Implement an 8-order low-pass filter with a cut-off frequency of 25 kHz.

i) **Find the feedforward and feedback coefficients from MATLAB. Show MATLAB simulation of the filter output.**

ii) **Write Verilog code to implement the IIR filter in hardware. Verify your hardware code by writing test bench with a number of test cases.**

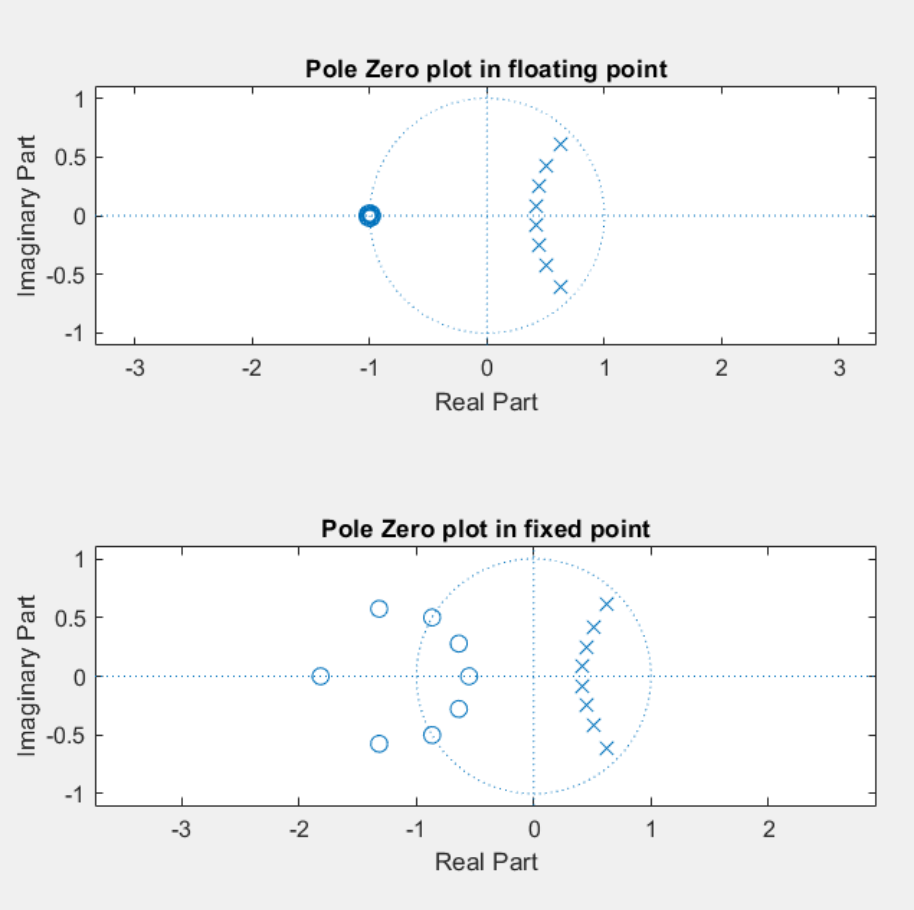
iii) **Show the output of your Verilog system Verilog code by running sample audio in your simulated hardware.**

Answer i)

MATLAB code:

