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

2018

Table of Contents

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 [1]. The integrated modules include the Voice Activity Detector (VAD) discussed in [2], the noise reduction module discussed in [3] and the compression module discussed in [4] and [5]. 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.

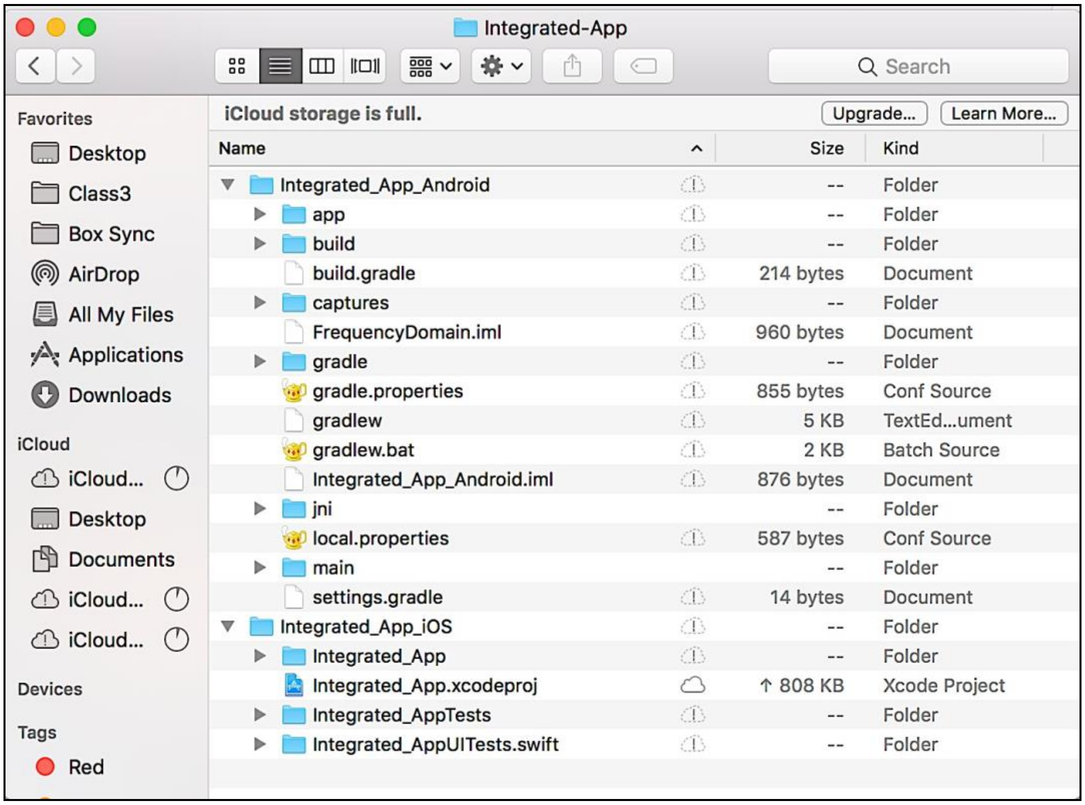
Devices used for running the integrated app

- 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. 1 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



iCloud storage is full.

Name	Size	Kind
▼ Integrated_App_Android	--	Folder
▶ app	--	Folder
▶ build	--	Folder
build.gradle	214 bytes	Document
▶ captures	--	Folder
FrequencyDomain.iml	960 bytes	Document
▶ gradle	--	Folder
gradle.properties	855 bytes	Conf Source
gradlew	5 KB	TextEd...ument
gradlew.bat	2 KB	Batch Source
Integrated_App_Android.iml	876 bytes	Document
▶ jni	--	Folder
local.properties	587 bytes	Conf Source
▶ main	--	Folder
settings.gradle	14 bytes	Document
▼ Integrated_App_iOS	--	Folder
▶ Integrated_App	--	Folder
Integrated_App.xcodeproj	↑ 808 KB	Xcode Project
▶ Integrated_AppTests	--	Folder
▶ Integrated_AppUITests.swift	--	Folder

Fig. 1: Integrated app folder contents

Part 1

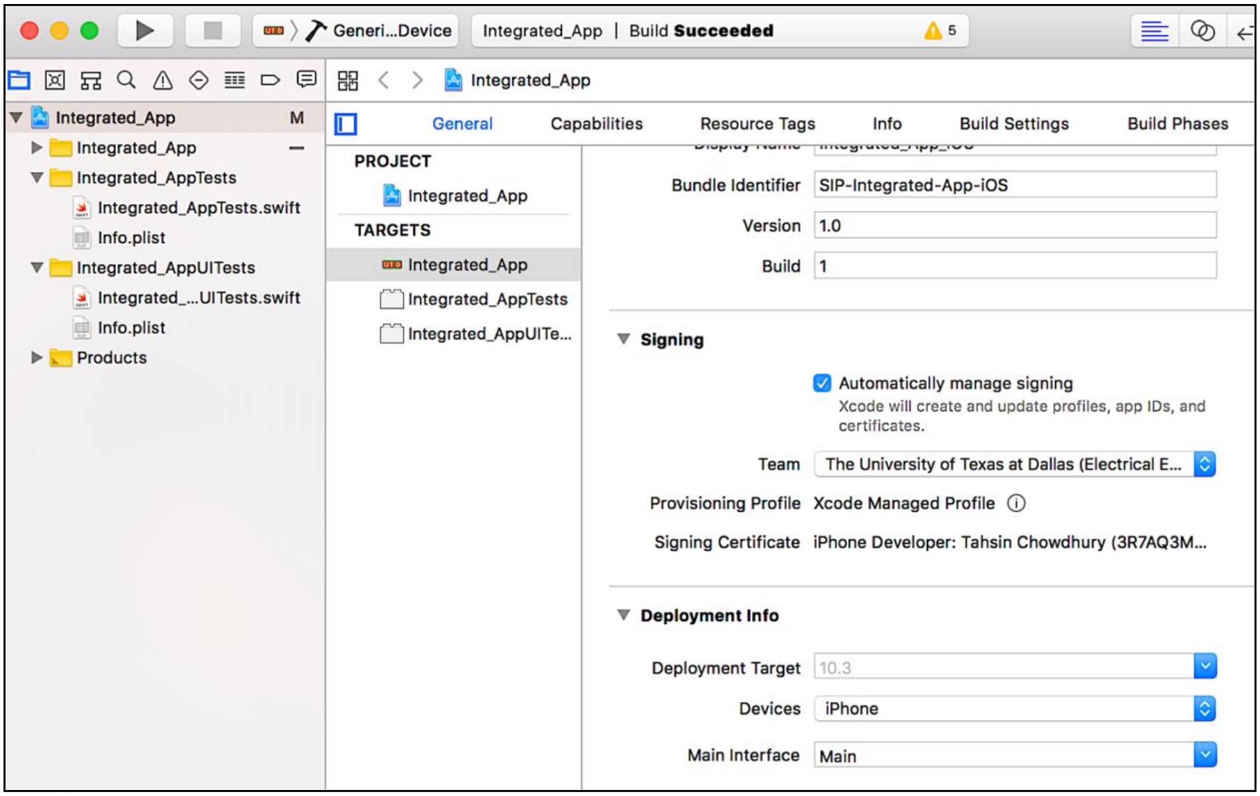
iOS

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 [6] for more details.

