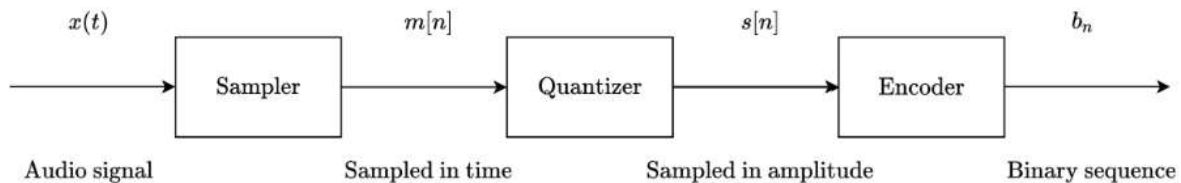


## LABORATORY 4: DIGITIZATION

HKU SPACE Community College, Fall 2022

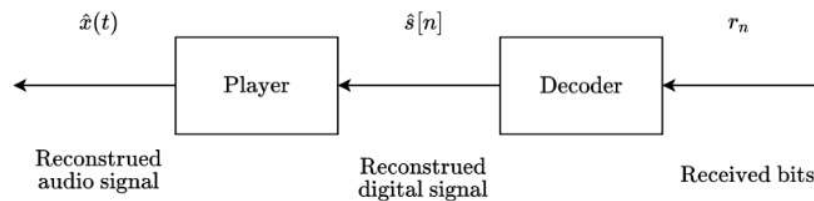
In this laboratory we are going to simulate the analog-to-digital conversion process. The system flow chart is as follows.



**Figure: System Flow Chart of an A/D Converter.**

Here,  $x(t)$  denotes the audio signal we intend to process,  $m[n]$  is the sampled version of  $x(t)$ , and  $s[n]$  is the quantized version of  $x[n]$ . Finally,  $b_n$  is the binary symbol stream which is to be stored in digital devices.

We will then model a noisy environment of transmission in Octave. The received bit streams  $r_n$  will then be decoded back in to  $s[n]$  and then we will play the audio  $x(t)$ . We shall compare the sound quality of the received message. As to be seen in the subsequent figure, it basically manifests an inverse operation of the A/D conversion.



**Figure: System Flow Chart of a D/A Converter.**

## 1 Digitization: A Three Steps Approach

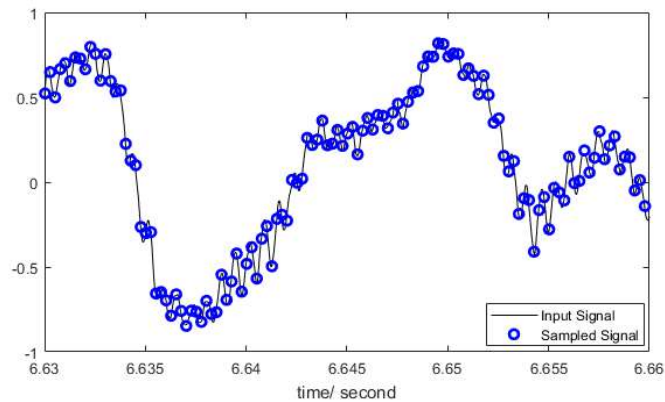
We begin by describing the functions given. `EIS_Sampling.m` is a function that converts a .wav file into a certain sampling frequency. For instance, if you read a .wav file and type:

```
>> [x, fs] = audioread('demo.wav');
>> sound(x, fs);
```

you will be able to hear the audio frame stored in the file `demo.wav`. You may now initiate the sampling process using a simple command line

```
>> fnew = 8000;
>> m = EIS_Sampling(x, fs, fnew);
```

to acquire the sampled signal  $m(n)$  with sampling frequency 4kHz — which is stored in the array  $m$ .

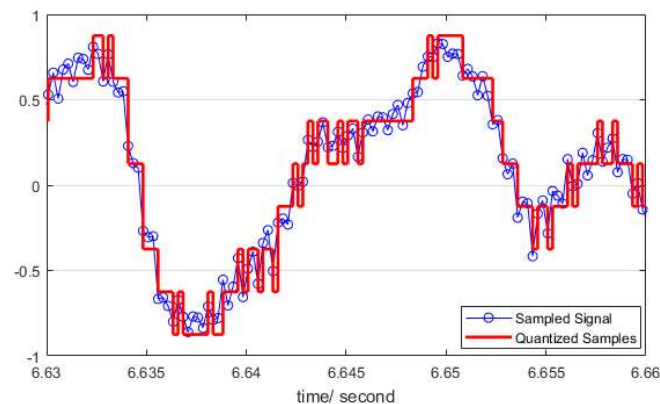


One may try a different sampling frequency, say for instance, 2kHz, 8kHz or 16kHz. The length of the resulting array  $m$  will vary with this value. Think: why? Also, what are the differences when you hear the sampled signals with different sampling rates?

Next, consider the quantizer function `EIS_Quant.m`. This is a function that quantises the sampled signal  $m[n]$  into a finite amount (i.e.  $M$  levels) of amplitudes. For instance, if you want to quantise a signal with  $M = 2$ , you may type:

```
>> M = 8;
>> [v, lv] = EIS_Quant(m, fnew, M);
```

to turn the sampled signal  $m[n]$  into the quantized samples  $s[n]$ . You will be able to see the quantized signal as the red line in the subsequent figure.

