Echo, FIR, and IIR Filtering in MATLAB and C

Kendrick Nguyen
ECE 161B: Digital Signal Processing
kan006@ucsd.edu
PID: A16889378

Introduction

The focus of this lab is to create filters that add effects and remove noise at a certain frequency. Schemes to add echo to a .wav file, create a notch filter from a lowpass filter, and design both finite and infinite impulse response filters were implemented in MATLAB and C. Spectrograms show the differences before and after when the filters were applied.

1 Echo Filter in MATLAB

The speech.wav array was min-max normalized on the range [-1,1] for accuracy. The difference equation for the echo is

$$y[n] = 0.8x[n] + 0.2x[n - N_e]$$

where 0.8 is the attenuation for the original sample and 0.2 is the attenuation for the echoed sample. N_e is the delay at F_s. For this lab, $N_e = 0.5$ seconds at $F_s = 48$kHz.

1.1 Code

```
%% Task 1 Echo
[x,fs] = audioread("speech.wav");
x = normalize(x, "range", [-1 1]);
Ne = round(0.5*fs);

y = zeros(size(x));
a = [0.8 0.2];

for n= Ne + 1:length(x)
    y(n) = a(1) * x(n) + a(2) * x(n-Ne);
end

audiowrite("outputecho.wav",fs);
```

Listing 1: Echo Filter

The realization is fairly straightforward for an echo effect. For the output waveform, the speech.wav has a quieter repeat of the words said 0.5 seconds before throughout the duration of the track. There is also a 0.5 delay to the start of the soundfile.

2 Sinusoidal Interference to `speech.wav`

Purposeful corruption is added to the original sample in order to filter it out with a notch filter with both FIR and IIR filtering. The sinusoidal signal $v[n]$ is the same length of speech.wav with a frequency of 2.4 kHz and a sampling frequency of 48 kHz. A helper function called `generate_sigs()`6 will assist in creating this sinusoidal wave. The corruption will be added to the original sample such that

$$w[n] = x[n] + v[n]$$

where $w[n]$ is the output waveform. A spectrogram measured in frequency over time is created for each signal.

