Indian Statistical Institute, Kolkata

Project for 'Statistical Methods -I'

Bachelors of Statistics (Hons.), 2023-26

De-noising Audio

Adeesh Ajay Devasthale (BS2303)
Aditya Aryan (BS2305)
Deep Jayesh Hariya (BS2324)
Nishant Lamboria (BS2334)
Sauparna Kar (BS2343)
Srijan Bhowmick (BS2352)

Advisor/Instructor
DR. ARNAB CHAKRABORTY
Associate Professor, Applied Statistics Unit

Monday 27th November, 2023

Contents

1 Abstract

This project delves into analyzing audio clips capturing people's speech amidst mild background noises. The primary goal is to effectively remove the pauses between voices to enhance clarity.

The current approach leans towards simplicity, employing fundamental statistical tools like variance and mode. It operates under the assumption that pause intervals exhibit low variance due to consistent background noise. The results validate the method's accuracy in detecting and successfully de-noising these pauses.

Additionally, the implementation of the Fast Fourier Transform algorithm explores avenues for complete background noise removal, mimicking functionalities found in modern software.

2 Initial Efforts

In our initial approach, we aimed to break down the signal into simpler sinusoidal components to discern the presence of voice or noise frequencies within it. To accomplish this, we explored the widely recognized Fast Fourier Transform (FFT) algorithm. The Fourier transformation, a mathematical tool employed to dissect a time-domain signal and portray it in the frequency domain, served as our primary method. Here's an illustration:

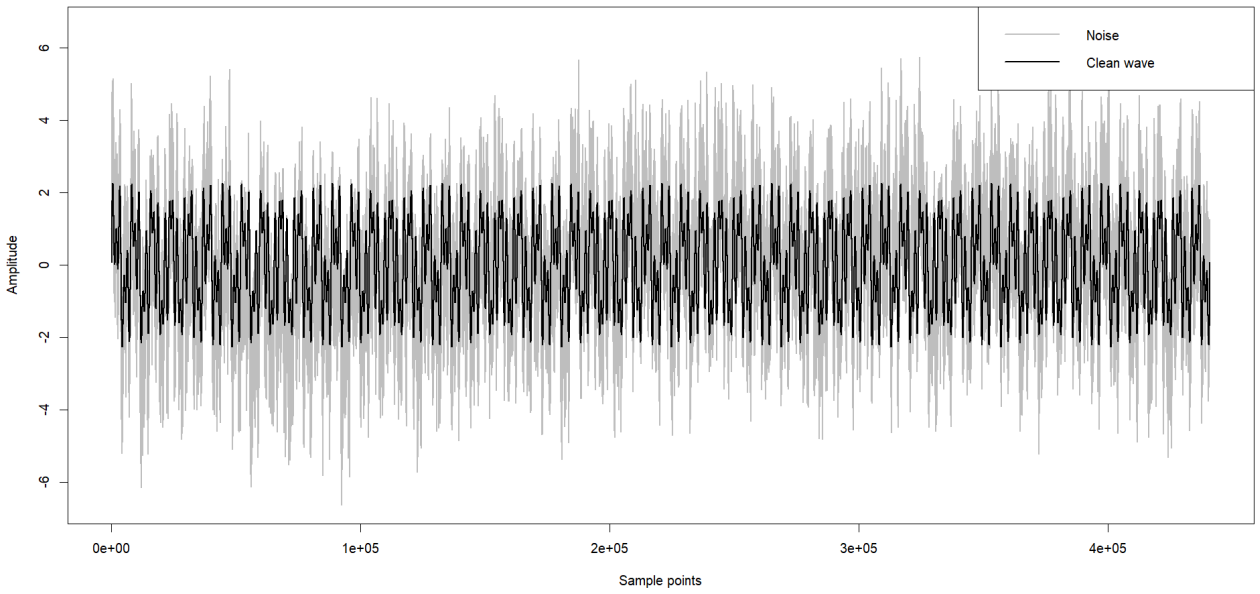

Figure 2.1: A sine wave mixed with random noise

We applied the algorithm on a signal composed of three sine waves with frequencies 6Hz, 17Hz, and 29Hz, heavily perturbed by random noise (refer to figure 2.1). The resulting output is presented below.

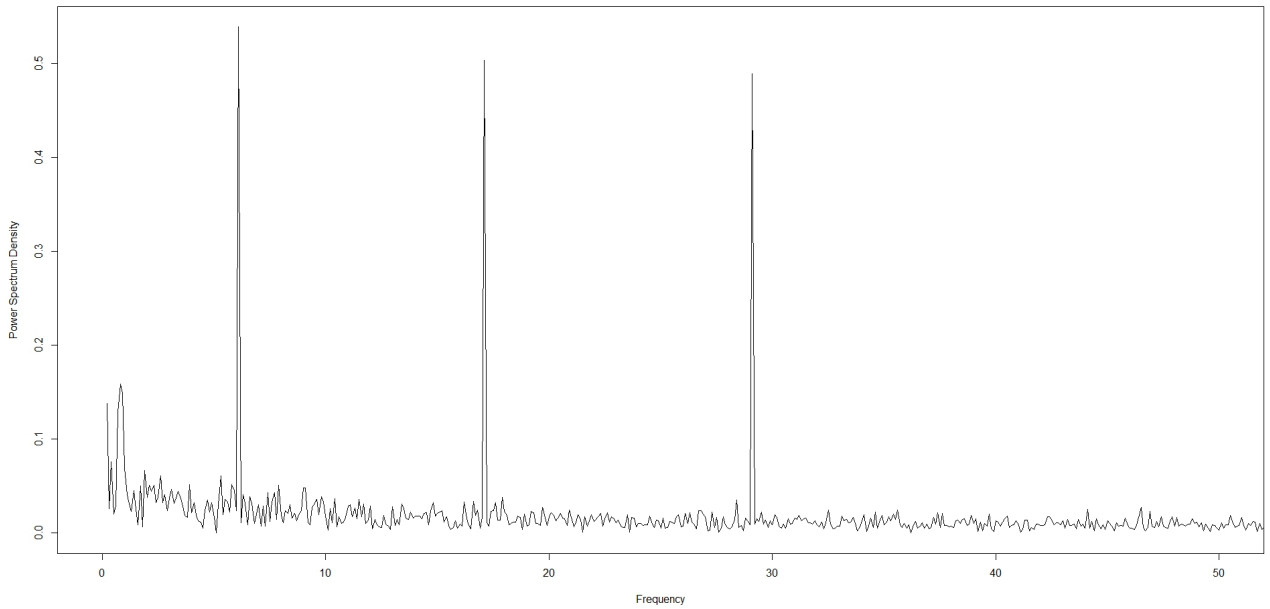

Figure 2.2: Zoomed plot of the Power Spectrum Density of the wave in figure 2.1

The algorithm demonstrates precision in identifying consistent frequencies within signals. However, its performance tends to be less satisfactory when applied to real-world audio signals.

