

REAL-TIME SMARTPHONE APPS INTEGRATING  
SIGNAL PROCESSING MODULES OF HEARING AIDS

by

Tahsin Ahmed Chowdhury



APPROVED BY SUPERVISORY COMMITTEE:

---

Dr. Nasser Kehtarnavaz, Chair

---

Dr. Chin-Tuan Tan

---

Dr. William (Bill) Swartz

Copyright 2018

Tahsin Ahmed Chowdhury

All Rights Reserved

REAL-TIME SMARTPHONE APPS INTEGRATING  
SIGNAL PROCESSING MODULES OF HEARING AIDS

By

TAHSIN AHMED CHOWDHURY

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirement

for the Degree of

MASTER OF SCIENCE IN

ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT DALLAS

May 2018

## ACKNOWLEDGEMENTS

First of all, I would like to thank GOD for His endless graces and blessings.

I would like to express my sincere appreciation to my advisor, Dr. Nasser Kehtarnavaz for his continuous guidance, patience and motivation in my Master's thesis.

Besides my advisor, I would like to thank Dr. Chin-Tuan Tan and Dr. William (Bill) Swartz for their time and support as committee members.

I would also like to thank my lab colleagues in the Signal and Image Processing Lab, Abhishek Sehgal, Nasim Alamdari and Neha Dawar for their valuable support and discussions.

Finally, I would like to thank my parents, Taher Ahmed Chowdhury and Romana Parvin, my wife, Armina Jahan, my brother, Tausif Ahmed Chowdhury and my sister, Tasnim Chowdhury Hridita for their moral support in my life.

Tahsin Ahmed Chowdhury

May 2018

# REAL-TIME SMARTPHONE APPS INTEGRATING SIGNAL PROCESSING MODULES OF HEARING AIDS

Tahsin Ahmed Chowdhury, MS  
The University of Texas at Dallas, 2018

Supervisory Professor: Dr. Nasser Kehtarnavaz

This thesis covers the integration of several signal processing modules that appear in digital hearing aids as real-time smartphone apps. The modules that are considered for integration in the developed apps in this thesis are voice activity detector, supervised noise classifier, noise reduction, and compression. The objective of this work has been to lay the foundation for real-time implementation of integrating various signal processing pipelines that are used in digital hearing aids on the smartphone platform as an open source platform which is widely used and carried by people. More specifically, two integrations are covered in this thesis. The first integration involves combining a previously developed voice activity detector and a previously developed supervised noise classifier into a real-time smartphone app. The second integration involves combining another previously developed voice activity detector, a previously developed noise reduction, and the compression module in the MATLAB Audio System Toolbox into a real-time smartphone app. The results obtained indicate that the developed integrations run in real-time on both iOS and Android smartphones.

## TABLE OF CONTENTS

ACKNOWLEDGMENT .....	iv
ABSTRACT.....	v
TABLE OF CONTESTS.....	vi
LIST OF FIGURES.....	vii
LIST OF TABLES.....	ix
CHAPTER 1 INTRODUCTION.....	1
CHAPTER 2 USER’S GUIDE: A SMARTPHONE APP INTEGRATING VAD AND SUPERVISED NOISE CLASSIFIER FOR HEARING IMPROVEMENT STUDIES....	3
CHAPTER 3 INTEGRATING SIGNAL PROCESSING MODULES OF HEARING AIDS INTO A REAL-TIME SMARTPHONE APP .....	24
CHAPTER 4 NOISE REDUCTION STUDY OF INTEGRATION .....	36
CHAPTER 5 USER’S GUIDE: A SMARTPHONE APP INTEGRATING SIGNAL PROCESSING MODULES OF HEARING AIDS.....	47
REFERENCES.....	88
BIOGRAPHICAL SKETCH.....	91
CURRICULUM VITAE.....	92

## LIST OF FIGURES

1. Contents of the VAD and NoiseClassifier integrated app folders .....	5
2. VAD and NoiseClassifier integrated app GUI – iOS version.....	7
3. VAD and NoiseClassifier integrated app code flow – iOS version .....	10
4. VAD and NoiseClassifier integrated app GUI – Android version.....	16
5. VAD and NoiseClassifier integrated app code flow – Android version.....	19
6. Integration of signal processing modules of hearing aids as a real-time smartphone app.....	27
7. A typical compression function.....	29
8. GUI of the integrated app (main page).....	33
9. CPU and memory consumptions of the integrated app .....	34
10. PESQ measure for fixed and adaptive noise estimation.....	35
11. Objective evaluation of noise reduction with fixed and adaptive noise estimation at SNR 5dB.....	39
12. Objective evaluation of noise reduction with fixed and adaptive noise estimation at SNR 0dB.....	40
13. Objective evaluation of noise reduction with fixed and adaptive noise estimation at SNR -5dB.....	42
14. Compression app: iOS.....	43
15. Noise estimation before speech activity frames.....	45
16. Moving average of all previous noise frames estimated before speech activity frames.....	45
17. Integrated app folder contents.....	49

18. Signing with Apple Developer ID.....	51
19. Integrated app iOS GUI: Main View.....	52
20. Integrated app iOS GUI: Noise Reduction Settings View.....	54
21. Integrated app iOS GUI: Compression Settings view.....	57
22. Integrated app iOS code flow.....	58
23. Further breakdown of native code modules in iOS.....	60
24. Supporting files.....	62
25. Setting optimization level in Xcode for the integrated app iOS.....	67
26. Integrated app Android GUI: Main view.....	70
27. Integrated app Android GUI: Noise Reduction Settings view.....	72
28. Integrated app Android GUI: Compression Settings view.....	75
29. Integrated app Android version: project organization.....	76
30. Further breakdown of native code modules in Android.....	80
31. Add native folder paths.....	82
32. Creating native linking function.....	83
33. Setting optimization level in Android Studio for the integrated app Android.....	86



## LIST OF TABLES

1. Listening effort scale.....	45
2. Subjective evaluation on noise reduction for different noise estimations.....	46
3. Timing difference between the iOS and Android versions of the integrated app.....	87

# **CHAPTER 1**

## **INTRODUCTION**

As per the World Health Organization, over 450 million people worldwide have some level of hearing impairment. As a solution to this problem, initially analog and now digital hearing aids have been developed by many companies. The main function of a hearing aid is to amplify sound within the hearing range of the person having a hearing impairment. Due to the limitation of the processing power of processors in hearing aids, researchers have started using smartphones as an assistive device to hearing aids. Smartphones have powerful processors that can be used in conjunction with or in addition to hearing aid processors to address hearing impairments.

Software tools have been developed by the Signal and Image Processing (SIP) Laboratory at the University of Texas at Dallas which enable real-time implementation of signal processing algorithms on smartphones. Furthermore, based on these software tools, a number of smartphone apps have been developed in this lab that run one component of the signal processing pipeline of hearing aids on smartphones as an open platform or source manner. These apps include voice activity detection, noise classification, noise reduction, and compression.

This thesis involves the integration of the previously developed individual signal processing components into smartphone apps that run in real-time and with low audio latency. More specifically, two integrations are covered in this thesis. The first integration involves combining a previously developed voice activity detector and a previously developed supervised noise

classifier into a real-time smartphone app. This first integration was carried out as part of an NIH sponsored project. The second integration involves combining another previously developed voice activity detector, a previously developed noise reduction, and the compression module in the MATLAB Audio System Toolbox into a real-time smartphone app.

The rest of the thesis is organized into the following four chapters:

Chapter 2 covers the user's guide describing how to run and use the app code for the integration of a previously developed voice activity detector app and a previously developed supervised noise classifier app. The work in this chapter was carried out as part of the NIH project entitled "Smartphone-Based Open Research Platform for Hearing Improvement Studies."

Chapter 3 covers the work done in the SIP Lab to integrate another previously developed voice activity detector app, a previously developed noise reduction app, and a newly developed compression app based on the MATLAB Audio System Toolbox.

Chapter 4 provides further analysis of the integrated app in Chapter 3. Finally, Chapter 5 covers the user's guide describing how to run and use the app code for the integrated app discussed in Chapter 3.

## **CHAPTER 2**

### **USER’S GUIDE: A SMARTPHONE APP INTEGRATING VAD AND SUPERVISED NOISE CLASSIFIER FOR HEARING IMPROVEMENT STUDIES**

T. Chowdhury, A. Sehgal, and N. Kehtarnavaz

Signal and Image Processing Lab

University of Texas at Dallas Copyright 2018

800 West Campbell Road

Richardson, Texas 75080-3021

The work in this chapter was supported by the National Institute of the Deafness and Other Communication Disorders (NIDCD) of the National Institutes of Health (NIH) under the award number 1R01DC015430-01. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIH.

## **Introduction**

This user's guide describes how to run an integrated smartphone app consisting of two previously developed apps of Voice Activity Detector (VAD) covered in [1, 2] and the supervised noise classifier covered in [3, 4].

The first part of this guide covers the iOS version of this integrated app and the second part covers its Android version. Each part consists of four sections. The first section discusses the GUI of the app. The second section covers the code flow of the app. In the third section, the process of data collection for training is mentioned. Finally, how to run the MATLAB training code to generate the parameters associated with the classifiers, is covered in the fourth section.

## **Devices used for running this integrated app**

- iPhone7 as iOS platform
- Google Pixel as Android platform

## **Folder Description**

The codes for the Android and iOS versions of the app are arranged as described below. These codes can be downloaded from the website at [5]. Fig. 1 lists the contents of the folder containing the app codes. These contents include:

- “VAD\_NC\_iOS” includes the iOS Xcode project. To open the project, double click on the “VAD\_NC iOS.xcodeproj” inside the folder.
- “VAD\_NC\_Android” includes the Android Studio project. To open the project, open Android Studio, click on “Open an existing Android Studio Project” and navigate to the project” VAD\_NC\_Android”.

- “VAD\_RandomForest\_Training” contains the MATLAB code to train the random forest classifier for VAD, based on data collected by the smartphone on which the app is to be run.
- “NoiseClassifier\_RandomForest\_Training” contains the MATLAB code to train the random forest classifier for NoiseClassifier, based on data collected by the smartphone on which the app is to be run.

Name		Size	Kind
▼ VAD_NC_iOS	📁	--	Folder
▶ VAD_NC_iOS	📁	--	Folder
▶ VAD_NC_iOSTests	📁	--	Folder
📄 VAD_NC iOS.xcodeproj	📄	379 KB	Xcode Project
▼ VAD_NC_Android	📁	--	Folder
▶ build	📁	--	Folder
▶ app	📁	--	Folder
▶ captures	📁	--	Folder
▶ gradle	📁	--	Folder
📄 VAD_NC_Android.iml	📄	868 bytes	Document
📄 FrequencyDomain.iml	📄	953 bytes	Document
📄 local.properties	📄	587 bytes	Conf Source
📄 build.gradle	📄	214 bytes	Document
📄 VADAndroid.iml	📄	864 bytes	Document
📄 gradle.properties	📄	855 bytes	Conf Source
📄 gradlew	📄	5 KB	Unix e...cutable
📄 gradlew.bat	📄	2 KB	Batch Source
📄 settings.gradle	📄	15 bytes	Document
▶ VAD_RandomForest_Training	📁	--	Folder
▶ NoiseClassifier_RandomForest_Training	📁	--	Folder

Figure 1: Contents of the VAD and NoiseClassifier integrated app folders

## **PART 1**

### **IOS**

## Section 1: iOS GUI

This section covers the iOS GUI of the integrated app consisting of VAD and Noise Classifier and its entries. The GUI consists of 4 views, see Fig. 2:

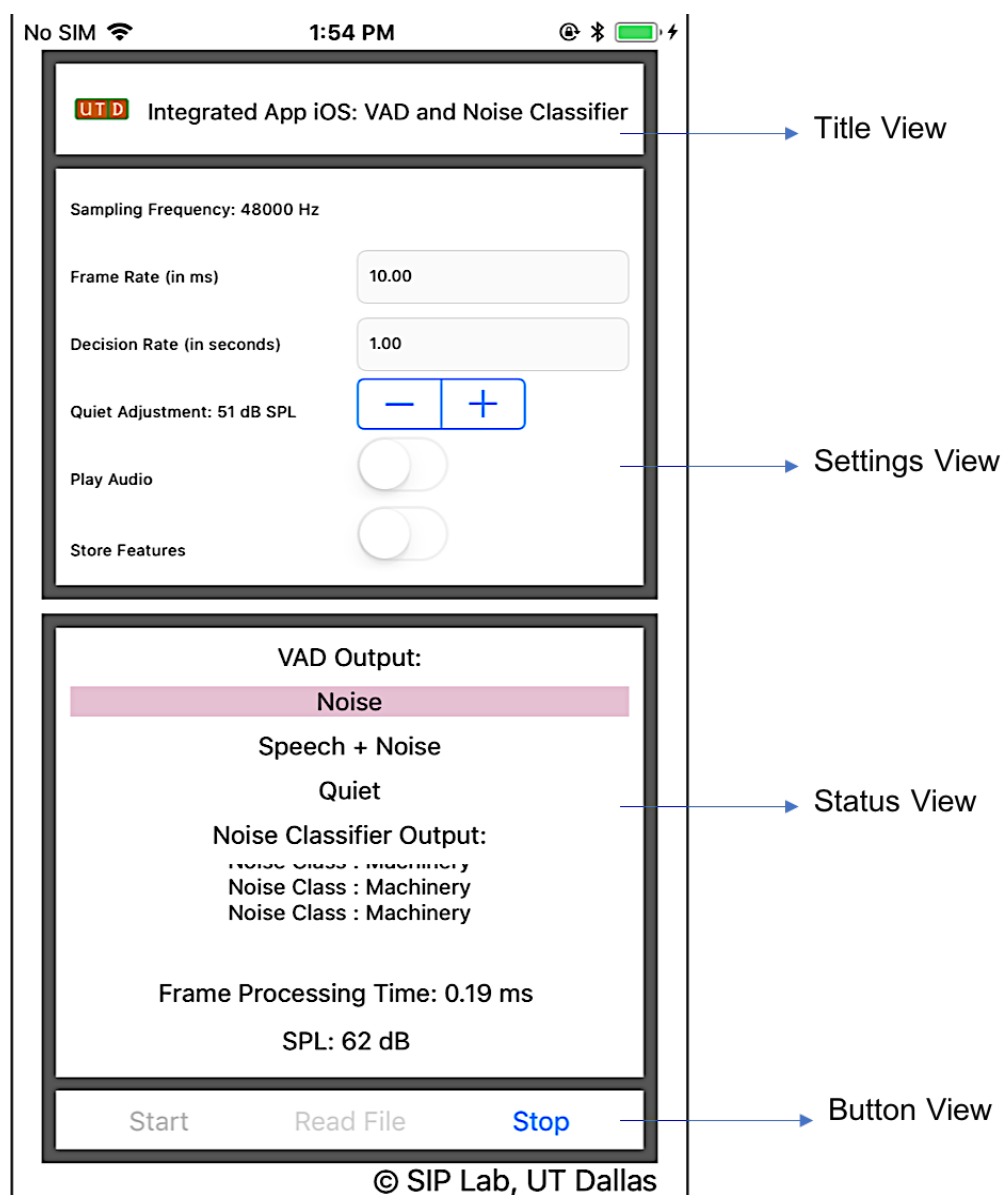


Figure 2: VAD and NoiseClassifier integrated app GUI – iOS version



- Title View
- Settings View
- Status View
- Button View

### 1.1 Title View

The title view displays the title of the app.

### 1.2 Settings View

This view controls the app settings as follows:

- **Sampling Frequency** in Hertz (Hz),
- **Frame Rate** in milliseconds (ms),
- **Decision Rate** in seconds,
- **Quite Adjustment** in dB SPL (sound pressure level),
- **Play Audio**, a switch to play audio signal to the smartphone speaker,
- **Store Features**, a switch to enable feature collection from audio environment.

Details of these settings are described in the user's guides for the individual VAD and Noise Classifier apps in [2] and [4]. For integrating these two modules, the sampling frequency is kept fixed at 48kHz in order to have the lowest audio latency imposed by the i/o hardware of smartphones. Other settings can be changed by the user before the app is run.

### **1.3 Status View**

The status view provides real-time feedback on:

- Whether captured audio signal frames are noise or speech+noise;
- If noise is detected, it shows the noise type (babble, machinery or traffic) for which the app has been trained;
- How much processing time is taken per frame in milliseconds; and
- SPL in dB.

### **1.4 Button View**

This view allows the user to start and stop the app. The app can also operate on audio files of “.wav” format. The user’s guides in [2] and [4] describe how to add an audio file to the app project.

## Section 2: Code Flow

This section states the app code flow. The user can view the code by running “VAD\_NC iOS.xcodeproj” as shown in Fig. 1. The code is divided into 3 parts, see Fig. 3:

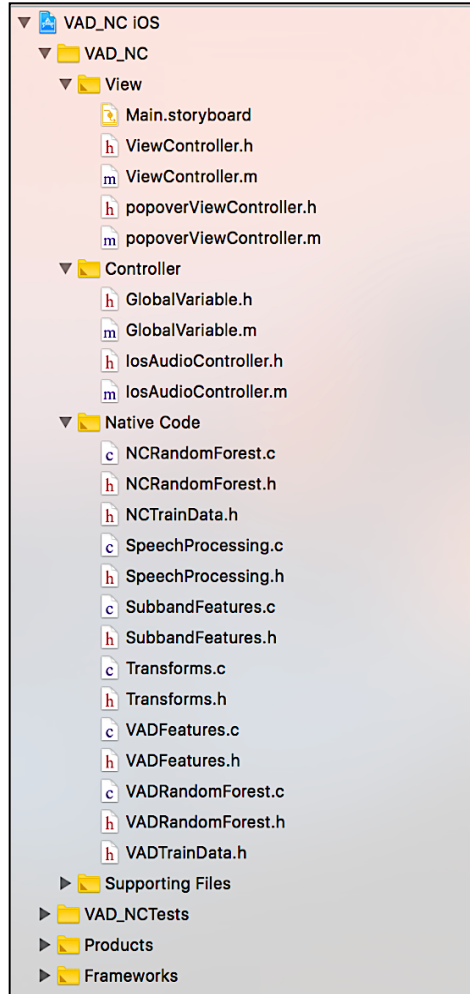


Figure 3: VAD and NoiseClassifier integrated app code flow – iOS version

- View
- Controller
- Native Code

## 2.1 View

This part provides the setup for the GUI of the app and has the following components:

- **Main.storyboard:** This component provides the layout of the app GUI.
- **ViewController:** This component provides the GUI elements and actions of the GUI View.
- **popoverViewController:** This component enables popping over the app GUI when the “Read File” button is clicked, populating a table view of audio files.

## 2.2 Controller

This part provides the controllers to pass data from the GUI to the native code and to update the status from the native code to appear in the GUI. The components are:

- **GlobalVariable:** This component acts as a bridge between the view and the native code.
- **IosAudioController:** This component controls the audio i/o setup for processing incoming audio frames by calling the native code.

## 2.3 Native Code

This part comprises the integrated app modules written in C. It is divided into the following components:

- **Speech Processing:** This component is the entry point of the native codes of the app. It initializes all the settings for the two modules and then processes incoming audio signal frames to detect noise/speech+noise and noise type. It consists of:

- **Transform:** This component computes the FFT of the incoming audio frames.
- **SubbandFeatures:** This component uses the FFT to extract subband features used by both the VAD and NoiseClassifier modules of the app.
- **VADFeatures:** Besides the subband features, this component is used to extract the additional features required by the VAD module of the app.
- **VADRandomForest:** This component classifies the incoming feature vector as noise or speech+noise.
- **NCRandomForest:** This component classifies the noise type if the VAD detects the incoming feature vector as noise.
- **VADTrainData:** This component is the random forest classifier designed for the VAD module. Its parameters are obtained by performing training in MATLAB on the HINT sentences audio files [6].
- **NCTrainData:** This component is the random forest classifier designed for the NoiseClassifier module. Its parameters are obtained by performing training in MATLAB on the HINT sentences audio files [6].

