School of Electrical Engineering and Computer Science

Communication Systems

8-PAM Sigma-Delta Modulation and Demodulation

Submitted to: Salman Ghafoor

Contents

INTRODUCTION:

In this project we needed to implement Sigma-Delta Modulation and demodulation on an 8- PAM radio frequency (RF). The file was already provided to us. We developed a system to implement SD Modulation and Demodulation using different functions and features of MATLAB.
An 8-PAM (8-level Pulse Amplitude Modulation) signal is a digital modulation scheme where the amplitude of the signal is varied to represent different digital symbols. In 8-PAM, eight different levels of amplitude can be used to encode data. Each of these levels can represent a unique combination of bits. 8-PAM is used in various communication systems where the goal is to transmit digital data efficiently over a communication channel while mitigating the effects of noise and interference. It provides higher spectral efficiency compared to simpler modulation schemes like binary modulation. Sigma-delta modulation (Σ-Δ modulation) (SDM) is a type of analogue-to-digital conversion (ADC) and digital-to-analogue conversion (DAC) technique commonly used in digital communication systems and analogue-to-digital converters (ADCs) for high-resolution applications. In single-bit sigma-delta modulation, the output is a single-bit data stream, typically a high-frequency stream of 1s and 0s. The technique involves oversampling the input signal, quantizing the difference between the input signal and the output signal, and feeding back the quantization error through a feedback loop. Single-bit sigma-delta modulators are widely used in audio applications, such as digital audio converters (DACs) and analog-to-digital converters (ADCs).

WORKING OF MATLAB CODE

i. Closing All Previous Files:

```
%clearing command window and closing previously opened graphs
clear;
clc;
close all;
```

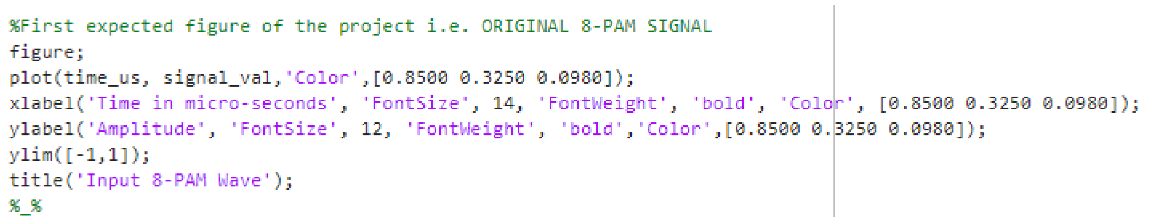
These commands are used to clear any previous memory, cleans space for new output and close all open figure windows.

ii. Opening and Reading Excel File:

```
%Reading Data from 8-PAM xlsx file%
original_signal = xlsread('8-PAM.xlsx');
time_us= original_signal(:,1);
signal_val= original_signal(:,2);
```

As we were provided with an excel file for the project, we are reading that file through **xlsread** function and stored it in variable **original_signal**. As the file had two columns one contained time information and the other column had electrical signal information. We stored the values of time column in the variable **time_us** from our previously made variable named original_signal by taking all of its rows and 1st column. The same goes for electrical signal value which is he second column of original_signal.

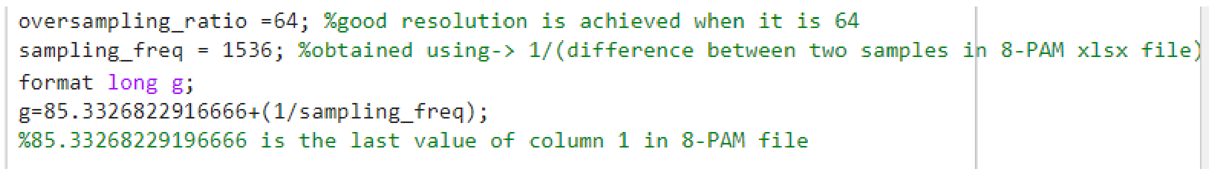
iii. Plotting the Excel File Signal:



```
%First expected figure of the project i.e. ORIGINAL 8-PAM SIGNAL
figure;
plot(time_us, signal_val,'Color',[0.8500 0.3250 0.0980]);
xlabel('Time in micro-seconds', 'FontSize', 14, 'FontWeight', 'bold', 'Color', [0.8500 0.3250 0.0980]);
ylabel('Amplitude', 'FontSize', 12, 'FontWeight', 'bold','Color',[0.8500 0.3250 0.0980]);
ylim([-1,1]);
title('Input 8-PAM Wave');
%_%
```

Here we are plotting the signal with respect to time. We also labelled x-axis as Time axis and y-axis as Amplitude axis. The whole plot name is Input 8-PAM Wave. We also changed the colours, font sizes and font weights etc to make the graph more visually pleasing and distinguishable.

