Jake Haapoja

Computer Science

Saint Cloud State University

Saint Cloud, MN, United States

JLHaapoja@stcloudstate.edu

**Abstract– The objective of this paper is to present an analysis of a basic Least Mean Squares (LMS) adaptive filtering algorithm as applied to speech recordings with various amplitudes of Additive Gaussian White Noise (AGWN). The sample data consists of audio recordings of 6 sentences from 4 individuals. These recordings were processed in python with various libraries, where a Gaussian distributed random list was generated and applied to the audio recording before being fed into the Least Mean Squares algorithm, using another sample of Gaussian noise as a reference signal. Results indicate that a naïve implementation of the algorithm is insufficient for audio processing, and actually has a negative impact on the recording quality. Future research can study the effects of altering the learning rate and filter order, as well as preprocessing the recordings to employ feed-forward techniques that will generate a more accurate reference signal.**

***Keywords–Adaptive Filtering, Least-Mean-Squares, LMS, Noise Cancellation***

**I. Introduction**

Noise cancellation is an important problem in many signal processing applications, such as speech recognition, audio processing, and telecommunications. One of the commonly used techniques for noise cancellation is the least mean squares (LMS) algorithm, which is a type of adaptive filter that can adaptively adjust its coefficients to minimize the mean square error between the filtered output and the desired signal. In this paper, we present a study of the basic LMS algorithm for noise cancellation and its performance in different scenarios. We investigate the effectiveness of the LMS algorithm in reducing gaussian noise in audio recordings. The results of this study can be useful for researchers and practitioners working in the field of signal processing and noise cancellation, and speech recognition.

**II. Implementation**.

The algorithm was implemented in multiple phases. Phase one consists of audio intake and processing, and had multiple parts. Phase two then involves analyzing and visualizing the data from each part of phase one. Each phase is handled by its own python program, with dependencies on the numpy library[4], the os library[5], the scipy library[6], the parselmouth library[7], and the matplotlib library[8]. All of these are included in the python virtual environment which is built on python version 3.10.9 AMD64 released Dec 6, 2022.[9] The algorithm used is based on the one described by Matous, C. [3]

The program requires audio to be a single channel recording in .wav format, recorded at 44100 samples per second. The recordings should also be placed within a folder titled ‘Data’ with subfolders as shown.

Data  
 |[sentenceFileName]  
 | |\_0riginal  
 | | |audio

| | | |recording 1.wav

| | | |...  
 | |\_[%noiseAmplitude]\_percent  
 | | |filtered  
 | | |noisy

| | |noise\_references  
 | |...

| |analysis

| |figures

**A. Phase One: Processing the Audio**

This first phase is handled by the program found in the file titled *processAudio.py* with the functions *noisify() and LMS()*. During this phase, the noise amplitudes to generate are first stored in a list variable. Then the list of sentence folders is extracted from the Data folder using the *os.listdir()* method. The folder list is then scanned through, with each audio file being processed individually, and the resulting noisy audio and filtered audio, along with an audio file representing the reference noise into their relevant folders.

The *noisify()* method takes a string containing the path to the source audio file, and a path to the storage location, along with an amplitude argument which defaults to 0.1. The amplitude argument is a floating point number between 0 and 1 and represents the fraction of the maximum absolute amplitude in the audio file. The method begins by validating the files, printing an error if the files don’t exist or are of the wrong type. It then formats the output file name with the proper amplitude percent number. Noise is then generated with a gaussian distribution mean of zero and standard deviation of one. Then it is scaled to the correct amplitude, and applied to the pure audio using the method described by commenter Noel Evans on stackoverflow.com. [10]

sRate, audioData = wf.read(inFile)  
noise = np.random.normal(0, 1, len(audioData))  
noise = noise \* amp \* max(abs(audioData))  
noisyAudio = audioData + noise

The resulting audio data is then saved to a new file in the relevant folder with the amplitude percent as an integer and noisy to indicate as such. The method then checks that the file was successfully created and returns to the main method.