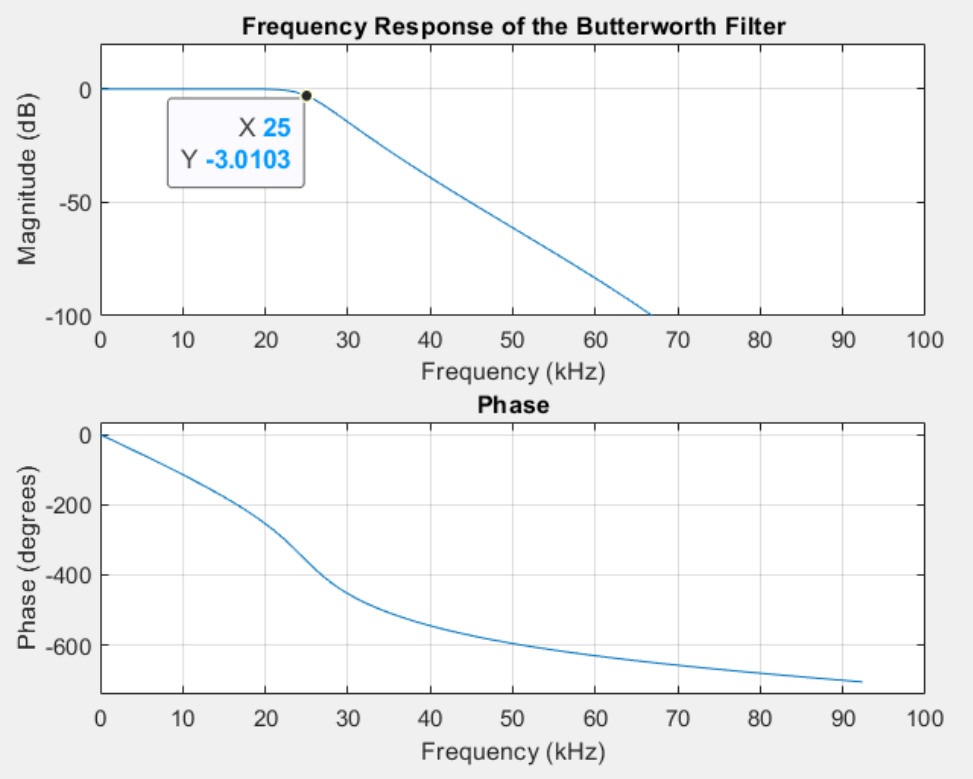
```
clc
fc = 25000;
fs = 200000;

[b,a] = butter(8,fc/(fs/2),"low")

scale_bits = 20;
b_fixed = round(b * 2^scale_bits)
a_fixed = round(a * 2^scale_bits)

figure
subplot(2,1,1)
zplane(b,a)
title('Pole Zero plot in floating point');
subplot(2,1,2)
zplane(b_fixed,a_fixed)
title('Pole Zero plot in fixed point');

figure;
subplot(2,1,1)
freqz(b,a,[],fs)
ylim([-100 20]);

title('Frequency Response of the Butterworth Filter');
```

Pole – zero plot:



Since we need to deal with integer numbers in hardware, it is a better practice to scale the coefficients to integer number instead of dealing with direct floating points. 20 was chosen as the scaling factor which deals with enough precision as well as adequate headroom for signs, necessary multiplication and summation later on. From the pole zero plot, the system is stable for the choice of coefficients before and after scaling.

Now plotting the filter response:



It shows unity gain at low frequencies and -3 dB gain at 25kHz which is the cut-off.

Answer ii)

Verilog code for the base module:

```
module butterworth #(parameter N=8, parameter width = 32)(
input logic clk,
input logic rst,
input logic [width-1:0] xin,
output logic [width-1:0] yout
);

//filtercoefficients
parameter signed [width-1:0] b [0:N] = '{
    32'sd113,
    32'sd905,
    32'sd3168,
    32'sd6337,
    32'sd7921,
    32'sd6337,
    32'sd3168,
    32'sd905,
    32'sd113
};

parameter signed [width-1:0] a [0:N] = '{
    32'sd1048576,
    -32'sd4177301,
    32'sd7902314,
    -32'sd9017560,
    32'sd6711048,
    -32'sd3309332,
    32'sd1050355,
    -32'sd195394,
    32'sd16261
};

//delay registers
logic signed [width-1:0] xreg [0:N];
logic signed [width-1:0] yreg [1:N]; //no use of y0
logic signed [(2*width)-1:0] sum;
```

```
always_comb
  begin
        sum = 0;
        for (int i = 0; i <= N; i++)
          begin
                sum += b[i] * xreg[i];
          end
        for (int i = 1; i <= N; i++)
          begin
                sum -= a[i] * yreg[i];
          end
        yout = sum >>> 20; //yout is combinational
  end

always_ff @(posedge clk or posedge rst)
        if (rst)
          begin
                xreg[0] <= 0;
                for (int i = 1; i <= N; i++)
                  begin
                        xreg[i] <= 0;
                        yreg[i] <= 0;
                  end
          end
        else
          begin
                xreg[0] <= xin;
                xreg[1] <= xreg[0];
                for (int i = N; i > 1; i--)
                  begin
                        xreg[i] <= xreg[i-1];
                        yreg[i] <= yreg[i-1];
                  end
                yreg[1] <= yout;
          end

endmodule
```

Here we choose 32 bits register for regular registers and 64 bits for the sum register to accommodate large values. The output signal is set to be combinational whereas input is set to be sequential.

