ECE 438 - Laboratory 8
Number Representation and Waveform Quantization

Last Updated on May 5, 2022

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import soundfile as sf
import IPython.display as ipd
from helper import xcorr, lloyds
```

In [2]:
```
# make sure the plot is displayed in this notebook
%matplotlib inline
# specify the size of the plot
plt.rcParams['figure.figsize'] = (16, 6)

# for auto-reloading extenrnal modules
%load_ext autoreload
%autoreload 2
```

1. Introduction

This lab presents two important concepts for working with digital signals. The first section discusses how numbers are stored in memory. Numbers may be either in fixed point or floating point format. Integers are often represented with fixed point format. Decimals, and numbers that may take on a very large range of values would use floating point. The second issue of numeric storage is quantization. All analog signals that are processed on the computer must first be quantized. We will examine the errors that arise from this operation, and determine how different levels of quantization affect a signal's quality. We will also look at two types of quantizers. The **uniform quantizer** is the simpler of the two. The **max quantizer** is optimal, in that it minimizes the mean square error between the original and quantized signals.

2. Review of Number Representations

There are two types of numbers that a computer can represent: integers and decimals. These two numbers are stored quite differently in memory. Integers (e.g., 27, 0, -986) are usually stored in fixed point format, while decimals (e.g., 12.34, -0.98) most often use floating point format. Most integer representations use four bytes of memory; floating point values usually require eight.

There are different conventions for encoding fixed point binary numbers because of the different ways of representing negative numbers. Three types of fixed point formats that accommodate negative integers are **sign-magnitude**, **one's-complement**, and **two's-complement**. In all three of these "signed" formats, the first bit denotes the sign of the number: 0 for positive, and 1 for negative. For positive numbers, the magnitude simply follows the first bit. Negative numbers are handled differently for each format.

Of course, there is also an **unsigned** data type which can be used when the numbers are known to be non-negative. This allows a greater range of possible numbers since a bit isn't wasted on the negative sign.

2.1. Sign-magnitude Representation

Sign-magnitude notation is the simplest way to represent negative numbers. The magnitude of the negative number follows the first bit. If an integer was stored as one byte, the range of possible numbers would be -127 to 127.

The value $+27$ would be represented as

$$0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1$$

The value -27 would be represented as

$$1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1$$

2.2. One's-complement

To represent a negative number, the complement of the bits for the positive number with the same magnitude are computed. The positive number 27 in one's-complement form would be written as