To open the app project, double click on the “Integrated_App.xcodeproj” in the “Integrated_App_iOS” folder, refer to Fig. 2.

To run the project, enable developer mode in iPhone if required (see [7]), 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. 2.



Bundle Identifier SIP-Integrated-App-iOS

Version 1.0

Build 1

Signing

☑ Automatically manage signing

Xcode will create and update profiles, app IDs, and certificates.

Team The University of Texas at Dallas (Electrical E...

Provisioning Profile Xcode Managed Profile ⓘ

Signing Certificate iPhone Developer: Tahsin Chowdhury (3R7AQ3M...

Deployment Info

Deployment Target 10.3

Devices iPhone

Main Interface Main

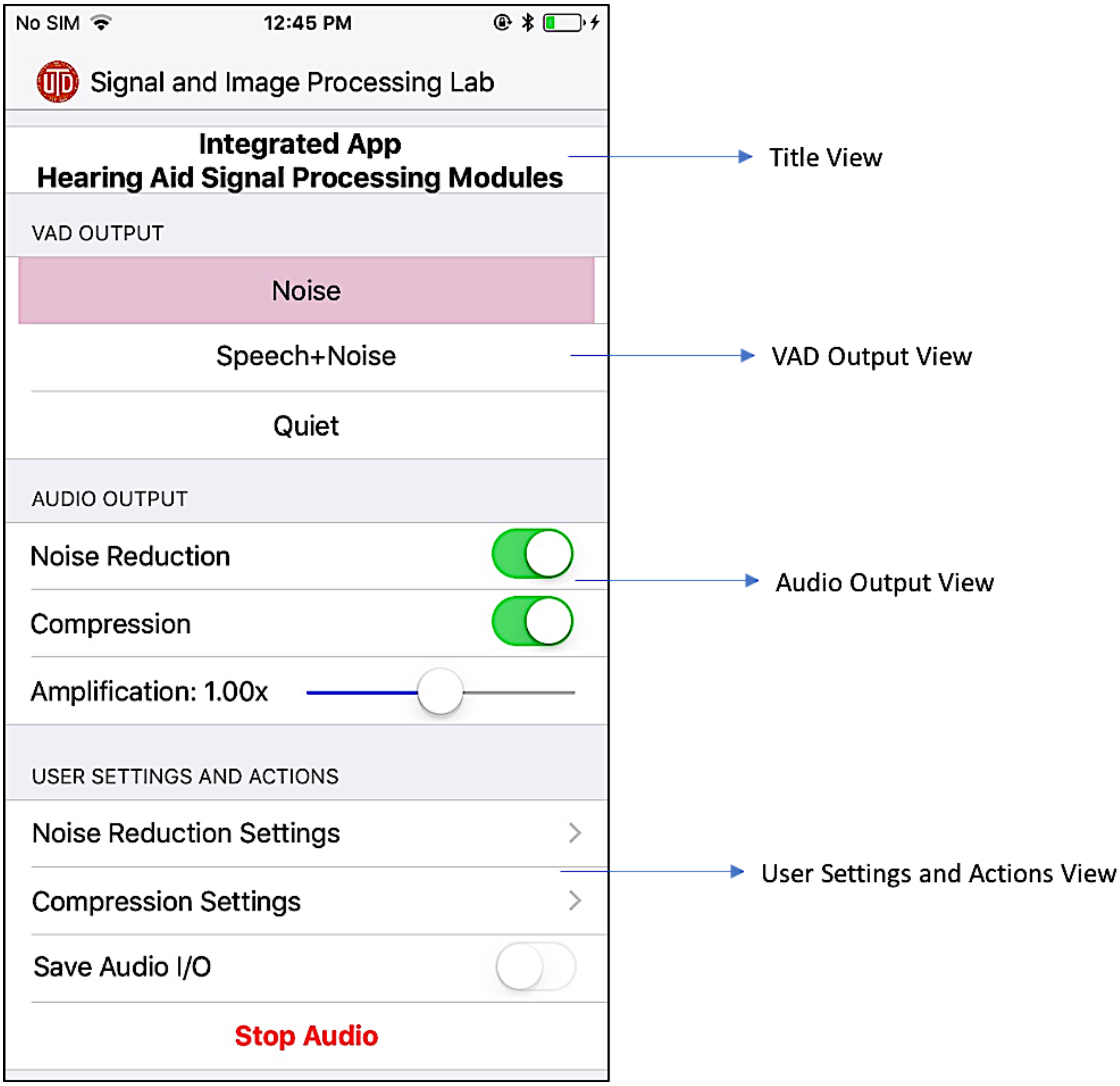
Fig. 2: Signing with Apple Developer ID

Section 2: iOS GUI

This section covers the GUI of the developed integrated app and its entries. The GUI consists of three views, see Fig. 3:

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

2.1 Main View



No SIM 12:45 PM

Signal and Image Processing Lab

Integrated App
Hearing Aid Signal Processing Modules

VAD OUTPUT

Noise

Speech+Noise

Quiet

AUDIO OUTPUT

Noise Reduction ●

Compression ●

Amplification: 1.00x

USER SETTINGS AND ACTIONS

Noise Reduction Settings >

Compression Settings >

Save Audio I/O ○

Stop Audio

Fig. 3: Integrated qpp iOS GUI: Main view

Main view consists of 4 segments, see Fig. 3.