The processed output obtained after applying the algorithm to the sound wave depicted in figure 3.1 is presented below.

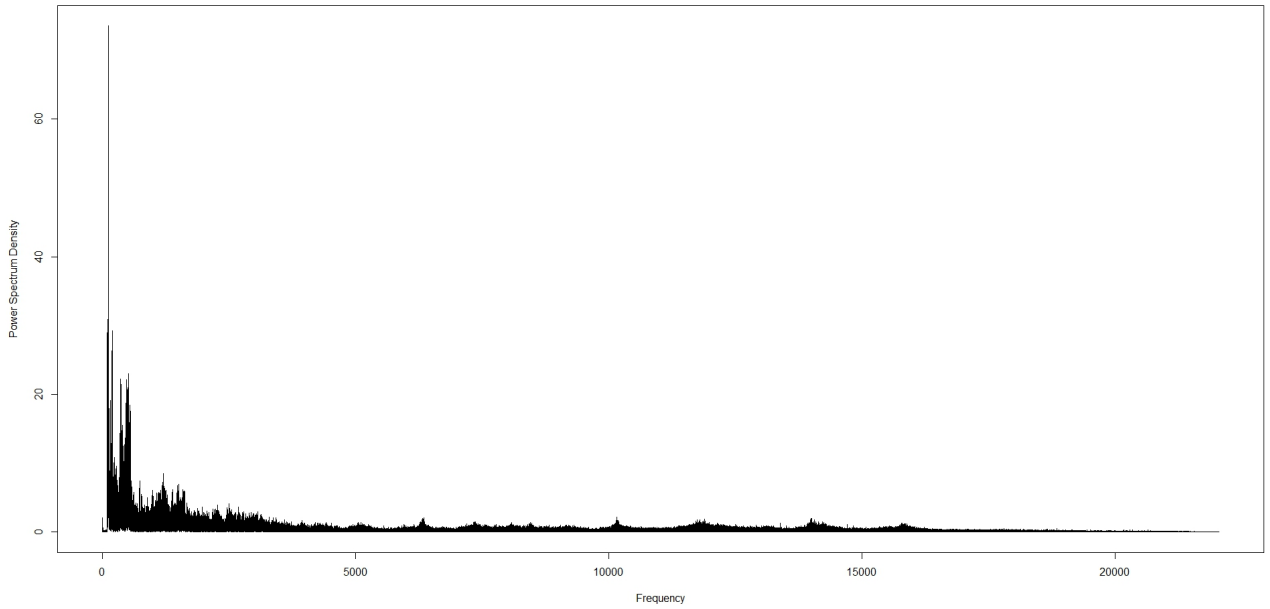

Figure 2.3: Power Spectrum Density of a real-world Audio Clip

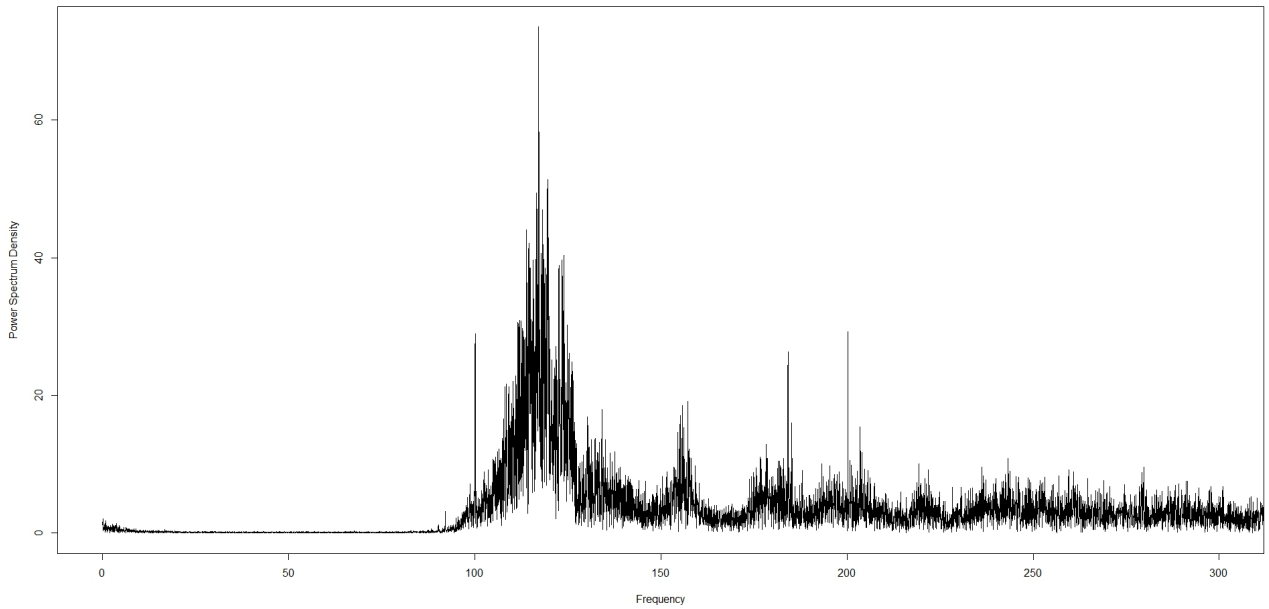

Figure 2.4: Zoomed Power Spectrum Density

The power spectrum density displayed multiple peaks. Moreover, the frequency range of the human voice falls within the broader range of frequencies for noise. As a result, we were unable to advance this particular idea any further.

3 A Statistical Approach

Building upon insights gained from our previous unsuccessful attempt, we opted for a fresh perspective to tackle the challenge. Our exploration led us to the intriguing capability of the human eye in discerning between areas of voice and noise within an audio clip with a steady mild background noise.

Inspired by this observation, we endeavored to translate this “Fuzzy Logic” into a practical algorithm, primarily leveraging simple statistical measures, particularly variance.

We examined the amplitude graph of the audio clip. In areas containing both voice and noise, the amplitudes were notably higher compared to regions consisting solely of noise (refer to figure 3.1).