$$0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1$$

but the value -27 would represented as

$$1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0$$

2.3. Two's-complement

The problem with each of the above notations is that two different values represent zero Two's-complement notation is a revision to one's-complement that solves this problem. To form negative numbers, the positive number is subtracted from a certain binary number. This number has a one in the most significant bit (MSB), followed by as many zeros as there are bits in the representation. If 27 was represented by an eight-bit integer, -27 would be represented as:

$$\begin{array}{rcccccccc} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ - & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ = & & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \end{array}$$

Notice that this result is one plus the one's-complement representation for -27 (modulo-2 addition). What about the second value of 0? That representation is

$$\begin{array}{cccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

This value equals -128 in two's-complement notation!

$$\begin{array}{rcccccccc} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ - & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ = & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

The value represented here is -128; we know it is negative, because the result has a 1 in the MSB. Two's-complement is used because it can represent one extra negative value. More importantly, if the sum of a series of two's-complement numbers is within the range, overflows that occur during the summation will not affect the final answer! The range of an 8-bit two's complement integer is $[-128, 127]$.

2.4. Floating Point

Floating point notation is used to represent a much wider range of numbers. The tradeoff is that the resolution is variable: it decreases as the magnitude of the number increases. In the fixed point examples above, the resolution was fixed at 1. It is possible to represent decimals with fixed point notation, but for a fixed word length any increase in resolution is matched by a decrease in the range of possible values.

A floating point number, F, has two parts: a **mantissa**, M, and an **exponent**, E.

$$F = M * 2^E$$

The mantissa is a signed fraction, which has a power of two in the denominator. The exponent is a signed integer, which represents the power of two that the mantissa must be multiplied by. These signed numbers may be represented with any of the three fixed-point number formats. The IEEE has a standard for floating point numbers (IEEE 754). For a 32-bit number, the first bit is the mantissa's sign. The exponent takes up the next 8 bits (1 for the sign, 7 for the quantity), and the mantissa is contained in the remaining 23 bits. The range of values for this number is $(-1.18 \times 10^{-38}, 3.40 \times 10^{38})$.