Verilog code for the testbench:

```
module audiotb #(parameter width = 32);

    logic clk;
    logic rst;
    logic signed [width-1:0] xin;
    logic signed [width-1:0] yout;

    integer file, outfile;

    butterworth dut ( .clk(clk), .rst(rst), .xin(xin), .yout(yout));

    always #10 clk =~clk;
    initial begin
        clk = 0;
        rst = 1;
        xin = 0;

        //input and output text files
        file = $fopen("Input_signal.txt", "r");
        if (file == 0) begin
            $display("Error opening Input_signal.txt");
            $finish;
        end
        outfile = $fopen("filtered_signal.txt", "w");

        #10 rst = 0;

        while (!$feof(file)) begin
            @(posedge clk); #2;
            $fscanf(file, "%d\n", xin);
            @(negedge clk);
            $fwrite(outfile, "%d\n", yout);
        end

        $fclose(file);
        $fclose(outfile);
        $finish;
    end
endmodule
```

Here we take the values of yout at the negative edge of the clock since it is set as a combinational output. This allows enough time for the yout to reach a stable value before write operation.

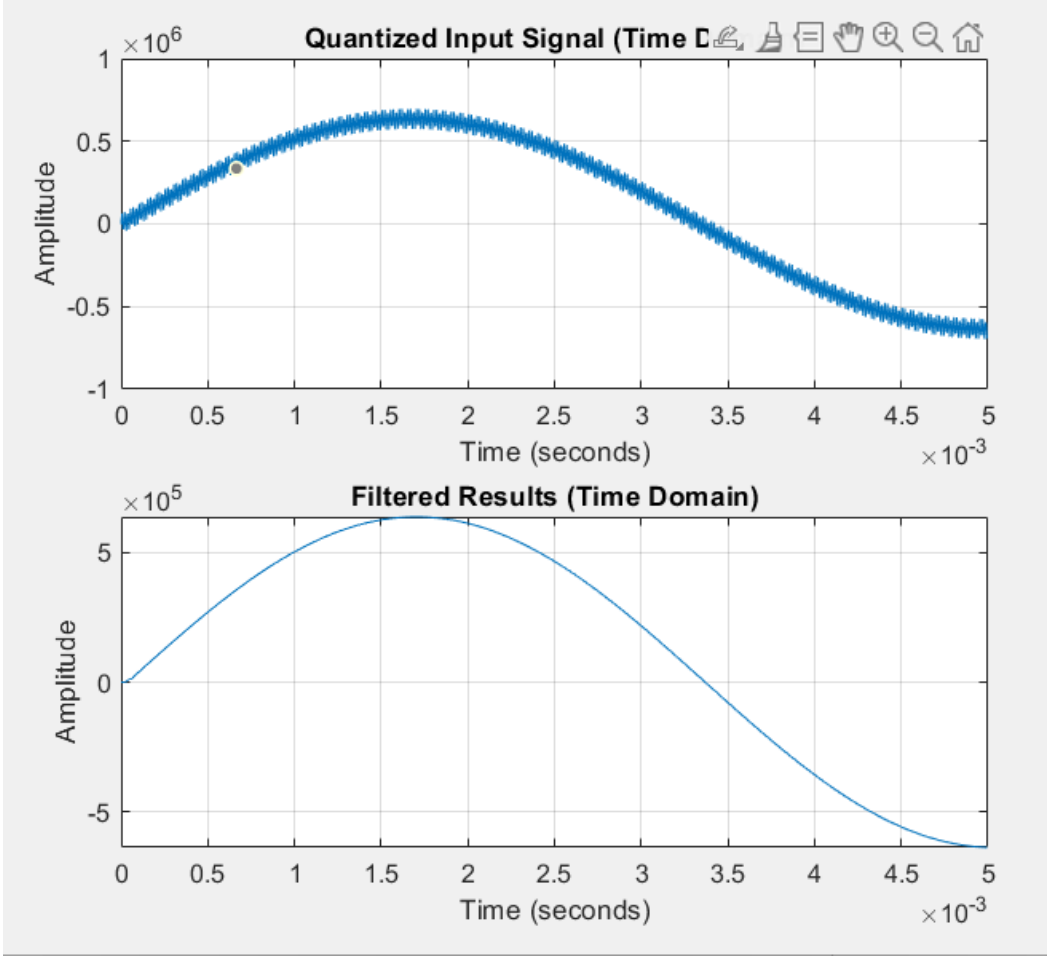
MATLAB Code for testbench stimulus:

```
Ts=1/fs;
n=0:999;
t=n*Ts;
x=1*sin(2*pi*150*t)+0.1*sin(2*pi*60000*t);
Vref = 3.3;

xq = (x / (Vref/2)) * (2^scale_bits);
xq_int = round(xq);
fid = fopen('input_signal.txt','w');
fprintf(fid,"%d\n",xq_int);
fclose(fid);
```