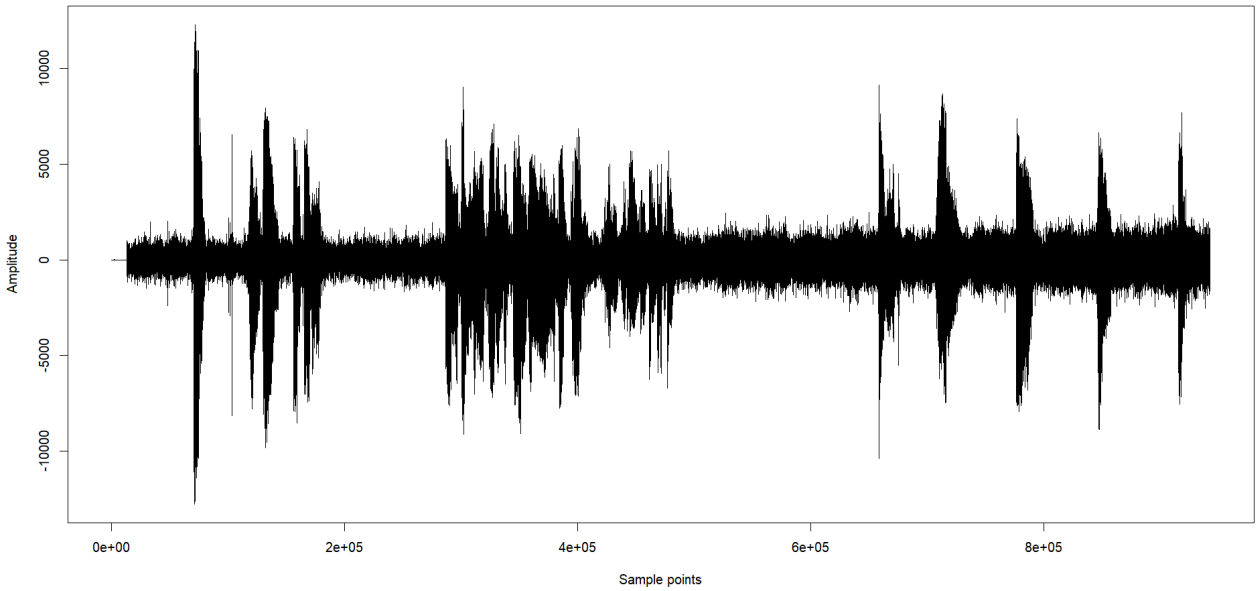

Figure 3.1: Sample audio clip

3.1 Fundamental Idea

Sound propagates as longitudinal waves. Given that voice amplitudes surpass those of noise, the distance between consecutive compression and rarefaction in voice are notably higher than those in noise. Consequently, the variation within a window exclusively containing noise was considerably lower in comparison (refer to figure 3.2).

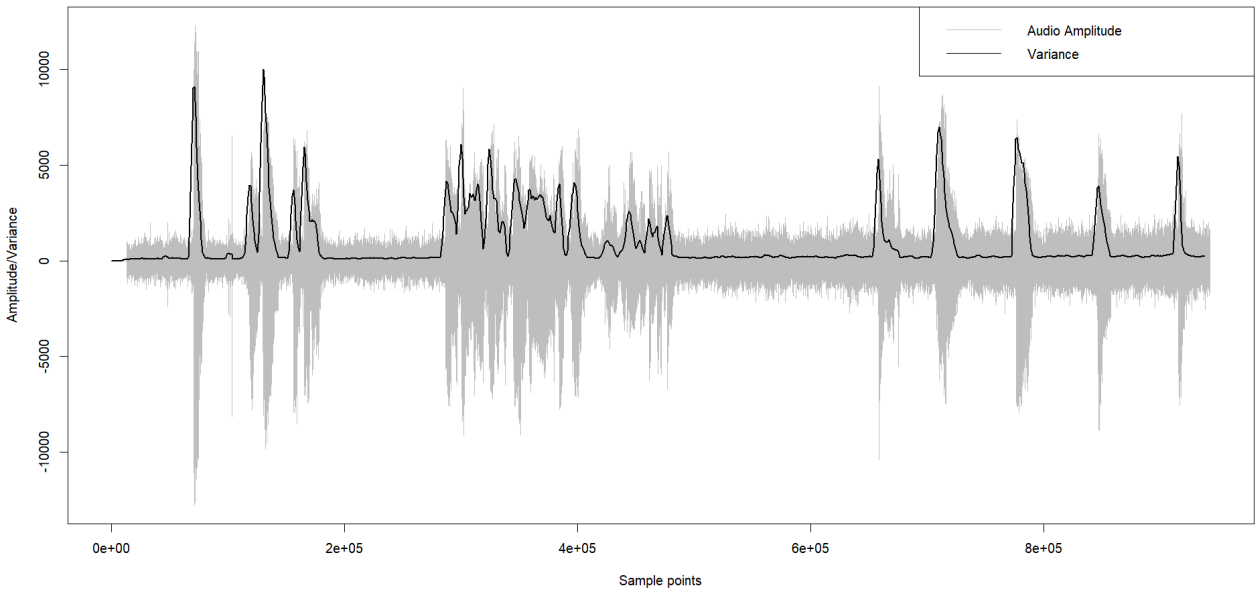

Figure 3.2: Amplitude and Variance Graph

3.2 Proposed Algorithm

Algorithm 1 De-noising Algorithm

Input: $Aud \leftarrow audio.wav$ ▷ The *.wav* file to be de-noised

$A_0 \leftarrow Aud$@left ▷ The array containing amplitude of the audio

$r \leftarrow Aud$@sampling.rate ▷ Number of data points of audio in a single second

$n \leftarrow \text{len}(A_0)$ ▷ The length of the audio

$w \leftarrow \lfloor \frac{r}{10} \rfloor - 1$ ▷ The window width

$V_0 = [\,]$ ▷ Initialising array to store variance values

$X \leftarrow \text{sequence}(\text{start} = 1; \text{stop} = n - w; \text{step} = 1000)$ ▷ Sequence of window steps

for $i \leftarrow X$ **do**

    $V_0.\,\text{append}(\text{var}(A_0[i : (i + w)]))$

end for

$V \leftarrow 1000 \cdot \dfrac{V_0 - \min(V_0)}{\max(V_0) - \min(V_0)}$ ▷ Scaling V_0 appropriately

$Z \leftarrow \lfloor \frac{V}{25} \rfloor$ ▷ Putting variance values in groups of length 25