The *LMS()* method then takes this noisy file and applies a least mean squares adaptive filter, which will be explained in further detail in another section. This filtered data is also saved to a new file in the relevant folder.

**B. Phase 2: Data Analysis**

The second phase handles analyzing the files produced during phase one. It begins similarly by scanning through the ‘Data’ folder to get a list of sentence folders. Then this list is looped through and for each sentence the *process\_data()* method is called by passing the sentence folder path as a string, and a list of amplitudes being analyzed.

The method begins by checking the validity of the sentence folder, then scanning the \\_0riginal\audio folder for a list of the pure audio files. Empy lists are created to store average error data, then the list of files is looped through. For each file, a string is created which is appended to and eventually printed to an analysis text file. After that, the *analyze\_waveforms()* method is called, passing the file name, the path to the original audio, and the list of amplitudes. This method gathers the files, then creates a 2x2 table of waveform graphs, with original in the top left, noisy in the top right, filtered in the bottom left, and the noise reference in the bottom right, all saved to a single image file. Then the average error between original and both error and filtered audios is calculated and returned. The values returned represent an average percent error for both the noisy and filtered audio at each amplitude setting. The numbers returned are the percentage difference from the original recording, and are rounded to the nearest integer. Then finally, the *draw\_spectrograms()* method is called with the same parameters as the waveforms method, which draws spectrograms for each of the files in the same 2x2 grid format. Both the waveform and spectrogram graphs are saved to the figures folder for that sentence, and the analysis text file is saved to the analysis folder.

**C. The LMS Algorithm**

The least Mean Squares filtering algorithm is a type of linear regression, which uses a reference signal and a vector of initial weights to alter the values of each sample of the audio file progressively a number of samples at a time indicated by the order variable, which also determines the number of adaptive parameters used in the algorithm. The algorithm could be considered a type of artificial intelligence, despite each instance of the filter being single use with no persistent model between uses, it employes parameters that are adjusted as it ‘trains’ itself on the given signal and the reference signal. The core mathematical operation of the filter can be depicted by the following equation. This equation takes a number of samples equal to the number of adaptive parameters, here denoted by the vector xT(K) and multiplies it element wise by with vector of parameters w(k).

[1] y(k)=xT(k)w(k)

This equation gives a single output value, shown as y(k) which is then stored at the index of the start of the sample vector. The sample vector is then shifted one item at a time along the length of the source data list.

At each time step k, the weights are adapted by calculating an error value for the filter output and changing the weights in the filter coefficients vector by the following equation. Where μ is the learning rate, e(k) is the error, and x(k) is the sample vector.

[2] Δw(k)=0.5(μ∂e2(k)/∂w(k)) =μ⋅e(k)⋅x(k)

**III. Results & Analysis**

Reviewing the analysis files produced by phase 2, it is evident that the filter at default settings of mu = 0.01 and order 100 drastically decreased the audio quality. The amount varies between sentences between a 10% and 50% increase in the error value from noisy to filtered when compared to the original audio. This was consistent among all amplitude levels tested.

This indicates that other techniques are required, or the implementation is flawed in some way. A potential confounding factor is that the reference signal generated was orders of magnitude smaller than the noise signal added. This is very likely as the errors were in the hundreds of thousands of percent in both noisy and filtered audio, with some exceeding well over 1 million. Graphical analysis for a subset of source files can be found in the appendix, and the figures generated for all files can be found in the accompanying results binder.

**IV. Conclusion**

The goal of this research was to assess the effectiveness of the LMS signal filtering algorithm which is used in many applications to clear noise from digital audio or other digital signals. The algorithm remains one of the most commonly used methods of noise reduction for audio signals, however the results of this study suggest that alone it is insufficient to produce high quality audio from noisy audio. These findings suggest that other techniques must be used in tandem with the Least Mean Squares algorithm to see significant improvements, and that alone, and with the wrong initial parameters and reference signals, it can actually reduce the quality of the signal.