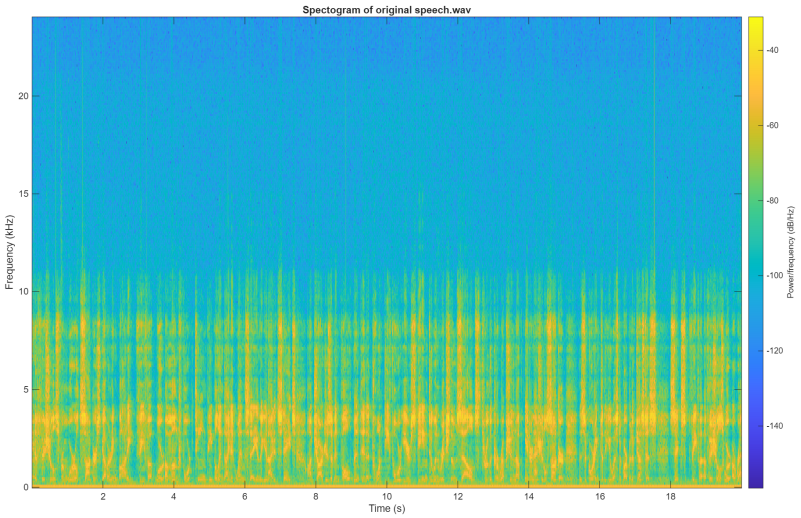

Figure 1: Spectrogram of `speech.wav`

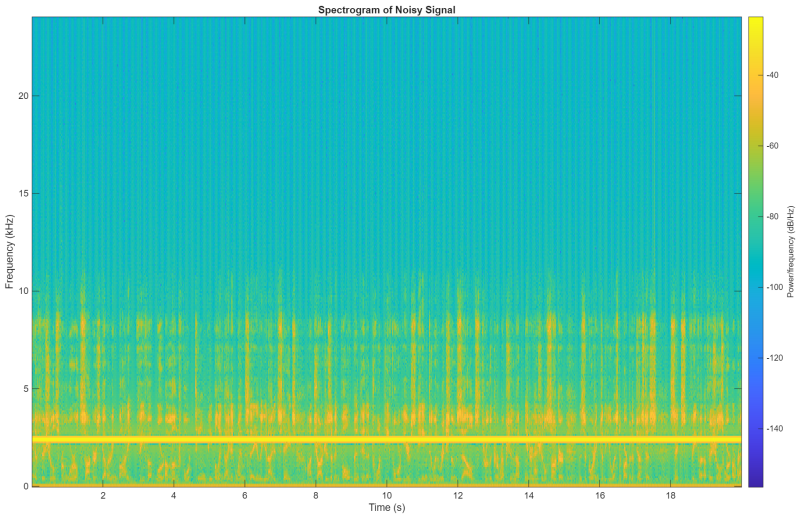

Figure 2: Spectrogram of Corrupted Signal

From the two spectrograms, it is clear that the added sinusoidal signal adds a constant additional noise at 2.4 kHz. Notice how the sinusoidal wave is above -40 dB/Hz, which is more powerful than the original signal. This means that the additive noise will be the more prominent sound when the corrupted speech is played.

3 Notch Filter

To remove this 2.4 kHz sinusoidal signal, a notch filter must be implemented with a narrow stop-band at 2.4 kHz. The notch filter coefficients can be derived from an ideal Type-I linear-phase lowpass filter where the middle coefficient can be manually tuned to meet specs.

3.1 Notch Filter From Lowpass Filter Derivation

Let H be an ideal Type-I linear-phase lowpass filter and G be the desired notch filter. We need to normalize $H(z)$ s.t. $H_{ampl} = 1$ and $G(z)$ s.t. $G_{ampl} = 1$.

Let $\omega_c = 0.5(\omega_p + \omega_s)$ i.e. the frequency where the notch occurs

If $G_{ampl}(\omega) = H_{ampl}(\omega) - 0.5$, then $G_{ampl}(\omega_c) \approx 0$

Converting each filter into its impulse response results in the desired condition of

$$g[n] = \begin{cases} h[n] - k & \text{n=L} \\ h[n] & \text{otherwise} \end{cases}$$

where k is a manually tuned constant to bring the notch closer to a desired frequency (2.4 kHz).

Let the filter order N = 14, the center index L = N/2=7. The central frequency needs to satisfy the frequency and sampling frequency conditions

$$w_c = \frac{f_{notch}}{f_s/2} = \frac{2.4kHz}{24kHz}$$

Once the LPF $h[n]$ is normalized with w_c, $g[n]$ becomes $2 * h[n]$ so the impulse response does not leave the range [-1,1] when subtracted by constant k. $g[n]$ is normalized so its sum is equal to 1, satisfying all conditions. Through manual tuning, k was found to be 1.43, leaving the final notch at 2.414 kHz. Code of this process is found in the appendix7.

3.2 Magnitude Response of the Notch Filter

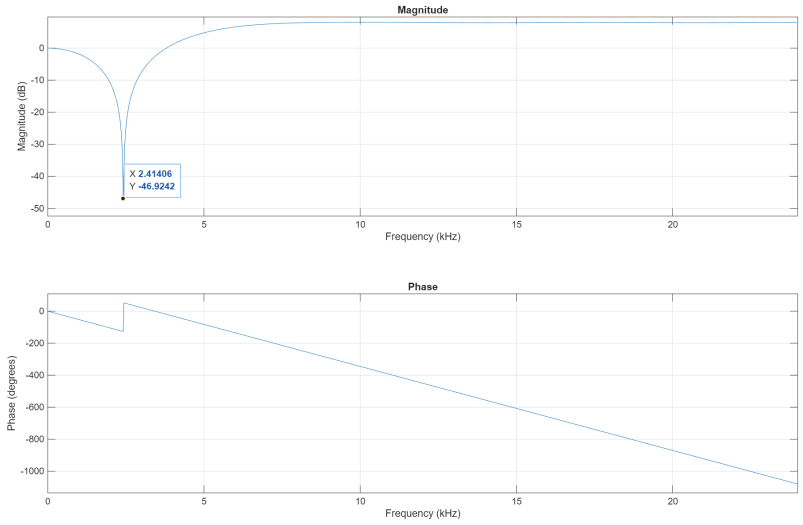

Figure 3: Notch Filter `freqz()` Magnitude and Phase Response

From the notch filter derivation, the notch frequency occurs at 2.414 kHz, approximately where the corrupted signal occurs. The length of both $h[n]$ and $g[n]$ is 15, which satisfies a 14th order filter.