Readers are referred to the user's guides of the individual apps in [2] and [4] for more details. The above components appear in a modular manner to allow their modification or replacement with ease.

### Section 3: Data Collection for Training

Considering that both the classifiers in the VAD and NoiseClassifier modules are supervised classifiers, data are needed to train them. Data can be collected using the “Store Features” switch button in the app. When this button is turned on and the app is run, the app stores the data needed for training the classifiers. The data get stored in the form of a “.txt” file with comma separated values and each data frame appears on a new line. The data corresponds to the following features extracted from audio signal frames:

- 4 band periodicities (BP1, BP2, BP3, BP4),
- 4 band entropies (BE1, BE2, BE3, BE4),
- Short-Time Energy Deviation (STED),
- Subband Power Spectral Deviation (SPSD), and
- Spectral Flux (SF).

All of the above 11 features are used by the VAD module. From these features, the first 8 modules (BPs and BEs) are also used by the NoiseClassifier module. The user’s guides in [2] and [4] provide more details on data collection.

## Section 4: Training Classifiers

After data get collected, the random forest classifiers need to be trained in MATLAB for obtaining their parameters. The obtained parameters for both of the classifiers are then assigned to the native code for actual operation.

The classification operation consists of two parts:

- **VAD operation:** This module classifies whether audio signal frames are noise or speech+noise. A random forest training script in MATLAB appears inside the folder “VAD\_RandomForest\_Training”, see Fig. 1. This generates a header file containing the model which is renamed as “VADTrainData.h” and used in the project.
- **NoiseClassifier operation:** This module classifies the noise type detected by the VAD among three previously trained classes of babble, machinery, and traffic. A random forest training script in MATLAB appears inside the folder “NoiseClassifier\_RandomForest\_Training”, see Fig. 1. This generates a header file containing the model which is renamed as “NCTrainData.h” and used in the project.

**PART 2**

**ANDROID**



## Section 1: Android GUI

This section covers the Android GUI of the integrated app consisting of VAD and Noise Classifier and its entries. The GUI consists of 4 views, see Fig. 4:

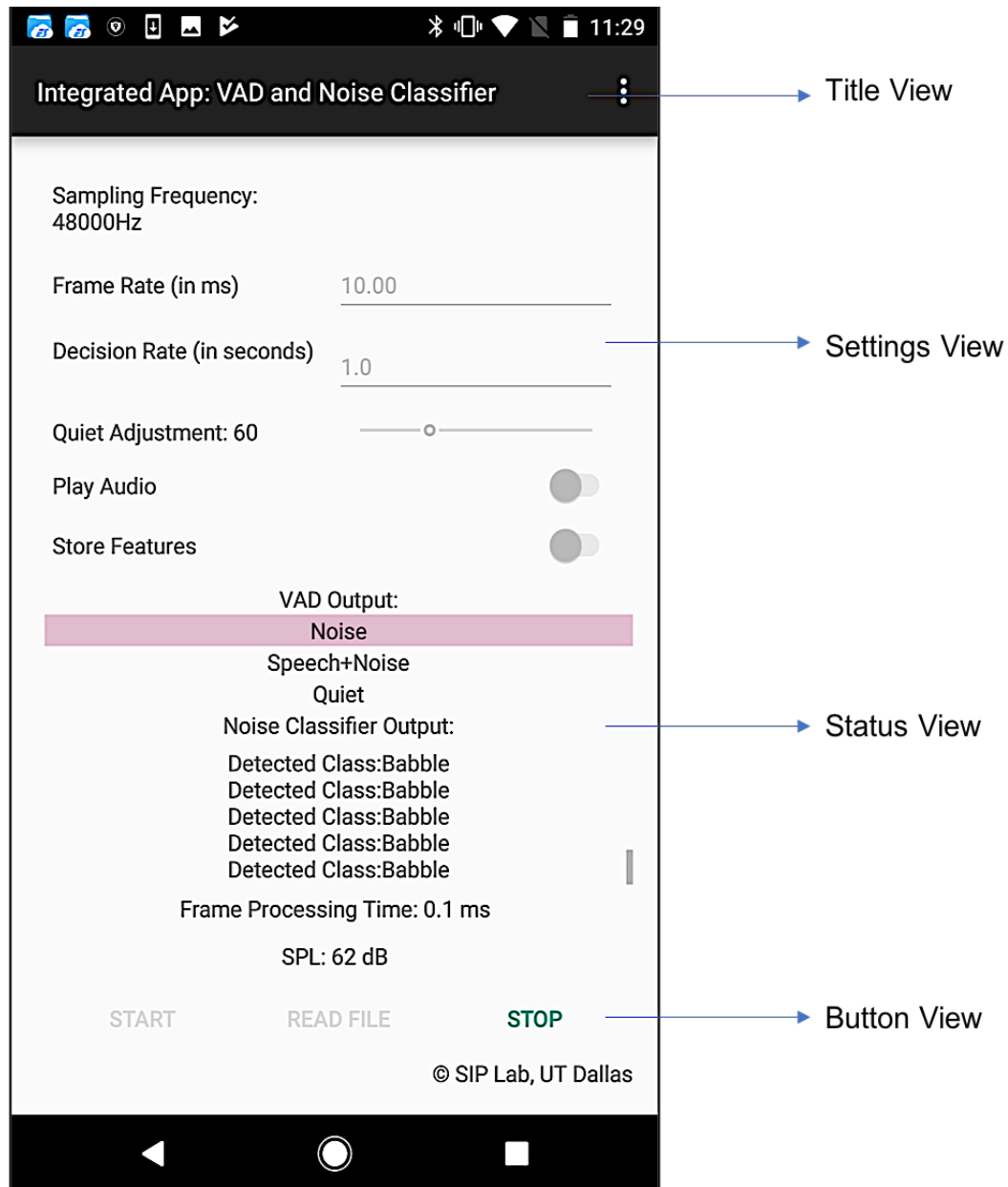


Figure 4: VAD and NoiseClassifier integrated app GUI – Android version

- Title View
- Settings View
- Status View
- Button View

### 1.1 Title View

The title view displays the title of the app.

### 1.2 Settings View

This view controls the app settings as follows:

- **Sampling Frequency** in Hertz (Hz),
- **Frame Rate** in milliseconds (ms),
- **Decision Rate** in seconds,
- **Quite Adjustment** in dB SPL (sound pressure level),
- **Play Audio**, a switch to play audio signal to the smartphone speaker,
- **Store Features**, a switch to enable feature collection from audio environment.

Details of these settings are described in the user's guides for the individual VAD and Noise Classifier apps in [2] and [4], respectively. For integrating these two modules, the sampling frequency is kept fixed at 48kHz in order to have the lowest audio latency imposed by the i/o hardware of smartphones. Other settings can be changed by the user before the app is run.

### **1.3 Status View**

The status view provides real-time feedback on:

- Whether captured audio signal frames are noise or speech+noise;
- If noise is detected, it shows the noise type (babble, machinery or traffic) for which the app has been trained;
- How much processing time is taken per frame in milliseconds; and
- SPL in dB.

### **1.4 Button View**

This view allows the user to start and stop the app. The app can also operate on audio files of “.wav” format. The user’s guides in [2] and [4] describe how to add an audio file to the app project.

## **Section 2: Code Flow**

To be able to run the Android version of the app, it is required to link the Superpowered SDK library [7] to it. The steps described in the user’s guides in [2] and [4] need to be followed to open the “VAD\_NC\_Android” project and run the app.

### **2.1 Project Organization**

After the project is opened in Android Studio, the project organization appears as shown in Fig.

5.

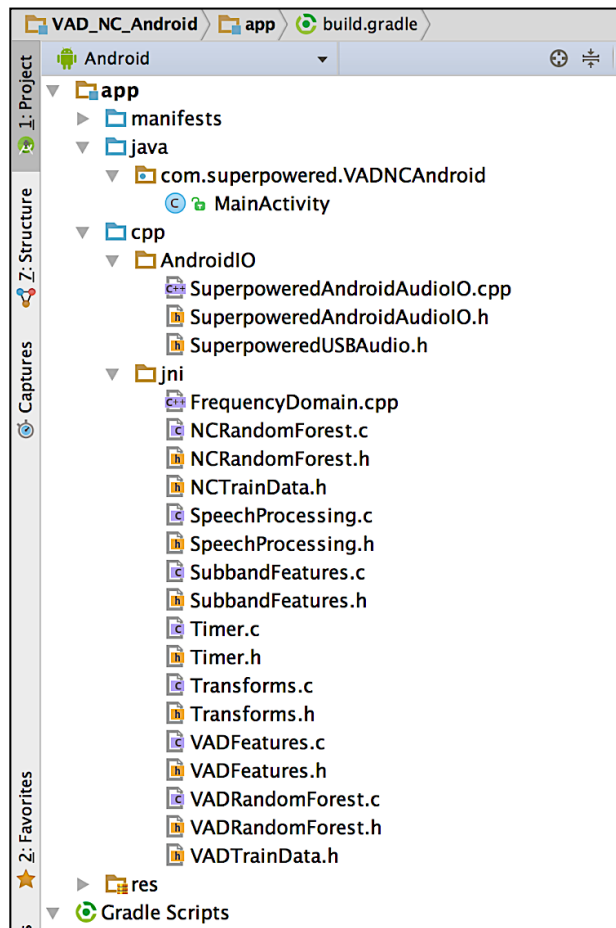


Figure 5: VAD and NoiseClassifier integrated app code flow – Android version

- **java:** This java folder contains the “MainActivity.java” file which handles all the operations of the app and allows the user to link the GUI to the native code.
- **cpp:** This folder contains the native code divided into two subfolders:
  - **AndroidIO:** This subfolder contains the Superpowered sources root files which control the audio interface to the app.
  - **jni:** This subfolder includes the native codes to start audio i/o and also the audio processing files of the VAD and NoiseClassifier modules.

## 2.2 Native Code

The native code part comprises the integrated app modules written in C and a bridging file written in C++. It is divided into the following components:

- **Frequency Domain:** This component is the C++ file that acts as a bridge between the native code and the java GUI. This file creates an audio i/o interface with the GUI settings.
- **Speech Processing:** This component is the entry point of the native codes of the app. It initializes all the settings for the two modules and then processes incoming audio signal frames to detect noise/speech+noise and noise type. It consists of:
  - **Transform:** This component computes the FFT of the incoming audio frames.
  - **SubbandFeatures:** This component uses the FFT to extract subband features used by both the VAD and NoiseClassifier modules of the app.
  - **VADFeatures:** Besides the subband features, this component is used to extract the additional features required by the VAD module of the app.
  - **VADRandomForest:** This component classifies the incoming feature vector as noise or speech+noise.
  - **NCRandomForest:** This component classifies the noise type if the VAD detects the incoming feature vector as noise.
  - **VADTrainData:** This component is the random forest classifier designed for the VAD module. Its parameters are obtained by performing training in MATLAB on the HINT sentences audio files [6].

- **NCTrainData**: This component is the random forest classifier designed for the NoiseClassifier module. Its parameters are obtained by performing training in MATLAB on the HINT sentences audio files [6].

Readers are referred to the individual user's guides in [2] and [4] for more details. The above components appear in a modular manner to allow their modification or replacement with ease.

### Section 3: Data Collection for Training

Considering that both the classifiers in the VAD and NoiseClassifier modules are supervised classifiers, data are needed to train them. Data can be collected using the “Store Features” switch button in the app. When this button is turned on and the app is run, the app stores the data needed for training the classifiers. The data get stored in the form of a “.txt” file in the device storage under the folder labelled “VAD\_NCAndroid”, with comma separated values and each data frame appears on a new line. The data corresponds to the following features extracted from audio signals:

- 4 band periodicities (BP1, BP2, BP3, BP4),
- 4 band entropies (BE1, BE2, BE3, BE4),
- Short-Time Energy Deviation (STED),
- Subband Power Spectral Deviation (SPSD), and
- Spectral Flux (SF).

All of the above 11 features are used by the VAD module. From these features, the first 8 modules (BPs and BEs) are also used by the NoiseClassifier module. The user's guides in [2]

and [4] provide more details on data collection.

#### Section 4: Training Classifiers

After data get collected, the random forest classifiers need to be trained in MATLAB for obtaining their parameters. The obtained parameters for both of the classifiers are then assigned to the native code for actual operation.

The classification operation consists of two parts:

- **VAD operation:** This module classifies whether audio signal frames are noise or speech+noise. A random forest training script in MATLAB appears inside the folder “VAD\_RandomForest\_Training”, see Fig. 1. This generates a header file containing the model which is renamed as “VADTrainData.h” and used in the project.
- **NoiseClassifier operation:** This module classifies the noise type detected by the VAD among three previously trained classes of babble, machinery, and traffic. A random forest training script in MATLAB appears inside the folder “NoiseClassifier\_RandomForest\_Training”, see Fig. 1. This generates a header file containing the model which is renamed as “NCTrainData.h” and used in the project.

## **CHAPTER 3**

# **INTEGRATING SIGNAL PROCESSING MODULES OF HEARING AIDS INTO A REAL-TIME SMARTPHONE APP**

Authors - Tahsin A. Chowdhury, Abhishek Sehgal, Nasser Kehtarnavaz,

Signal and Image Processing Lab

University of Texas at Dallas

800 West Campbell Road

Richardson, Texas 75080-3021

At the time of this writing, this chapter has been accepted for publication in Proceedings of the 40<sup>th</sup> International Conference of IEEE Engineering in Medicine and Biology Society (EMBC, 18). © 2018 IEEE



**Abstract**— This paper presents the integration of three major modules of the signal processing pipeline that go into a typical digital hearing aid as a real-time smartphone app. These modules include voice activity detection, noise reduction, and compression. The steps taken to allow the real-time implementation of this integration or signal processing pipeline are discussed. These steps can be utilized to create similar signal processing pipelines or integrated apps to evaluate hearing improvement algorithms. The real-time characteristics of the developed integrated app are reported as well as an objective evaluation of its noise reduction.

## 1. Introduction

In order to give more control to hearing aid users, smartphones can be used to run in real-time the modules that form the signal processing pipeline of a typical digital hearing aid. For example, the modules of noise reduction, compression, and amplification can be designed to run on a smartphone and the smartphone can then be interfaced wirelessly with low-latency Bluetooth hearing aids such as Starkey Halo2 [8]. The use of smartphones allows different algorithms for each hearing aid module to be easily tested. In other words, smartphones can be used as a research platform to evaluate different hearing improvement algorithms that form the signal processing pipeline of digital hearing aids.

Steps have been taken by our research team to use smartphones as an open source platform for hearing improvement studies noting that currently no open source, programmable and mobile platform exists in the public domain for carrying out hearing improvement studies. A smartphone-based platform enables the utilization of a programmable, mobile, and widely used device by researchers and audiologists towards exploring and field testing new and existing hearing improvement algorithms. The smartphone platform offers the following benefits: (a)

smartphones have powerful ARM processors enabling the real-time implementation of computationally intensive signal processing algorithms, (b) smartphones are already possessed and carried by most people (thus making them effectively a cost-free mobile platform), and (c) software development tools of smartphones are free of charge and are well maintained by smartphone companies.

A major challenge in using smartphones as a research platform lies in the fact that the programming languages used by the great majority of researchers when developing hearing improvement algorithms is C/MATLAB while the software development environment of iOS smartphones is based on Objective-C and the software development environment of Android smartphones is based on Java. In other words, one needs to address the mismatch between C/MATLAB programming that researchers use and Objective-C/Java programming environments of smartphones. We have previously met this challenge by developing software shells in [9, 10] which allow running C/MATLAB codes seamlessly within the software environments of iOS and Android smartphones. The development of these open source shells has enabled keeping the programming languages in using smartphones the same as the programming languages used by the great majority of researchers, namely C/MATLAB.

A number of smartphone apps have already been developed by our research team for an individual or a specific module of the signal processing pipeline of a digital hearing aid, for example smartphone apps for voice activity detection [1, 11], background noise reduction [12], and noise classification [3]. Each of these apps performs a specific task or one module of the signal processing pipeline of hearing aids. This paper presents the integration of three individual

modules to form an integrated app running in real-time on smartphones. The modules that are chosen for this integration include voice activity detection, noise reduction, and compression. The paper discusses the steps taken in order to enable this integrated app or in general any integrated app to run in real-time and with low audio latency when interfaced with Bluetooth-capable hearing aids. The output of the integrated app can also be heard simply by using a headphone audio cable. It is worth stating that this work is an extension of the solution submitted to the NSF Hearables Challenge that was selected as the second winner of this challenge [13].

The rest of the paper is organized as follows: In Section 2, the integrated signal processing modules to form the pipeline are discussed. Then, the implementation issues to run this signal processing pipeline as an integrated smartphone app in real-time are covered in Section 3. Section 4 provides the integrated app real-time characteristics and results. Finally, the paper is concluded in Section 5.

## **2. Integrated Signal Processing Modules**

Considering that the functions of voice activity detection, noise reduction, and compression constitute three major functions or modules in a digital hearing aid, we have considered these functions to form our signal processing pipeline. Fig. 6 illustrates all the modules that are integrated to form the pipeline which includes both the input/output (i/o) modules for performing frame-based signal processing and the processing modules consisting of voice activity detection, noise reduction, and compression. In what follows, each of these modules is described in the order they appear in the figure or the pipeline.

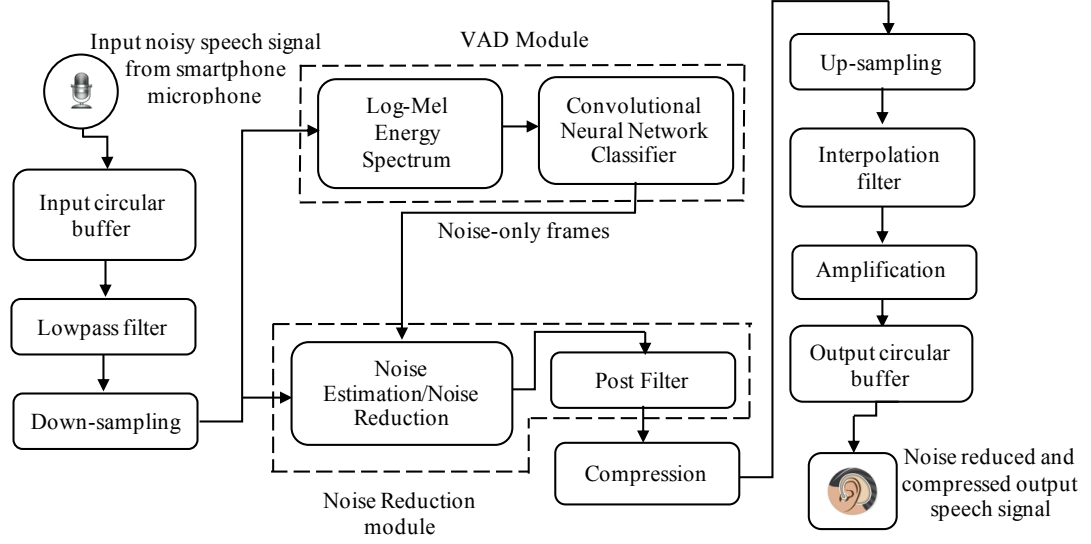


Figure 6. Integration of signal processing modules of hearing aids as a real-time smartphone app.