**V. References**

[1]

“Privateline.com Telephone History: Alec Reeves,” *web.archive.org*, May 11, 2014. https://web.archive.org/web/20140511182503/http://www.privateline.com/TelephoneHistory2/reeves.html (accessed Apr. 28, 2023).

[2]

[3]

C. Matousc, “Least-mean-square (LMS) — Padasip 1.2.1 documentation,” *matousc89.github.io*. https://matousc89.github.io/padasip/sources/filters/lms.html

[4]

“NumPy Documentation,” *numpy.org*. https://numpy.org/doc/

[5]

Python, “os — Miscellaneous operating system interfaces — Python 3.8.0 documentation,” *Python.org*, 2019. https://docs.python.org/3/library/os.html

[6]

SciPy, “SciPy.org — SciPy.org,” *Scipy.org*, 2020. https://scipy.org/

[7]

“Parselmouth – Praat in Python, the Pythonic way — Parselmouth 0.4.3 documentation,” *parselmouth.readthedocs.io*. https://parselmouth.readthedocs.io/en/stable/

[8]

Matplotlib, “Matplotlib: Python plotting — Matplotlib 3.1.1 documentation,” *Matplotlib.org*, 2012. https://matplotlib.org/

[9]

“Python Release Python 3.10.9,” *Python.org*. https://www.python.org/downloads/release/python-3109/ (accessed Apr. 28, 2023).

[10]

N. Evans, “adding noise to a signal in python,” *Stack Overflow*. https://stackoverflow.com/a/26181710 (accessed Apr. 28, 2023).

[11]

M. Do, “Linear Regression and Least-Squares Filtering.” Accessed: Apr. 28, 2023. [Online]. Available: https://courses.engr.illinois.edu/ece420/fa2017/AdaptiveSignalProcessing.pdf

**V. Appendix**

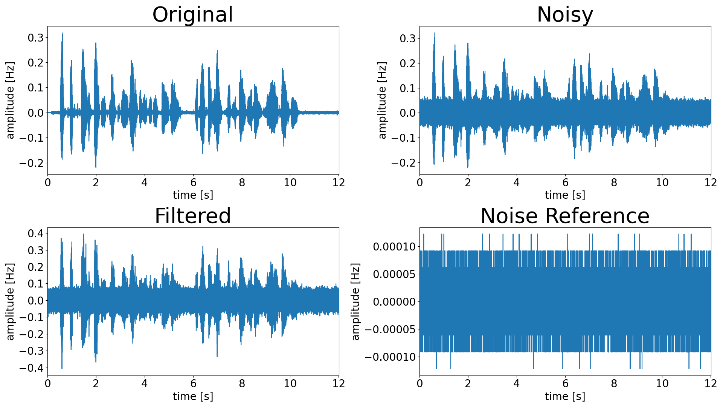
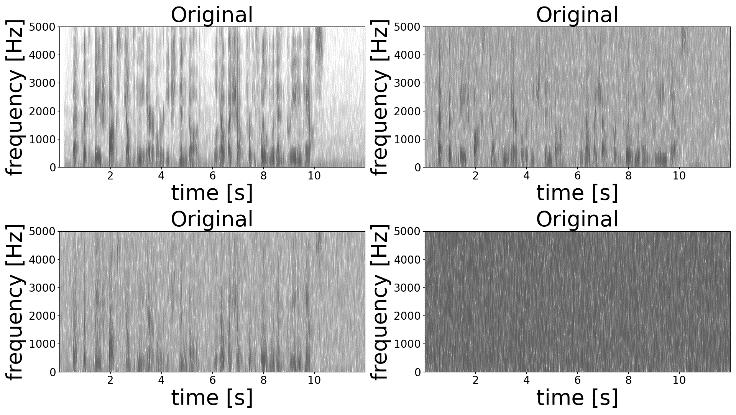
**A. Program Code**