- 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

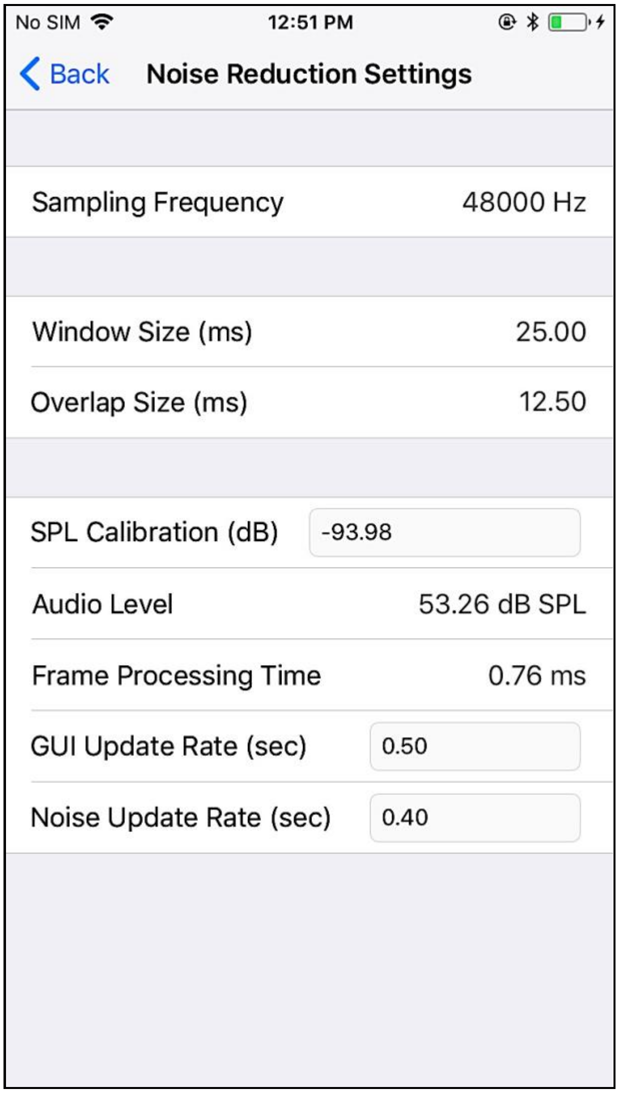
2.2 Noise Reduction Settings View

This view contains the settings for the noise reduction and audio processing. As shown in Fig. 4, the fields in this view are:

- **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 [1].
- **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.

- **<u>Overlap Size (ms):</u>** 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.



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

Fig. 4: Integrated app iOS GUI: Noise Reduction Settings view

- **<u>SPL Calibration (dB):</u>** 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.
- **<u>Audio Level:</u>** 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

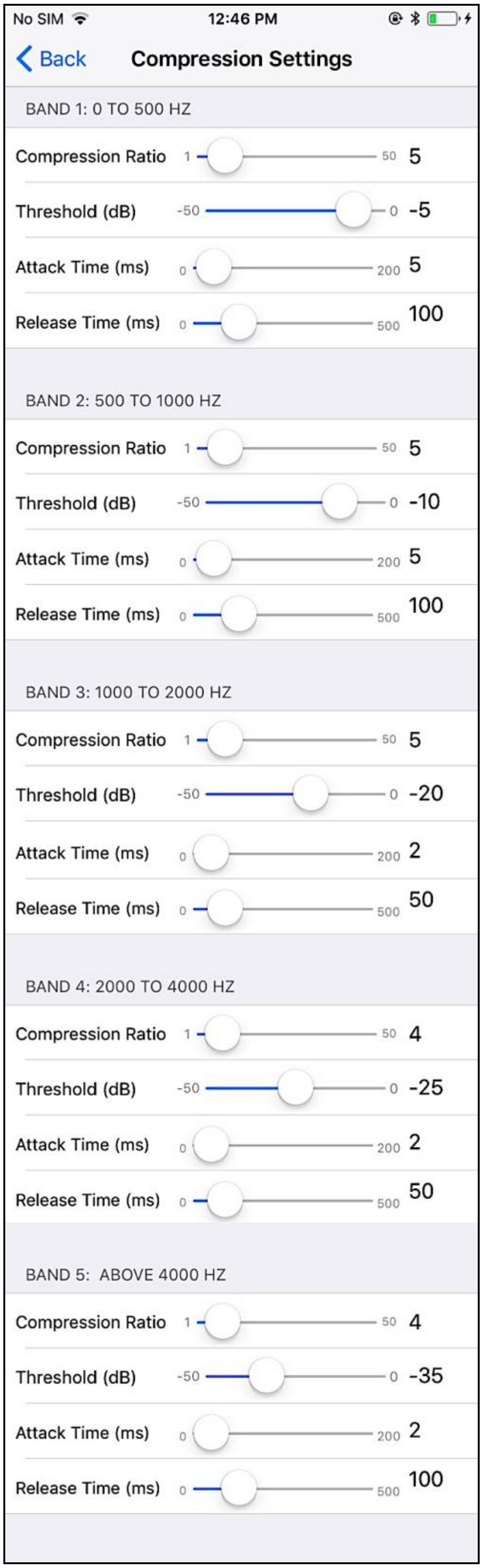
This view shows various settings for the five frequency bands of the compression module that the user can set, see Fig 5. These settings include:

- **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. 5.