MATLAB Code for plotting:

```
% Load the quantized signal (from your previous step)
quantizedSignal = dlmread('input_signal.txt');

% Load the simulation results (output from the filter)
simResults = dlmread('filtered_signal.txt');

% Time vector for plotting with respect to time
N = length(quantizedSignal);  % Number of samples
time = (0:N-1) / fs;          % Time vector

% Plot the signals with respect to time in a new figure
figure;
subplot(2, 1, 1);
plot(time, quantizedSignal);
title('Quantized Input Signal (Time Domain)');
xlabel('Time (seconds)');
ylabel('Amplitude');
grid on;

subplot(2, 1, 2);
plot(time, simResults);
title('Filtered Results (Time Domain)');
xlabel('Time (seconds)');
ylabel('Amplitude');
grid on;
```

Signal Figure:



We can see the signal with low frequency noise being smoothed after passing through the filter. The noise frequency was 60 kHz whereas the main signal frequency was 150 Hz. The digital low pass IIR filter successfully filtered out the noise. A total of thousand test points were processed.

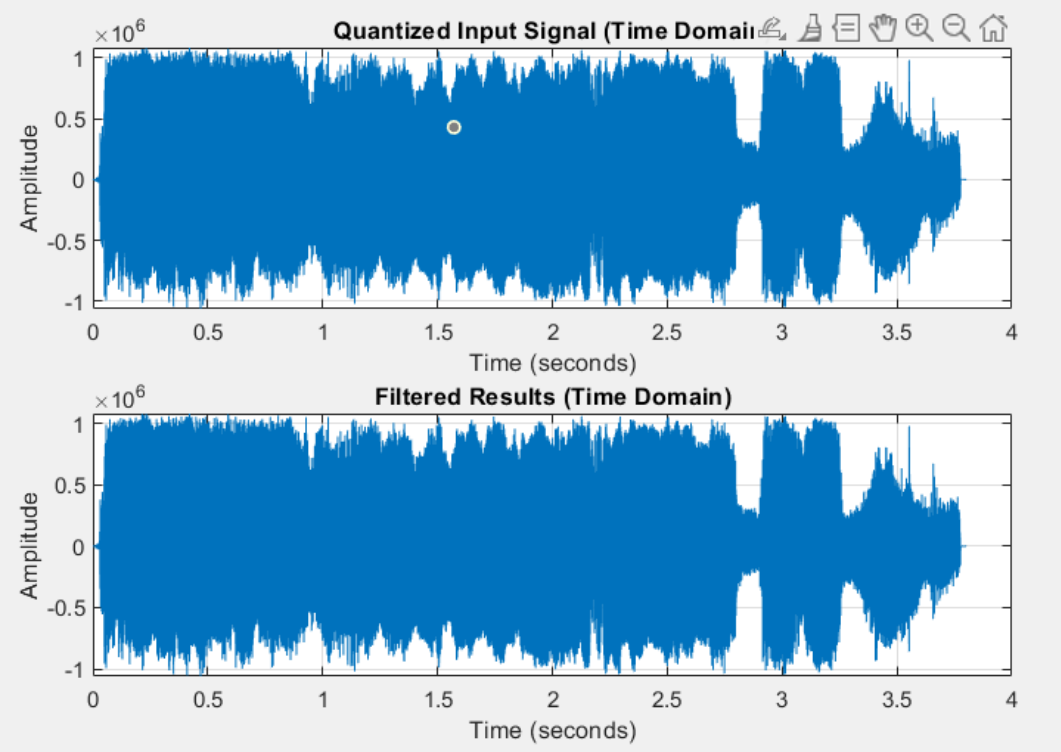
Answer ii)

Now for the audio signal we chose a short audio signal and resampled it at our sampling frequency. Since real life ADC usually has a bit range of 12 – 24 bits, choice of 20-bit quantization is more than sufficient for this example.