## 2.1 Input and Output Circular Buffers

As discussed in [14], the input audio signal from the smartphone microphone is captured via an input circular buffer and is outputted to the smartphone speaker or to a hearing aid via an output circular buffer in order to achieve the frame size corresponding to the lowest audio latency associated with the i/o hardware of smartphones.

## 2.2 Lowpass Filtering, Down-Sampling, Up-Sampling, and Interpolation Filtering

After circular buffering, as discussed in [14], a lowpass filter is used to remove signal activity beyond the frequency range of speech. Then, down-sampling is applied to decrease the sampling rate for the purpose of gaining computational efficiency or real-time throughput. In order to meet the hardware constraint associated with the lowest audio latency on a smartphone device, the down-sampling process is reversed by performing up-sampling or adding zeros between samples and by an interpolation filter to remove up-sampling artifacts.

### **2.3 Voice Activity Detection**

After the i/o modules, a Voice Activity Detector (VAD) module is activated in order to separate the noise-only segments of an input noisy speech signal from segments that contain speech activity. This module is the one that was developed in [11]. It consists of two sub-modules: an image formation sub-module which generates log-mel energy spectrum images, and a classification sub-module which involves a convolutional neural network (CNN) classifier. The VAD output is used to make the noise reduction module noise adaptive. That is, noise estimation is carried out during noise segments of an input noisy speech signal and noise reduction is carried out during speech activity segments of an input noisy speech signal.

### **2.4 Adaptive Noise Reduction**

The noise reduction module developed in [13] is executed next in the pipeline. This module uses a Wiener filter for noise reduction and is made noise adaptive in this work. Two types of noise estimation adaptive to the background noise environment are considered: (i) Noise power or variance is obtained for noise-only frames occurring before speech activity frames and the average value over these frames is used to reduce noise when the VAD specifies there is speech activity. (ii) Noise power is obtained for all noise-only frames as specified by the VAD and a moving average is used to reduce noise for speech activity frames. The first approach was found to generate audio fluctuations in realistic audio environments due to noise power changing across different noise frames. The second approach is thus implemented in our integrated app. Furthermore, a postfilter as discussed in [13] is used to reduce musical noise artifacts introduced by the Wiener filter in the noise reduction module.

## 2.5 Multiband Dynamic Range Compression

Compression is a key module in hearing aids. This module is used to amplify weak signals and suppress loud signals to bring them into the hearing range or hearing comfort zone of those suffering from hearing loss. Detailed descriptions of the compression module are provided in [15, 16].

There are many compression algorithms in the literature. Here, as part of our integrated app, we have used the so-called Dynamic Range Compression (DRC) module that is provided in the MATLAB Audio System Toolbox [17]. This module divides the input signal into five frequency bands and a compression function is applied to each band. The app implemented uses the following five frequency bands: 0-500Hz, 500-1000Hz, 1000-2000Hz, 2000-4000Hz and above 4000Hz. The major parameters associated with a compression function include (see Fig. 7):

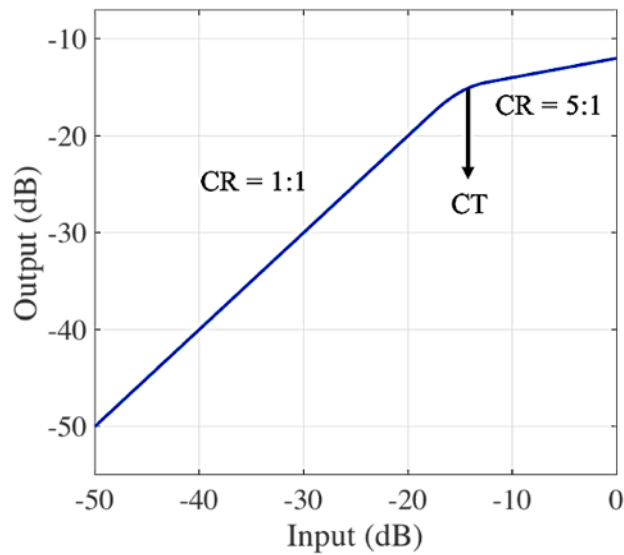


Figure 7. A typical compression function.

(i) Compression Threshold (CT) - This parameter indicates the point after which the compression is applied. (ii) Compression Ratio (CR) - This parameter indicates the amount of compression. (iii) Attack Time (AT) – This parameter indicates the time it takes for the compression module to respond when the signal level changes from a high to a low value. (iv) Release Time (RT) – This parameter indicates the time it takes for the compression module to respond when the signal level changes from a low to a high value. After compression, a final amplification is applied to the output signal before passing it to the speaker or sending it via Bluetooth to a hearing aid.

### **3. Real-time Implementation**

This section discusses the steps taken to incorporate the modules mentioned above as a smartphone app on iOS and Android devices. Both iOS and Android versions were developed. It should be noted that the iOS version running on iPhones exhibits a lower audio latency (10-15ms) as compared to Android smartphones. For example, for Google Pixel Android smartphone, the audio latency is about 40ms. It should be noted that since different Android smartphone manufacturers use different audio i/o hardware, audio latency of Android smartphones varies from phone to phone and in many cases, it is higher than 40ms.

All the algorithms were written in C or ported to C from MATLAB using the MATLAB Coder [18]. The smartphone shells developed in [9, 10] were used to incorporate the modules as an integrated app. These shells are written in Objective-C for iOS and in Java for Android. To deploy the apps on iPhones, the software tool Xcode IDE [19] was used and on Android smartphones the software tool Android Studio IDE [20] was used. The smartphones used for the experimentation reported in the next section were iPhone7 and Google Pixel. The low-latency

audio i/o setup for iOS was done using the software package CoreAudio API [21] and for Android was done using the software package Superpowered SDK [7]. It is worth noting that the integration of the modules is made modular in the integrated app by sharing common supporting files so that each module can be easily replaced by other similar modules than the ones used in this implemented integrated app. This modular design allows similar integrated app to be generated by using other VAD, noise reduction, and compression algorithms.

Due to the limitation imposed by the i/o hardware of smartphones, for the integrated app to have the lowest audio latency, the audio i/o needs to run at 48kHz and the i/o buffer size needs to be kept at 64 samples or  $64/48000=1.33\text{ms}$  for iOS smartphones and 192 samples or  $192/48000 = 4\text{ms}$  for Android smartphones (Pixel). The hearing aid modules run in frame-based manner at 16kHz with a 25ms processing frame size and with 50% overlap, that is a frame gets processed every 12.5ms. To synchronize the audio i/o and the hearing aid modules, circular buffers are utilized. An input circular buffer collects input samples from the audio i/o buffer till the overlapped frame size of 12.5ms (600 samples) is reached. It is then down-sampled and decimated by a factor of 3 and fed into the hearing aid modules. After a frame is passed through the modules, it is up-sampled and interpolated before being placed into an output circular buffer, which outputs the audio at the i/o rate of 64 samples at 48kHz, thus maintaining the lowest audio latency that is offered by the i/o hardware of smartphones.

The VAD app reported in [11] took about 0.43ms per audio frame and the noise reduction app reported in [12] took about 4ms per audio frame. The compression module when implemented as



an app took about 1ms per audio frame. For iOS smartphones, the processing time that is available to go through all the hearing aid modules of the pipeline with the lowest latency corresponds to the audio i/o time of 1.33ms per audio frame. That is, in order to have the lowest audio latency, if the processing time of a frame exceeds 1.33ms, it causes frames to get skipped. Initially, the integrated app took between 1.2ms to 1.5ms which caused some frames getting skipped due to not always meeting the lowest latency i/o time of 1.33ms. This problem was alleviated by using the GCC compiler optimization level 2. As a result, on average, the processing time of any frame going through the entire pipeline was brought down to 0.75ms on iPhone7 which enabled no frames getting skipped and a real-time operation was achieved. On the Pixel smartphone, the processing time for the entire pipeline was slightly higher but still within the available processing time of 4ms leading to a real-time operation as well.

#### **4. Results of Real-time Operations**

The GUI of the integrated app (iOS version) is shown in Fig. 8. On the GUI main page, the entries include: The VAD output (noise, speech+noise, or quiet), the option to turn on and off the noise reduction module, the option to turn on and off the compression module, the option to adjust the final output amplification, the settings associated with the noise reduction and the compression modules, and finally the option to store input/output signals.

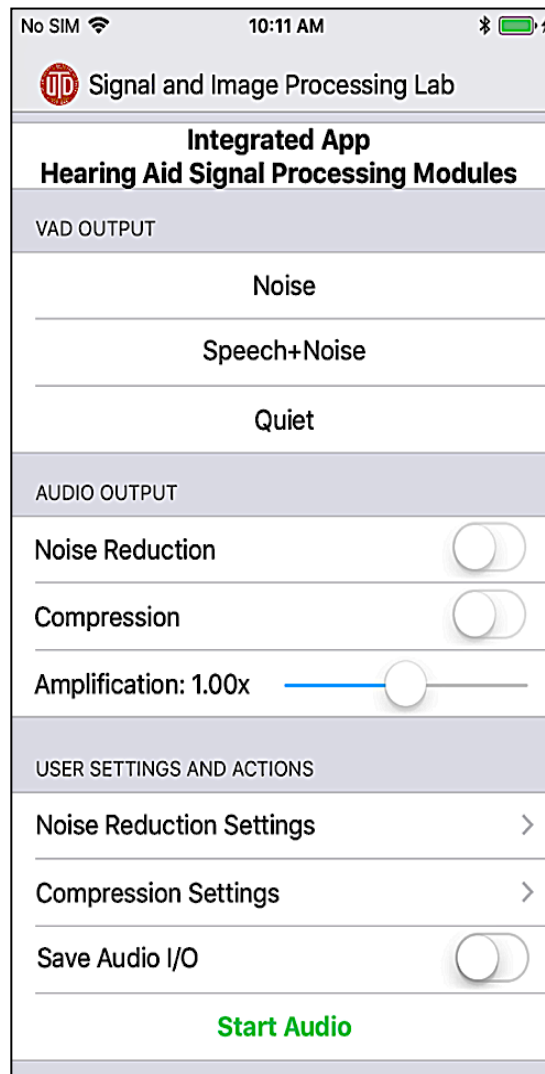
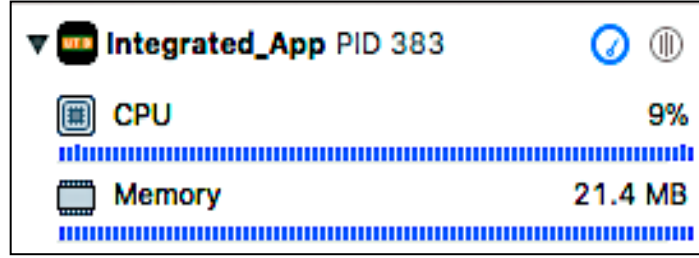
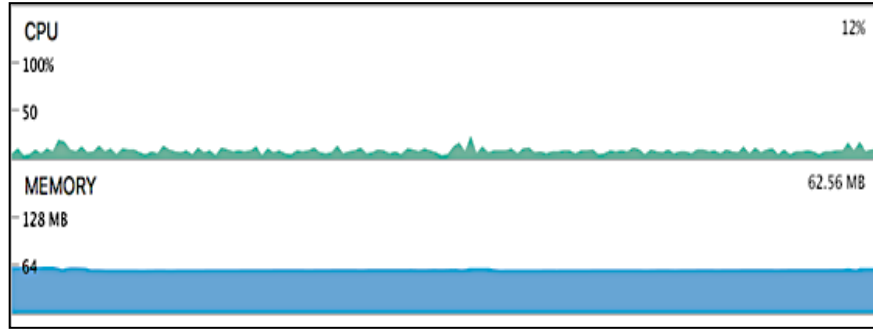


Figure 8. GUI of the integrated app (main page).

The CPU and memory consumptions of the integrated app on iPhone 7 and Pixel smartphones when all the modules are turned on is shown in Fig. 9. This information was generated via the Xcode IDE/Android Studio software tools. As can be seen from this figure, both the CPU and memory consumptions of the developed integrated app are low.



(a) iOS



(b) Android

Figure 9. CPU and memory consumptions of the integrated app.

Furthermore, an experiment was conducted by comparing the noise reduction of the developed integrated app with the noise reduction app reported in [12] where noise power is computed only once with no adaptation. In the app reported in [12], it is required to start with noise and with no speech presence in order to estimate the noise power. In the developed integrated app, this requirement is eased. The comparison was done by examining the widely used speech quality measure of PESQ described in [22] for the HINT sentences commonly used in audiology [6]. Babble and machinery noises were added to the sentences at 0dB and 5dB SNR levels and the effectiveness of the noise reduction was measured by computing the PESQ measure when the noise reduction module was turned on. The results obtained are shown in Fig. 10 where the PESQ measure is averaged over all the sentences. As can be seen from this figure, the adaptive

noise estimation approach implemented in the integrated app achieved higher PESQ measures compared to the fixed noise estimation approach in [6].

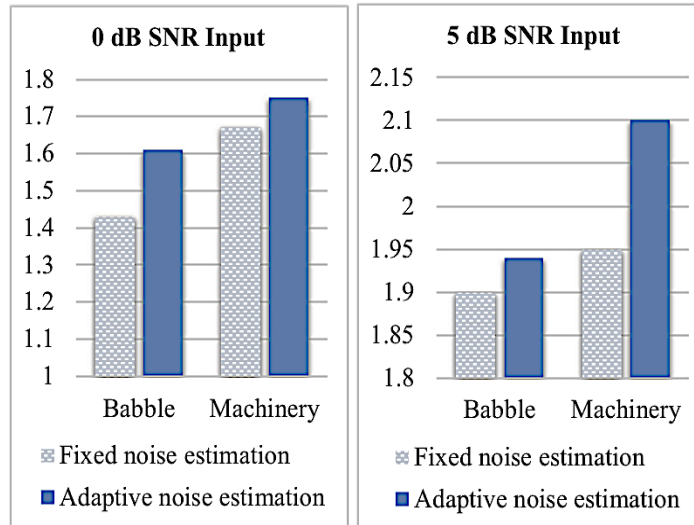


Figure 10. PESQ measure for fixed and adaptive noise estimation .

A video clip of the developed integrated app as well as its settings can be viewed at this link:

[www.utdallas.edu/~kehtar/IntegratedApp.mp4](http://www.utdallas.edu/~kehtar/IntegratedApp.mp4) .

## 5. Conclusion

In this paper, three major modules of digital hearing aids consisting of voice activity detection, noise reduction, and compression have been integrated to run in real-time and with low audio latency as a smartphone app. The steps discussed in this work to achieve a real-time operation of this integration can be utilized to develop other integrated apps or similar signal processing pipelines. In our future work, it is planned to develop similar integrated apps in which other algorithms in the signal processing pipeline of digital hearing aids are incorporated.

## **CHAPTER 4**

### **NOISE REDUCTION STUDY OF INTEGRATION**

This chapter provides a noise reduction study of the integrated app discussed in Chapter 3. To reduce noise, proper noise estimation is essential to obtain good performance from the noise-reduction module. The module discussed in [12] estimates the average power of first six incoming audio frame as noise, which gets updated based on the “incoming audio signal to previously estimated noise ratio”. One can easily see that this approach is not an effective way of estimating noise, because it is estimated in a fixed manner, only at the beginning; those six frames may contain noise or may contain speech signal. Furthermore, while running the noise reduction module as an app, it is required to first wait with no speech presence for the app to estimate the noise and then proceed with the noise reduction task. A modification is done here to avoid this problem by estimating the noise in an adaptive manner as described below.

The Voice Activity Detection (VAD) module is used to provide a separation between noise and speech+noise frames. Initially, a supervised Random Forest (RF) classifier based on subband features, described in [4], was used as the VAD. Later on, a Convolutional Neural Network (CNN) classifier based on log-mel energy spectrum features, described in [11], was used as the VAD to separate noise and speech+noise frames. The VAD allowed the noise estimation to be done only during the noise frames.

An analysis was performed to study the noise estimation when it was done in an adaptive manner based on the VAD decision. Audio files (both noise and clean speech) used here were obtained

from [6]. The speech data were mixed with noise to create noisy speech data at the following Signal-to-Noise-Ratios (SNRs):

- i. SNR level at -5dB, 0dB and 5 dB,
- ii. Babble and Machinery noise types for the following scenarios:
  - a. 10 separate speech (HINT) sentences.
  - b. Repeating the sentences with the same interval in between to mimic a conversation.
  - c. Repeating the sentences with different intervals in between to mimic a more natural conversation.

These audio data were analyzed frame by frame. Each frame was given a label whether it was noise or speech+noise.

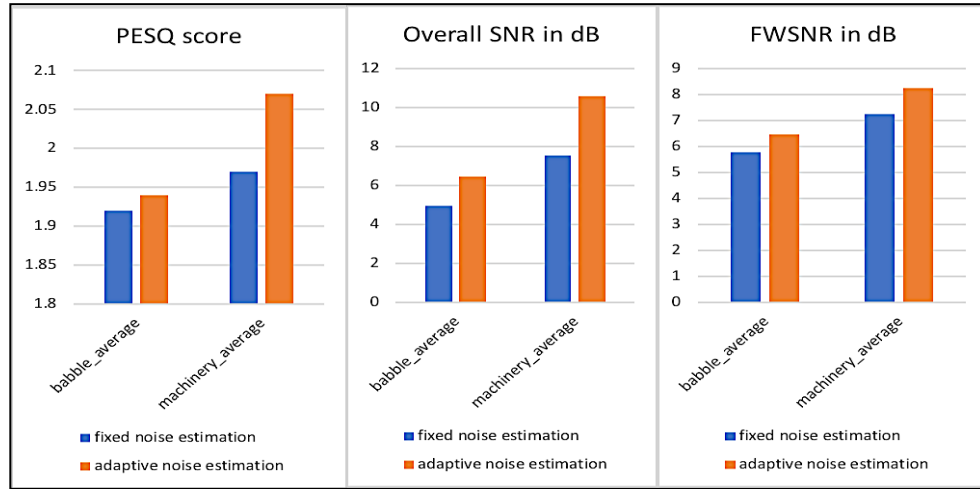
The processed output was evaluated using the following three objective measures as discussed in [22]:

- i. PESQ: The PESQ (Perceptual Evaluation of Speech Quality) measure is an ITU-T standardization measure that takes into consideration distortions that generally take place when speech goes through a telecommunication channel. This measure is widely used for speech quality evaluation.
- ii. FWSEG: The FWSEG (Frequency Weighted Segmented) SNR measure is known to show high correlation with subjective listening tests.
- iii. Overall SNR: The Signal to Noise (SNR) measure is a general approach to measure the quality of the processed output signal compared to the environmental noise.