4 Creating an `fir_filter()` in MATLAB

The created function `fir_filter(w,h)` 8 takes in the corrupted signal w and the filter coefficients h. For the function, a convolution was made between the signal and the filter coefficients that satisfies

$$\sum_{k=0}^{M} w[k]h[n-k]$$

using the previously derived notch coefficients in $g[n]$ and the corrupted signal, the following spectrogram is created

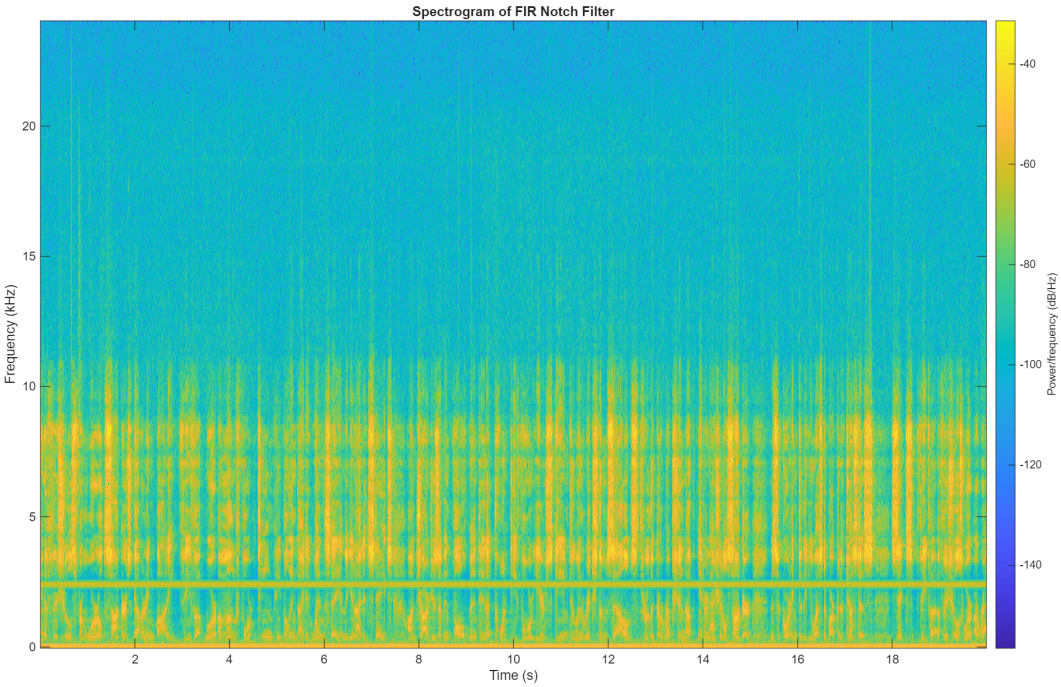

Figure 4: Spectrogram of FIR filter

Compared to the spectrogram of the corrupted speech, the power/frequency of the 2.4 kHz sinusoid has been diminished to the levels of the original signal. Playing the FIR filtered speech has a present hum of the 2.4 kHz sinusoid, but the speech is more audible compared to before.

5 `iir_filter()` in MATLAB

An `iir_filter(x,a,b)` 9 will be used to filter out the corrupted signal in the speech. A notch ω_0 must be created for the transfer function of this IIR filter. For a second order IIR filter, the transfer function is

$$H(z) = \frac{(z - e^{j\omega_0})(z - e^{-j\omega_0})}{(z - re^{j\omega_0})(z - re^{-j\omega_0})}$$

where r is the pole radius. Using trigonometric identities and Euler's formula, the transfer function can be rewritten as

$$H(z) = \frac{1 - 2\cos(\omega_0)z^{-1} + z^{-2}}{1 - 2r\cos(\omega_0)z^{-1} + r^2z^{-2}}$$

This form of the transfer function gives us the a and b coefficients of needed for the function where

$$a = [1, -2r\cos(\omega_0), 1] \text{ and } b = [1, -2r\cos(\omega_0), r^2]$$

The output of the IIR filter produces the following spectrogram Notice how not only the 2.4 kHz