MATLAB code:

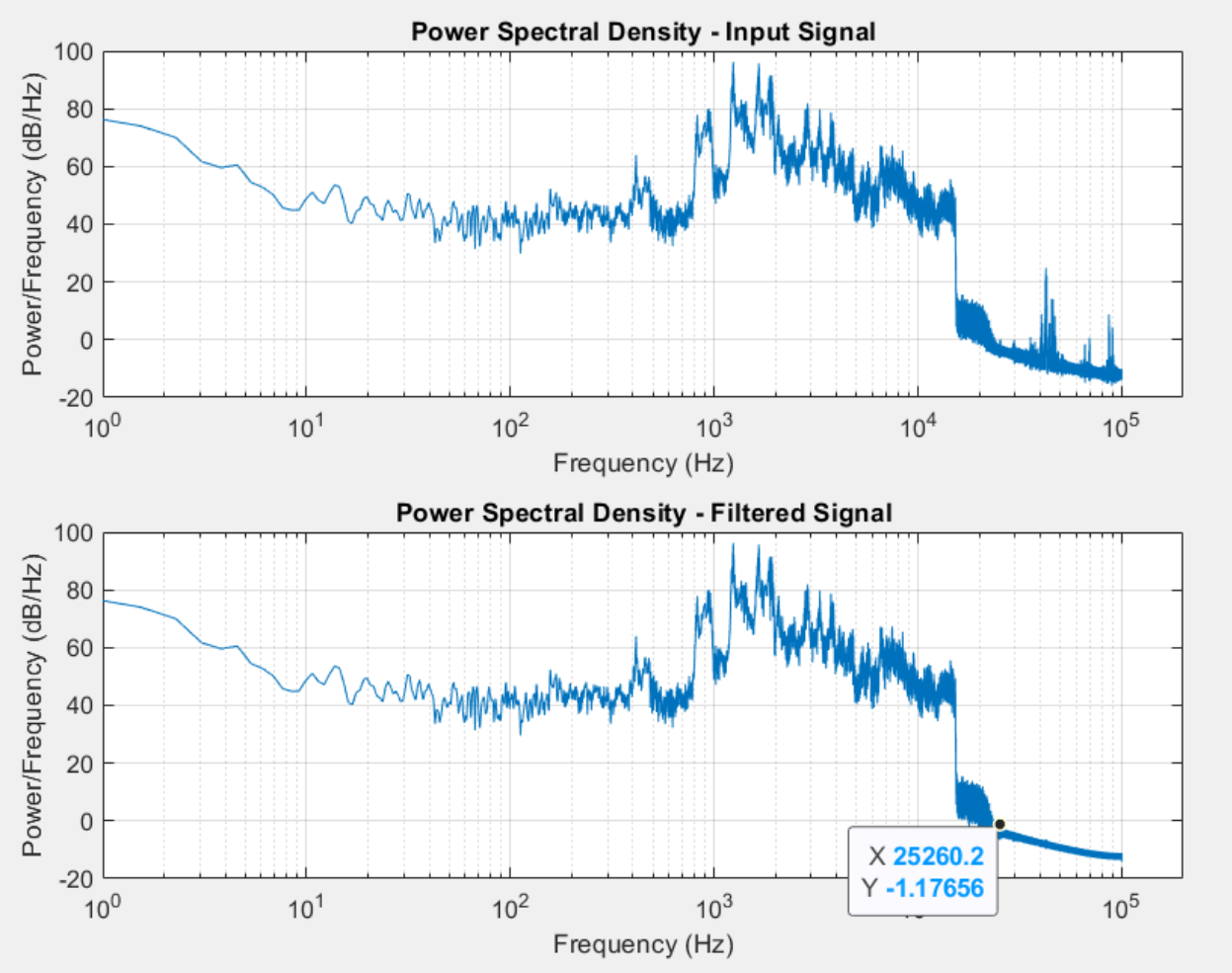
```
[audioSignal,  originalFs] = audioread('E:\OneDrive - BUET\4-2\VLSI\IIR\magic.wav');
audioSignal = audioSignal(:, 1);
if originalFs ~= fs
    audioSignal = resample(audioSignal, fs, originalFs);
end %audioread already normalized audio signal value

quantizedSignal = round(audioSignal * 2^scale_bits);
dlmwrite('input_signal.txt', quantizedSignal, 'delimiter', '\n');
```

Wave Signal:



The filtered output is slightly smoother but it is not as noticeable since most of the frequencies in the audio signal is below 25kHz. To understand better a power spectrum is presented next.

Power Spectral Density:



Here it is evident that the power for frequencies above 25kHz has gone below zero after filtering process. This proves the low pass filter design was accurate and robust.