#""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""  
#  
# processAudio.py  
# noisifies, then filters the audio samples, saving the noisy,  
# filtered, and reference noise signals as sound files  
#  
# Preconditions: Data folder is formatted as shown below:  
# sentence folder names can be arbitrary but must have subfolders  
# structured as below  
#  
# Data  
# |[sentenceFileName]  
# | |\_0riginal  
# | | |audio  
# | |\_[%noiseAmplitude]\_percent  
# | | |filtered  
# | | |noisy  
# | |... (repeat for number of amplitudes to study)  
#  
# Postconditions: all audio files in all sentence folders will have  
# been noisified, filtered, and then stored in their respective  
# folders  
#  
# Author: Jacob Haapoja  
# ©2023  
#  
#""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""  
  
#import dependencies  
import numpy as np  
import scipy.io.wavfile as wf  
import os  
import os.path as path  
  
def noisify(inFile: str, outFile: str, amp = 0.1):  
 if path.isfile(inFile) and inFile.endswith(".wav"):  
 outFile = outFile.replace(".wav", f"\_{int(amp\*100)}\_noisy.wav")  
 sRate, audioData = wf.read(inFile)  
 noise = np.random.normal(0, 1, len(audioData))  
 noise = noise \* amp \* max(abs(audioData))  
 noisyAudio = audioData + noise  
 wf.write(outFile, sRate, noisyAudio.astype(np.int16))  
  
 if path.exists(outFile):  
 return True  
 else:  
 print('ERROR: problem writing to file, exiting noisify...')  
 return False  
 else:  
 print('ERROR: problem reading from file or incorrect file type, exiting noisify...')  
 return False  
  
# applies a least mean squares filter to the sound indicated by inFile  
def LMS(inFile: str, outFile: str, ref\_out: str, amp = 0.1, lRate=0.01, fOrder=100):  
 sRate, audioData = wf.read(inFile)  
  
 reference = np.random.normal(0, 1, len(audioData))  
  
 filtCoef = np.zeros(fOrder)  
 filteredAudio = np.zeros(len(audioData))  
  
 for n in range(fOrder, len(audioData) - 100):  
 noiseInput = reference[n - fOrder:n]  
 filtOutput = np.dot(filtCoef, noiseInput)  
 error = audioData[n] - filtOutput  
 filtCoef += lRate \* error \* noiseInput  
 filteredAudio[n] = audioData[n] - filtOutput  
  
 wf.write(ref\_out.replace(".wav", f"\_{int(amp \* 100)}\_noise\_ref.wav"), sRate, np.int16(reference))  
 wf.write(outFile.replace(".wav", f"\_{int(amp \* 100)}\_filtered.wav"), sRate, np.int16(filteredAudio))  
 return True  
  
  
def main():  
 # declare amplitudes to generate  
 noiseAmplitudes = [0.05, 0.25, 0.5]  
  
 # get list of sentence files from 'Data' directory  
  
 senFolders = os.listdir("Data")  
  
 # For each Sentence Folder  
 for sentence in senFolders:  
 # define orginal data directory path  
 origin = "Data\\" + sentence + "\\\_0riginal\\audio"  
 # get list of recording files  
 recordings = os.listdir(origin)  
 # for each recording file do:  
 for rec in recordings:  
 if rec.endswith(".wav"):  
 path = origin + "\\" + rec  
 for amp in noiseAmplitudes:  
 outPath = origin.replace("\_0riginal\\audio", f"\_{int(amp \* 100)}\_percent\\noisy\\") + rec  
 noisify(path, outPath, amp)  
 path\_noisy = outPath.replace(".wav", f"\_{int(amp\*100)}\_noisy.wav")  
 outPath = origin.replace("\_0riginal\\audio", f"\_{int(amp \* 100)}\_percent\\filtered\\") + rec  
 ref = outPath.replace("filtered", "noise\_references")  
 LMS(path\_noisy, outPath, ref, amp)  
 return True  
   