$cut \leftarrow 25 \cdot (\text{mode}(Z) + 1) + 1$ ▷ The proposed cutoff

$A \leftarrow A_0$ ▷ Replicating A_0 in new variable A and editing it to get final audio

for $i \leftarrow 1, 2, \ldots, \text{len}(V)$ **do**

    if $V[i] < cut$ **then**

        $t \leftarrow X[i]$

        $A[t : (t + w)] \leftarrow 0$ ▷ Silencing the part below cutoff

    else if $cut \leqslant V[i] < (1.5) \cdot cut$ **then**

        $t \leftarrow X[i]$

        $A[t : (t + w)] \leftarrow \dfrac{A[t : (t + w)]}{2}$ ▷ Scaling additional parts for cushioning

    end if

end for

$OutAud \leftarrow Aud$ ▷ Replicating original wave for output

$OutAud$@left $\leftarrow A$ ▷ Replacing the amplitude array with the updated version

Output: $OutAud$ ▷ The de-noised *.wav* file

We used a sliding window and continuously calculated its variance. The width of the window used here is one-tenth of a second (4410 sample points), and the sliding step was 1000 sample points. We mapped all the obtained variances to the interval $[0, 1]$ linearly such that $\min(Var) = 0$ and $\max(Var) = 1$ and then scaled them by 1000 to get a good visual idea.

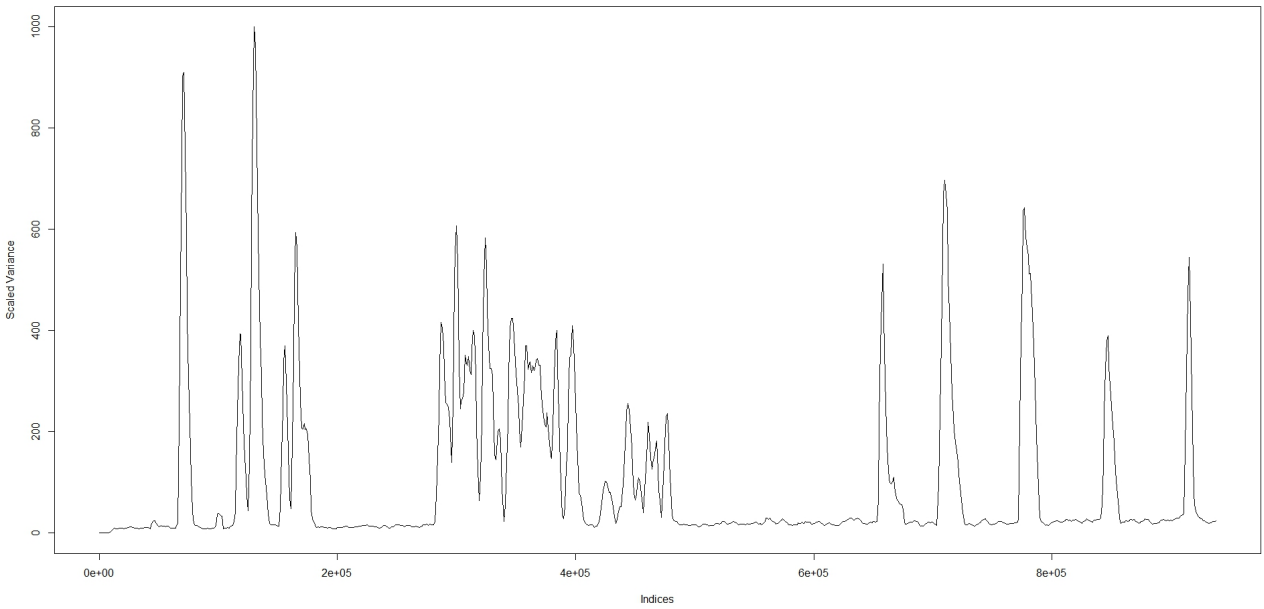

Figure 3.3: Scaled Variance

To determine the cutoff point, we organized variances into 40 equivalence classes, each with a consistent width of 25 units. Identifying the class with the highest frequency as the modal class, we established the cutoff as the value just beyond the upper limit of this modal class.

For variances below the cutoff, we systematically reduced the corresponding amplitudes to zero, effectively silencing purely noisy windows. However, for instances where windows contained both noise and voice elements, we considered the range $[cut, 1.5cut)$. We halved the amplitudes of all windows within this variance range.

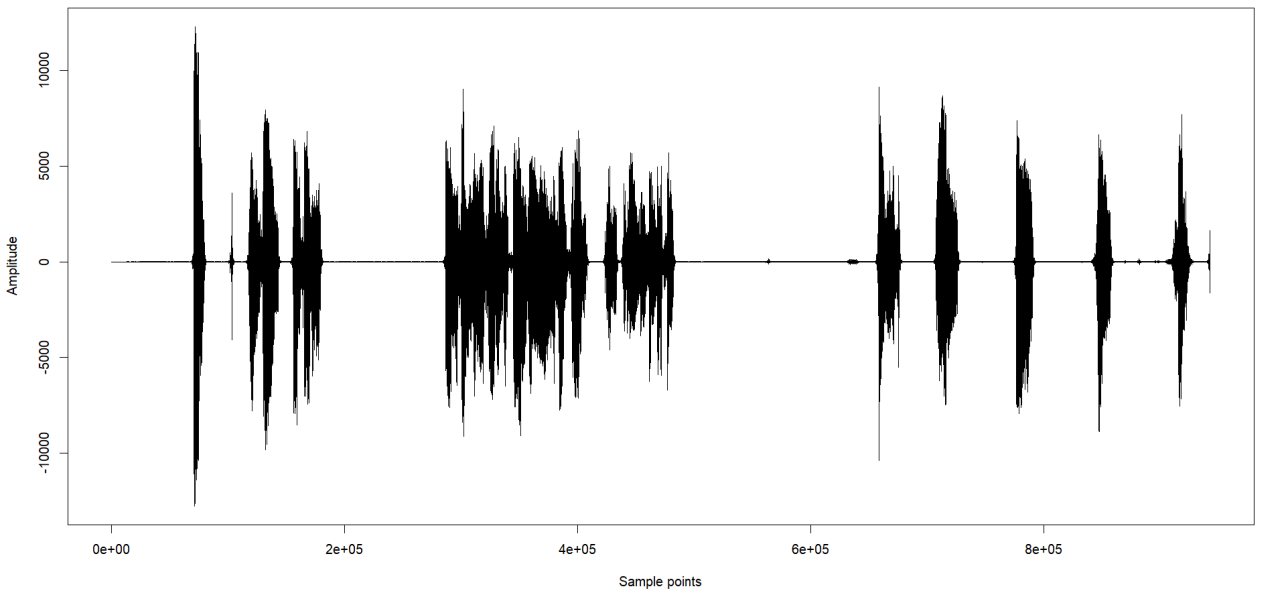

Figure 3.4: Final Output

4 Results

Below are the results achieved on two more sample audios with varied settings:

4.1 Sample Audio 1

This audio contained voices of two people instead of just one.