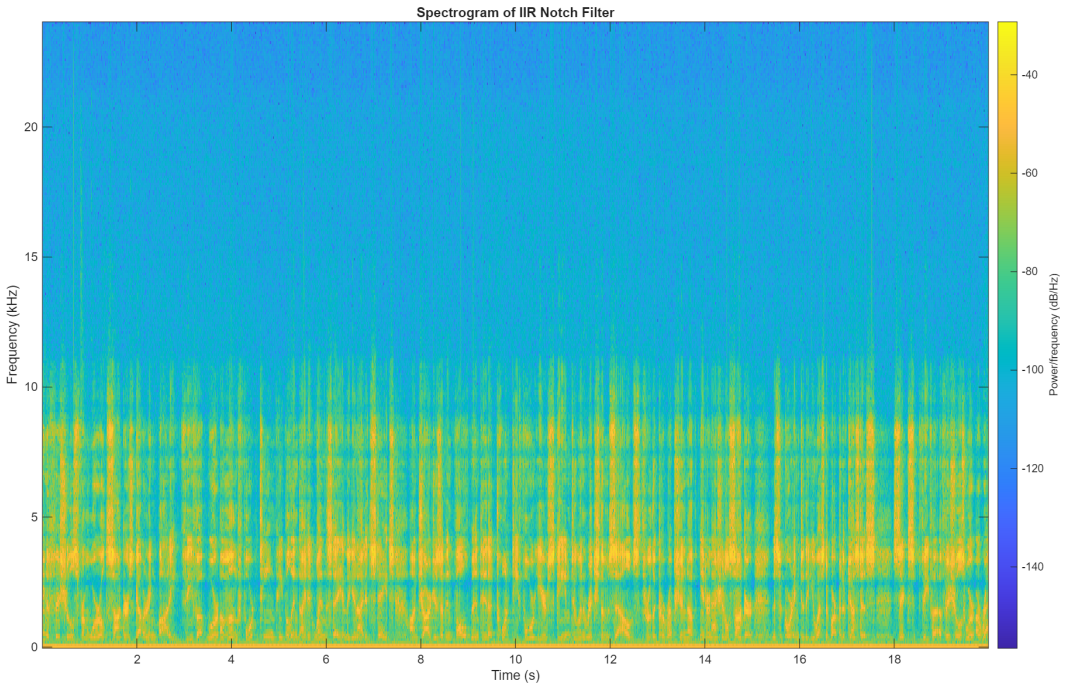

Figure 5: Spectrogram of the IIR Filter

sinusoidal signal has been removed, but also some of the original audio at 2.4 kHz. The resulting filtered speech has the 2.4 kHz hum completely removed; the speech is as clear as the original sample to the human ear. The IIR filter is more efficient at removing the noise compared to the FIR filter thanks to its feedback and feedforward properties. The IIR filter can filter at a lower order with better results at the cost of being more complex to implement.

6 Echo in C

All filters and effects can be reproduced in C language programming. The echo effect can be achieved through dynamic memory allocation while establishing the same initial conditions in MATLAB. A function was created to write the output signal after the echo was applied. Full code can be found in the appendix.10

```
float echo(float x_in, float *buffer, int Ne, int *index);
...
int main(int argc, char *argv[]) {
int ii;
...
int Ne = (int)(0.5 * Fs); // Number of samples corresponding to the delay
     time
  float *echo_buffer = (float *)calloc(Ne, sizeof(float)); // Buffer to
    store delayed samples for echo filter
  int echo_index = 0; // Index for circular buffer in echo filter
  for(ii = 0; ii < sndInfo.frames; ii++)
  {
    sf_readf_float(sndFile, &sig_in[ii], 1); // Read one sample at a time
    sig_out[ii] = echo(sig_in[ii], echo_buffer, Ne, &echo_index); //
    Apply echo filter to current input sample
    sf_writef_float(sndFileOut, &sig_out[ii], 1); // Write one sample at
    a time
  }
    free(echo_buffer); // Free echo buffer after processing
...
return 1;
}

float echo(float x_in, float *buffer, int Ne, int *index){
  float a[]= {0.8, 0.2}; // Original and echo coefficients
  float y_out = a[0] * x_in + a[1] * buffer[*index]; // Output sample is
    the sum of current input and delayed sample
  buffer[*index] = x_in; // Update delay buffer with current input sample
  *index = (*index + 1) % Ne; // Update index for circular buffer
  return y_out;
}
...
```

Listing 2: Echo Filter in main() and as a function

The output file provides an echoed version of the original speech, similar to the MATLAB version.