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

#""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""  
#  
# processData.py  
# takes in a series of sound recordings and uses parselmouth,   
# Librosa, and Pandas to create praat annotations for each  
# recording using the corresponding english transcription  
# to prepare the data for insertion to a Recurrent Neural Network  
#  
# Preconditions: processAudio has been run and data files are in  
# the correct tree format  
#  
# Postconditions: a 2x2 grid for the waveforms and spectrograms  
# have been generated in the figures folder, along with an  
# analysis file listing average percent error rounded to the  
# nearest whole number  
#  
# Author: Jacob Haapoja  
# ©2023  
#  
#""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""  
  
# import dependencies  
import parselmouth as pm  
import numpy as np  
import matplotlib.pyplot as plt  
import os.path as path  
import os  
  
# draws the spectrograms for one sound file  
def draw\_spectrograms(fileName: str, orig\_path: str, amplitudes=[0.05, 0.25, 0.5], dynamic\_range = 70):  
 # get the original sound  
 original = pm.Sound(orig\_path)  
 # set figure save location  
 figPath = orig\_path.replace("\_0riginal\\audio\\" + fileName, "figures\\")  
  
 for i, amp in enumerate(amplitudes):  
 # set the location of the files for amplitude amp  
 data\_path = orig\_path.replace("\_0riginal\\audio\\" + fileName, f"\_{int(amp \* 100)}\_percent\\")  
 noisy\_path = data\_path + "noisy\\" + fileName.replace(".wav", f"\_{int(amp \* 100)}\_noisy.wav")  
 filt\_path = data\_path + "filtered\\" + fileName.replace(".wav", f"\_{int(amp \* 100)}\_filtered.wav")  
 ref\_path = data\_path + "noise\_references\\" + fileName.replace(".wav", f"\_{int(amp \* 100)}\_noise\_ref.wav")  
 # import processed audio as Sound objects  
 noisy = pm.Sound(noisy\_path)  
 filtered = pm.Sound(filt\_path)  
 reference = pm.Sound(ref\_path)  
 # create array containing all sound objects to study  
 sounds = [original, noisy, filtered, reference]  
 # set figure size to 1930x1080  
 plt.figure(figsize=(19.2, 10.8))  
 # for each sound  
 for j, snd in enumerate(sounds):  
 # specify subplot  
 plt.subplot(2, 2, j+1)  
 # extract spectrogram object  
 spectrogram = snd.to\_spectrogram()  
 # draw spectrogram  
 X, Y = spectrogram.x\_grid(), spectrogram.y\_grid()  
 sg\_db = 10 \* np.log10(spectrogram.values)  
 plt.pcolormesh(X, Y, sg\_db, vmin=sg\_db.max() - dynamic\_range, label="spectrogram", cmap='binary', alpha=0.7)  
 plt.ylim([spectrogram.ymin, 5000])  
 # determine title  
 if i == 0:  
 plt.title("Original", fontsize=40)  
 elif i == 1:  
 plt.title("Noisy", fontsize=40)  
 elif i == 2:  
 plt.title("Filtered", fontsize=40)  
 else:  
 plt.title("Noise Reference", fontsize=40)  
 # set labels and font sizes  
 plt.xlabel("time [s]", fontsize=40)  
 plt.xticks(fontsize=20)  
 plt.ylabel("frequency [Hz]", fontsize=40)  
 plt.yticks(fontsize=20)  
 # end loop  
 # adjust layout so labels don't overlap  
 plt.tight\_layout()  
 # save the waveform plots to a png file corresponding to the noise amplitude  
 plt.savefig(figPath + fileName.replace(".wav", f"\_{int(amp \* 100)}\_spectrograms.png"))  
 # close the window to free memory space  
 plt.close()  
 return True  
  
# draws the waveforms for one file, calculates the avg error, and returns  
# these quantities as a list  
def analyze\_waveforms(fileName: str, orig\_path: str, amplitudes: list):  
 # initialize error% list for the file  
 analysis = [[], []]  
 # get the original sound  
 original = pm.Sound(orig\_path)  
 # set figure save location  
 figPath = orig\_path.replace("\_0riginal\\audio\\" + fileName, "figures\\")  
  