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

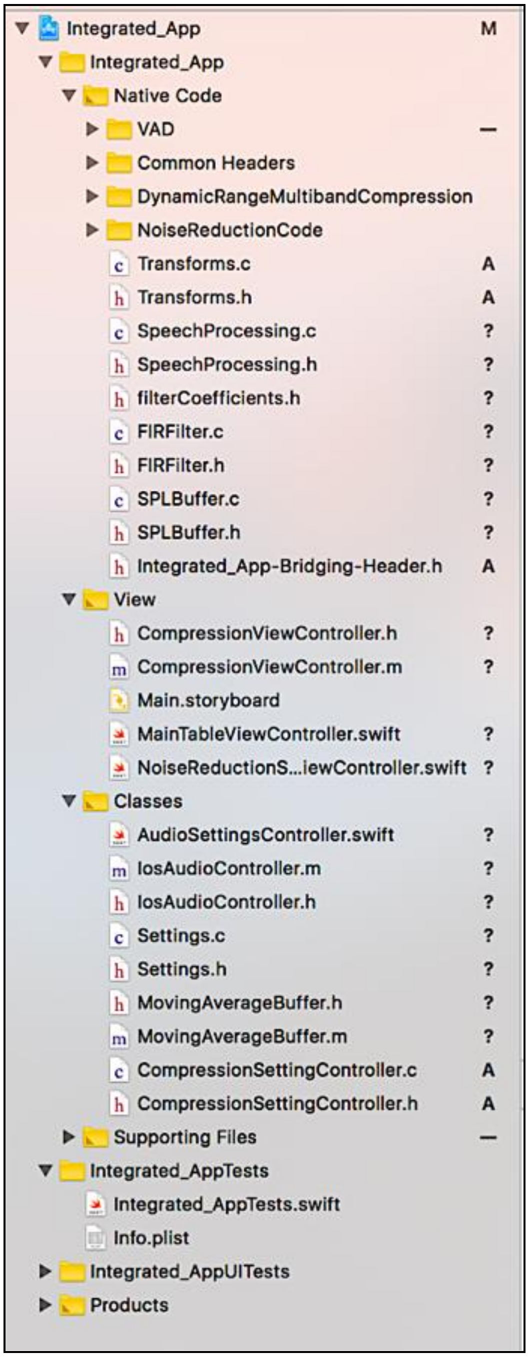
Attack Time (ms) 0 200 2

Release Time (ms) 0 500 100

Fig. 5: 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. 1. The code is divided into 3 parts, see Fig. 6:



```
▼ Integrated_App                                      M
  ▼ Integrated_App
    ▼ Native Code
      ▶ VAD                                           –
      ▶ Common Headers
      ▶ DynamicRangeMultibandCompression
      ▶ NoiseReductionCode
        Transforms.c                                  A
        Transforms.h                                  A
        SpeechProcessing.c                            ?
        SpeechProcessing.h                            ?
        filterCoefficients.h                          ?
        FIRFilter.c                                   ?
        FIRFilter.h                                   ?
        SPLBuffer.c                                   ?
        SPLBuffer.h                                   ?
        Integrated_App-Bridging-Header.h              A
    ▼ View
        CompressionViewController.h                   ?
        CompressionViewController.m                   ?
        Main.storyboard
        MainTableViewController.swift                 ?
        NoiseReductionS...iewController.swift         ?
    ▼ Classes
        AudioSettingsController.swift                 ?
        IosAudioController.m                          ?
        IosAudioController.h                          ?
        Settings.c                                    ?
        Settings.h                                    ?
        MovingAverageBuffer.h                         ?
        MovingAverageBuffer.m                         ?
        CompressionSettingController.c                A
        CompressionSettingController.h                A
    ▶ Supporting Files                                –
  ▼ Integrated_AppTests
      Integrated_AppTests.swift
      Info.plist
  ▶ Integrated_AppUITests
  ▶ Products
```

Fig. 6: Integrated app iOS code flow

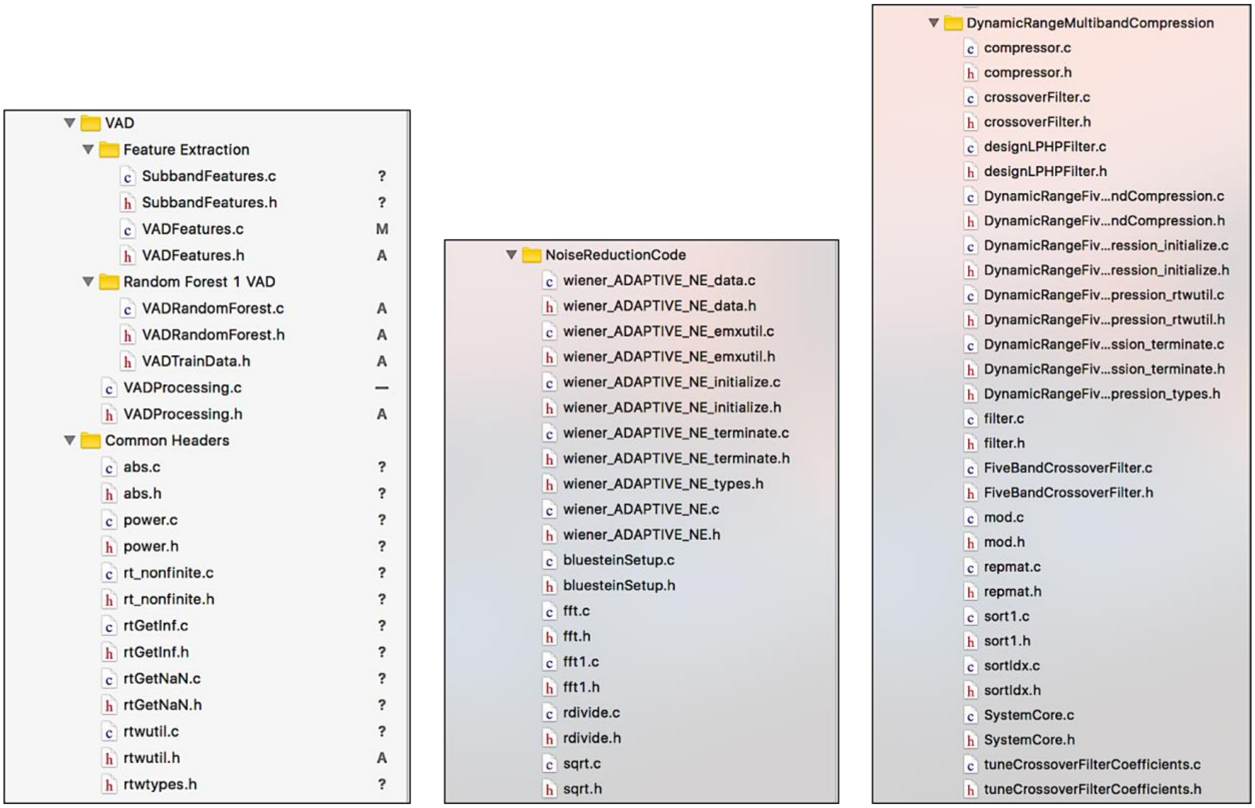
- Native Code
- View
- Classes

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 [1].
- **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 [8] 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 [9], which was initially developed in MATLAB and then converted into C using the MATLAB Coder [10].
- **DynamicRangeMultibandCompression:** This is the code for the compression module, which is also developed in MATLAB and then converted into C using the MATLAB Coder [10].
- **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. 7.



```
▼ VAD
  ▼ Feature Extraction
      SubbandFeatures.c        ?
      SubbandFeatures.h        ?
      VADFeatures.c            M
      VADFeatures.h            A
  ▼ Random Forest 1 VAD
      VADRandomForest.c        A
      VADRandomForest.h        A
      VADTrainData.h           A
    VADProcessing.c            —
    VADProcessing.h            A
▼ Common Headers
    abs.c                      ?
    abs.h                      ?
    power.c                    ?
    power.h                    ?
    rt_nonfinite.c             ?
    rt_nonfinite.h             ?
    rtGetInf.c                 ?
    rtGetInf.h                 ?
    rtGetNaN.c                 ?
    rtGetNaN.h                 ?
    rtwutil.c                  ?
    rtwutil.h                  A
    rtwtypes.h                 ?
```

```
▼ NoiseReductionCode
    wiener_ADAPTIVE_NE_data.c
    wiener_ADAPTIVE_NE_data.h
    wiener_ADAPTIVE_NE_emxutil.c
    wiener_ADAPTIVE_NE_emxutil.h
    wiener_ADAPTIVE_NE_initialize.c
    wiener_ADAPTIVE_NE_initialize.h
    wiener_ADAPTIVE_NE_terminate.c
    wiener_ADAPTIVE_NE_terminate.h
    wiener_ADAPTIVE_NE_types.h
    wiener_ADAPTIVE_NE.c
    wiener_ADAPTIVE_NE.h
    bluesteinSetup.c
    bluesteinSetup.h
    fft.c
    fft.h
    fft1.c
    fft1.h
    rdivide.c
    rdivide.h
    sqrt.c
    sqrt.h
```

```
▼ DynamicRangeMultibandCompression
    compressor.c
    compressor.h
    crossoverFilter.c
    crossoverFilter.h
    designLPHPFilter.c
    designLPHPFilter.h
    DynamicRangeFiv…ndCompression.c
    DynamicRangeFiv…ndCompression.h
    DynamicRangeFiv…ression_initialize.c
    DynamicRangeFiv…ression_initialize.h
    DynamicRangeFiv…pression_rtwutil.c
    DynamicRangeFiv…pression_rtwutil.h
    DynamicRangeFiv…ssion_terminate.c
    DynamicRangeFiv…ssion_terminate.h
    DynamicRangeFiv…pression_types.h
    filter.c
    filter.h
    FiveBandCrossoverFilter.c
    FiveBandCrossoverFilter.h
    mod.c
    mod.h
    repmat.c
    repmat.h
    sort1.c
    sort1.h
    sortIdx.c
    sortIdx.h
    SystemCore.c
    SystemCore.h
    tuneCrossoverFilterCoefficients.c
    tuneCrossoverFilterCoefficients.h
```

Fig. 7: 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 [11].
- **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.