7 FIR Filter in C

The FIR filter was more complex to implement than its MATLAB counterpart. Two additional functions were created: one for generating FIR filter coefficients and one for the FIR filtering. Dynamic memory allocation was used to store the FIR coefficients. The resulting .wav file produced worse results compared to the MATLAB FIR filter. The 2.4 kHz sinusoid was more powerful after the C FIR filtering than the MATLAB FIR filtering. Manual tuning was harder to achieve with the C design.

```
...
float fir_filter(float x_in, double *g, float *x, int N);
void generate_fir_coefficients(double *g, int N, float notch_freq);
...
int main(int argc, char *argv[])
{
...
  float notch_freq = 2400; // Notch frequency
  int N = 14; // Filter order
  double *g = (double *)malloc((N+1)*sizeof(double)); // FIR coefficients
```

```
  if(g == NULL) {
    fprintf(stderr, "Could not allocate memory for FIR coefficients\n");
    sf_close(sndFile);
    return 1;
  }

  generate_fir_coefficients(g, N, notch_freq); // Generate FIR
    coefficients for the
  printf("FIR coefficients:\n");
  for(int i = 0; i <= N; i++){
    printf("%f ", g[i]);
  }
  printf("\n");

  float *fir_buffer = (float *)calloc(N+1, sizeof(float)); // Buffer to
    store delayed samples for FIR filter
  if(fir_buffer == NULL) {
    fprintf(stderr, "Could not allocate memory for FIR buffer\n");
    sf_close(sndFile);
    return 1;
  }

  for(ii = 0; ii < sndInfo.frames; ii++)
  {
    sf_readf_float(sndFile, &sig_in[ii], 1); // Read one sample at a time
    sig_out[ii] = fir_filter(sig_in[ii], g, fir_buffer, N); // Apply FIR
    filter to current input sample
    sf_writef_float(sndFileOut, &sig_out[ii], 1); // Write one sample at
    a time
  }
  free(g);
  free(fir_buffer); // Free FIR buffer after processing
    ...
    return 1;
}
...
void generate_fir_coefficients(double *g, int N, float notch_freq){
  int L = N/2; // Center Frequency
  double wc = 2 * PI * notch_freq / Fs; // Normalized Frequency
  double h[N+1]; // Impulse Response of Ideal Notch Filter
  double sum_h = 0.0; // Sum of Impulse Response

  for(int n = 0; n<=N; n++){
    if(n == L){
      h[n] = wc / PI;
    }
    else{
      h[n] = sin(wc * (n - L)) / (PI * (n - L));
    }
    sum_h += h[n];
  }


  double sum_g = 0.0; // Sum of FIR Coefficients
  for(int n = 0; n<=N; n++){
    h[n] = h[n] / sum_h; // Normalize the impulse response
    g[n] = -2 * h[n] *cos(wc*(n-L)); // FIR Coefficients
    if(n==L){
      g[n] += 1; // Add 1 to the center coefficient
    }
    sum_g += g[n];
  }

}
...
```

```
72
73 float fir_filter(float x_in, double *g, float *x, int N){
74
75   for(int i = N; i > 0; i--){
76     x[i] = x[i-1]; // Shift input buffer
77   }
78   x[0] = x_in; // Current input sample
79
80   float y_out = 0.0f; // Current output sample
81   for(int i = 0; i <= N; i++){
82     y_out += g[i] * x[i]; // Convolution of input signal with FIR
       coefficients
83   }
84
85   return y_out;
86
87 }
88 ...
```