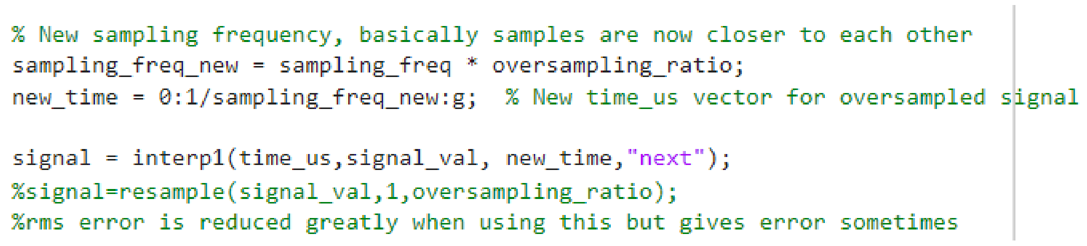
iv. Initializing Parameters for Sigma Delta Modulation:



```
oversampling_ratio =64; %good resolution is achieved when it is 64
sampling_freq = 1536; %obtained using-> 1/(difference between two samples in 8-PAM xlsx file)
format long g;
g=85.3326822916666+(1/sampling_freq);
%85.33268229196666 is the last value of column 1 in 8-PAM file
```

As we know in sigma delta the modulation, we do oversample to get better resolution. So, first we initialized the **oversampling ratio** by a factor of 64 because 64 is a commonly used factor to get high resolution. Then we obtained sampling frequency by subtracting two corresponding time values and taking their reciprocal i.e., **[1/ (T_2- T_1)].** This sampling frequency almost remains the same for every two consecutive time samples given in the excel file. The variable **g** is the value till where we will get our signal in the plot. To get digits after decimal point than the normal we used **long** which is of 32 bits.

v. Defining New sampling Factors:

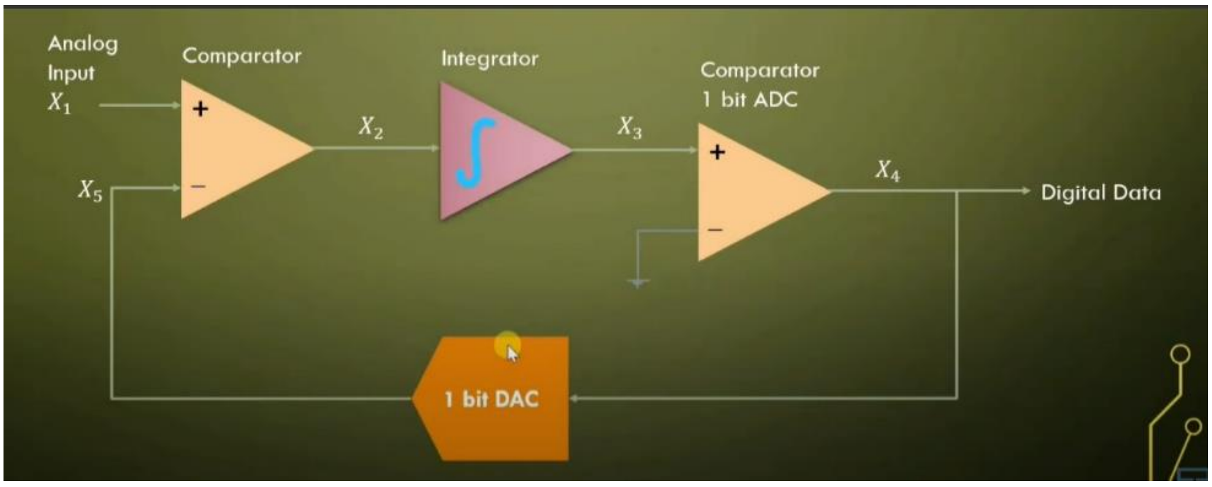


```
% New sampling frequency, basically samples are now closer to each other
sampling_freq_new = sampling_freq * oversampling_ratio;
new_time = 0:1/sampling_freq_new:g;  % New time_us vector for oversampled signal

signal = interp1(time_us,signal_val, new_time,"next");
%signal=resample(signal_val,1,oversampling_ratio);
%rms error is reduced greatly when using this but gives error sometimes
```