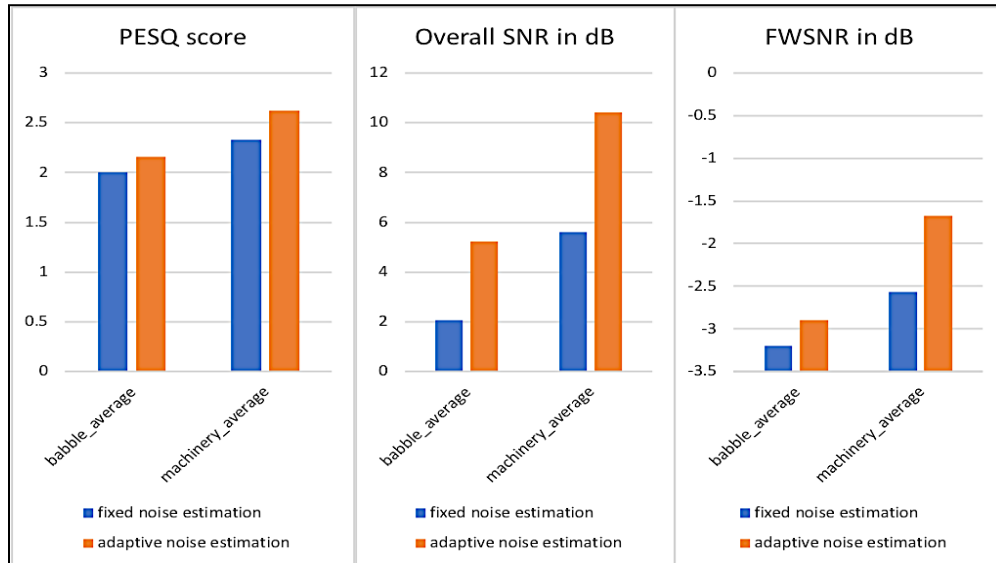
The outcome of the above analysis appears below. PESQ score was found to be quite sensitive to the alignment of clean speech with the noisy speech and it was sometimes challenging to obtain good scores for low SNR -5dB, while frequency weighted SNR and total SNR exhibited good

scores when the noise estimation was done adaptively. Fig. 11, 12 and 13 provide a graphical comparison with the noise estimation in [12] and the adaptive noise estimation adopted in this work.

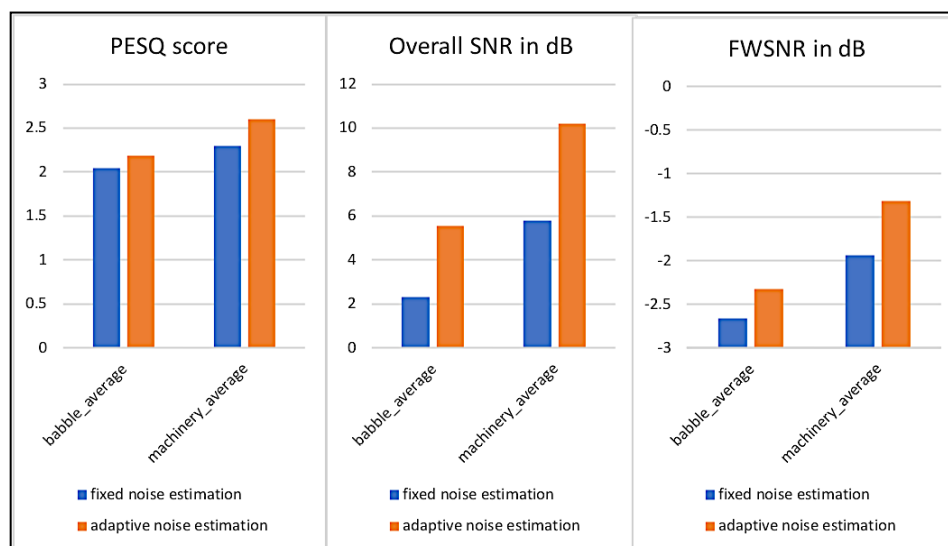
**i. SNR 5dB:**



(a) Single Sentences



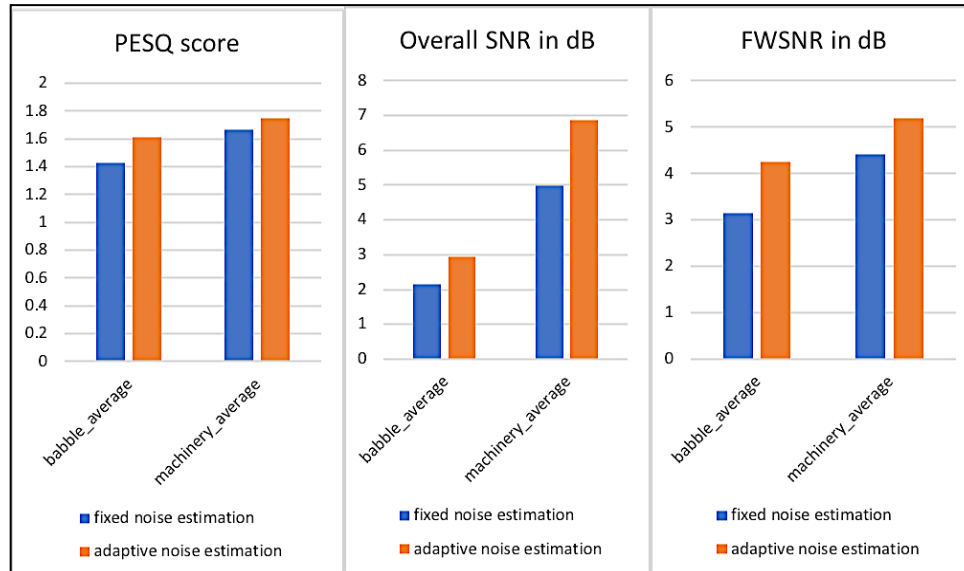
(b) Repeating sentences to mimic a conversation with the same interval in between



(c) Repeating sentences to mimic a conversation with different intervals

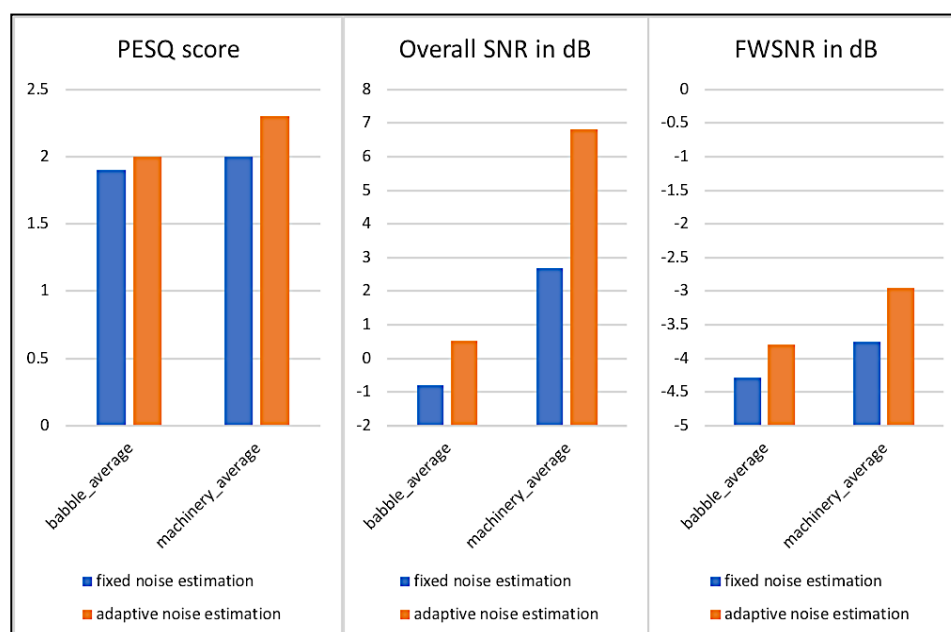
Figure 11. Objective evaluation of noise reduction with fixed and adaptive noise estimation at SNR 5dB

## ii. SNR 0dB:

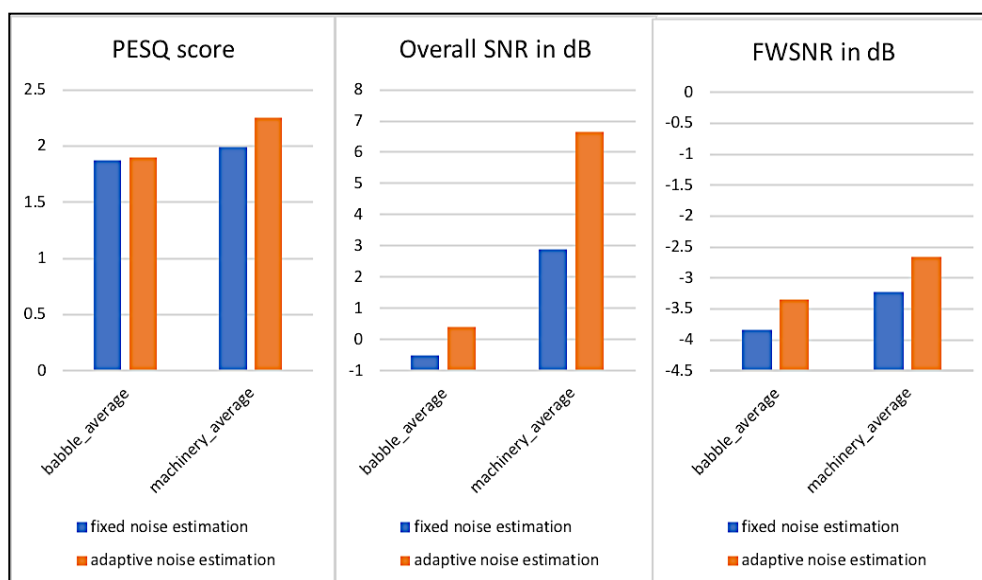


(a) Single Sentences





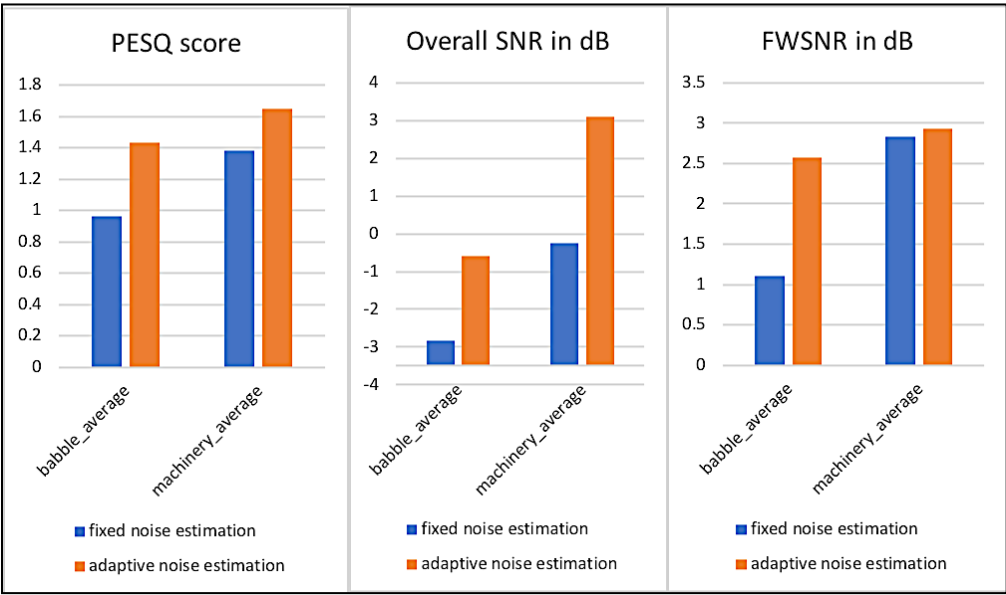
(b) Repeating sentences to mimic a conversation with the same interval in between



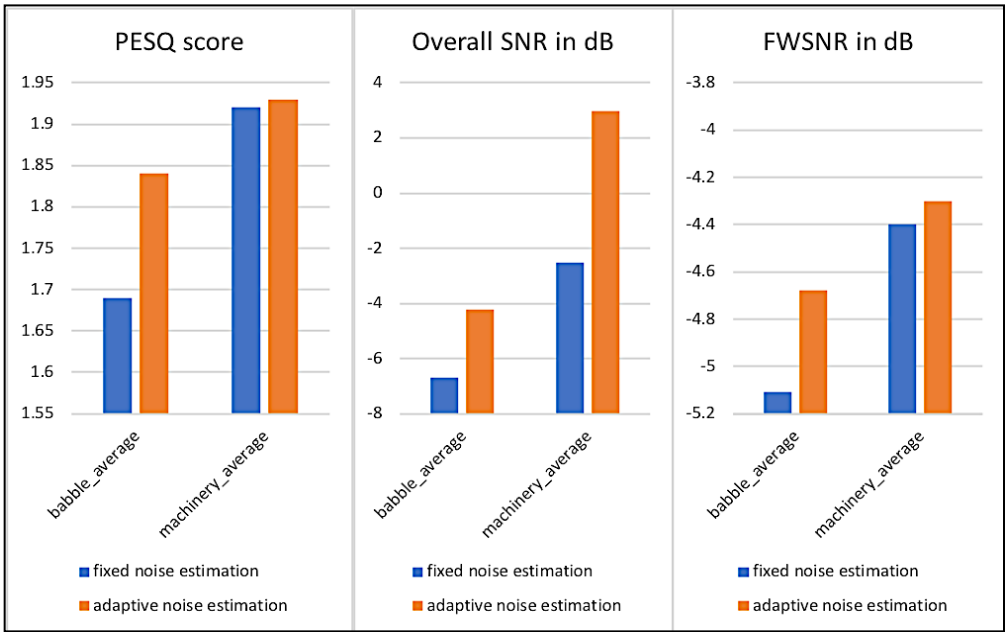
(c) Repeating sentences to mimic a conversation with different intervals

Figure 12. Objective evaluation of noise reduction with fixed and adaptive noise estimation at SNR 0dB

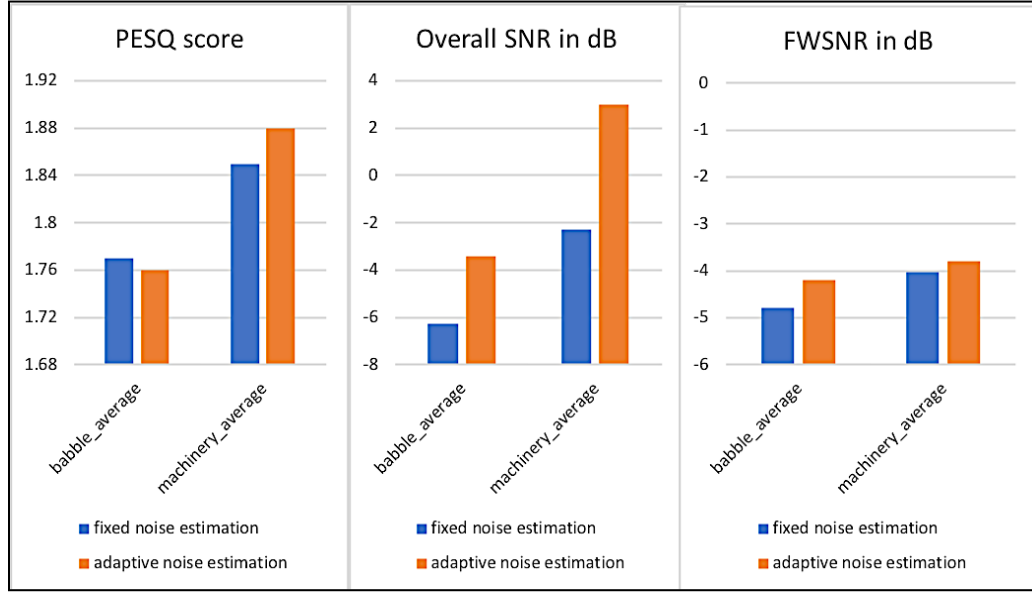
iii. SNR -5dB



(a) Single Sentences



(b) Repeating sentences to mimic a conversation with the same interval in between



(c) Repeating sentences to mimic a conversation with different intervals

Figure 13. Objective evaluation of noise reduction with fixed and adaptive noise estimation at SNR -5dB

It is worth pointing out that the noise reduction module used had already been evaluated using objective measures in [12] and the analysis conducted here was only meant to provide the justification for using the VAD for a better noise estimation.

#### 4.1 Implementation of Multi-Band Dynamic Range Compression as a Separate Smartphone App

A separate compression smartphone app was also developed here. The app was developed in such a way that the code and the GUI (see, Fig. 14) could easily be added to another app without the need to make any major changes. This app was developed using the pipeline described in [14] which brought down the frame processing time from 4.6ms to 1.00ms. The code for this app was converted to C code using the MATLAB Coder.

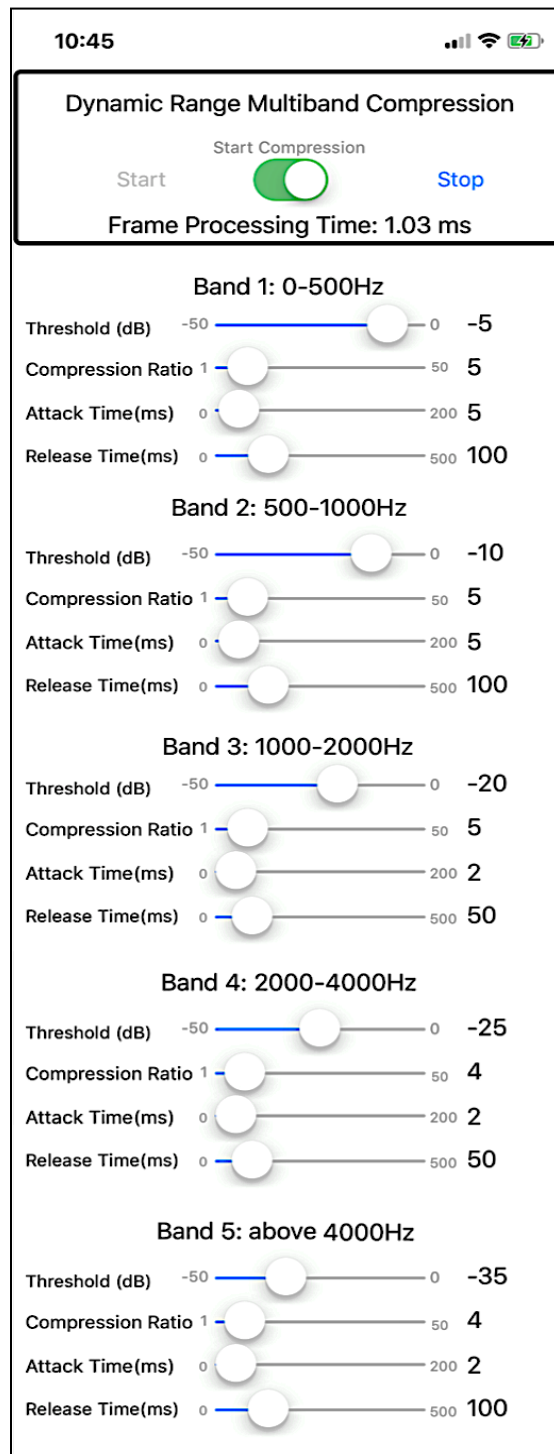


Figure 14. Compression app: iOS version

## **4.2 Compiler Optimization**

Initially, some frames in the integrated app got skipped due to not meeting the available time bandwidth. The following three approaches were considered to address any frame getting skipped:

- Reducing filter coefficients: Reducing the number of coefficients of the lowpass and interpolation filter in the pipeline provided some computational improvement. However, this led to relatively poor filtering functionality.
- Increasing audio i/o buffer size: Increasing the audio i/o buffer size extended the available time bandwidth for processing audio frames. However, this created a higher audio latency.
- GCC compiler level optimization: GCC compiler optimization at level 2 (-O2) provided a significant improvement in computational time leading to less than 1.00ms frame processing time for the entire integrated pipeline.

## **4.3 Adaptive Noise Estimation Approaches**

As mentioned in Chapter 3, two types of noise estimation adaptive to the background noise environment were examined:

- Noise power or variance was obtained for noise-only frames occurring before speech activity frames and the average value over these frames was used to reduce noise when the VAD specified there was speech activity as shown in Fig.15.