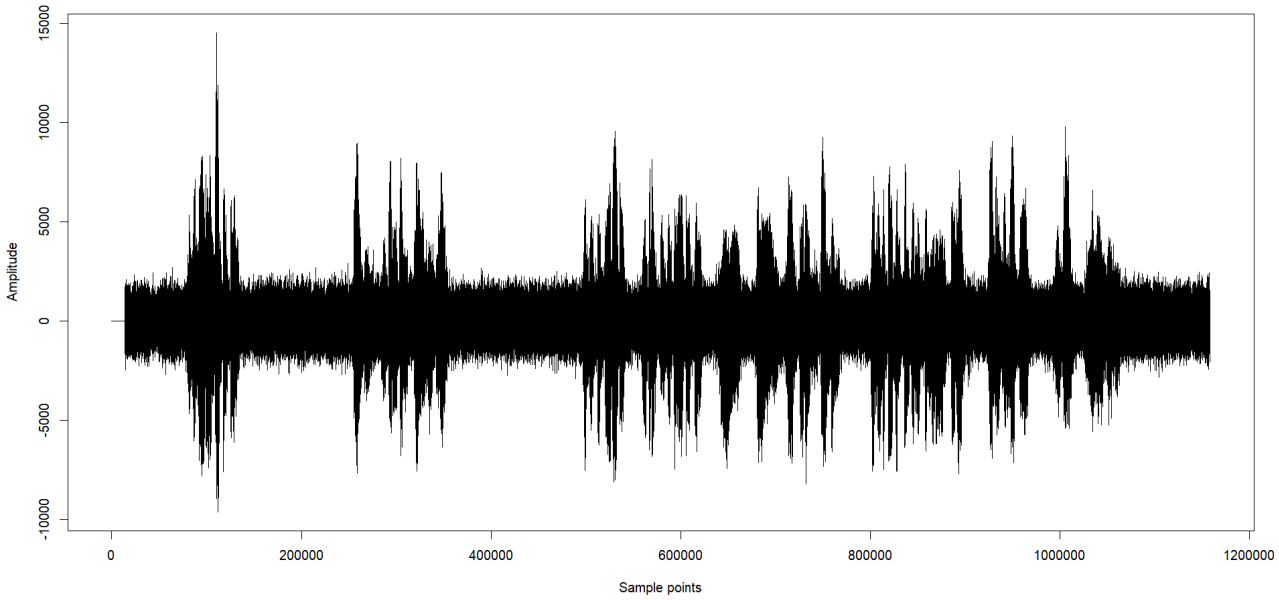

Figure 4.1: Sample Audio 1

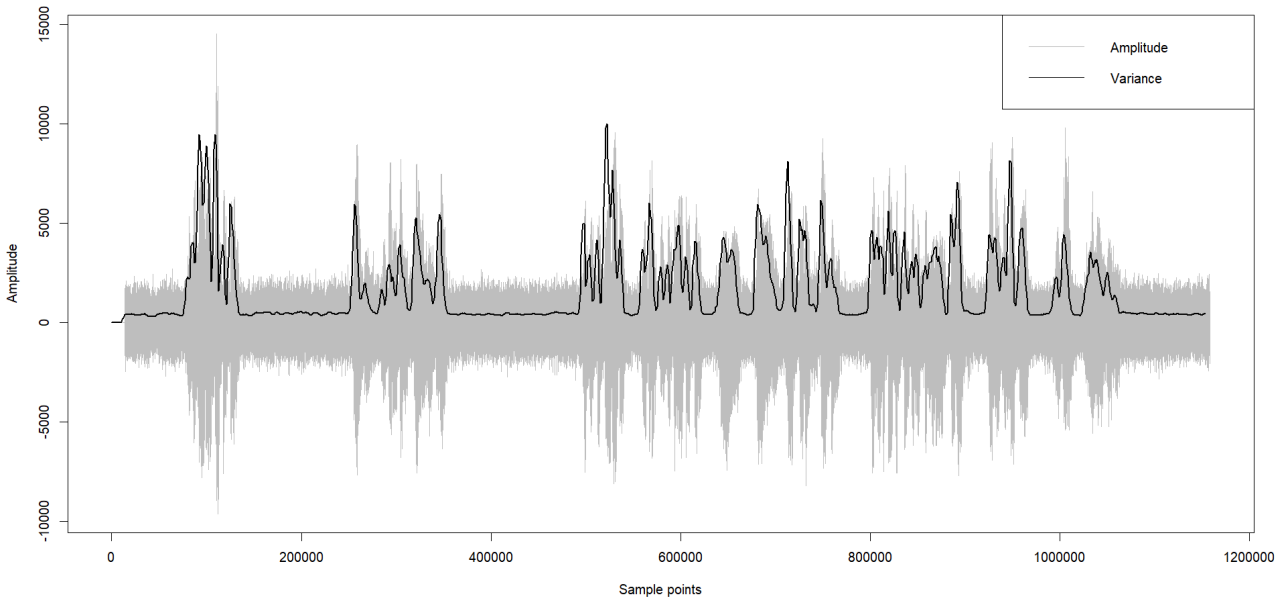

Figure 4.2: Amplitude and Variance Graph

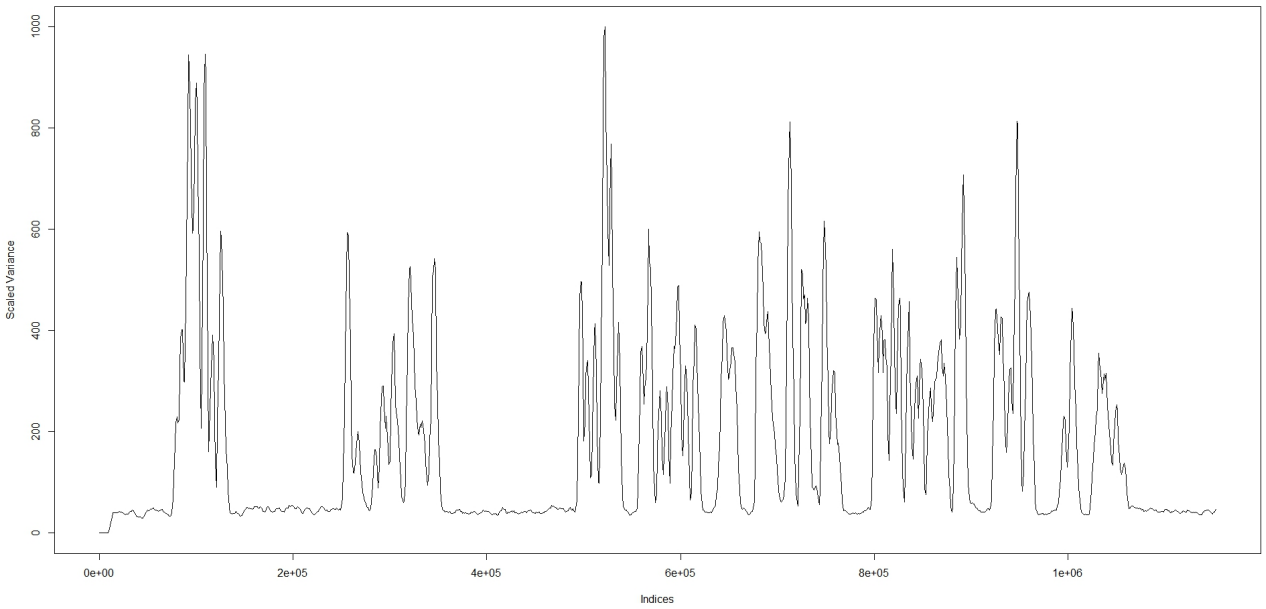

