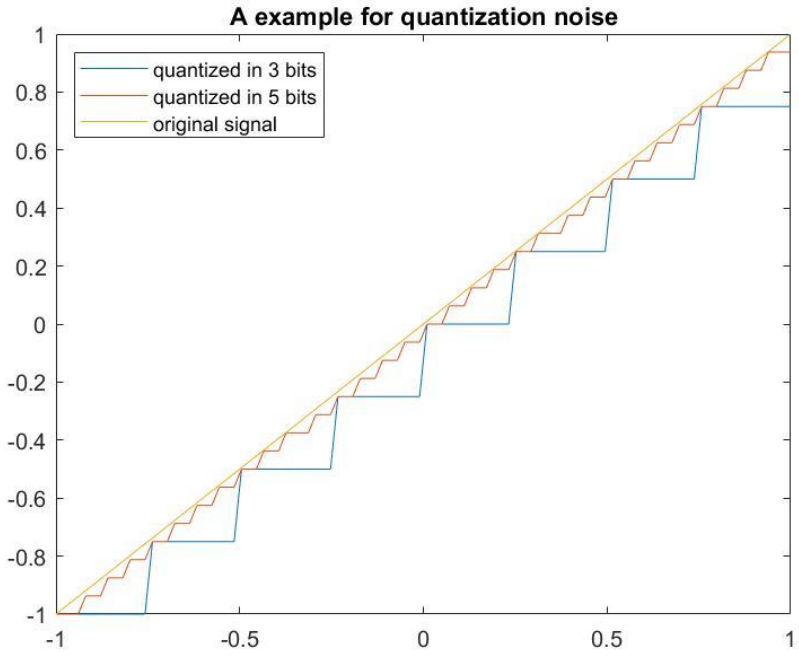

Figure 17. An example for quantization noise.

Acknowledgement

This is the second time I have attended the professor Kie B. Eom’s class, he is a very nice man. In this semester, I have learned much more from him. I did much better than before because of the improvement of ability and the experience from previous semester. Without his course, I don't even know how to write MATLAB code, I’m so appreciate to him. Moreover, I also would like to thank my classmates for the academic discussion with them.

References

[1] IMA Digital Audio Focus and Technical Working Groups. *Recommended Practices for Enhancing Digital Audio Compatibility in Multimedia Systems*, vol. 3, pp. 3-5, October 21, 1992.

[2] Sklar, Bernard. *Digital Communications: Fundamentals and Applications*. Englewood Cliffs, N.J.: Prentice-Hall, 1988.

[3] Cisco, *Waveform Coding Techniques*, Document ID:8123, February 2, 2006