Listing 3: FIR filter main() and functions

8 IIR Filter in C

The IIR filter followed the same conditions for the a and b coefficients as well as the difference equation implementation. The resulting .wav file produced a clear speech, similar to the original .wav file. This is comparable to the MATLAB IIR Filter implementation.

```
1 ...
2 float iir_filter(float x_in);
3 ...
4 int main(int argc, char *argv[])
5 {
6 ...
7 for(ii = 0; ii < sndInfo.frames; ii++)
8   {
9     sf_readf_float(sndFile, &sig_in[ii], 1); // Read one sample at a time
10    sig_out[ii] = iir_filter(sig_in[ii]);
11    sf_writef_float(sndFileOut, &sig_out[ii], 1); // Write one sample at
      a time
12  }
13    ...
14    return 1;
15 }
16 ...
17 float iir_filter(float x_in)
18 {
19   float r = 0.95; // Pole radius
20   float w0 = 2* PI * 2400/Fs; // Normalized Frequency
21
22   float b[] = {1, -2*cos(w0), 1}; // IIR coefficients
23   float a[] = {1, -2*r*cos(w0), r*r}; // IIR coefficients
24
25   static float x[3] = {0,0,0}; // Input signal buffer for IIR filter
26   static float y[3] = {0,0,0}; // Output signal buffer for IIR filter
27
28   for(int i = 2; i > 0; i--){
29     x[i] = x[i-1]; // Shift input buffer
30     y[i] = y[i-1]; // Shift output buffer
31   }
32
33   x[0] = x_in; // Current input sample
34   float y_out =  (b[0]*x[0] + b[1]*x[1] + b[2]*x[2] - a[1]*y[1] - a[2]*y
       [2])/a[0]; // Calculate current output sample using difference
     equation
```

```
35   y[0] = y_out; // Update output buffer with current output sample
36   return y_out;
37 }
```

Listing 4: IIR filter main() and functions

9 Conclusion

A waveform can be introduced to effects like an echo or filtered from noise with FIR or IIR filtering. These methods can be implemented with MATLAB or C based on the same difference equations and formulations for the filter coefficients.

A Appendix

```
%% Task 1 Echo
[x,fs] = audioread("speech.wav");
x = normalize(x, "range", [-1 1]);
Ne = round(0.5*fs);

y = zeros(size(x));
a = [0.8 0.2];

for n= Ne + 1:length(x)
    y(n) = a(1) * x(n) + a(2) * x(n-Ne);
end
sound(y,fs);
nfft = 512;
window = hamming(nfft);
figure;
spectrogram(y,window, [], nfft,fs, 'yaxis');
```

Listing 5: Echo MATLAB Code

```
%% Task 2 Noisy Signal
signal_length = length(x);
v = generate_sigs(fs, 2400, signal_length);

w = zeros(size(x));
for n = 1:length(x)
    w(n) = x(n) + v(n);
end

nfft = 512;
window = hamming(nfft);

figure;
spectrogram(x,window, [], nfft, fs, 'yaxis');
title("Spectogram of original speech.wav")
figure;
% Plot the spectrogram of the noisy signal
spectrogram(w, window, [], nfft, fs, 'yaxis');
title('Spectrogram of Noisy Signal');
...
function sigs = generate_sigs(fs,f,duration)
    ts = 1/fs;
    t = ts*(0:duration-1);
    sigs = sin(2*pi*f*t);
end
```

Listing 6: Noisy Signal MATLAB code

```
%% Task 3 Notch Filter
f_notch = 2400;
N=14;
L = N/2; %Center index


wc_norm = f_notch/(fs/2);
h=fir1(N,wc_norm); %a 14th order lowpass FIR filter
h=h/sum(h);%normalization of h

g=2*h; %scaled h so subtracting constant doesn't go out of range [-1,1]
g(L+1) = g(L+1)-1.43;
g = g/sum(g); %normalization of g
figure;
freqz(g,1,1024,fs);
```

Listing 7: Notch Filter Coefficients MATLAB Code

```
%% Task 4 FIR Notch Filter
s = fir_filter(w,g);
figure;
spectrogram(s,window,400,nfft,fs, "yaxis");
title("Spectrogram of FIR Notch Filter");
...
function sig = fir_filter(w,h)
    N = length(w);
    M = length(h);
    sig = zeros(1,N);

    for n = 1:N
        for k = 1:M
            if(n-k+1) >0
                sig(n)=sig(n) + h(k) * w(n-k+1);
            end
        end
    end
end
```

Listing 8: FIR MATLAB Code

```
%% Task 5 IIR Notch Filter
r = 0.95;
w0 = 2*pi*2400/fs;
b_iir = [1, -2*cos(w0),1];
a_iir = [1,-2*r*cos(w0),r^2];

iir_filtered_s = iir_filter(w,a_iir,b_iir);
figure;
spectrogram(iir_filtered_s, window, 400, nfft, fs, "yaxis");
title("Spectrogram of IIR Notch Filter");
...
function y = iir_filter(x,a,b)
y = zeros(size(x));
for n = 3:length(x)
    y(n) = (b(1)*x(n)+b(2)*x(n-1)+b(3)*x(n-2)-a(2)*y(n-1)...
        -a(3)*y(n-2))/a(1);

end
end
```