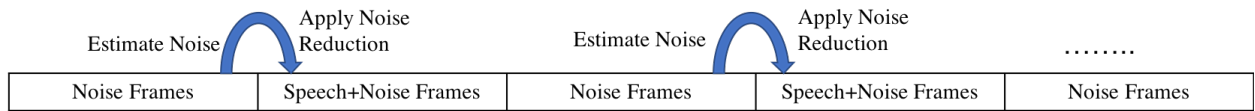


Figure 15. Noise estimation before speech activity frames

This approach exhibited fluctuations in noise reduction. This occurred due to noise power changing across different noise frames. As an alternative, the following second approach was considered:

- Noise power was obtained for all noise-only frames as specified by the VAD and a moving average was used to reduce noise for speech activity frames as shown as Fig.16.

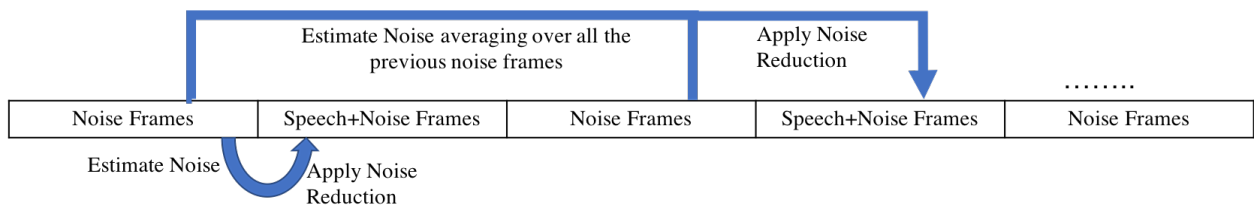


Figure 16. Moving average of all previous noise frames estimated before speech activity frames

A subjective study was conducted to see which noise estimation preferred from a subjective point-of-view based on the listening effort scale shown in Table 1:

**TABLE 1. LISTENING EFFORT SCALE**

Effort required to understand sentences	Score
Complete relaxation possible: no effort required	5
Attention necessary: no appreciable required	4
Moderate effort required	3
Considerable effort required	2
No meaning understood with any feasible effort	1

Table 2 presents the scores at various noisy environments including restaurant, coffee shop, shopping mall, etc., for both of the noise estimation approaches with comments appearing in parentheses. As noted in this table, it was noticed that the first approach causes suffering from fluctuations while the second approach was found to be more consistent. The second approach was thus the approach that was considered for implementation in the integrated app.

**TABLE 2. SUBJECTIVE EVALUATION ON NOISE REDUCTION FOR DIFFERENT NOISE ESTIMATIONS**

<b># Evaluations</b>	<b>1. Noise Estimation Before Speech Activity</b>	<b>2. Moving Averaged Noise Estimation Before Speech Activity</b>
<b>1</b>	4 (fluctuation)	5 (consistent)
<b>2</b>	4 (fluctuation)	4
<b>3</b>	4 (fluctuation)	5 (consistent)
<b>4</b>	3 (fluctuation)	5 (consistent)
<b>5</b>	2 (echo + fluctuation)	4 (better, stable)
<b>6</b>	3 (fluctuation)	4 (better, stable)
<b>7</b>	4 (fluctuation)	5 (stable)
<b>8</b>	4 (fluctuation)	5 (consistent)
Average	3.5	4.625

## **CHAPTER 5**

### **USER'S GUIDE: A SMARTPHONE APP INTEGRATING SIGNAL PROCESSING MODULES OF HEARING AIDS**

T. Chowdhury, A. Sehgal, and N. Kehtarnavaz

Signal and Image Processing Lab

University of Texas at Dallas

800 West Campbell Road

Richardson, Texas 75080-3021



## **Introduction**

This user's guide covers how to use a smartphone app developed in the Signal and Image Processing Laboratory at the University of Texas at Dallas. This app is an integration of three signal processing modules encountered in digital hearing aids, all running together in real-time as discussed in Chapter 3. The integrated modules include the Voice Activity Detector (VAD) discussed in [1], the noise reduction module discussed in [12] and the compression module discussed in [15] and [16]. Interested readers are referred to these references for the details of these modules or algorithms.

This user's guide is divided into two parts. The first part covers the iOS version of the integrated app and the second part covers the Android version. Each part consists of four sections. The first section discusses the steps to be taken to run the app. The second section covers the GUI of the app. In the third section, the app code flow is explained. Finally, the modularity and modification of the app are mentioned in the fourth section.

## **Devices used for running the integrated app**

- iPhone7 as iOS platform
- Google Pixel as Android platform

## **Integrated App Folder Description**

The codes for the Android and iOS versions of the app are arranged as described below. Fig. 17 lists the contents of the folder containing the app codes. These contents include:

- “Integrated\_App\_Android” denoting the Android Studio project
- “Integrated\_App\_iOS” denoting the iOS Xcode project

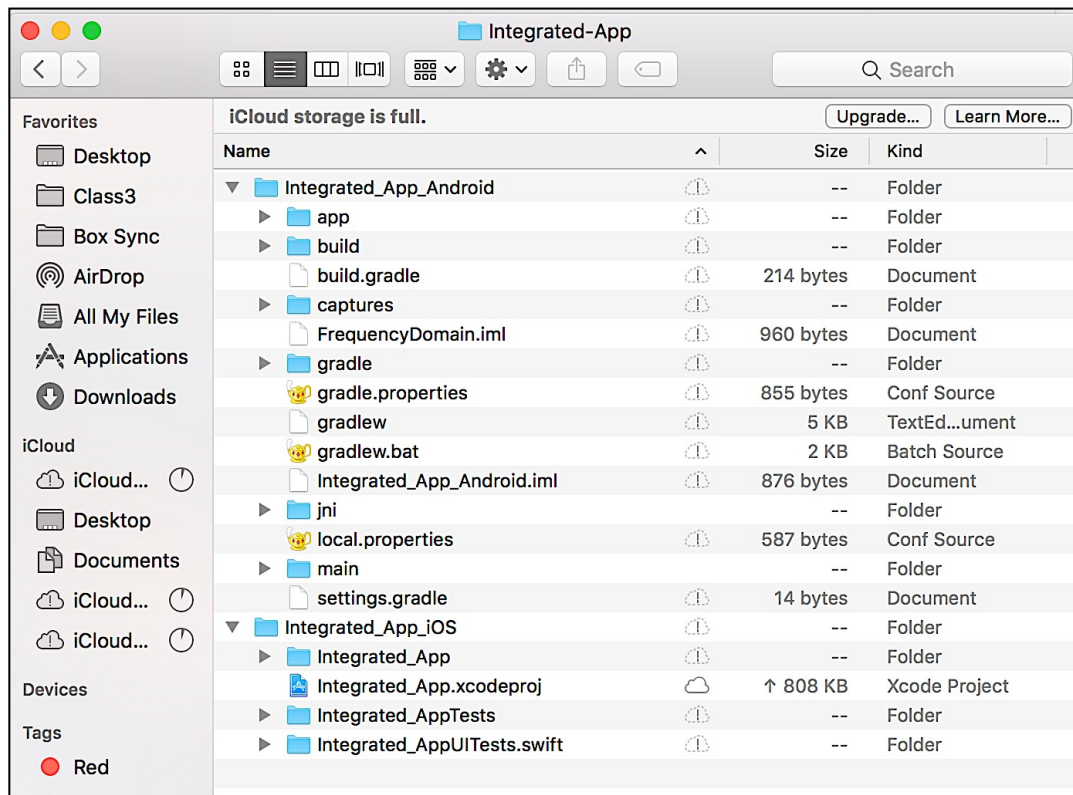


Figure. 17: Integrated app folder contents

## **PART 1**

### **IOS**

## Section 1: Running the App

The iOS version of the integrated app was developed using the Xcode development tool (Version 9.0). It is required to have an Apple ID to run the iOS version of the app, see [19] for more details.

To open the app project, double click on the “Integrated\_App.xcodeproj” in the “Integrated\_App\_iOS” folder, refer to Fig. 17.

To run the project, enable developer mode in iPhone if required (see [23]), connect iPhone to a MAC computer using a USB cable, and then run the app by Command+R. The app gets installed. Make sure the Xcode is signed into using your Apple Developer ID, refer to Fig. 18.

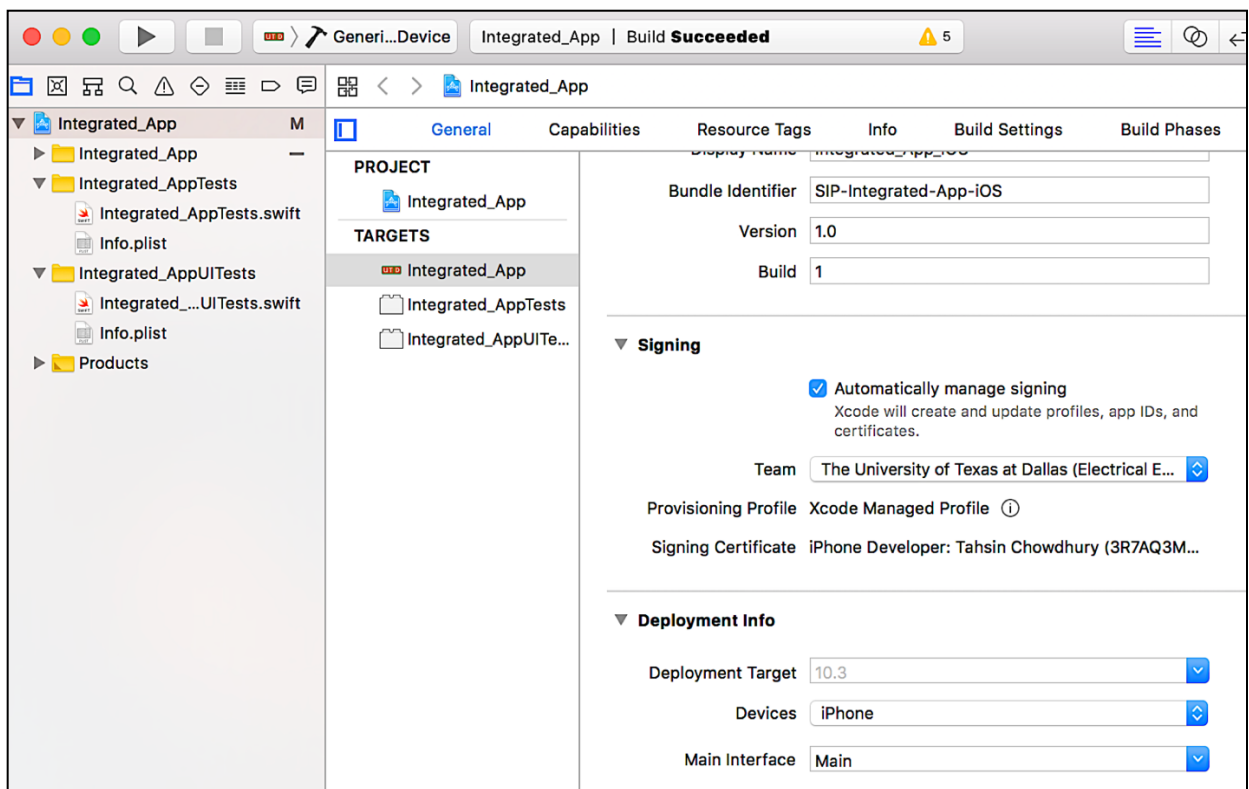


Figure. 18: Signing with Apple Developer ID

## Section 2: iOS GUI

This section covers the GUI of the developed integrated app and its entries. The GUI consists of three views.

- Main View
- Noise Reduction Settings View
- Compression Settings View

### 2.1 Main View

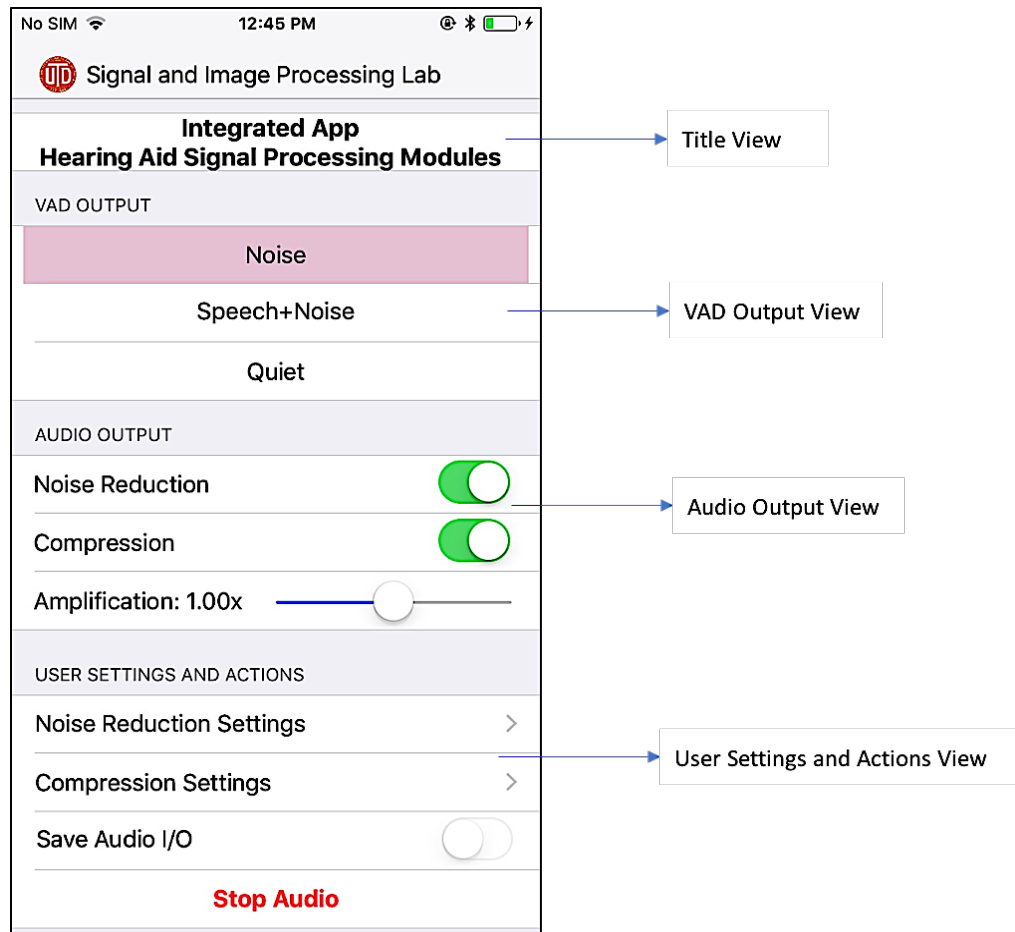


Figure. 19: Integrated app iOS GUI: Main view

Main view consists of 4 segments, see Fig. 19.

- Title View
- VAD Output View
- Audio Output View
- User Settings and Actions View

### **2.1.1 Title View**

The title view displays the title of the app.

### **2.1.2 VAD Output View**

This view displays the VAD output, that is, noise, speech+noise, or quiet.

### **2.1.3 Audio Output View**

This view shows:

- A switch to turn on and off the noise reduction module
- A switch to turn on and off the compression module
- A slider to control final amplification

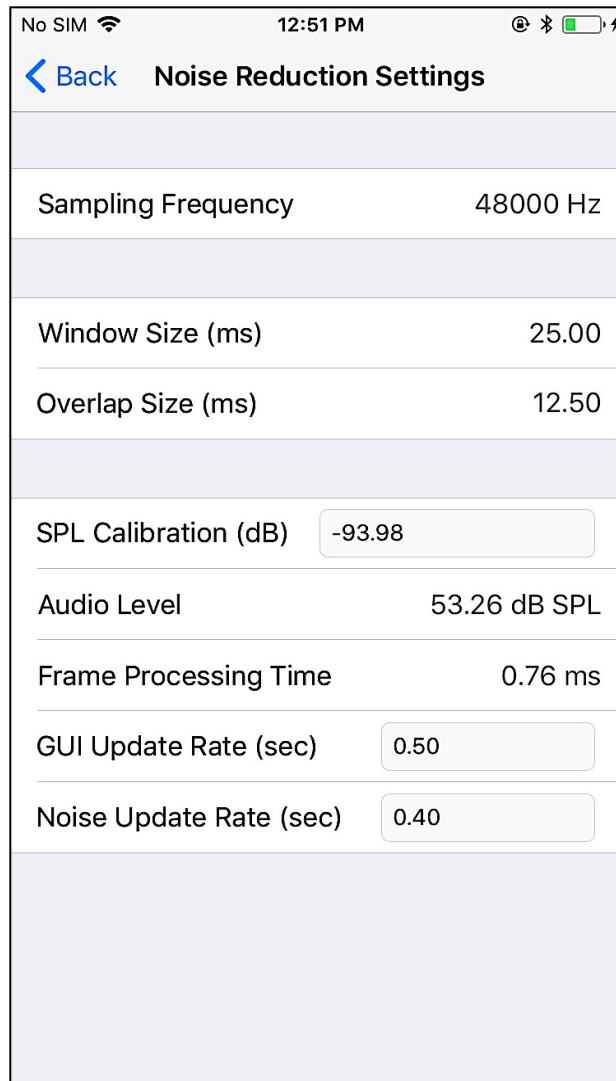
### **2.1.4 User Settings and Actions Views**

This view displays the following entries:

- Noise reduction (and audio) settings
- Compression settings
- An option to save input/output audio signals
- A button for starting and stopping the app

## 2.2 Noise Reduction Settings View

This view contains the settings for the noise reduction and audio processing. As shown in Fig. 20, the fields in this view are:



No SIM 12:51 PM	
<a href="#">Back</a> Noise Reduction Settings	
Sampling Frequency	48000 Hz
Window Size (ms)	25.00
Overlap Size (ms)	12.50
SPL Calibration (dB)	-93.98
Audio Level	53.26 dB SPL
Frame Processing Time	0.76 ms
GUI Update Rate (sec)	0.50
Noise Update Rate (sec)	0.40

Figure. 20: Integrated app iOS GUI: Noise Reduction Settings view.

- **Sampling Frequency:** This field shows the sampling frequency. To obtain the lowest latency when using iOS mobile devices, this value is set to 48000 Hz. The audio signal is down-sampled to a sampling frequency of 16000 Hz in order to make the processing time computationally efficient as described in chapter 3.
- **Window Size (ms):** This field shows the window (frame) size in milliseconds. Basically, this entry indicates how many data samples (of this window length) get processed by the integrated app. To obtain the number of samples, multiply the window size in seconds with the sampling frequency.
- **Overlap Size (ms):** This field shows the overlap (step) size in milliseconds. Since audio frames are processed with 50% overlap, a processing frame is updated at half of the window size time.
- **SPL Calibration (dB):** This field allows the user to set a calibration constant which converts the audio level from dB FS (full scale) to dB SPL (sound pressure level). This value needs to get properly set before the app is activated.
- **Audio Level:** This field shows the measured sound pressure level (SPL) in dB of the audio signal using the calibration constant.
- **Frame Processing Time:** This field shows the processing time of the integrated app in milliseconds per data frame.
- **GUI Update Rate (sec):** This field allows the user to set the time at which the audio level and frame processing time are updated. This time can be adjusted by the user.



- **Noise Update Rate (sec):** This field allows the user to set the time at which the noise estimation is updated for noise reduction. This time can be adjusted by the user.

### 2.3 Compression Settings View

This view shows various settings for the five frequency bands of the compression module that the user can set, see Fig 21. These settings include:

- **Compression Ratio:** This parameter indicates the amount of compression.
- **Compression Threshold (dB):** This parameter indicates the point after which the compression is applied.
- **Attack Time (ms):** This parameter indicates the time it takes for the compression module to respond when the signal level changes from a high to a low value.
- **Release Time (ms):** This parameter indicates the time it takes for the compression module to respond when the signal level changes from a low to a high value.