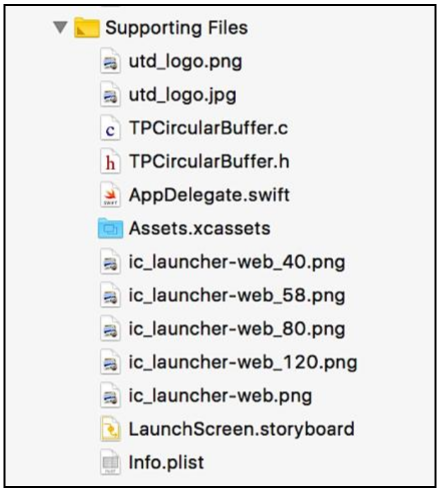
This is written in Swift.

- **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. 8. Along with the app logo and launcher images, there are:

- **TPCircularBuffer:** This is an implementation of the circular buffer in [12], 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.



```
▼ Supporting Files
    utd_logo.png
    utd_logo.jpg
    TPCircularBuffer.c
    TPCircularBuffer.h
    AppDelegate.swift
    Assets.xcassets
    ic_launcher-web_40.png
    ic_launcher-web_58.png
    ic_launcher-web_80.png
    ic_launcher-web_120.png
    ic_launcher-web.png
    LaunchScreen.storyboard
    Info.plist
```

Fig. 8: Supporting files

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 = 1.33ms.

- **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.

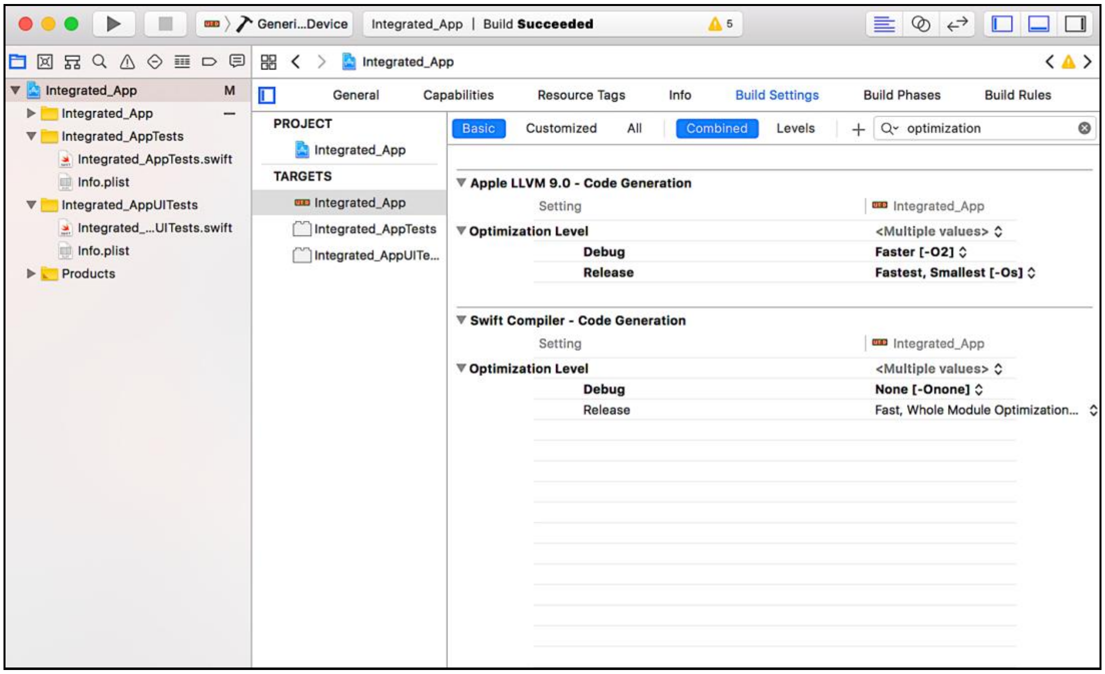
"MainTableViewController" takes this decision from the setting variable. Since "MainTableViewController" is written in Swift, the access to Settings is bridged through "Integrated_App-Bridging-Header.h" which is included inside the native code folder (refer to Fig 6). To train the VAD module of the app, use the VAD app as described in [8] and available at [13]. 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->playAudo" 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 [14] and [15]. The user can change them according to specific requirements. To set these flags, follow the steps noted below, see Fig. 9.
    - 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.



Apple LLVM 9.0 - Code Generation

Setting	Integrated_App
Optimization Level	<Multiple values>
Debug	Faster [-O2]
Release	Fastest, Smallest [-Os]

Swift Compiler - Code Generation

Setting	Integrated_App
Optimization Level	<Multiple values>
Debug	None [-Onone]
Release	Fast, Whole Module Optimization...

Fig. 9: 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 [16]. 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.