Listing 9: IIR MATLAB Code

```
#include <stdlib.h>
#include <stdio.h>
#include <float.h>
#include <sndfile.h>
#include <math.h>

#define PI 3.14159265
#define Fs 48000.0f

float echo(float x_in, float *buffer, int Ne, int *index);
float iir_filter(float x_in);
float fir_filter(float x_in, double *g, float *x, int N);
void generate_fir_coefficients(double *g, int N, float notch_freq);


int main(int argc, char *argv[])
{
  int ii;

  //Require 2 arguments:input file,  and output file
  if(argc < 2)
```

```
{
  printf("Not enough arguments \n");
  return -1;
}

SF_INFO sndInfo;
SNDFILE *sndFile = sf_open(argv[1], SFM_READ, &sndInfo);
if (sndFile == NULL) {
  fprintf(stderr, "Error reading source file '%s': %s\n", argv[1],
  sf_strerror(sndFile));
  return 1;
}

SF_INFO sndInfoOut = sndInfo;
sndInfoOut.format = SF_FORMAT_WAV | SF_FORMAT_PCM_16;
sndInfoOut.channels = 1;
sndInfoOut.samplerate = sndInfo.samplerate;
SNDFILE *sndFileOut = sf_open(argv[2], SFM_WRITE, &sndInfoOut);

// Check format - 16bit PCM
if (sndInfo.format != (SF_FORMAT_WAV | SF_FORMAT_PCM_16)) {
  fprintf(stderr, "Input should be 16bit Wav\n");
  sf_close(sndFile);
  return 1;
}

// Check channels - mono
if (sndInfo.channels != 1) {
  fprintf(stderr, "Wrong number of channels\n");
  sf_close(sndFile);
  return 1;
}

// The following code is a only a template, please rewrite

// Allocate memory
// float *sig_in = ... ;
// float *sig_out = ... ;
// if (buffer == NULL) {
//  fprintf(stderr, "Could not allocate memory for file\n");
//  sf_close(sndFile);
//  return 1;
// }

float *sig_in = (float *)malloc(sndInfo.frames * sndInfo.channels *
  sizeof(float)); // Input buffer
if (sig_in == NULL) {
  fprintf(stderr, "Could not allocate memory for file\n");
  sf_close(sndFile);
  return 1;
}

float *sig_out = (float *)malloc(sndInfo.frames * sizeof(float)); //
  Output buffer
if (sig_out == NULL) {
  fprintf(stderr, "Could not allocate memory for output file\n");
  sf_close(sndFile);
  return 1;
}

// float y_fir = fir_filter(sig_in); // Example usage of fir_filter
  function