The integrated app provides the following 5 frequency bands:

- 0 - 500 Hz
- 500 – 1000 Hz
- 1000 – 2000 Hz
- 2000 – 4000 Hz
- above 4000 Hz

The compression function using the above 4 parameters is applied to each band. The app uses a scrolling option for the large view shown in Fig. 21.

No SIM
12:46 PM

Back

## Compression Settings

BAND 1: 0 TO 500 HZ

Compression Ratio
1
50
5

Threshold (dB)
-50
0
-5

Attack Time (ms)
0
200
5

Release Time (ms)
0
500
100

BAND 2: 500 TO 1000 HZ

Compression Ratio
1
50
5

Threshold (dB)
-50
0
-10

Attack Time (ms)
0
200
5

Release Time (ms)
0
500
100

BAND 3: 1000 TO 2000 HZ

Compression Ratio
1
50
5

Threshold (dB)
-50
0
-20

Attack Time (ms)
0
200
2

Release Time (ms)
0
500
50

BAND 4: 2000 TO 4000 HZ

Compression Ratio
1
50
4

Threshold (dB)
-50
0
-25

Attack Time (ms)
0
200
2

Release Time (ms)
0
500
50

BAND 5: ABOVE 4000 HZ

Compression Ratio
1
50
4

Threshold (dB)
-50
0
-35

Attack Time (ms)
0
200
2

Release Time (ms)
0
500
100

Fig. 21: Integrated app iOS GUI: Compression Settings view

### Section 3: Code Flow

This section states the app code flow. The user can view the code by running “Integrated\_App.xcodeproj” in the folder “Integrated-App” as shown in Fig. 17. The code is divided into 3 parts, see Fig. 22:

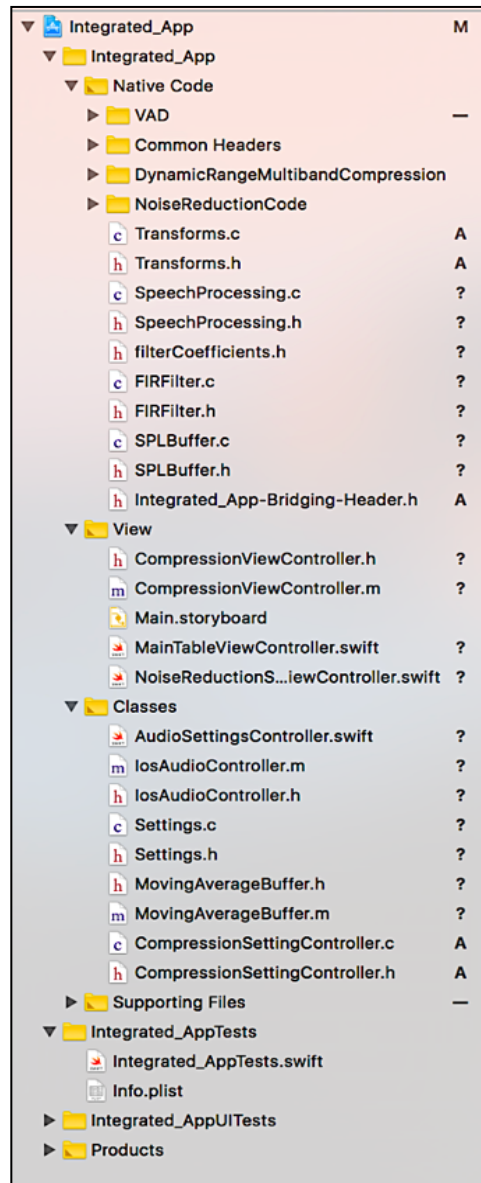


Figure. 22: Integrated app iOS code flow

- Native Code
- View
- Classes

### 3.1 Native Code

The native code section comprises the hearing aid modules written in C. It is divided into the following components:

- **Speech Processing:** This is the entry point of the native codes of the hearing aid modules. It initializes all the settings for the three modules and then processes the incoming audio signal according to the signal processing pipeline described in chapter 3.
- **FIRFilter:** FIR filtering is done for lowpass filtering before down-sampling and interpolation filtering is done after up-sampling.
- **Transform:** This component computes the FFT of the incoming audio frames.
- **SPLBuffer:** This component computes the average SPL over the GUI update time (noted in section 2.2).
- **VAD:** This component applies the codes as described in [2] for VAD. It includes:
  - **Feature Extraction:** This extracts the sub-band features using FFT and other features.
  - **Random Forest Classifier:** This is the random forest classifier detecting whether the incoming signal is noise or speech+noise based on the extracted feature vector.

- **VADProcessing:** This calls the feature extraction and random forest classifier acting together as VAD.
- **NoiseReductionCode:** This is the code for the noise reduction module as described in [12], which was initially developed in MATLAB and then converted into C using the MATLAB Coder [18].
- **DynamicRangeMultibandCompression:** This is the code for the compression module, which is also developed in MATLAB and then converted into C using the MATLAB Coder [18].
- **Common Headers:** This provides some common headers shared by both the noise reduction and compression modules. Note that these files are generated by the MATLAB Coder by converting MATLAB codes into C codes.

The breakdown of the three modules along with the common header is shown in Fig. 23.

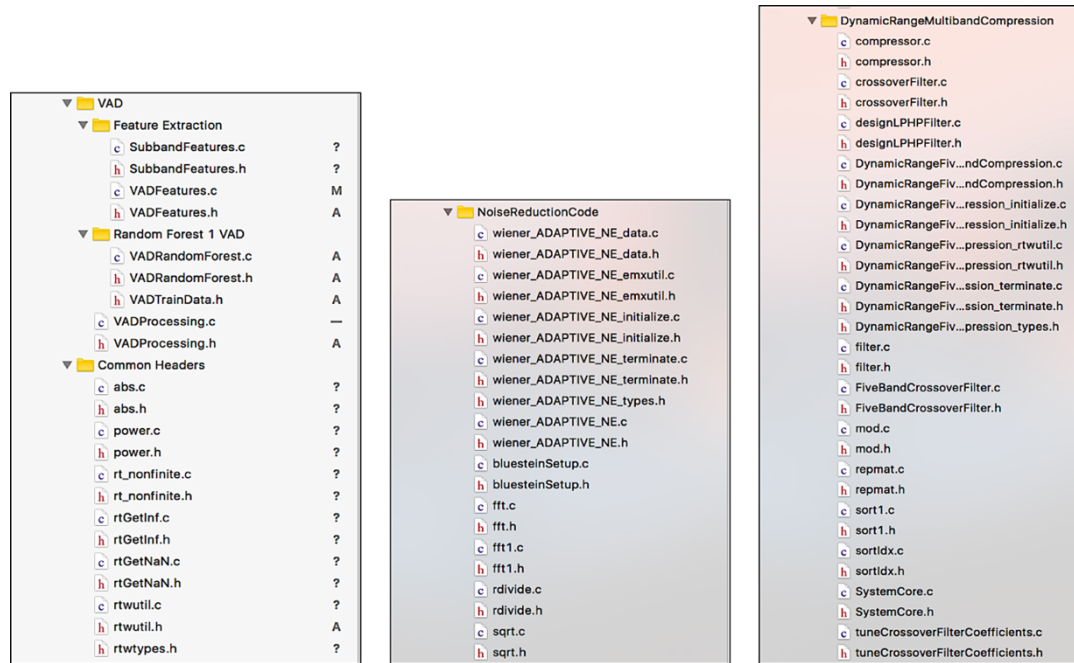


Figure. 23: Further breakdown of native code modules in iOS

### 3.2 View:

This section provides the setup for the GUI of the integrated app. The GUI has the following components:

- **Main.storyboard:** This provides the layout of the integrated app GUI.
- **MainTableViewController:** This provides the GUI elements and actions of the main GUI view. This is developed using Swift [24].
- **NoiseReductionSettingsTableViewController:** This provides the GUI elements and actions for the noise reduction and audio settings. This is developed using Swift.
- **CompressionViewController:** This provides the GUI elements and actions for the compression settings. This is developed in Objective-C.

### 3.3 Classes:

This section covers the controllers to pass data from the GUI to the native code and to update the status from the native code to appear in the GUI. The components are:

- **AudioSettingsController:** This provides the controls for audio processing and settings. This is written in Swift.
- **Settings:** This provides the variables for the audio control settings. Native codes use these variables settings for audio processing. These variables are updated through AudioSettingsController appearing in the GUI. This is written in C.
- **IosAudioController:** This controls the audio i/o setup for processing incoming audio frames by calling the native code. This is written in Objective-C. This loads/destroys the

settings and native variables by calling their initializers/destructors.

- **MovingAverageBuffer:** This provides a separate class written in Objective-C to compute the frame processing time. It provides the average processing time over the GUI Update Time mentioned in section 2.2.
- **CompressionSettingController:** This provides a separate variable array for the compression settings. Any update from the GUI elements of “CompressionViewController” gets updated here and the array of compression parameters for the 5 bands is used by the native code for compression. This is written in C.

#### **\*Supporting Files:**

There are supporting files as shown in Fig. 24. Along with the app logo and launcher images, there are:

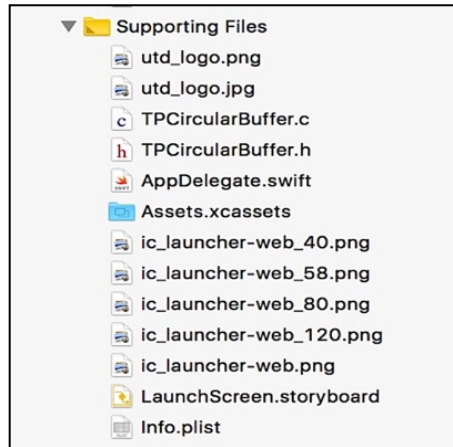


Figure 24: Supporting files

- **TPCircularBuffer:** This is an implementation of the circular buffer in [25], which is used in the integrated app to obtain a desirable frame size for audio processing.

- **AppDelegate:** This is called when the app is launched and initializes AudioSettingController.

#### **Section 4: Modularity and Modification**

This section covers the modularity of the integrated app and the steps to be taken to modify the app or any portion of the modules if needed.

- **Available bandwidth:** The app is developed in such a way that the hearing aid modules used here (i.e., VAD, Noise Reduction and Compression) can be replaced with other similar modules. It is also possible to add new modules if the available processing time bandwidth is not exceeded. The available processing time bandwidth can be obtained from the sampling frequency and audio i/o buffer size of iOS smartphones. For the lowest latency, iOS smartphones process 64 samples of incoming audio signal per frame at 48KHz sampling frequency which translates into an available processing time bandwidth of  $64/48000 = 1.33\text{ms}$ .
- **App Work Flow:** The work flow of the integrated app is straightforward and includes:
  - Detection of speech or noise activity of incoming audio frames by the VAD.
  - Noise estimation and reduction of incoming audio frames based on the VAD decision.
  - Compression of the noise reduction module output.

The declarations and definitions of initializations and destructions as well as the main function that call these modules are written the “SpeechProcessing” source files (.h and .c).



- **Replace or Add modules:** To replace or add module(s), include/remove:
  - Corresponding headers in “SpeechProcessing.h”
  - Set variables in the “VADNoiseReductionCompression” structure (same header).
  - Initialize variables inside the “initVAD\_NoiseReduction\_Compression” function as defined in “SpeechProcessing.c”.
  - Destruct variables inside the function named “destroyVAD\_NoiseReduction\_Compression” for optimized memory allocation and usage.
  - Function(s) that calls an updated module at the proper place inside the main function is named “doNoiseReduction\_Compression\_withVAD”.
- **Data transaction between the GUI and native code:** The “Settings” source files (.h and .c) provide an interface between the GUI and the native code for exchanging audio settings parameters. These parameters are:
  - **VAD Results:** The VAD decision is saved in “settings->classLabel”. If the user wishes to replace it with a new VAD, the decision can be saved in this variable. “MainTableViewController” takes this decision from the setting variable. Since “MainTableViewController” is written in Swift, the access to Settings is bridged through “Integrated\_App-Bridging-Header.h” which is included inside the native code folder (refer to Fig. 22). To train the VAD module of the app, use the VAD app as described in [2] and available at [26]. Make sure the app is used with the proper window size and sampling frequency (decimated sampling frequency).
  - **AudioOutput controls:**

- “settings->noiseReductionOutputType” saves the switch status for noise reduction.
- “settings->compressionOutputType” saves the switch status for compression.
- “settings->amplification” saves final amplification value from the slider.
- “settings->doSaveFile” saves the status of save i/o data switch.
- “settings->playAudio” saves the start/stop button status.
- “settings->fs” provides the sampling frequency.
- “settings->frameSize” provides the window size.
- “settings->stepSize” provides the overlap size.
- “setting->calibration” saves the SPL calibration value.
- “settings->guiUpdateInterval” saves the time at which audio level and processing time get updated.
- “settings->dbpower” provides dB SPL power computed in “Transform” and averaged in “SPLBuffer” over the “guiUpdateInterval” time.
- “settings->processTime” provides the average frame processing time computed by “MovingAverageBuffer” over the “guiUpdateInterval” time.
- “settings->noiseEstimateTime” saves the time over which noise parameters are estimated and averaged.

These data are passed to the GUI through “AudioSettingsController.swift”. The user needs to update this file. Also, in the view files if there are any changes (replace/addition), they appear in the “Settings” source files.

- **Compression Controls:** “CompressionSettingsController” provides a separate set of parameters for the compression module that are passed between the native code and “CompressionViewController”. The compression parameters set by the user are saved in the “dataIn” array variable and 5 separate flags are set to update the compression function or curves for 5 frequency bands of the compression module. The native compression module calls the “CompressionSettingsController” header.
- **Compiler optimization:** Using compiler optimization improves the code performance and/or size depending on which optimization level is used. For the iOS version of the integrated app, “-O2” optimization level is used for debugging and “-Os” is used for releasing. Details are given in [27] and [28]. The user can change them according to specific requirements. To set these flags, follow the steps noted below, see Fig. 25.
  - Select the project name on the left in the project view.
  - Select the project under the “TARGETS” option in the middle.
  - Select “Build Settings” from the toolbar above.
  - In the search field, type “optimization”.
  - Under optimization level, set optimization flags accordingly.

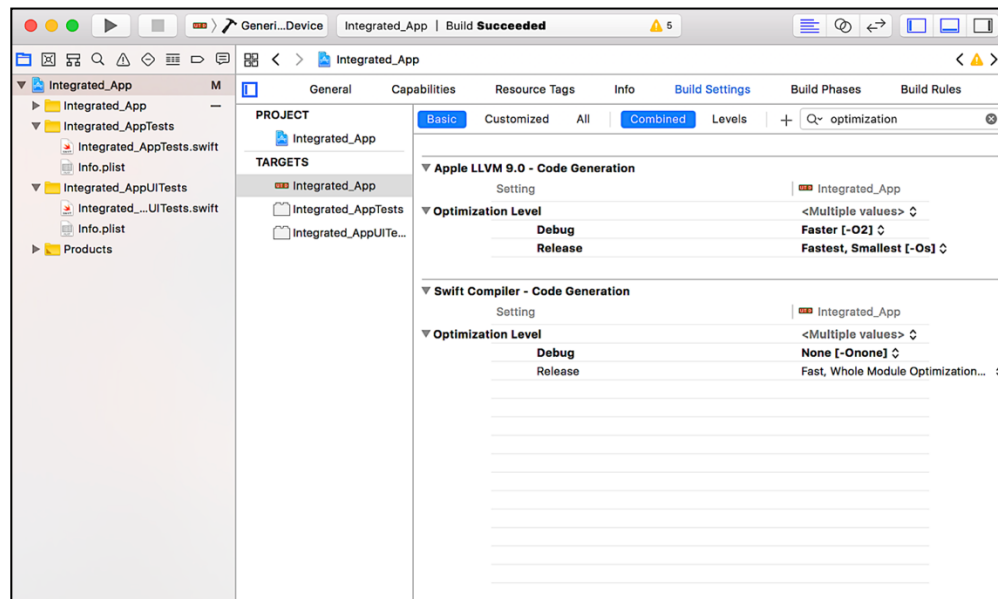


Figure. 25: Setting optimization level in Xcode for the integrated app iOS

**PART 2**

**ANDROID**

## **Section 1: Running the App**

The Android version of the integrated app was developed using Android Studio (Version 2.3.3). To run the Android version of the integrated app, it is necessary to have Superpowered SDK which can be obtained from the link at [7]. The use of SuperpoweredSDK enables low latency audio processing.

To open and run the app:

- Open Android Studio.
- Click on ““Open an existing Android Studio project””.
- Navigate to the app location and open it.
- Make sure that the NDK is installed. The proper locations of ndk, sdk and superpoweredSDK are given in “local.properties”.
- Make sure the environment has the proper platform and build tools version.
- Enable the developer option on the Android smartphone to be used.
- Connect the Android smartphone using a USB cable and allow data access and debugging to the smartphone.
- Clean the project first, then click run button from Android Studio and select the device.

More details of the above steps are covered in [29].

## **Section 2: Android GUI**

This section covers the GUI of the developed integrated app and its entries. The GUI consists of three views:

- Main View

- Noise Reduction Settings View
- Compression Settings View

## 2.1 Main View

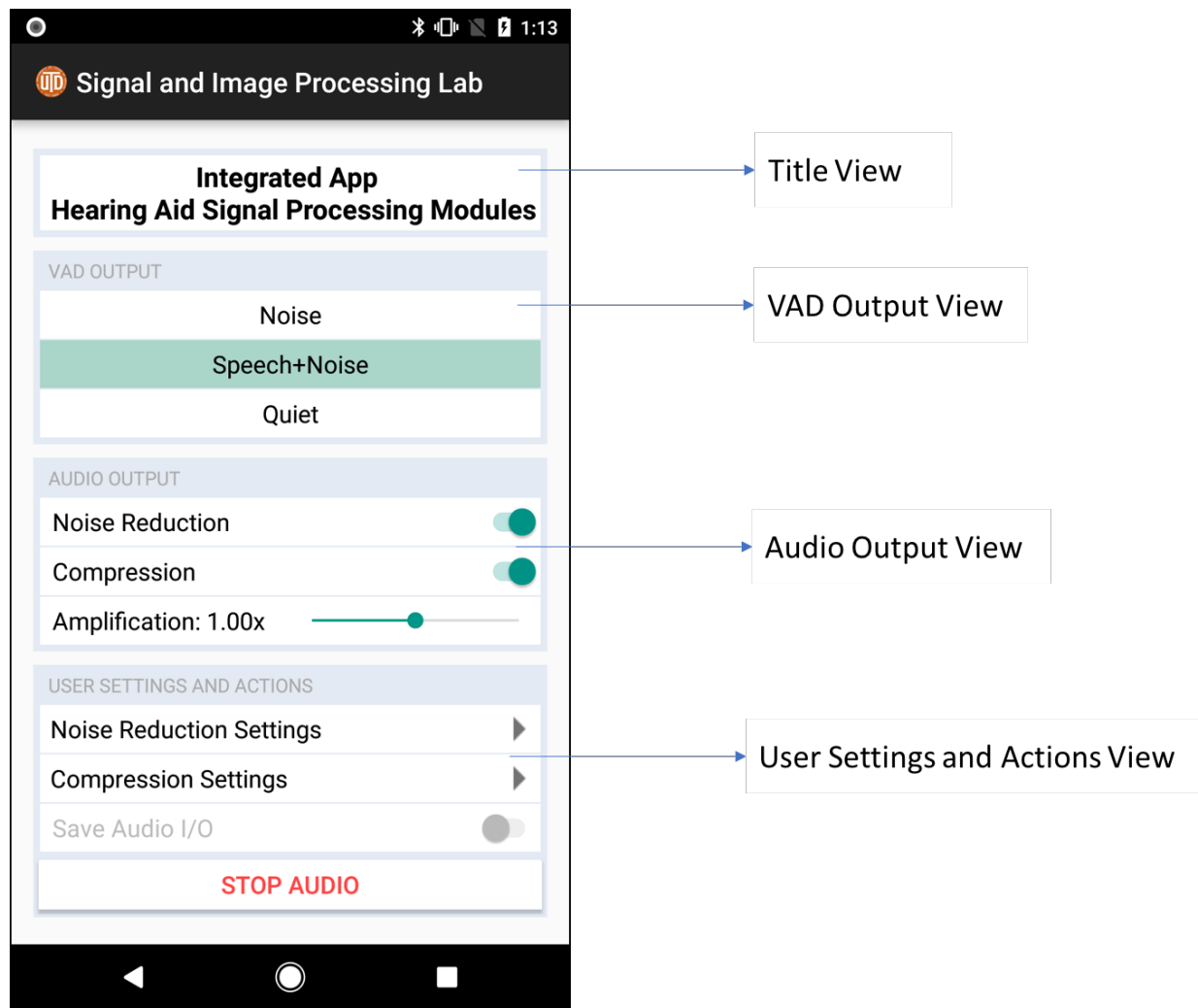


Figure. 26: Integrated app Android GUI: Main view

Main view consists of 4 views, see Fig. 26:

- Title View

- VAD Output View
- Audio Output View
- User Settings and Actions View

### **2.1.1 Title View**

The title view displays the title of the app.