To add two floating point numbers, the exponents must be the same. If the exponents are different, the mantissa is adjusted until the exponents match. If a very small number is added to a large one, the result may be the same as the large number! For instance, if $0.15600 \cdots 0 \times 2^{30}$ is added to $0.62500 \cdots 0 \times 2^{-3}$, the second number would be converted to $0.0000 \cdots 0 \times 2^{30}$ before addition. Since the mantissa only holds 23 binary digits, the decimal digits 625 would be lost in the conversion. In short, the second number is rounded down to zero. For multiplication, the two exponents are added and the mantissas multiplied.

3. Quantization

3.1. Introduction

Quantization is the act of rounding off the value of a signal or quantity to certain discrete levels. For example, digital scales may round off weight to the nearest gram. Analog voltage signals in a control system may be rounded off to the nearest volt before they enter a digital controller. Generally, all numbers need to be quantized before they can be represented in a computer.

Digital images are also quantized. The gray levels in a black and white photograph must be quantized in order to store an image in a computer. The "brightness" of the photo at each pixel is assigned an integer value between 0 and 255 (typically), where 0 corresponds to black, and 255 to white. Since an 8-bit number can represent 256 different values, such an image is called an "8-bit grayscale" image. An image which is quantized to just 1 bit per pixel (in other words only black and white pixels) is called a halftone image. Many printers work by placing, or not placing, a spot of colorant on the paper at each point. To accommodate this, an image must be halftoned before it is printed.

Quantization can be thought of as a functional mapping $y = f(x)$ of a real-valued input to a discrete-valued output. An example of a quantization function is shown in Figure 1, where the x-axis is the input value, and the y-axis is the quantized output value.

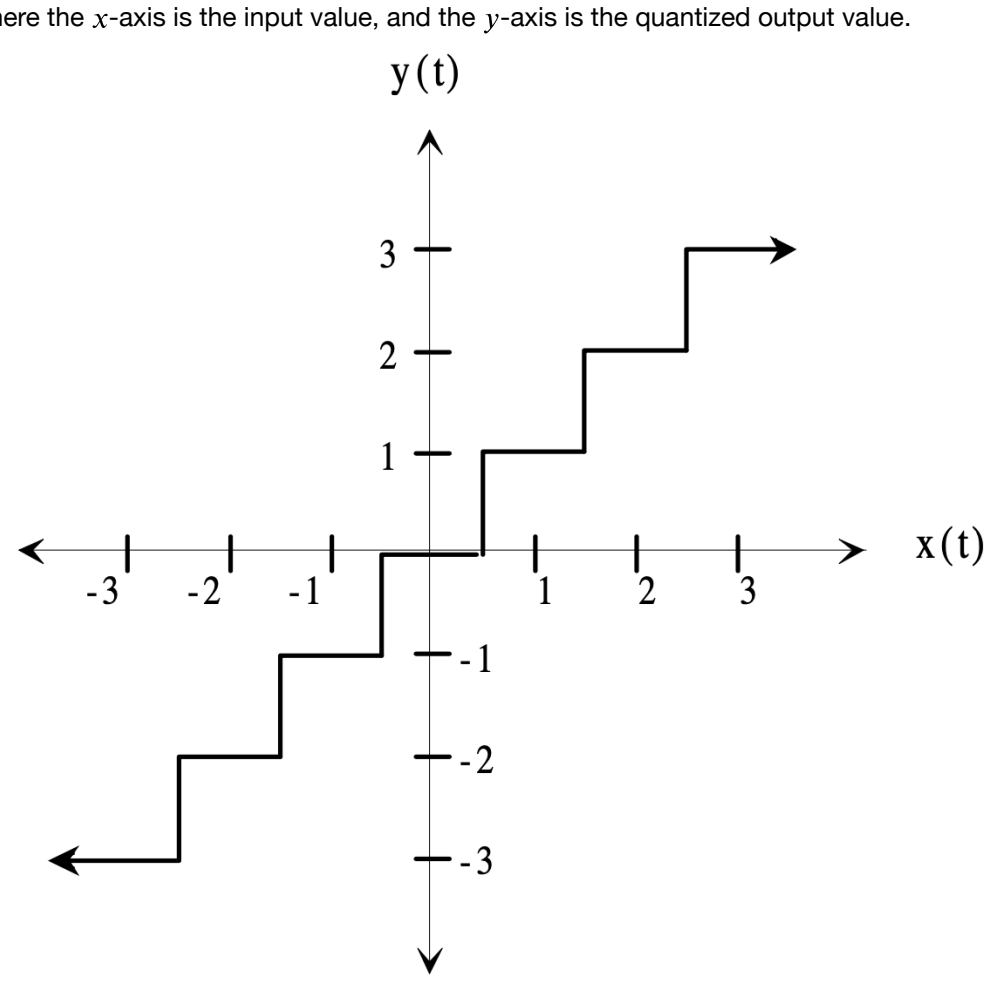

Figure 1: Input-output relation for a 7-level uniform quantizer

3.2. Quantization and Compression

Quantization is sometimes used for compression. As an example, suppose we have a digital image which is represented by 8 different gray levels: $[0, 31, 63, 95, 159, 191, 223, 255]$. To directly store each of the image values, we need at least 8-bits for each pixel since the values range from 0 to 255. However, since the image only takes on 8 different values, we can assign a different 3-bit integer (a code) to represent each pixel: $[000, 001, \ldots, 111]$. Then, instead of storing the actual gray levels, we can store the 3-bit code for each pixel. A look-up table, possibly stored at the beginning of the file, would be used to decode the image. This lowers the cost of an image considerably: less hard drive space is needed, and less bandwidth is required to transmit the image (i.e. it downloads quicker). In practice, there are much more sophisticated methods of compressing images which rely on quantization.

Exercise 3.3: Image Quantization

In the current folder, there is an image file named `fountainbw.tif`, which contains an 8-bit grayscale image. We will investigate what happens when we quantize it to fewer bits per pixel (b/pel).

1. Load the image and display it using the following sequence of commands.