Figure 4.3: Scaled Variance Graph

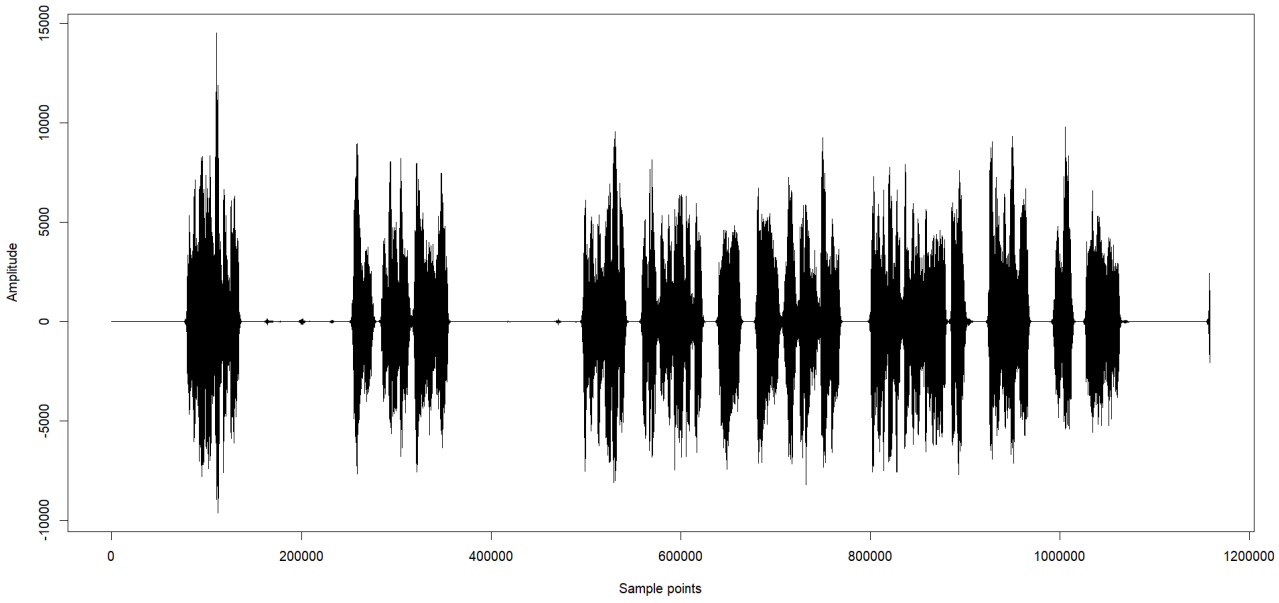

Figure 4.4: De-noised Sample Audio 1

4.2 Sample Audio 2

This audio didn't have any significant silence periods.