### **2.1.2 VAD Output View**

This view corresponds to the VAD output, that is, noise, speech+noise, or quiet.

### **2.1.3 Audio Output View**

This view consists of:

- A switch to turn on and off the noise reduction module
- A switch to turn on and off the compression module
- A seek bar (in Android, slider is named a seek bar) to control final amplification.

### **2.1.4 User Settings and Actions Views**

This view provides the following entries:

- Noise reduction (and audio) settings
- Compression settings
- An option to save input/output audio signals
- A button for starting and stopping the app



## 2.2 Noise Reduction Settings View

This view contains various settings for the noise reduction and audio processing. As shown in Fig. 27, the items of this settings view are:

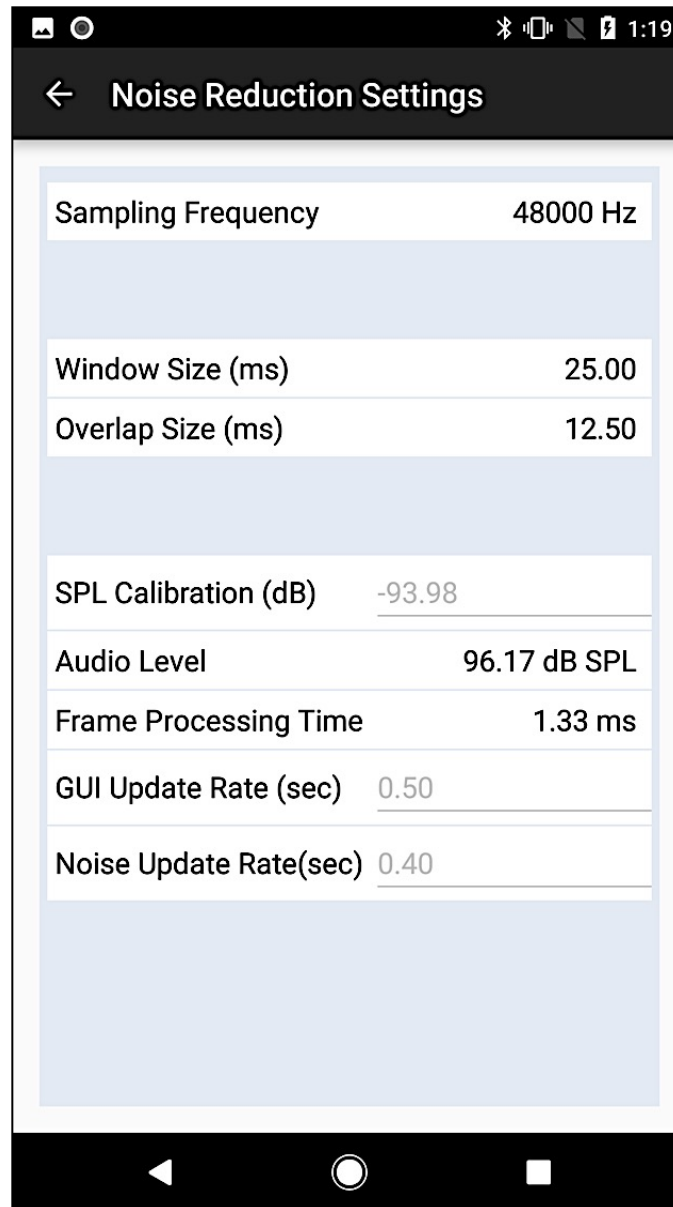


Fig. 27: Integrated app Android GUI: Noise Reduction Settings view

- **Sampling Frequency:** This field shows the sampling frequency. To obtain the lowest latency for Android mobile devices, this value is set to 48000 Hz. The audio signal is down-sampled to a sampling frequency of 16000 Hz in order to make the processing computationally more efficient.
- **Window Size (ms):** This field shows the window (frame) size in milliseconds. Basically, this indicates how many data samples (of this window length) get processed by the integrated app. To obtain the number of samples, multiply the window size in seconds with the sampling frequency.
- **Overlap Size (ms):** This field shows the overlap (step) size in milliseconds. Since audio frames are processed with 50% overlap, the content of a processing frame is updated at half of the window size time.
- **SPL Calibration (dB):** This field allows the user to set a calibration constant which converts the audio level from dB FS (full scale) to dB SPL (sound pressure level). This value can be adjusted by the user.
- **Audio Level:** This field shows the measured sound pressure level (SPL) in dB of the audio signal using the calibration constant.
- **Frame Processing Time:** This field shows the processing time in milliseconds taken by the integrated app per data frame.
- **GUI Update Rate (sec):** This field allows the user to set the time at which the audio level and frame processing time are updated. The audio level and frame processing time are averaged over this time. This time can be adjusted by the user.

- **Noise Update Rate (sec):** This field allows the user to set the time at which the noise estimation is updated for performing noise reduction. This time can be adjusted by the user.

## 2.3 Compression Settings View

This view shows various settings for the compression module for the five frequency bands, see Fig 28. These settings include:

- **Compression Ratio:** This parameter indicates the amount of compression.
- **Compression Threshold (dB):** This parameter indicates the point after which the compression is applied.
- **Attack Time (ms):** This parameter indicates the time it takes for the compression module to respond when the signal level changes from a high to a low value.
- **Release Time (ms):** This parameter indicates the time it takes for the compression module to respond when the signal level changes from a low to a high value.

The integrated app has the following 5 frequency bands:

- 0 - 500 Hz
- 500 – 1000 Hz
- 1000 – 2000 Hz
- 2000 – 4000 Hz
- above 4000 Hz

The compression function based on the above 4 parameters is applied to each band. The app uses a scrolling option for the large view of the compression settings, see Fig. 28.

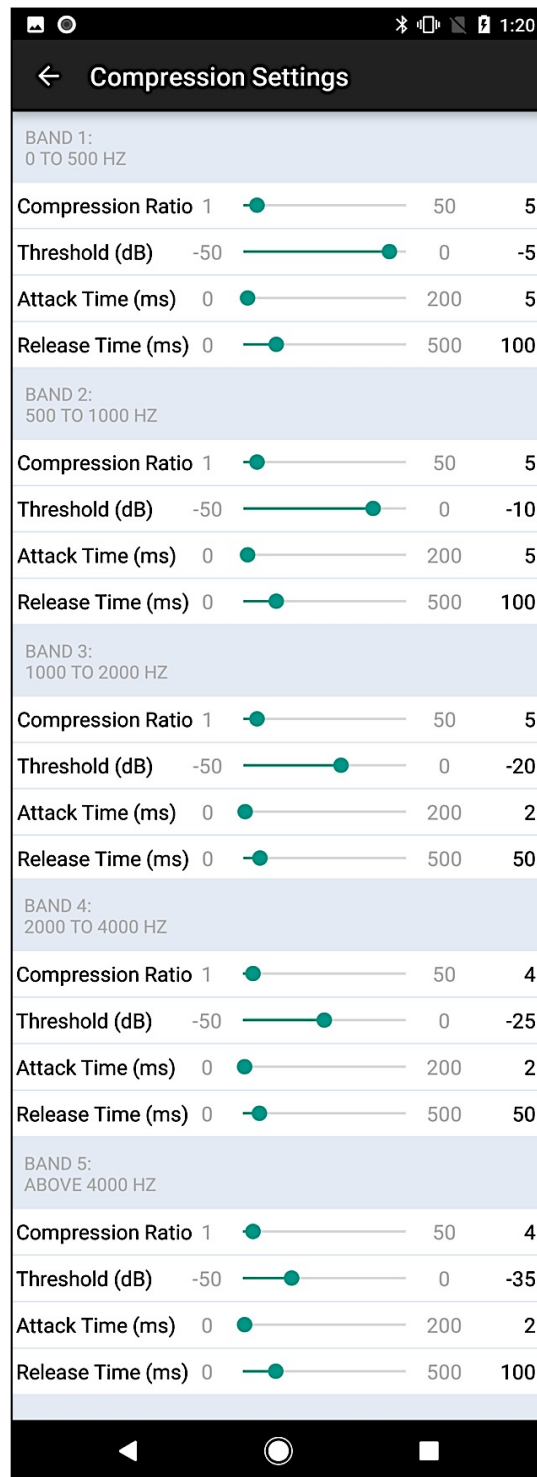


Fig. 28: Integrated app Android GUI: Compression Settings view

### Section 3: Code Flow

This section covers the integrated app code flow. The folder organization of the app can be seen under the “Android” view, see Fig. 29.

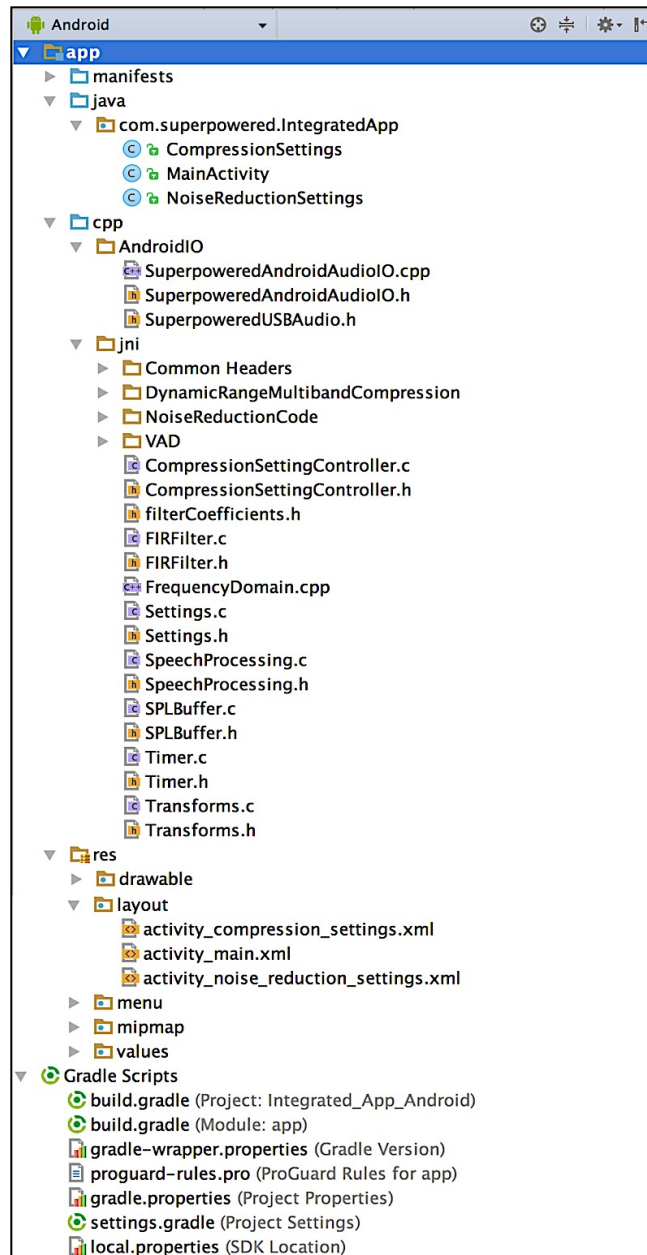


Fig. 29: Integrated app Android version: project organization

The Android project of the app is organized into the following code flow:

- **java:** The java folder contains three “.java” files, which handle all the operations of the app and provide the link between the GUI and the native code.
  - **MainActivity:** This contains the GUI elements and controls the Main view of the app.
  - **NoiseReductionSettings:** This contains the GUI elements and controls the Noise Reduction Settings view.
  - **CompressionSettings:** This contains the GUI elements and controls the Compression Settings view.
- **cpp:** This folder contains the native code, which is subdivided into these two folders:
  - **AndroidIO:** This subfolder provides the Superpowered source root files to control the audio i/o interface of the app.
  - **jni:** This folder contains all the function calls to start the audio i/o and the C codes of the implemented hearing aid modules.
- **res->layout:** The layout subfolder of the recourse folder contains the GUI layouts of the integrated app appearing in these three .xml files:
  - **activity\_main.xml:** It provides the layout for Main view.
  - **activity\_noise\_reduction\_settings.xml:** It provides the layout for Noise Reduction Settings view.
  - **activity\_compression\_settings.xml:** It provides the layout for Compression Settings view.

### 3.1 Native Code and Settings

This native code section states the implementation aspects of the hearing aid modules and settings in C++/C code. It is divided into the following:

- **FrequencyDomain:** This file acts as a connection or bridge between the native code and the java activities/GUI. It contains the necessary function calls and initializations for creating the audio i/o interface with the GUI settings and processing of the hearing aid modules per frame. The result of the processed audio is passed back to the app GUI. This file is written in C++. The other parts stated below are written in C.
- **Speech Processing:** This is the entry point of the native codes of the hearing aid modules. It initializes all the settings for the three modules and then processes the incoming audio signal according to the signal processing pipeline described in chapter 3.
- **FIRFilter:** FIR filtering is done for lowpass filtering before down-sampling and interpolation filtering is done after up-sampling.
- **Transform:** This computes the FFT of incoming audio frames.
- **SPLBuffer:** This computes the average SPL over the GUI update time (mentioned earlier in section 2.2).
- **Settings:** This provides the variables for the audio control settings. The native codes use the variables in this settings (in a structure) for audio processing. “MainActivity” initially loads this settings structure when the app is loaded. The corresponding variables are updated through FrequencyDomain appearing in the GUI.
- **CompressionSettingController:** This provides a separate variable array for

compression settings. Any update from the GUI elements of the “CompressionSettings” activity gets updated here and the array of compression parameters for the 5 bands is used by the native code for compression.

- **VAD:** This corresponds to the VAD codes as described in [2] including:
  - **Feature Extraction:** This extracts the subband features using FFT and other VAD features.
  - **Random Forest Classifier:** This is the classifier that detects whether the incoming signal is noise or speech+noise based on the extracted feature vector.
  - **VADProcessing:** This calls the feature extraction and random forest classifier to perform VAD.
- **NoiseReductionCode:** This corresponds to the codes as described in [29] for the noise reduction module, which is developed in MATLAB and then converted into C using the MATLAB Coder [18].
- **DynamicRangeMultibandCompression:** This corresponds to the codes for the compression module, which is also developed in MATLAB and then converted into C using the MATLAB Coder [18].
- **Common Headers:** This provides some common headers shared by both the noise reduction and compression modules. Note that these files are generated by the MATLAB Coder by converting MATLAB codes into C codes.

The breakdown of the three modules along with the common header is shown in Fig. 30.



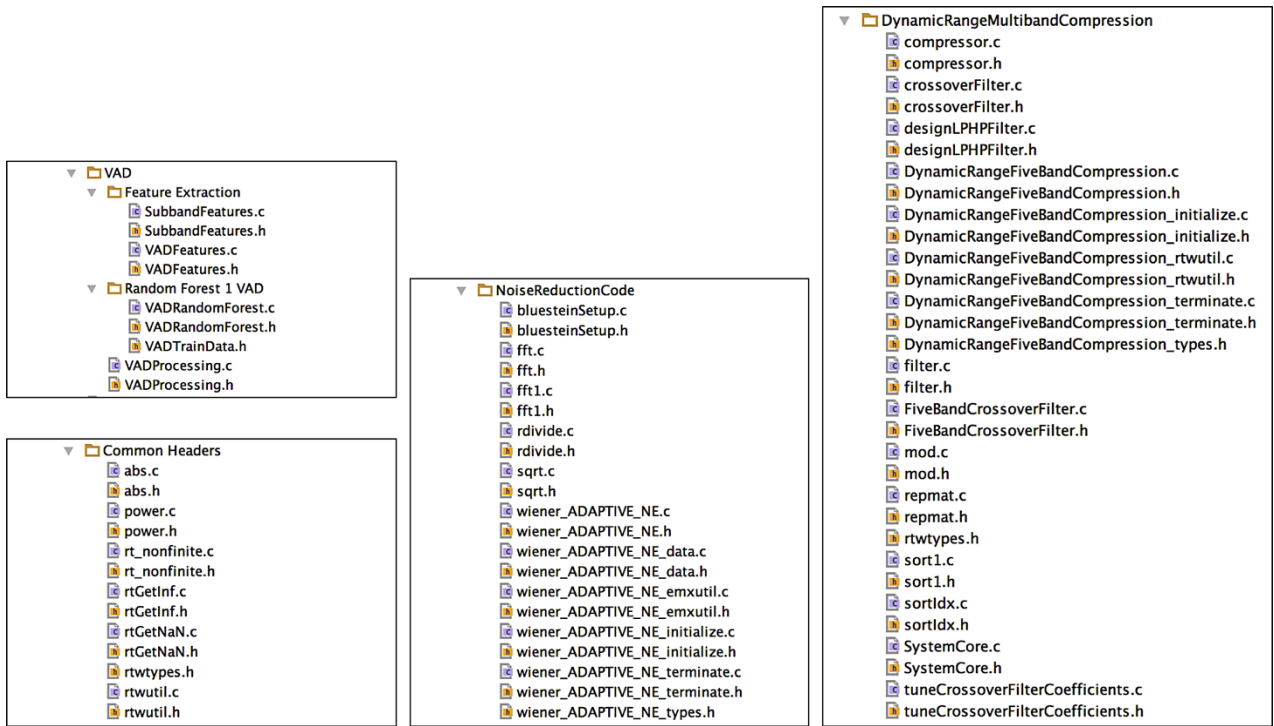


Figure 30: Further breakdown of native code modules in Android

## Section 4: Modularity and Modification

This section covers the modularity of the integrated app and the steps one needs to take to modify the app or any portion of the modules if needed.

- **Available bandwidth:** The app is developed in such a way that the hearing aid modules used (i.e., VAD, Noise Reduction and Compression) can be replaced with other similar modules. It is also possible to add new modules if the available processing time bandwidth is not exceeded. The processing time bandwidth can be obtained from the sampling frequency and audio i/o buffer size of the Android smartphone. For the lowest latency, many Android smartphones process 192 samples of incoming audio signal per

frame at 48KHz sampling frequency. This translates into an available processing time bandwidth of  $192/48000 = 4\text{ms}$ . This number of samples (192) can be obtained by calling “getProperty()” API of Android’s “AudioManager” with the attribute “PROPERTY\_OUTPUT\_FRAMES\_PER\_BUFFER”.