 # repeat the following for each amplitude at index j  
 for j, amp in enumerate(amplitudes):  
 # set the location of the files for amplitude amp  
 data\_path = orig\_path.replace("\_0riginal\\audio\\" + fileName, f"\_{int(amp \* 100)}\_percent\\")  
 noisy\_path = data\_path + "noisy\\" + fileName.replace(".wav", f"\_{int(amp \* 100)}\_noisy.wav")  
 filt\_path = data\_path + "filtered\\" + fileName.replace(".wav", f"\_{int(amp \* 100)}\_filtered.wav")  
 ref\_path = data\_path + "noise\_references\\" + fileName.replace(".wav", f"\_{int(amp \* 100)}\_noise\_ref.wav")  
 # import processed audio as Sound objects  
 noisy = pm.Sound(noisy\_path)  
 filtered = pm.Sound(filt\_path)  
 reference = pm.Sound(ref\_path)  
 # create array containing all sound objects to study  
 sounds = [original, noisy, filtered, reference]  
 # set figure size to 1930x1080  
 plt.figure(figsize=(19.2, 10.8))  
 # plot the waveform of each sound to one quadrant of the figure  
 for i, snd in enumerate(sounds):  
 plt.subplot(2, 2, i+1)  
 plt.plot(snd.xs(), snd.values.T)  
 plt.xlim([snd.xmin, snd.xmax])  
 if i == 0:  
 plt.title("Original", fontsize=40)  
 elif i == 1:  
 plt.title("Noisy", fontsize=40)  
 elif i == 2:  
 plt.title("Filtered", fontsize=40)  
 else:  
 plt.title("Noise Reference", fontsize=40)  
 plt.xlabel("time [s]", fontsize=20)  
 plt.xticks(fontsize=20)  
 plt.ylabel("amplitude [Hz]", fontsize=20)  
 plt.yticks(fontsize=20)  
 # end loop  
 # adjust layout so labels don't overlap  
 plt.tight\_layout()  
 # save the waveform plots to a png file corresponding to the noise amplitude  
 plt.savefig(figPath + fileName.replace(".wav", f"\_{int(amp \* 100)}\_waveforms.png"))  
 # close the window to free memory space  
 plt.close()  
 # calculate the average error between original vs noisy and original vs filtered  
 noisy\_err = calc\_avg\_error(sounds[0], sounds[1])  
 filtered\_err = calc\_avg\_error(sounds[0], sounds[2])  
 # save the errors on the row corresponding to the noise amplitude  
 analysis[0].append(noisy\_err)  
 analysis[1].append(filtered\_err)  
 # end loop  
 return analysis  
  
# calculates the average error between two sounds  
def calc\_avg\_error(acc: pm.Sound, exp: pm.Sound): # acc -> accepted values (original audio) exp -> experimental values (filtered/noisy)  
 err = np.zeros(acc.values.size)  
 for i in range(acc.values.size):  
 if np.isnan(acc.values[0][i]):  
 acc.values[0][i] = 0  
 if np.isnan(exp.values[0][i]):  
 exp.values[0][i] = 0  
 if acc.values[0][i] != 0:  
 err[i] = (abs(acc.values[0][i] - exp.values[0][i]) / abs(acc.values[0][i]))\*100  
 return round(sum(err)/acc.values.size)  
  