```
image = plt.imread("fountainbw.tif")
plt.imshow(image, cmap='gray', vmin=0, vmax=255)
plt.axis('image')
plt.show()
```

In [3]:
```
# insert your code here
```

The image array will initially be of type *uint8*, so you will need to convert the image matrix to type `float` before performing any computation. Use the following command for this:

```
image = image.astype(float)
```

2. Print the data type of this image, then convert the image matrix to type `float`, and print the data type of this image again.

- Use `image.dtype` to get the data type of `image`.

In [4]:
```
# insert your code here
```

There is an easy way to uniformly quantize a signal. Let

$$\Delta = \frac{\max X - \min X}{N - 1}$$

where X is the signal to be quantized, and N is the number of quantization levels. To force data to have a uniform quantization step of Δ,

- subtract $\min X$ from the data and divide the result by Δ.
- round the data to the nearest integer.
- multiply the rounded data by Δ and add $\min X$ to convert the data back to its original scale.

3. Complete the function below which will uniformly quantize an input array `x` (either a vector or a matrix) to an `numBits`-bit array.

In [5]:
```
def Uquant(X, numBits):
    """
    Parameters
    ---
    X: the input array to be quantized
    numBits: the number of bits. The number of quantization levels will be 2^numBits.

    Returns
    ---
    Y: the quantized array
    """

    Y = None
    return Y
```

4. Use this function to quantize the fountain image to $7, 6, 5, 4, 3, 2, 1$ b/pel, and display and observe the output images. Don't forget the titles of the images.

- To display a grayscale image `image`, use the following commands:

```
plt.imshow(image.astype(np.uint8), cmap='gray', vmin=0, vmax=255)
plt.title("title")
plt.show()
```

In [6]:
```
# insert your code here
```

5. Describe the artifacts (errors) that appear in the image as the number of bits is lowered.

insert your answer here

6. Note the number of b/pel at which the image quality noticeably deteriorates.

insert your answer here

7. Compare each of four quantized images (7, 4, 2 and 1 b/per) to the original.

insert your answer here

Exercise 3.4: Audio Quantization

If an audio signal is to be coded, either for compression or for digital transmission, it must undergo some form of quantization. Most often, a general technique known as **vector quantization** is employed for this task, but this technique must be tailored to the specific application so it will not be addressed here. In this exercise, we will observe the effect of uniformly quantizing the samples of two audio signals.

1. Use your function `Uquant()` to quantize each of these signals: `speech.au` and `music.au` to 7, 4, 2 and 1 bits/sample. Listen to the original and quantized signals.

- To read an audio file:

```
speech, fs = sf.read("speech.au")  # speech is the signal vector, and fs is the sampling frequency
```

- To play a signal

```
ipd.Audio(speech, rate=fs)
```

In [7]:
```
# insert your code here
```

2. For each signal, describe the change in quality as the number of b/sample is reduced.

insert your answer here

3. For each signal, is there a point at which the signal quality deteriorates drastically? At what point (if any) does it become incomprehensible?

insert your answer here

4. Which signal's quality deteriorates faster as the number of levels decreases?

insert your answer here

5. Do you think 4 b/sample is acceptable for telephone systems? What about 2 b/sample?

insert your answer here

6. Plot the four quantized `speech` signals over the index range $[7200 : 7400)$. Generate a similar figure for the `music` signal, using the same indices.

In [8]:
```
# insert your code here
```

Exercise 3.5. Error Analysis

As we have clearly observed, quantization produces errors in a signal. The most effective methods for analysis of the error turn out to be probabilistic. In order to apply these methods, however, one needs to have a clear understanding of the error signal's statistical properties. For example, can we assume that the error signal is white noise? Can we assume that it is uncorrelated with the quantized signal? As you will see in this exercise, both of these are good assumptions if the quantization intervals are small compared with sample-to-sample variations in the signal.

If the original signal is X, and the quantized signal is Y, the error signal is defined by the following:

$$E = Y - X$$

1. Compute the error signal for the quantized speech for 7, 4, 2 and 1 b/sample.

In [9]:
```
# insert your code here
```

When the spacing, Δ, between quantization levels is sufficiently small, a common statistical model for the error is a uniform distribution from $-\frac{\Delta}{2}$ to $\frac{\Delta}{2}$.

2. Use the command `plt.hist(E, bins=20)` (https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.hist.html) to generate 20-bin histograms for each of the four error signals.

In [10]:
```
# insert your code here
```

3. How does the number of quantization levels seem to affect the shape of the distribution?

insert your answer here

4. Explain why the error histograms you obtain might not be uniform?

insert your answer here

Next we will examine correlation properties of the error signal.

5. Compute and plot an estimate of the autocorrelation function for each of the four error signals using the following commands:

```
lags, r = xcorr(E, maxlags=200)
plt.plot(lags, r)
plt.show()
```

Hint: function `xcorr` is provided in the file `helper.py`

In [11]:
```
# insert your code here
```

6. Now compute and plot an estimate of the cross-correlation function between the quantized speech Y and each error signal E using