To effectively oversample the signal, we need another sampling frequency. This frequency is obtained by multiplying the previous sampling frequency with the oversampling ratio. This will increase the number of samples of the signal. For these new samples we need an updated time vector too which can clearly show these samples. The sampling vector i.e., **new_time** starts from 0 and goes to g with the increment equal to **(1/Sampling frequency).** Then we used **interp1** function to interpolate the **signal_val** data at time points specified by **new_time** using the nearest neighbor interpolation method as specified by **‘next’.** This process generates an oversampled version of the signal based on the original sample data for improving the resolution. We also tried to use **resample** function to oversample the signal, the error was also less i.e., 0.2 as compared to intep1 i.e., 0.49 but it gave error sometimes on our PCs so we didn’t use it.

vi. Initializing the Arrays:



```
integrator_state = zeros(size(signal));
adc_comparator = zeros(size(signal));
dac_output = zeros(size(signal));
error=zeros(size(signal));
initial_state=0;
```

As we can the above diagram were the working of a SDM ADC is shown, we define arrays of zeros with length equal to the length of the signal which ensures that these arrays have same no of elements as the signal. We also initialized the initial state as zero. Theses arrays are used to store the subsequent values at different points as shown in the diagram as the simulation runs. The initial state is made zero which means we are starting from the very start and there is no other value stored at the start. The zeros of these arrays get modified as the SDM ADC simulation runs, and more values are processed.

vii. Sigma Delta Modulation ADC:

```
for n = 2 :length(signal)
  error(n) = signal(n) - dac_output(n-1)+error(n-1);


    if (error(n) >= 0)
       adc_comparator(n) = 1;

    else
        adc_comparator(n) =0;

    end
```

Here we have implemented a loop starting from 2 and going till the length of the signal. The starting is 2 as in matlab the array index starts from 1 so if we take 1 it will make error of index 0 which is problematic. S the integrator operation integrates the error, and the previous error is provided through feedback loop to the system to reduces the quantization noise and smoothens the signal and enhances the ability of ADC to captures small variations in input signal giving higher resolution. Hence, we represented this error as **error(n)** which subtracts signal value from DAC output and add the previous error to accumulate error over time which is a feature of sigma-delta modulation. Then we check if the error is non-negative. The result generates a stream of 1s and zeros.

viii. Sigma Delta Modulation DAC:

```
if ( adc_comparator(n) == 1)
    dac_output(n)=1;
else
    dac_output(n)=-1;
end
end
adc_comparator1=adc_comparator-mean(adc_comparator);
```

These lines of code provide us with the output of the digital to analog converter. We check if the analog to digital converter's output is 1 then the output of DAC will also be 1 otherwise it will be -1 as we had values between 1 and -1. Then we calculate the normalized value of **adc_comparator** by subtracting it from its mean value. Hence in the plot we will be able to see the values between 1 and -1.

ix. Saving the ADC Output:

```
%_Saving ADC Output in .mat file_%
%constructing the filename using the first names of the group members
filename= "AnserZuhaNaimaInput";
save(filename, 'adc_comparator1');
%_%
```

Here we are saving the output of ADC as asked with our group member names. We used the **save** function to do so.

x. Applying filter:

```
filter_order =1;

adc_comparator2 = interp1(new_time, adc_comparator, time_us, "next");
%adc_comparator2 = resample(adc_comparator1, 1, oversampling_ratio,1);
%->  [b,a] = butter(n,Wn)
[b, a] = butter(filter_order, 767/sampling_freq/2);
dac_output1 = filter(b, a, adc_comparator2);
dac_output1 = 2 * dac_output1 - 1;
```