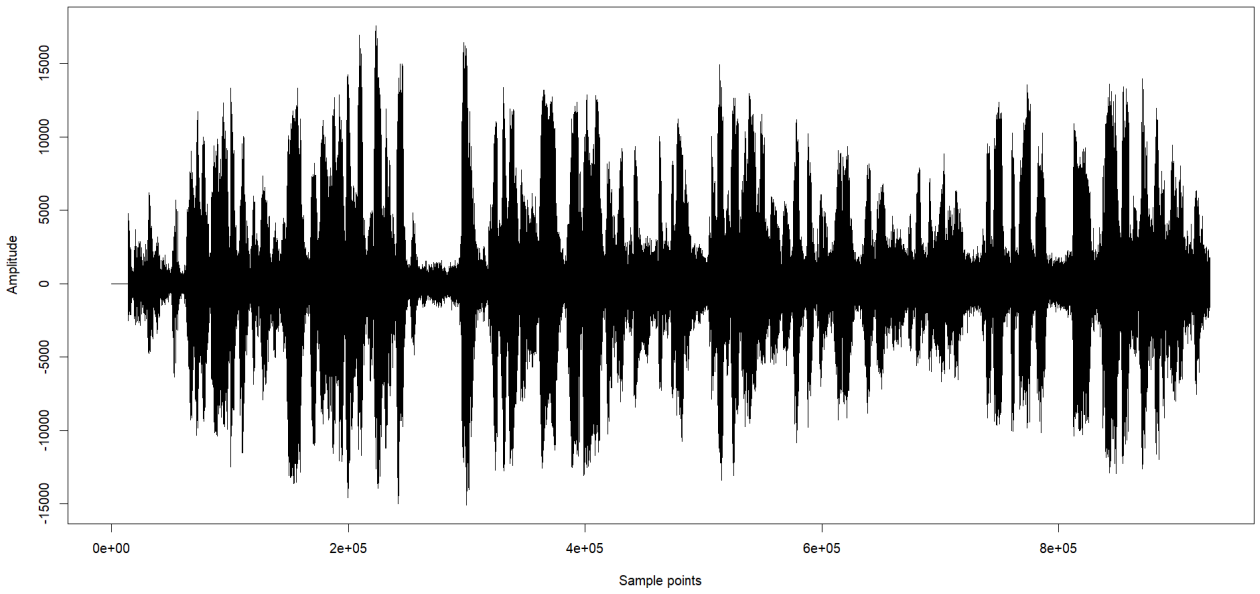

Figure 4.5: Sample Audio 2

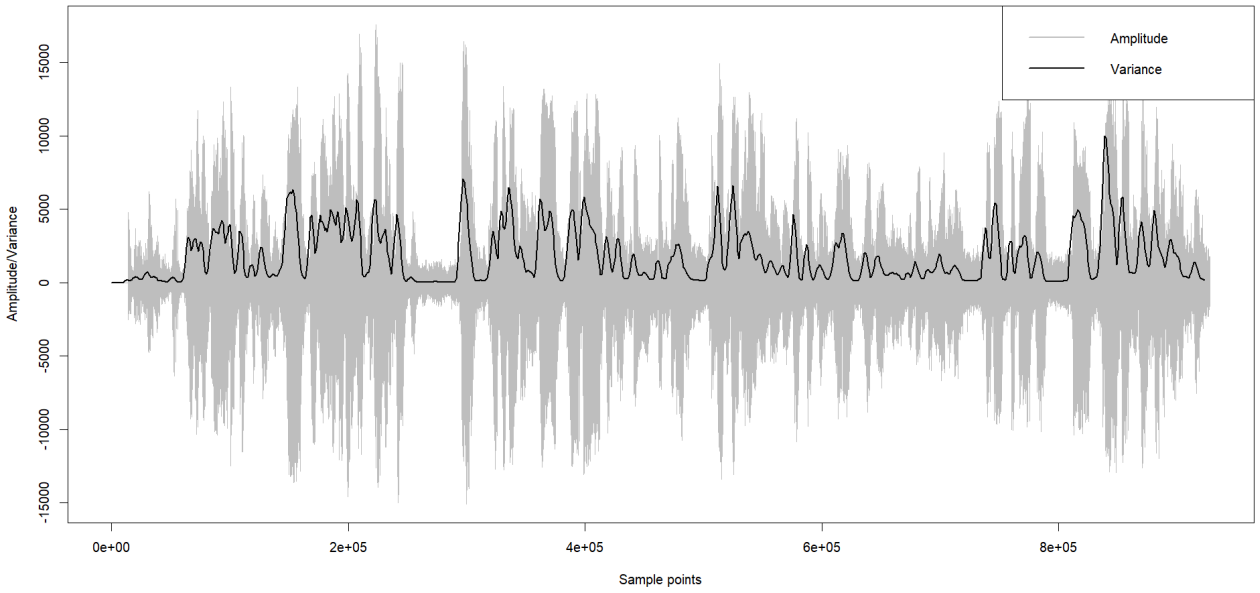

Figure 4.6: Amplitude and Variance Graph

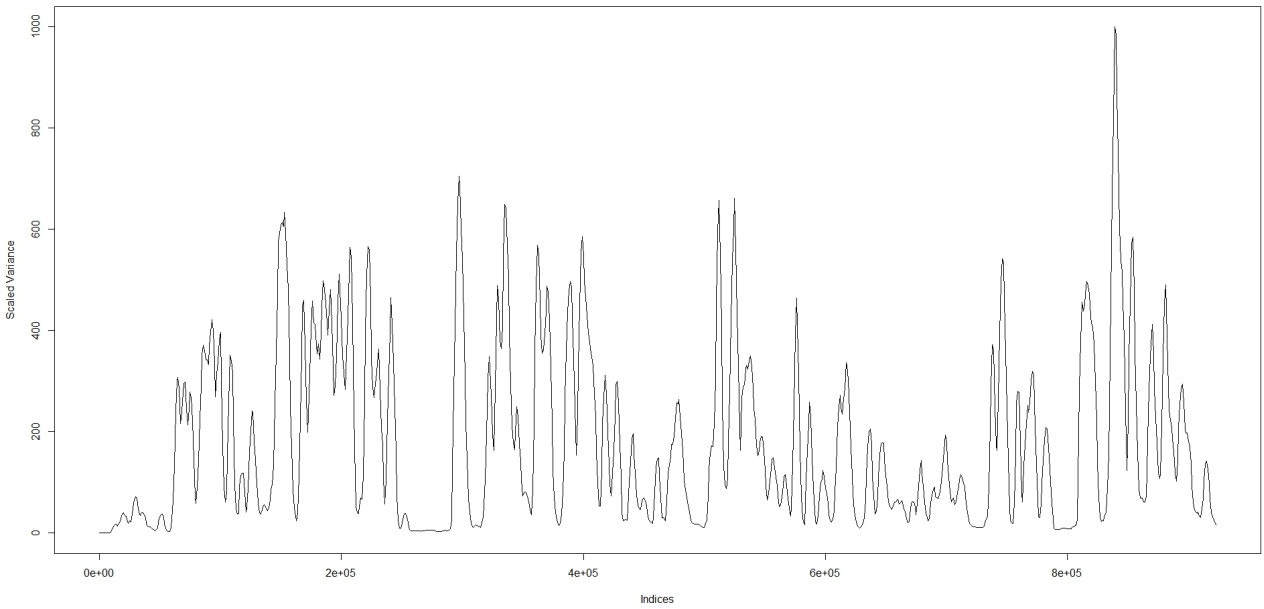

Figure 4.7: Scaled Variance Graph

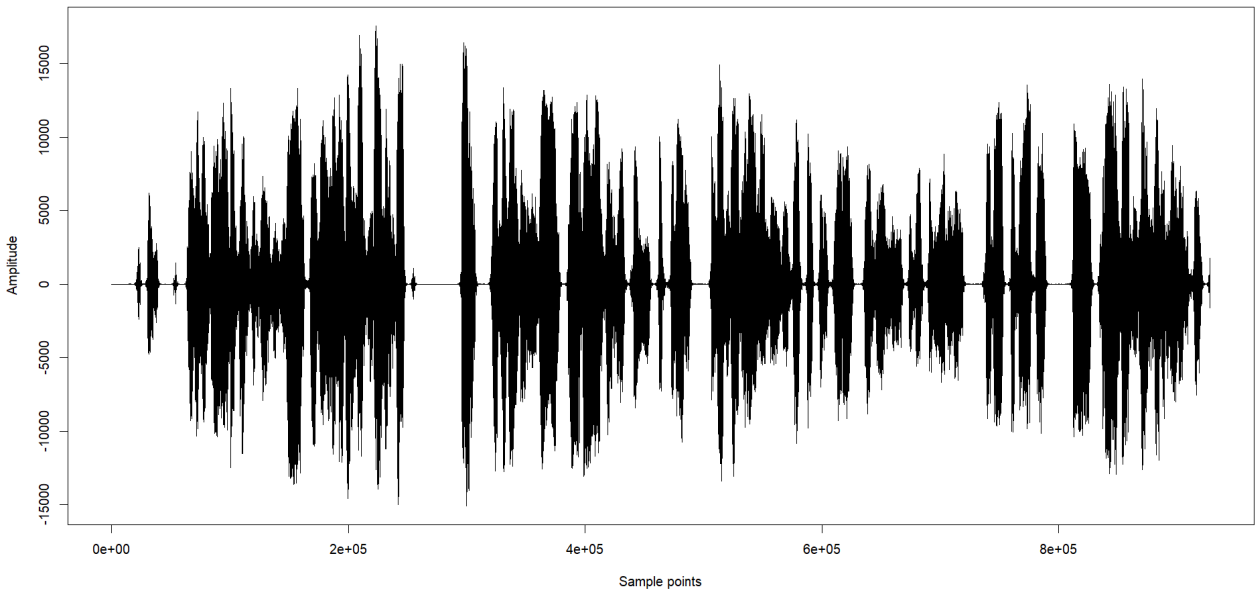

Figure 4.8: De-noised Sample Audio 2

5 Conclusion

In conclusion, our project marked a significant success as we effectively implemented a mechanism to identify and suppress noise during silent intervals within audio clips. This accomplishment showcases the potential of our approach in enhancing audio quality by eliminating unwanted disturbances.

While our current implementation excelled in silencing periods of inactivity, there exists room for future improvements. Notably, we can further refine our algorithm to address noise challenges during speech intervals, opening avenues for more comprehensive noise reduction. Additionally, considering the algorithm's current reliance on consistent and mild background noise, future enhancements could involve adapting it to handle scenarios with high-variance noise levels.

Throughout this project, we gained invaluable insights into the intricacies of digital signals, their generation, transmission, and storage. The hands-on experience with processing *.wav* files using R has fortified our skill set, and the exploration of the Fast Fourier Transform has expanded our understanding of modern signal processing techniques.

A Appendix

Here are the final R codes we used:

A.1 Main Algorithm

```
# Loading necessary libraries.
library(DescTools) #Required for implementing the mode function.
library(tuneR)     #Required for reading the input audio wave.

# Reading input audio file and extracting relevant information.
aud_wave = readWave("Audio/Main␣Audio.wav")
aud_data = aud_wave@left
sam_rate = aud_wave@samp.rate #Sampling rate
sam_count = length(aud_data)

# Defining window size based on sample rate.
win_size = as.numeric(sam_rate/10)-1

# Initializing variables for variance values (VAR) and sequence of indices.
VAR = c()
indices = seq(1, sam_count - win_size, 1000)

# Appending variance values for the overlapping windows.
for(j in indices){
  VAR = append(VAR, var(aud_data[j:(j+win_size)]))
}

# Normalizing variance values to a scale of 0 to 1000.
VAR = 1000*((VAR-min(VAR))/(max(VAR)-min(VAR)))

# Uncomment plot commands to view.
# plot(indices, VAR, type = 'l')

# Defining the cutoff for the noisy audio.
int_var = as.integer(VAR/25)
mode_var = Mode(int_var)
cut_off = 25*(mode_var+1)+1
# plot(int_var, type = 'l')

# Initializing a variable to store the de-noised audio data.
den_aud_data = aud_data