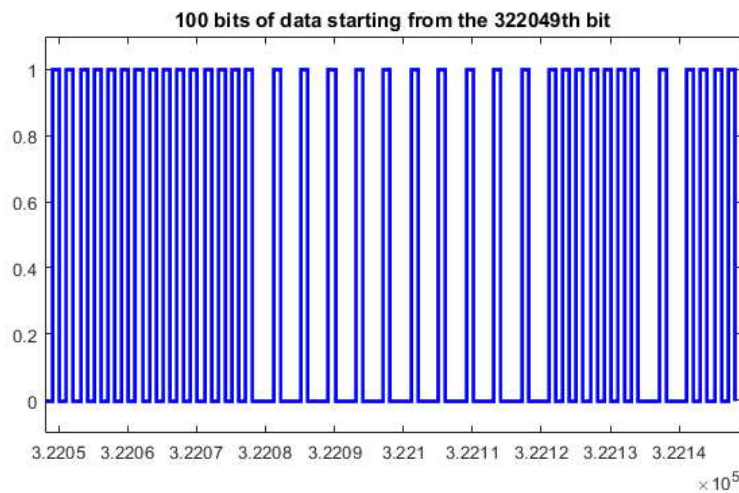


Note that you are allowed to hear the quantized samples using the command line `sound(m, fnew)` as usual. Note that the signal values of the sampled signal have many possibilities, but the quantized signal only takes a finite amount of value. These quantization levels, technically known as the *codebook*, are stored in the return argument `lv`. Try different values of `M` and see what are the effects on the quantized signals.

We now turn to the encoding part. The function `EIS_Encode.m` will be used to turn the quantized samples into binaries, and will automatically plot a sequence of 200 bits upon completion of binarization. The line of command is straight forward:

```
>> b = EIS_Encode(v, lv);
```

and the binary sequence  $b_n$  will be stored in the array `b`. The figure will be visualized as:



Observe the length of the binary sequence. Make a guess on what happens to it and verify it by re-executing the above commands with different sampling frequency and quantization levels.

## 2 Binary Transmission With Noise

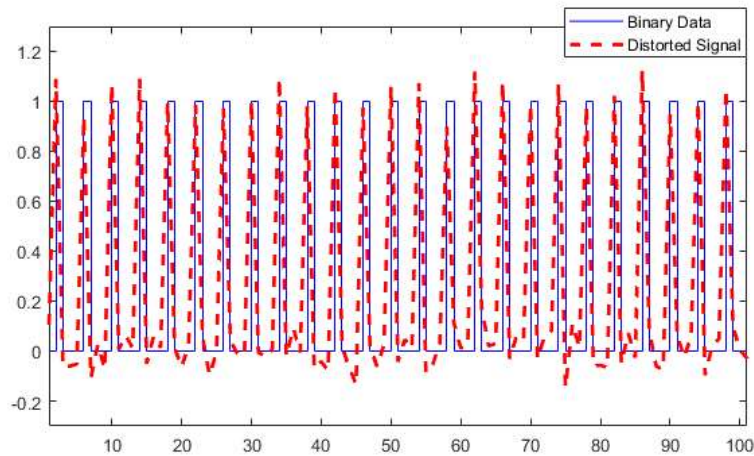
In many applications, transmission of signals are interfered with by the effect of noise. We try to simulate such an effect in this section. In reality, the effect of noise is completely random, i.e. we cannot precisely indicate a function that is a noise. In communication engineering, a popular choice of modeling noise is to use the additive white Gaussian noise (AWGN) assumption. Mathematically, it states a received sequence  $y_n$  can be modeled as

$$r_n = b_n + v_n$$

where  $b_n$  is a transmitted signal and  $v_n$  is a Gaussian random variable that indicates noise. To implement this, we will need to call the function `awgn` from the communication package.

```
>> y = awgn(b, SNR, 'measured');
```

Here, the input argument SNR is the preset signal-to-noise ratio in dB, specified as a scalar. The value of SNR dictates “how strong” the noise is being incurred. Below shows a plot of  $y_n$  versus  $b_n$  when SNR is fixed to 20dB.



Now that we have a receive signal  $y_n$  that is continuous-valued. All we need to do right now is to convert its values back into either zero or one. We can do it by the following line:

```
>> r = (y >= 1/2);
```

Here, the received bit stream  $r_n$  is returned by incurring the threshold level 0.5 on  $y_n$ , now stored in the binary array  $r$ . We can measure the number of bit error by:

```
>> bit_of_error = sum(r ~= b);
```

Normally speaking, the higher the signal-to-noise ratio, the smaller the bit of error we measure in this line. Finally, we need to decode the received bits  $r_n$  back to a continuous signal — so that we can hear the recovered message signal. We first decode  $r_n$  into an estimation of the quantised samples  $s[n]$  by using the function `EIS_Decode.m`:

```
>> s_hat = EIS_Decode(r, lv);
```

and thereby we will be able to listen to the received message by `sound(s_hat, fnew)` as usual.