```
lags, r = xcorr(E, Y, maxlags=200)
plt.plot(lags, r)
plt.show()
```

Hint: function `xcorr` is provided in the file `helper.py`

In [12]:
```
# insert your code here
```

7. Is the autocorrelation influenced by the number of quantization levels? Do samples in the error signal appear to be correlated with each other?

insert your answer here

8. Does the number of quantization levels influence the cross-correlation?

insert your answer here

Exercise 3.6: Signal to Noise Ratio

One way to measure the quality of a quantized signal is by the Power Signal-to-Noise Ratio (PSNR). This is defined by the ratio of the power in the quantized speech to power in the noise.

$$\text{PSNR} = \frac{P_Y}{P_E}$$

In this expression, the noise is the error signal E. Generally, this means that a higher PSNR implies a less noisy signal.

From previous labs we know the power of a sampled signal, $x[n]$, is defined by

$$P_x = \frac{1}{L}\sum_{n=1}^{L} x^2[n]$$

where L is the length of $x[n]$.

1. Complete the function below that calculates the power of a sampled signal `x`.

In [13]:
```
def get_power(x):
    """
    Parameters
    ---
    x: the input signal

    Returns
    ---
    P: the power of the signal
    """

    P = None
    return P
```

2. Compute the PSNR for the four quantized speech signals from the previous section.

In [14]:
```
# insert your code here
```

In evaluating quantization (or compression) algorithms, a graph called a “rate-distortion curve” is often used. This curve plots **signal distortion vs. bit rate**. Here, we can measure the distortion by $\frac{1}{\text{PSNR}}$, and determine the bit rate from the number of quantization levels and sampling rate. For example, if the sampling rate is 8000 samples/sec, and we are using 7 bits/sample, the bit rate is 56 kilobits/sec (kbps).

3. Assuming that the speech is sampled at 8kHz, plot the rate distortion curve using $\frac{1}{\text{PSNR}}$ as the measure of distortion. Generate this curve by computing the PSNR for $7, 6, 5, \ldots, 1$ bits/sample. Make sure the axes of the graph are in terms of *distortion* and *bit rate*.

In [15]:
```
# insert your code here
```

3.7. Max Quantizer

In this section, we will investigate a different type of quantizer which produces less noise for a fixed number of quantization levels. As an example, let’s assume the input range for our signal is $[-1, 1]$, but most of the input signal takes on values between $[-0.2, 0.2]$. If we place more of the quantization levels closer to zero, we can lower the average error due to quantization.

A common measure of quantization error is mean squared error (noise power). The **Max quantizer** is designed to minimize the mean squared error for a given set of training data. We will study how the Max quantizer works, and compare its performance to that of the uniform quantizer which was used in the previous sections.

Derivation

The Max quantizer determines quantization levels based on a data set’s probability density function, $f(x)$, and the number of desired levels, N. It minimizes the mean squared error between the original and quantized signals:

$$\varepsilon = \sum_{k=1}^{N} \int_{x_k}^{x_{k+1}} (q_k - x)^2 f(x) dx$$

where q_k is the k^{th} quantization level, and x_k is the lower boundary for q_k. The error ε depends on both q_k and x_k. (Note that for the Gaussian distribution, $x_1 = -\infty$, and $x_{N+1} = \infty$.) To minimize ε with respect to q_k, we must take $\frac{\partial \varepsilon}{\partial q_k} = 0$ and solve for q_k:

$$q_k = \frac{\int_{x_k}^{x_{k+1}} x f(x) dx}{\int_{x_k}^{x_{k+1}} f(x) dx} \tag{1}$$

We still need the quantization boundaries, x_k. Solving $\frac{\partial \varepsilon}{\partial x_k} = 0$ yields

$$x_k = \frac{q_{k-1} + q_k}{2}$$

This means that each non-infinite boundary is exactly halfway in between the two adjacent quantization levels, and that each quantization level is at the centroid of its region. Figure 2 shows a five-level quantizer for a Gaussian distributed signal. Note that the levels are closer together in areas of higher probability.

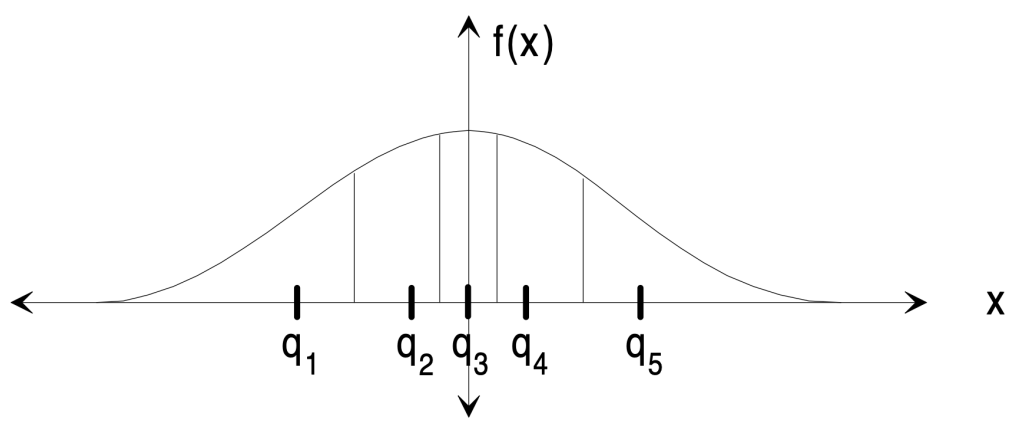

Figure 2: Five level Max quantizer for Gaussian-distributed signal

Implementation, Error Analysis and Comparison

In this section we will write code to compute an optimal quantizer, and compare its performance to the uniform quantizer. Since we almost never know the actual probability density function of the data that the quantizer will be applied to, we cannot use equation (1) to compute the optimal quantization levels. Therefore, a numerical optimization procedure is used on a training set of data to compute the quantization levels and boundaries which yield the smallest possible error for that set.

A Python function `lloyds()` has been provided in the file `helper.py` that performs this optimization. It's syntax is