We chose first order filter because they do not introduce phase distortion and also have fast transient response and simpler implementation. Then we used interpolation for oversampling of **adc_comparator** signal at the time specified by **time_us** using the nearest neighbor interpolation method as specified by **'next'**.
It makes the output synchronized with the original time axis i.e., **time_us**.
Then we used **butterworth filter** because it gives smooth frequency response, have adjustable order and normalized frequency response. **[767/sampling_freq/2]** calculates the normalized cutoff frequency for the filter. 767 is a specific value chosen for the **cutoff frequency**. After obtaining the value for **dac_output1** by applying filter to **adc_comparator** signal, we scaled the filtered signal to get the value between 1 and -1.

xi. Saving the DAC Output:

```
%_Saving DAC Output in .mat file_%
% Constructing the filename using the first names of the group members
filename= "AnserZuhaNaimaOutput";
save(filename, 'dac_output1');
%%
```

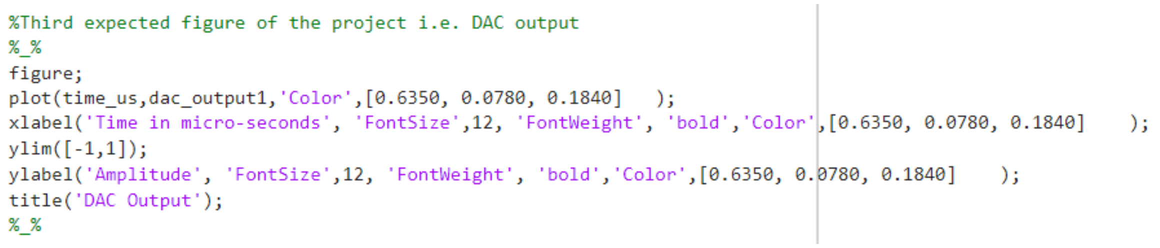
Here we are saving the output of DAC as asked with our group member names. We used the **save** function to do so.

xii. Plotting ADC Output:

```
%Second expected figure of the project i.e. ADC using sigma-delta modulator
%_%
figure;
plot(new_time, adc_comparator,'Color',[0, 0.4470, 0.7410]);
xlabel('Time in micro-seconds', 'FontSize', 14, 'FontWeight', 'bold','Color',[0, 0.4470, 0.7410]);
ylabel('Amplitude', 'FontSize', 14, 'FontWeight', 'bold','Color',[0, 0.4470, 0.7410]);
title('ADC Output');
%_%
```

Here we plotted the Figure 2 i.e., output of SDM ADC converter's output with respect to the **new_time** which was obtained from oversampled frequency. The x-axis was named **'Time in micro-seconds'** and the y-axis was labeled as **'Amplitude'**. The whole plot was named as **'ADC Output'**. To make the plot visually appealing, we changed the font size, weights and colors.

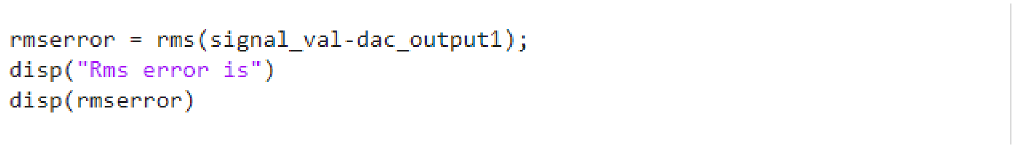
xiii. Plotting DAC Output:



```
%Third expected figure of the project i.e. DAC output
%_%
figure;
plot(time_us,dac_output1,'Color',[0.6350, 0.0780, 0.1840]   );
xlabel('Time in micro-seconds', 'FontSize',12, 'FontWeight', 'bold','Color',[0.6350, 0.0780, 0.1840]    );
ylim([-1,1]);
ylabel('Amplitude', 'FontSize',12, 'FontWeight', 'bold','Color',[0.6350, 0.0780, 0.1840]    );
title('DAC Output');
%_%
```

Here we plotted the Figure 3 i.e., output of SDM DAC converter's output with respect to the **time_us** which is the original time. The x-axis was named **'Time in micro-seconds'** and the y-axis was labeled as **'Amplitude'**. The whole plot was named as **'DAC Output'**. To make the plot visually appealing, we changed the font size, weights and colors.

xiv. RMS Value of Error:



```
rmserror = rms(signal_val-dac_output1);
disp("Rms error is")
disp(rmserror)
```

Here we obtained the value of error by subtracting the signal value from the output of dac converter. Then we used the **disp** function to display text and the value of error.