# processes all audio files in a single sentence folder  
def process\_data(sentence\_folder: str, amp=[0.05, 0.25, 0.5]):  
 if path.exists(sentence\_folder):  
 # get filenames of all sounds to analyze  
 files = os.listdir(sentence\_folder + "\\\_0riginal\\audio")  
 # initialize errors list for noisy / filtered  
 n\_errors = []  
 f\_errors = []  
 # create a sublist for each amplitude  
 for a in amp:  
 n\_errors.append([])  
 f\_errors.append([])  
 # for each file to analyze  
 for f in files:  
 # initialize analysis string to print to file  
 analysis = "recording\t\t|\t" \  
 "noise amp\t\t|\t" \  
 "avg error noisy\t\t|\t" \  
 "avg error filtered\n"  
 # create the full relative path for audio file of name f  
 original\_path = sentence\_folder + "\\\_0riginal\\audio\\" + f  
 # generate waveform images and return array of errors [[noisy\_err],[filt\_err]]  
 errors = analyze\_waveforms(f,  
 original\_path,  
 amp,  
 )  
 draw\_spectrograms(f,  
 original\_path,  
 amp)  
  
 # for each amplitude value  
 for i, a in enumerate(amp):  
 # add a line for that amplitude to analysis string  
 analysis += f"{f}\t\t|\t" \  
 f"{amp[i]} Hz\t\t|\t" \  
 f"{errors[0][i]} %\t\t|\t" \  
 f"{errors[1][i]} %\n"  
 # append error values to relevant errors sublist  
 n\_errors[i].append(errors[0][i])  
 f\_errors[i].append(errors[1][i])  
 # write analysis string to file  
 an\_file = open(sentence\_folder + "\\analysis\\" + f.replace(".wav", "\_analysis.txt"), "a+")  
 an\_file.write(analysis)  
 an\_file.close()  
 # write avg errors to another file  
 avgs = "Average Errors by Amplitude\n" \  
 "Amp\t\t|\tnoisy\t\t|\tfiltered\n"  
 for i, a in enumerate(amp):  
 avgs += f"{a}\t\t|\t" \  
 f"{np.mean(n\_errors[i]) \* 100}%\t\t|\t" \  
 f"{np.mean(f\_errors[i]) \* 100}" \  
 f"\n"  
 avgs\_file = open(sentence\_folder + "\\analysis\\averages.txt", "a+")  
 avgs\_file.write(avgs)  
 avgs\_file.close()  
  
  
  
 return True  
 else:  
 return False  
  
## main  
def main():  
  
 sentences = os.listdir("Data")  
 for sen in sentences:  
 process\_data("Data\\" + sen)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

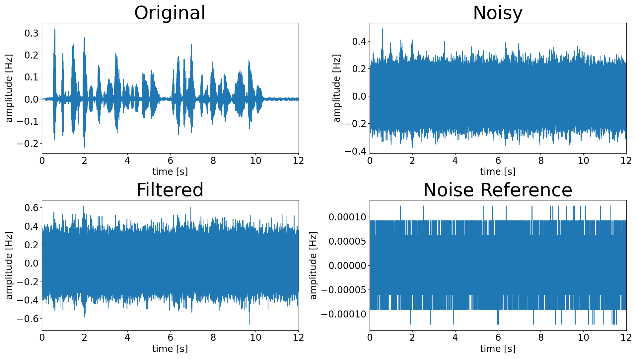
**B. Testing**

Haapoja\_beigefox\_5\_spectrograms & waveforms



Haapoja\_beigefox\_25\_spectrograms & waveforms

Timeline

Description automatically generated with medium confidence

**C. User Manual**

First, clone the repository from github at <https://github.com/jakadake/CSCI-334-Semester-Project-Audio-Filtering-and-Transcription>.

Then, create a sentence folder with subfolders structured as depicted in the main paper.

Once the folder is created, the \_0riginal\audio folder should be populated with the samples to be tested.

After the recordings have been loaded into the Data folder, execute the activate.bat file under the venv\Scripts folder. This will activate the virtual environment and load all dependencies automatically.

Once the virtual environment is activated, it should be possible to simply double click the processAudio.py file to execute it from the repository root directory.   
  
If this doesn’t work, then open a command prompt window and navigate to the repository root. From there type “python.exe processAudio” and the file will execute, generating noisy and filtered audio files.

Once processAudio has completed, you c