- **App Work Flow:** The working flow of the integrated app is straightforward and is as follows:
  - Detection of speech or noise activity of incoming audio frames by the VAD.
  - Noise estimation and reduction of incoming audio frames based on the VAD decision.
  - Applying compression on the output of the noise reduction module.

The declarations and definitions of initializations and destructions as well as the main function that calls these modules are written in the “SpeechProcessing” source files (.h and .c).

- **Replace or Add modules:** To replace or add module(s), include/remove the following:
  - Corresponding headers in “SpeechProcessing.h”.
  - Appropriate variables in the “VADNoiseReductionCompression” structure (same header).
  - Initialization of variables inside the “initVAD\_NoiseReduction\_Compression” function as defined in “SpeechProcessing.c”.
  - Destruction of variables inside the function named “destroyVAD\_NoiseReduction\_Compression” for optimized memory allocation and usage.

- Function(s) that calls the updated module at the proper place inside the main function named “doNoiseReduction\_Compression\_withVAD”.

All the native codes and headers folder path have to be included in the “build.gradle” file inside “android.sources.main.jni { exportedHeaders {....} }” as shown in Fig. 31. It is required to add the path of each folder and also its subfolders separately.



Fig. 31: Add native folder paths

- **Data transaction between the GUI and Native Code:** For the Android version of the developed integrated app, Java Native Interface (JNI) is used to interface with native codes in the Java environment. This approach is described in [9, 10, 30]. The following procedure needs to be followed:

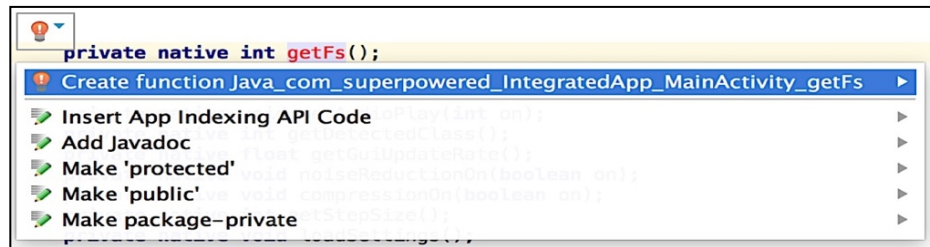
- To call any C function or update setting parameters, declare a linking function in the Java Activity file in which it will be utilized (MainActivity, NoiseReductionSettings, or CompressionSettings) using the Java keyword “native”.

**Example:** To get sampling frequency in MainActivity, declare “**private native int getFs()**” inside “**public class MainActivity extends AppCompatActivity {}**”. A red

inspection alert will pop up to create the linking function in FrequencyDomain.cpp, see Fig. 32a.

- Click on the create option. It will create a function in FrequencyDomain.cpp, see Fig. 32b.
- Modify the function definition using extern “C” keyword, see Fig. 32c, since it is calling the “setting->fs” variable from a C file.

Follow the steps for all the native calls. The entry point to Native Portion of the app from Java is obtained through the “**private native void** FrequencyDomain()” function after following the above steps.



(a)

```
JNIEXPORT jint JNICALL
Java_com_superpowered_IntegratedApp_MainActivity_getFs(JNIEnv *env, jobject instance) {

    // TODO
}
```

(b)

```
extern "C" JNIEXPORT int
Java_com_superpowered_IntegratedApp_MainActivity_getFs(JNIEnv* __unused env, jobject __unused instance) {

    if (settings != NULL) {
        return settings->fs;
    } else {
        return 0;
    }
}
```

(c)

Figure 32: Creating native linking function

The “Settings” source files (.h and .c) provide an interface between the GUI and the native code

to exchange audio settings parameters. The Main settings parameters are:

- **VAD Results:** VAD decision is saved in “settings->classLabel”. If the user wishes to include a new VAD, the decision can be saved in this variable. The “MainActivity” takes this decision from the settings variable. Since “MainActivity” is written in Java, the access to Settings is bridged through “FrequencyDomain.cpp” following the steps noted in Fig. 16. To train the VAD module of the app, use the VAD app guidelines as described in [2] which is available at [26]. Make sure to use the app with a proper window size and sampling frequency (decimated sampling frequency).
- **AudioOutput controls:**
  - “settings->noiseReductionOutputType” saves the switch status for noise reduction.
  - “settings->compressionOutputType” saves the switch status for compression.
  - “settings->amplification” saves final amplification value from the seek bar.
  - “settings->playAudio” saves the start/stop button status.
  - “settings->fs” provides the sampling frequency.
  - “settings->frameSize” provides the window size.
  - “settings->stepSize” provides the overlap size.
  - “setting->calibration” saves the SPL calibration value.

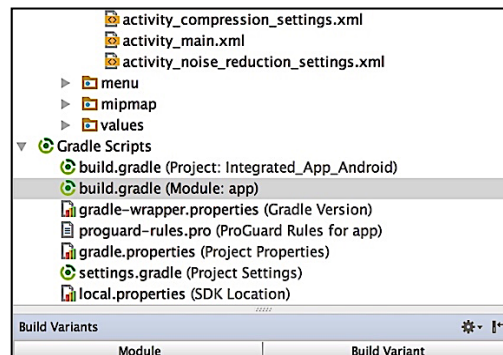
- “settings->guiUpdateInterval” saves the time at which audio level and processing time get updated.
- “settings->dbpower” provides the dB SPL power computed in “Transform” and averaged from “SPLBuffer” over the “guiUpdateInterval” time.
- “settings->noiseEstimateTime” saves the time over which noise parameters are estimated and averaged.

These data are passed to the GUI through “FrequencyDomain.cpp” as per the steps illustrated in Fig. 32. The user needs to update this file and also the activity files if there is any update (i.e. replace/addition) in the “Settings” source files.

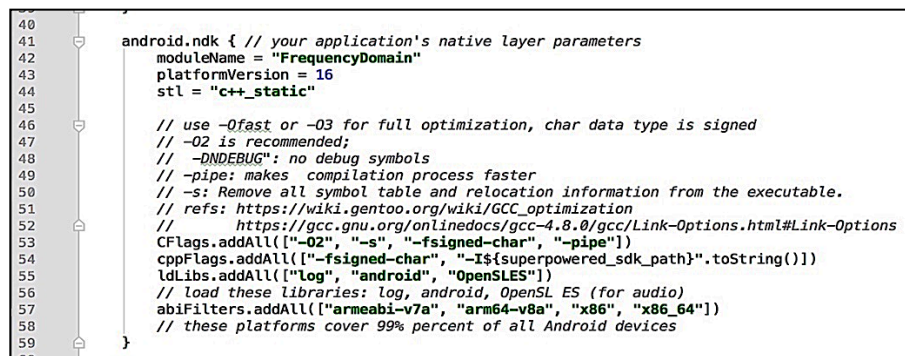
- **Compression Controls:** “CompressionSettingsController” provides a separate setting parameters for the compression module that are passed between the native code and the “CompressionSettings” activity. The compression parameters set by the user are saved in the “dataIn” array variable and 5 separate flags are set to update the compression function or curves for the 5 frequency bands of the compression module. The native compression module calls this “CompressionSettingsController” header. The interaction with the GUI is also done by the same steps as illustrated in Fig. 32.
- **Compiler optimization:** Using compiler optimization improves the code performance and/or size depending on which optimization level is used. For the Android version of the integrated app, several optimization flags are listed below:
  - -O2: Optimization level, recommended.

- `-s`: Removes all symbol table and relocation information from the executable.
- `-fsigned-char`: char data type is signed.
- `-pipe`: makes compilation process faster.

The details of the above are given in [27] and [28]. The user can change them according to specific requirements. To set these flags, follow the steps as noted in Fig. 33.



(left)



(right)

Fig 33: Setting optimization level in Android Studio for the integrated app Android

- Under “Android” of the project view explorer of Android Studio, select `build.gradle` (Module: app) under Gradle Scripts, see left side of Fig. 33.
- At the right side of Fig. 28, set the optimization flags under model → `android.ndk` → `CFlags.addAll()`;

### Timing Difference Between iOS and Android Versions of the Integrated App

The low-latency audio i/o setup for iOS is done using the software package CoreAudio API [21] and for Android using the software package Superpowered SDK [7]. Both the iOS and the Android version of the Integrated App use the same C codes for the lowpass filtering, down-sampling, implementing hearing aid modules, up-sampling and interpolation filtering. The average frame processing time for the complete pipeline with and without the implementation of hearing aid modules is given in the table below:

**TABLE 3: TIMING DIFFERENCE BETWEEN THE IOS AND ANDROID VERSIONS OF THE INTEGRATED APP**

<b>Version of the Integrated App</b>	<b>Overall frame processing time, T1 (ms)</b>	<b>Frame processing time without hearing aid modules, T2 (ms)</b>	<b>Processing time for hearing aid modules, T1-T2 (ms)</b>
iOS	0.75	0.6	0.15
Android	1.33	0.3	1.03

Table 3 indicates lower frame processing time for the iOS version than the Android version of the app. The iOS platform gives more compatibility and lower latency Bluetooth connection than the Android platform. In our experimentation, the lowest Bluetooth latency was achieved by using iPhone 7 with Starkey Halo2 [8] hearing aid.



## REFERENCES

1. A. Sehgal, F. Saki, and N. Kehtarnavaz, "Real-time implementation of voice activity detector on ARM embedded processor of smartphones," *Proceedings of IEEE 26th International Symposium on Industrial Electronics (ISIE)*, Edinburgh, UK, pp. 1285-1290, 2017.
2. A. Sehgal and N. Kehtarnavaz, 'User's Guide: A Voice Activity Detector (VAD) Smartphone App for Hearing Improvement Studies', 2017. [Online]. Available: <http://www.utdallas.edu/ssprl/files/Users-Guide-VAD.pdf>
3. F. Saki, A. Sehgal, I. Panahi and N. Kehtarnavaz, "Smartphone-based real-time classification of noise signals using subband features and random forest classifier," *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Shanghai, China, pp. 2204 – 2208, 2016.
4. A. Sehgal and N. Kehtarnavaz, 'User's Guide: A Noise Classifier Smartphone App for Hearing Improvement Studies', 2017. [Online]. Available: <http://www.utdallas.edu/ssprl/files/UsersGuide-NoiseClassifier-App1.pdf>
5. 'Integrated VAD and Noise Classifier'. [Online]. Available: <https://utdallas.app.box.com/v/SIP-IntegratedVAD-NoiseClass>
6. California Ear Institute, 'Hearing in Noise Test (HINT)', [Online]. Available: <http://www.californiaearinstitute.com/audiology-services-hint-bay-area-ca.php>
7. 'Superpowered: Fastest Mobile Audio Engine for Games, VR, Music and Interactive Audio Apps', [Online]. Available: <http://superpowered.com>
8. Starkey Hearing Technologies, 'Halo 2', [Online]. Available: <http://www.starkey.com/hearing-aids/technologies/halo-2-made-for-iphone-hearing-aids>
9. N. Kehtarnavaz and F. Saki, *Anywhere-Anytime Signals and Systems Laboratory: From MATLAB to Smartphones*, Morgan and Claypool Publishers, 2016.
10. N. Kehtarnavaz, S. Parris, A. Sehgal, *Smartphone-Based Real-Time Digital Signal Processing*, Morgan and Claypool Publishers, 2015.
11. A. Sehgal and N. Kehtarnavaz, "A convolutional neural network smartphone app for real-time voice activity detection," *IEEE Access*, online open access, Feb 2018.
12. A. Bhattacharya, A. Sehgal, and N. Kehtarnavaz, "Low-latency smartphone app for real-time noise reduction of noisy speech signals," *Proceedings of IEEE 26th International Symposium on Industrial Electronics (ISIE)*, Edinburgh, UK, pp. 1280 – 1284, 2017.

13. Nine Sights, ‘National Science Foundation Hearables Challenge’, 2017. [Online]. Available: <https://ninesights.ninesigma.com/web/hearables/innovationcontest>
14. A. Sehgal and N. Kehtarnavaz, "Utilization of two microphones for real-time low-latency audio smartphone apps," *Proceedings of IEEE International Conference on Consumer Electronics (ICCE)*, Las Vegas, NV, Jan 2018.
15. Starkeypro.com, ‘The Compression Handbook’, [Online]. Available: [https://starkeypro.com/pdfs/The\\_Compression\\_Handbook.pdf](https://starkeypro.com/pdfs/The_Compression_Handbook.pdf)
16. D. Giannoulis, M. Massberg, and J. Reiss, "Digital dynamic range compressor design — a tutorial and analysis," *Journal of Audio Engineering Society*, vol. 60, pp. 399–408, 2012.
17. Mathworks.com, ‘Multiband Dynamic Range Compression’, [Online]. Available: <https://www.mathworks.com/help/audio/examples/multiband-dynamic-range-compression.html>
18. Mathworks.com, ‘MATLAB Coder: Generate C and C++ code from MATLAB code’, [Online], Available: <https://www.mathworks.com/products/matlab-coder.html>
19. Developer.apple.com, ‘Xcode’, [Online]. Available: <https://developer.apple.com/xcode>
20. Developer.android.com, ‘Android Studio’, [Online]. Available: <https://developer.android.com/studio/index.html>
21. Developer.apple.com, ‘Core Audio’, [Online], Available: <https://developer.apple.com/documentation/coreaudio>
22. P. Loizou, *Speech Enhancement: Theory and Practice*, CRC Press, 2013.
23. Codewithchris.com, ‘How to Deploy your App on an iPhone’, 2016. [Online]. Available: <http://codewithchris.com/deploy-your-app-on-an-iphone/>
24. Developer.apple.com, ‘The Swift Programming Language’, [Online], Available: [https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift\\_Programming\\_Language/index.html](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/index.html)
25. M. Tyson, ‘A simple, fast circular buffer implementation for audio processing’, [Online]. Available: <https://github.com/michaeltyson/TPCircularBuffer>
26. ‘Voice Activity Detection (VAD)’, [Online]. Available: <https://utdallas.app.box.com/v/SIP-NP-VAD1>

27. Wiki.gentoo.org, 'GCC Optimization', [Online]. Available:  
[https://wiki.gentoo.org/wiki/GCC\\_optimization](https://wiki.gentoo.org/wiki/GCC_optimization)
28. Gcc.gnu.org, 'Options for Linking', [Online]. Available:  
<https://gcc.gnu.org/onlinedocs/gcc-4.8.0/gcc/Link-Options.html#Link-Options>
29. A. Sehgal, A. Bhattacharya, T. Chowdhury, and N. Kehtarnavaz, 'User's Guide: A Low-Latency Noise Reduction Smartphone App for Hearing Improvement Studies', 2017. [Online]. Available: <http://www.utdallas.edu/ssprl/files/Users-Guide-Noise-Reduction.pdf>
30. N. Kehtarnavaz and A. Sehgal, 'User's Guide: How to Run C/MATLAB Codes on Android Smartphones as Open Source and Portable Research Platforms for Hearing Improvement Studies', 2017. [Online]. Available:  
<http://www.utdallas.edu/ssprl/files/Users-Guide-Android.pdf>

## **BIOGRAPHICAL SKETCH**

Tahsin Ahmed Chowdhury received his Bachelor in Science degree in Electrical and Electronic Engineering from Khulna University of Engineering & Technology (KUET) in Bangladesh in 2013. He is currently pursuing his MS degree in Electrical Engineering at the University of Texas at Dallas. His research interests include real-time audio signal processing, signal and image processing.

## **CURRICULAM VITAE**

### **EDUCATION:**

- **The University of Texas at Dallas**, Richardson, TX May, 2018  
MS in Electrical Engineering, CGPA: 3.585/4
- **Khulna University of Engineering and Technology (KUET)**, Bangladesh. Sept, 2013  
BSc. in Electrical and Electronic Engineering, CGPA: 3.67/4

### **WORK EXPERIENCE:**

- Systems Engineer. Sales: Cisco Systems Inc., Richardson, Texas Jan 2018 – present
- Student Worker: SIP Lab, The University of Texas at Dallas Aug 2017 – Dec 2017
- Software Engineer: Samsung R&D Institute, Bangladesh (SRBD) May 2014 – Dec 2015
- System Support Engineer: Thakral Information Systems Pvt Ltd. Jan 2014 – Apr 2014

### **SKILL SET:**

- Programing Skill: C, C++, MATLAB, Java, Python and SQL.
- Machine Learning: Neural Networks, Random Forest, and other supervised methods.
- Coding optimization: NEON Instincts for vectorization in ARM processor (smartphones).
- Software Packages: Microsoft Word, Excel, PowerPoint, Visio, Visual Studio.
- ACM Solving: Solved 144 programming related problems in uVa online judge.
- Miscellaneous: Working knowledge of Android Studio, XCode, Windows, Mac OS, Ubuntu.
- Coursework: Pattern Recognition, Speech and Speaker Recognition, Microprocessor Systems, DSP-I, DSP-II, Computer Networks, Programming with C, C++, Matlab.

### **RESEARCH PROJECT:**

- **Real-Time Smartphone Apps Integrating Signal Processing Modules of Hearing Aids: MSEE Thesis**
  - Development of a smartphone app integrating voice activity detector (VAD) and supervised noise classifier for hearing improvement studies: To classify noise from environment based on the decision made by VAD.
  - Integrating signal processing modules of hearing aids into a real-time smartphone app: To provide an open source research platform where VAD, noise reduction and compression modules were incorporated together in a same pipeline for further studies in hearing improvement.
- **Performance Analysis on Handover Priority Schemes and a New Adaptive Channel Reservation for Handover Priority in Wireless Networks: BSc. Thesis**
  - An improved model for optimized channel utilization with better handover priority and new call blocking probability within acceptable range; used MATLAB for developing the model and analyzing the model properties.

### **ACADEMIC PROJECTS:**

- **Signal Processing Algorithm Implementation on Smartphones in C Programming Language: *An Independent Study***  
Converted an existing algorithm from MATLAB into C (without using MATLAB Coder) to obtain better process time for noise reduction in real time incoming audio signal from smartphone microphone.
- **Detecting Speech Activity Using Convolutional Neural Network(CNN): *Pattern Recognition project.***  
Classifying speech from noisy speech from given feature extracted data using CIFAR-10 framework of MatCovNet (MATLAB based) toolbox.
- **Comparative Classification of Speech Emotions Using MFCC Features: *Speech and Speaker Recognition project.***  
Detection of speech emotion (Happiness, sadness, anger and neutral) using neural network.
- **System Identification Using Real Data: *Digital Signal Processing II project***  
Identifying system using input and output audio data using DSP system toolbox in MATLAB, applied Adaptive Filtering (NLMS) algorithm.
- **Undergrad Projects:**
  - Development of Symmetrical Three-Phase Circuit Analyzing Tool using C programming language.
  - Development of a Simple Banking System Software using C++ Programming language.
  - Micro-controller Based Secured Room Access using Code Vision AVR with Atmega32 Microcontroller.

### **PUBLICATIONS:**

- T. A. Chowdhury, A. Sehgal, N. Kehtarnavaz, “Integrating Signal Processing Modules of Hearing Aids into A Real-Time Smartphone App,” accepted for publication in *Proceedings of the 40<sup>th</sup> International Conference of IEEE Engineering in Medicine and Biology Society*, Honolulu, Hawaii, 2018.
- T. A. Chowdhury, R. Bhattacharjee, M. Z. Chowdhury, “Handover priority based on adaptive channel reservation in wireless networks,” *International Conference on Electrical Information and Communication Technology (EICT)*, Khulna, Bangladesh, pp. 1-5, 2014.
- R. Bhattacharjee, T. A. Chowdhury, M. Z. Chowdhury, “Priority based adaptive guard channel for multi-class traffic in wireless networks,” *International Conference on Electrical Information and Communication Technology (EICT)*, Khulna, Bangladesh, pp. 1-4, 2014.