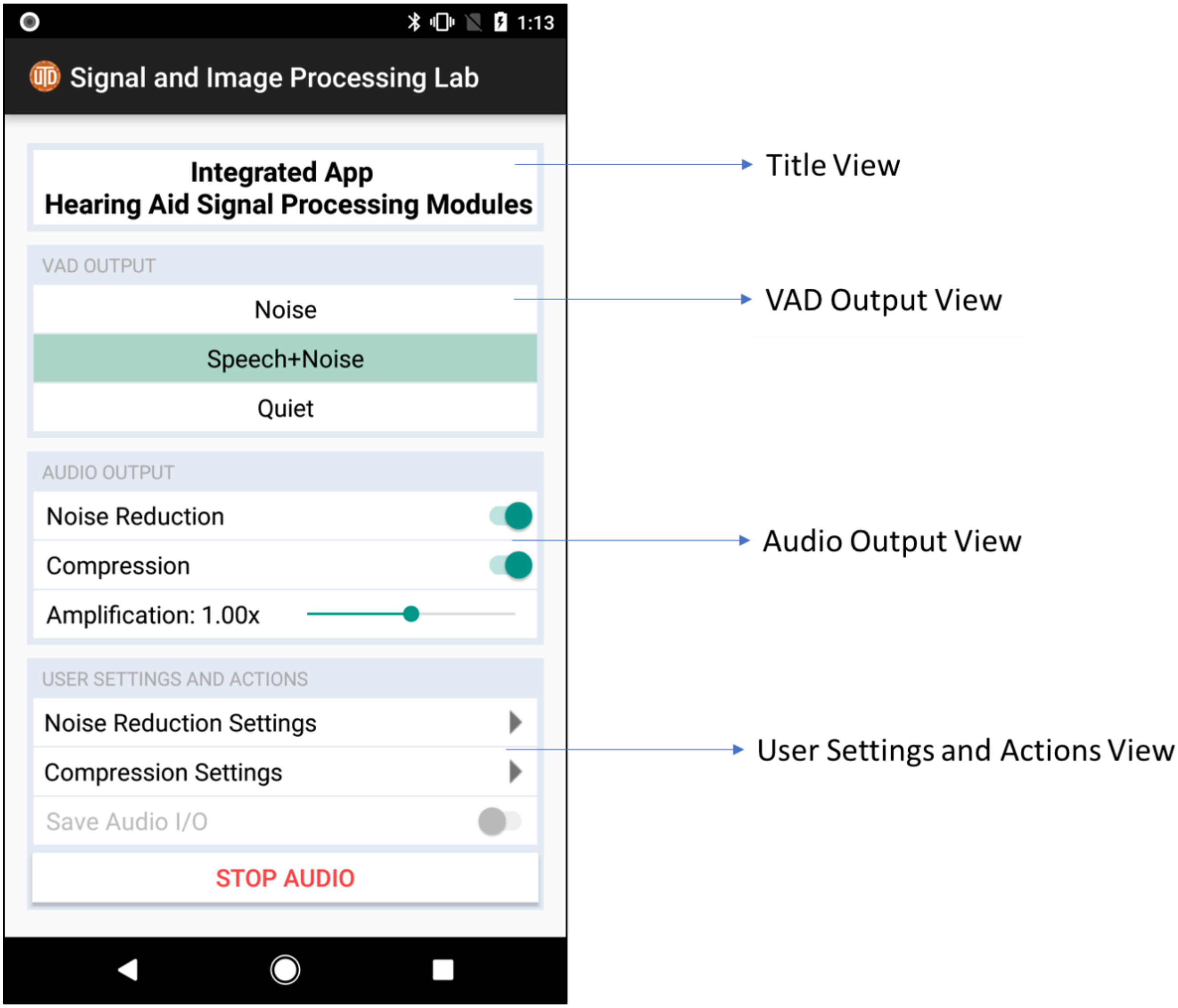
More details of the above steps are covered in [9].

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



1:13

Signal and Image Processing Lab

Integrated App
Hearing Aid Signal Processing Modules

VAD OUTPUT

Noise

Speech+Noise

Quiet

AUDIO OUTPUT

Noise Reduction ●

Compression ●

Amplification: 1.00x

USER SETTINGS AND ACTIONS

Noise Reduction Settings ▸

Compression Settings ▸

Save Audio I/O ○

STOP AUDIO

Fig. 10: Integrated app Android GUI: Main view

Main view consists of 4 views, see Fig. 10:

- 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

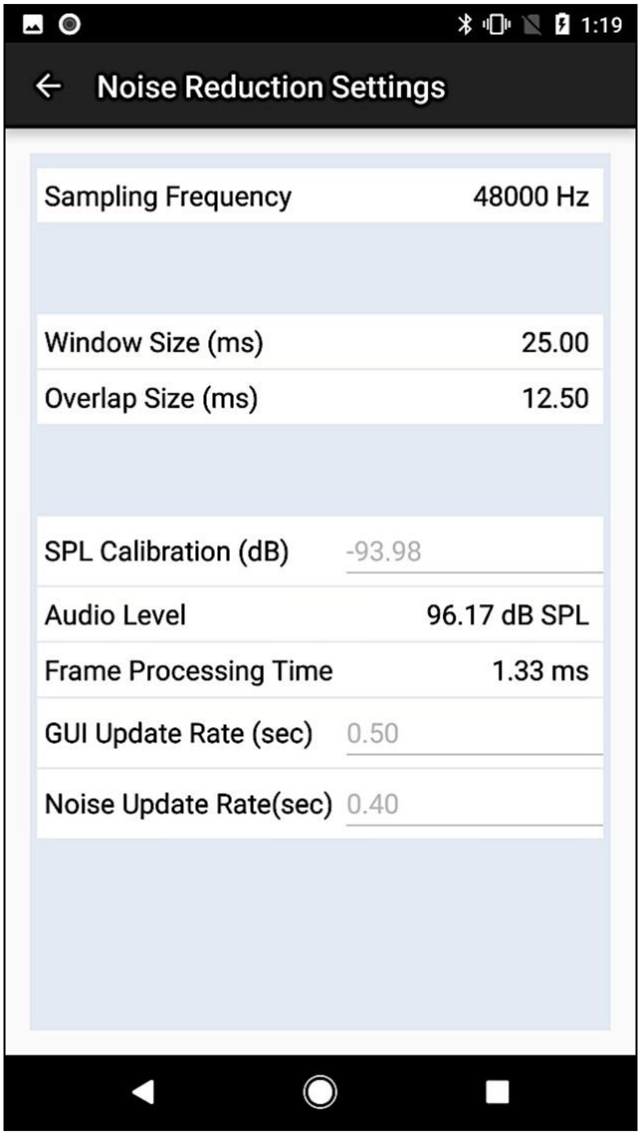
2.2 Noise Reduction Settings View

This view contains various settings for the noise reduction and audio processing. As shown in Fig. 11, the items of this settings view are:

- **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.