```

```
// Read data
// ...
//  sf_readf_float(sndFile, sig_in, 1);
// ...


////////////////// Uncomment each filter implementation for intended task ////////////////////////

/*//ECHO FILTER
int Ne = (int)(0.5 * Fs); // Number of samples corresponding to the delay time
float *echo_buffer = (float *)calloc(Ne, sizeof(float)); // Buffer to store delayed samples for echo filter
int echo_index = 0; // Index for circular buffer in echo filter
for(ii = 0; ii < sndInfo.frames; ii++)
{
  sf_readf_float(sndFile, &sig_in[ii], 1); // Read one sample at a time
  sig_out[ii] = echo(sig_in[ii], echo_buffer, Ne, &echo_index); // Apply echo filter to current input sample
  sf_writef_float(sndFileOut, &sig_out[ii], 1); // Write one sample at a time
}
  free(echo_buffer); // Free echo buffer after processing
*/ //End of Echo filter implementation



/*//FOR FIR FILTER
float notch_freq = 2400; // Notch frequency
int N = 14; // Filter order
double *g = (double *)malloc((N+1)*sizeof(double)); // FIR coefficients
if(g == NULL) {
  fprintf(stderr, "Could not allocate memory for FIR coefficients\n");
  sf_close(sndFile);
  return 1;
}

generate_fir_coefficients(g, N, notch_freq); // Generate FIR coefficients for the
printf("FIR coefficients:\n");
for(int i = 0; i <= N; i++){
  printf("%f ", g[i]);
}
printf("\n");

float *fir_buffer = (float *)calloc(N+1, sizeof(float)); // Buffer to store delayed samples for FIR filter
if(fir_buffer == NULL) {
  fprintf(stderr, "Could not allocate memory for FIR buffer\n");
  sf_close(sndFile);
  return 1;
}

for(ii = 0; ii < sndInfo.frames; ii++)
{
  sf_readf_float(sndFile, &sig_in[ii], 1); // Read one sample at a time
  sig_out[ii] = fir_filter(sig_in[ii], g, fir_buffer, N); // Apply FIR filter to current input sample
  sf_writef_float(sndFileOut, &sig_out[ii], 1); // Write one sample at a time
}
free(g);
free(fir_buffer); // Free FIR buffer after processing
*/ //End of FIR filter implementation
```

```
139
140
141
142
143   /*//FOR IIR FILTER
144   for(ii = 0; ii < sndInfo.frames; ii++)
145   {
146     sf_readf_float(sndFile, &sig_in[ii], 1); // Read one sample at a time
147     sig_out[ii] = iir_filter(sig_in[ii]);
148     sf_writef_float(sndFileOut, &sig_out[ii], 1); // Write one sample at
      a time
149   }
150   */ //End of IIR filter implementation
151
152   // Your implementation
153
154
155   // Write data
156   // ...
157   //  sf_writef_float(sndFileOut, sig_out, 1);
158   // ...
159
160   sf_close(sndFile);
161   sf_write_sync(sndFileOut);
162   sf_close(sndFileOut);
163   //Free all pointers
164   free(sig_in);
165   free(sig_out);
166
167   return 1;
168 }
169
170 float echo(float x_in, float *buffer, int Ne, int *index){
171   float a[]= {0.8, 0.2}; // Original and echo coefficients
172   float y_out = a[0] * x_in + a[1] * buffer[*index]; // Output sample is
      the sum of current input and delayed sample
173   buffer[*index] = x_in; // Update delay buffer with current input sample
174   *index = (*index + 1) % Ne; // Update index for circular buffer
175   return y_out;
176 }
177
178 void generate_fir_coefficients(double *g, int N, float notch_freq){
179   int L = N/2; // Center Frequency
180   double wc = 2 * PI * notch_freq / Fs; // Normalized Frequency
181   double h[N+1]; // Impulse Response of Ideal Notch Filter
182   double sum_h = 0.0; // Sum of Impulse Response
183
184   for(int n = 0; n<=N; n++){
185     if(n == L){
186       h[n] = wc / PI;
187     }
188     else{
189       h[n] = sin(wc * (n - L)) / (PI * (n - L));
190     }
191     sum_h += h[n];
192   }
193
194
195   double sum_g = 0.0; // Sum of FIR Coefficients
196   for(int n = 0; n<=N; n++){
197     h[n] = h[n] / sum_h; // Normalize the impulse response
198     g[n] = -2 * h[n] *cos(wc*(n-L)); // FIR Coefficients
199     if(n==L){
200       g[n] += 1; // Add 1 to the center coefficient
201     }
```

```
    sum_g += g[n];
  }

}

float fir_filter(float x_in, double *g, float *x, int N){

  for(int i = N; i > 0; i--){
    x[i] = x[i-1]; // Shift input buffer
  }
  x[0] = x_in; // Current input sample

  float y_out = 0.0f; // Current output sample
  for(int i = 0; i <= N; i++){
    y_out += g[i] * x[i]; // Convolution of input signal with FIR
    coefficients
  }

  return y_out;

}

float iir_filter(float x_in)
{
  float r = 0.95; // Pole radius
  float w0 = 2* PI * 2400/Fs; // Normalized Frequency

  float b[] = {1, -2*cos(w0), 1}; // IIR coefficients
  float a[] = {1, -2*r*cos(w0), r*r}; // IIR coefficients

  static float x[3] = {0,0,0}; // Input signal buffer for IIR filter
  static float y[3] = {0,0,0}; // Output signal buffer for IIR filter

  for(int i = 2; i > 0; i--){
    x[i] = x[i-1]; // Shift input buffer
    y[i] = y[i-1]; // Shift output buffer
  }

  x[0] = x_in; // Current input sample
  float y_out =  (b[0]*x[0] + b[1]*x[1] + b[2]*x[2] - a[1]*y[1] - a[2]*y
    [2])/a[0]; // Calculate current output sample using difference
    equation
  y[0] = y_out; // Update output buffer with current output sample
  return y_out;
}
```

Listing 10: Full C code