# Eliminating noise from the "noisy" parts determined by our cutoff.
for(i in 1:length(VAR)){
  if(VAR[i] < cut_off){
    k = indices[i]
    den_aud_data[k:(k+win_size)] = 0
  }
  else if(VAR[i] >= cut_off & VAR[i] < (1.5)*cut_off){
    k = indices[i]
    den_aud_data[k:(k+win_size)] = (den_aud_data[k:(k+win_size)])/2
  }
}

# plot(aud_data, type = 'l')
# plot(den_aud_data, type = 'l')

# Saving and writing the de-noised audio data.
den_aud_wave = aud_wave
den_aud_wave@left = den_aud_data

writeWave(den_aud_wave, "Output/denoised_Main␣Audio.wav")
```

A.2 FFT on the Sine Wave

```
#Loading necessary libraries
library(tuneR)

#Constructing a clean wave by combining 3 sine waves with different
#parameters including frequencies.

samp_rate = 44100 #Sampling rate
time = 10*samp_rate
clean_wave = sine(29, duration = time, from = 0.69, bit = 32) +
             sine(17, duration = time, from = 29, bit  = 32) +
             sine(6, duration = time, from = pi/23, bit = 32)

#Creating noise using the inbuilt noise function which is to be
#filtered by FFT.
noise_wave = 5*noise("power", duration = time, bit = 32)

#Combining the clean wave and noise.
comb_wave = clean_wave + noise_wave

#Plotting the combined wave and highlighting the clean and noisy parts.
plot(comb_wave@left, ty = 'l',col='grey',lwd=2,ylab='Amplitude',xlab='Sample␣points')
lines(clean_wave@left,lwd=2)
legend('topright',legend=c('Noise','Clean␣wave'),col=c('grey','black'), lty=1, lwd=2)

#Implementing Fast Fourier Transform on the combined wave and plotting it.
FFT = fft(comb_wave@left)
PSD = Mod(FFT)/time #Power Spectrum Density
freq = (samp_rate/time)*(1:time)
uplim = as.numeric(time/2) #Setting upper limit
plot(ylab='FFT',xlab='frequency',freq[2:uplim], PSD[2:uplim], ty = 'l')
#The three peaks correspond to the 3 frequencies of the original 3 sine waves.
```

A.3 FFT on the Audio Clip

```
#Loading necessary libraries
library(tuneR) #Required for reading the input audio wave

#Reading input audio file and extracting relevant information
aud_wave = readWave("Audio/audio_1.wav")
aud_data = aud_wave@left
sam_rate = aud_wave@samp.rate
sam_count = length(aud_data)

#Plotting the input audio wave
plot(xlab='Sample␣points',ylab='Amplitude',  aud_data, type = 'l')

#Implementing Fast Fourier Transform on the audio wave and plotting it
FFT = fft(aud_data)
PSD = Mod(FFT)/sam_count #Power Spectrum Density
freq = (sam_rate/sam_count)*(1:sam_count)
uplim = as.numeric(sam_count/2) #Setting upper limit
plot(ylab='FFT', xlab='Frequency',freq[2:uplim], PSD[2:uplim], type = 'l')
```