Noise Reduction Settings

Sampling Frequency 48000 Hz

Window Size (ms) 25.00

Overlap Size (ms) 12.50

SPL Calibration (dB) -93.98

Audio Level 96.17 dB SPL

Frame Processing Time 1.33 ms

GUI Update Rate (sec) 0.50

Noise Update Rate(sec) 0.40

Fig. 11: Integrated app Android GUI: Noise Reduction Settings view

- **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

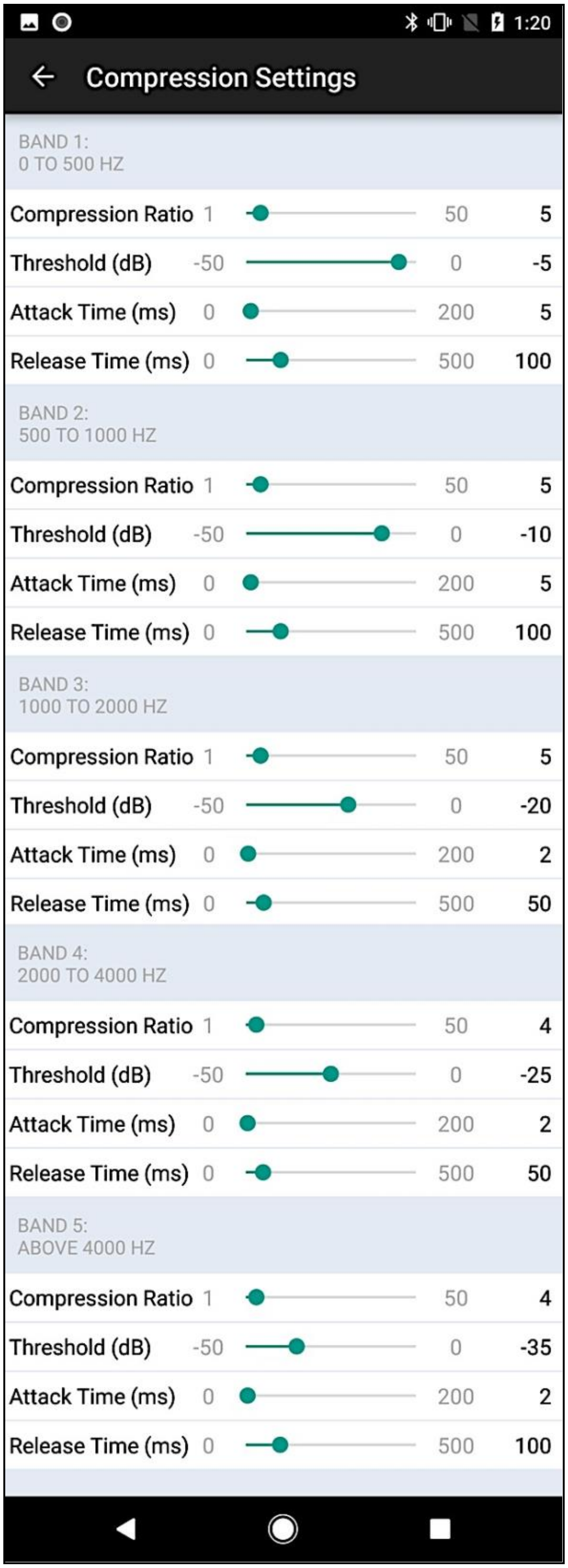
This view shows various settings for the compression module for the five frequency bands, see Fig 12. These settings include:

- **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. 12.



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

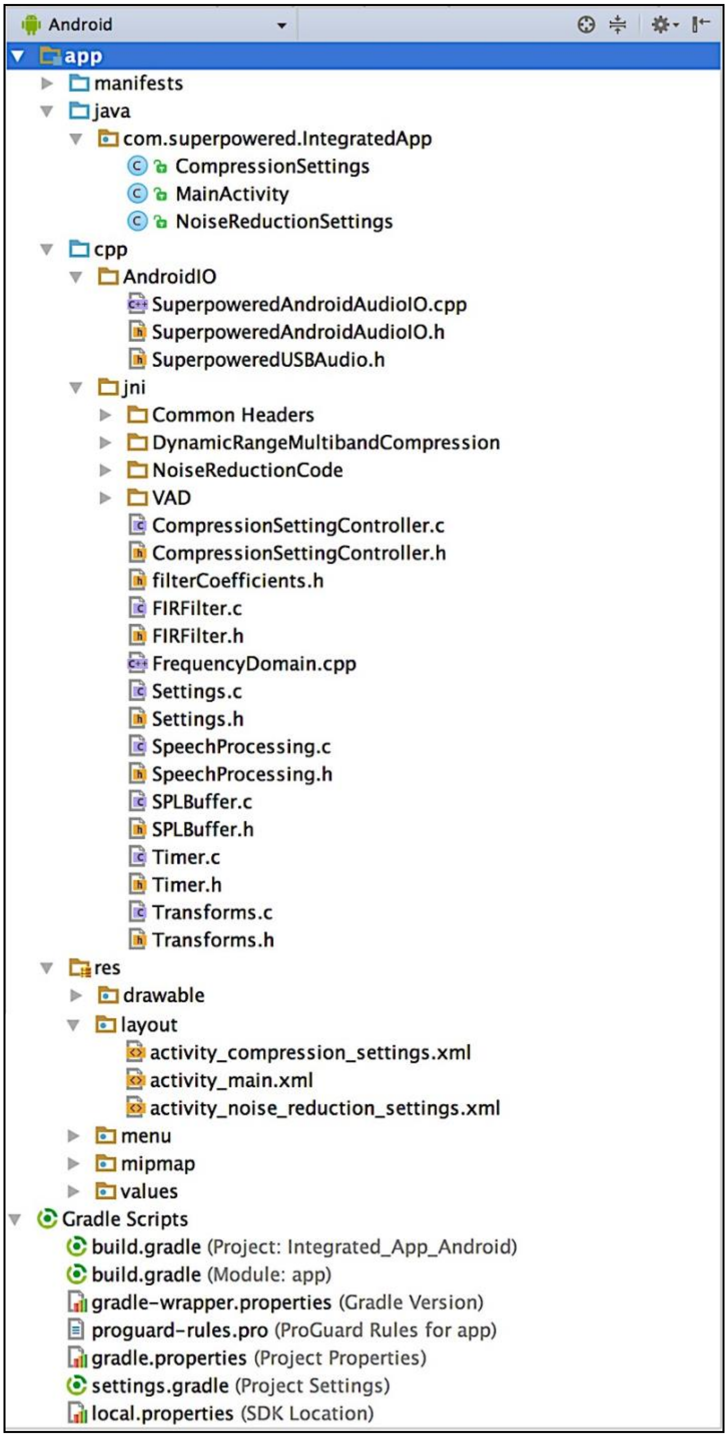
BAND 5:
ABOVE 4000 HZ

Compression Ratio	1	50	4
Threshold (dB)	-50	0	-35
Attack Time (ms)	0	200	2
Release Time (ms)	0	500	100

Fig. 12: 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. 13.



```
Android
▼ app
  ▶ manifests
  ▼ java
    ▼ com.superpowered.IntegratedApp
        CompressionSettings
        MainActivity
        NoiseReductionSettings
  ▼ cpp
    ▼ AndroidIO
        SuperpoweredAndroidAudioIO.cpp
        SuperpoweredAndroidAudioIO.h
        SuperpoweredUSBAudio.h
    ▼ jni
      ▶ Common Headers
      ▶ DynamicRangeMultibandCompression
      ▶ NoiseReductionCode
      ▶ VAD
        CompressionSettingController.c
        CompressionSettingController.h
        filterCoefficients.h
        FIRFilter.c
        FIRFilter.h
        FrequencyDomain.cpp
        Settings.c
        Settings.h
        SpeechProcessing.c
        SpeechProcessing.h
        SPLBuffer.c
        SPLBuffer.h
        Timer.c
        Timer.h
        Transforms.c
        Transforms.h
  ▼ res
    ▶ drawable
    ▼ layout
        activity_compression_settings.xml
        activity_main.xml
        activity_noise_reduction_settings.xml
    ▶ menu
    ▶ mipmap
    ▶ values
▼ Gradle Scripts
    build.gradle (Project: Integrated_App_Android)
    build.gradle (Module: app)
    gradle-wrapper.properties (Gradle Version)
    proguard-rules.pro (ProGuard Rules for app)
    gradle.properties (Project Properties)
    settings.gradle (Project Settings)
    local.properties (SDK Location)
```

Fig. 13: 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.