```
partition, codebook = lloyds(training_set, initial_codebook)
```

This function requires two inputs. The first is the training dataset `training_set`, from which it will estimate the probability density function. The second is a vector `initial_codebook` containing an initial guess of the optimal quantization levels. It returns the quantized signal `quantized_signal`.

Since this algorithm minimizes the error with respect to the quantization levels, it is necessary to provice a decent initial guess of the codebook is necessary to help ensure a valid result. If the initial codebook is significantly "far" away from the optimal solution, it's possible that the optimization will get trapped in a local minimum, and the resultant codebook may perform quite poorly. In order to make a good guess, we may first estimate the shape of the probability density function of the training set using a histogram. The idea is to divide the histogram into equal "areas" and choose quantization levels as the centers of each of these segments.

Exercise 3.8

1. First plot a 40-bin histogram of this speech signal using `plt.hist(speech, bins=40)`, and make an initial guess of the four optimal quantization levels. Print out the histogram and the initial guess of the quantization levels.

In [16]:
```
# insert your code here
```

2. Use the function `lloyds()` to compute an optimal 4-level codebook using `speech.au` as the training set.

In [17]:
```
# insert your code here
```

3. Once the optimal codebook is obtained, use the `codebook` and `partition` vectors to quantize the speech signal.

- This may be done with a *for* loop and *if* statements.

In [18]:
```
# insert your code here
```

4. Compute the error signal and PSNR.

In [19]:
```
# insert your code here
```

5. Plot the histogram in Q1 again. However, on this histogram plot, also mark where the optimal quantization levels fall along the x-axis.

- To draw a vertical line, use `plt.axvline(x=0.8, color='r')` to plot a vertical line $x = 0.8$ of red color.

In [20]:
```
# insert your code here
```

6. Play the quantized audio, and compare the sound quality of the uniform- and max-quantized signals.

In [21]:
```
# insert your code here
```

7. If the speech signal was uniformly distributed, would the two quantizers be the same? Explain your answer.

insert your answer here