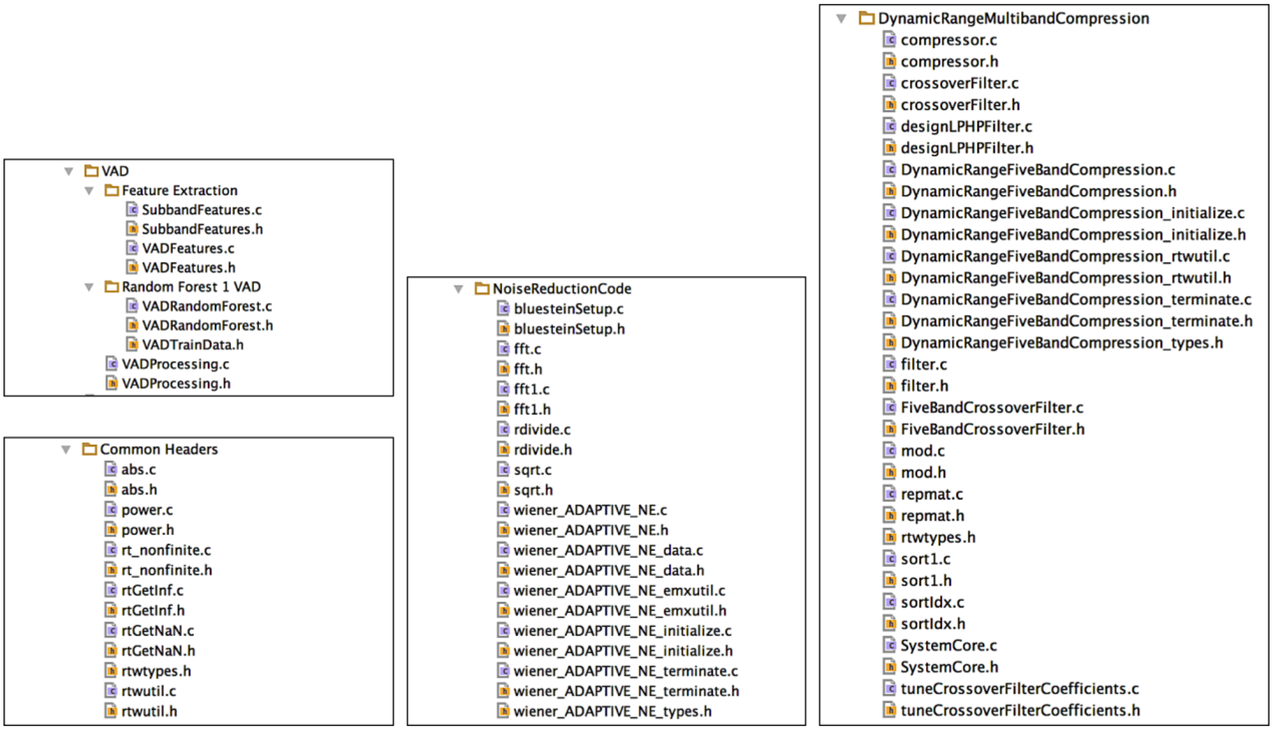
3.1 Native Code and Settings

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 [1].

- **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 [8] 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 [9] for the noise reduction module, which is developed in MATLAB and then converted into C using the MATLAB Coder [10].
- **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 [10].
- **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. 14.



```
▼ VAD
    ▼ Feature Extraction
        SubbandFeatures.c
        SubbandFeatures.h
        VADFeatures.c
        VADFeatures.h
    ▼ Random Forest 1 VAD
        VADRandomForest.c
        VADRandomForest.h
        VADTrainData.h
    VADProcessing.c
    VADProcessing.h
```

```
▼ Common Headers
    abs.c
    abs.h
    power.c
    power.h
    rt_nonfinite.c
    rt_nonfinite.h
    rtGetInf.c
    rtGetInf.h
    rtGetNaN.c
    rtGetNaN.h
    rtwtypes.h
    rtwutil.c
    rtwutil.h
```

```
▼ NoiseReductionCode
    bluesteinSetup.c
    bluesteinSetup.h
    fft.c
    fft.h
    fft1.c
    fft1.h
    rdivide.c
    rdivide.h
    sqrt.c
    sqrt.h
    wiener_ADAPTIVE_NE.c
    wiener_ADAPTIVE_NE.h
    wiener_ADAPTIVE_NE_data.c
    wiener_ADAPTIVE_NE_data.h
    wiener_ADAPTIVE_NE_emxutil.c
    wiener_ADAPTIVE_NE_emxutil.h
    wiener_ADAPTIVE_NE_initialize.c
    wiener_ADAPTIVE_NE_initialize.h
    wiener_ADAPTIVE_NE_terminate.c
    wiener_ADAPTIVE_NE_terminate.h
    wiener_ADAPTIVE_NE_types.h
```

```
▼ DynamicRangeMultibandCompression
    compressor.c
    compressor.h
    crossoverFilter.c
    crossoverFilter.h
    designLPHPFilter.c
    designLPHPFilter.h
    DynamicRangeFiveBandCompression.c
    DynamicRangeFiveBandCompression.h
    DynamicRangeFiveBandCompression_initialize.c
    DynamicRangeFiveBandCompression_initialize.h
    DynamicRangeFiveBandCompression_rtwutil.c
    DynamicRangeFiveBandCompression_rtwutil.h
    DynamicRangeFiveBandCompression_terminate.c
    DynamicRangeFiveBandCompression_terminate.h
    DynamicRangeFiveBandCompression_types.h
    filter.c
    filter.h
    FiveBandCrossoverFilter.c
    FiveBandCrossoverFilter.h
    mod.c
    mod.h
    repmat.c
    repmat.h
    rtwtypes.h
    sort1.c
    sort1.h
    sortIdx.c
    sortIdx.h
    SystemCore.c
    SystemCore.h
    tuneCrossoverFilterCoefficients.c
    tuneCrossoverFilterCoefficients.h
```

Fig. 14: 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.

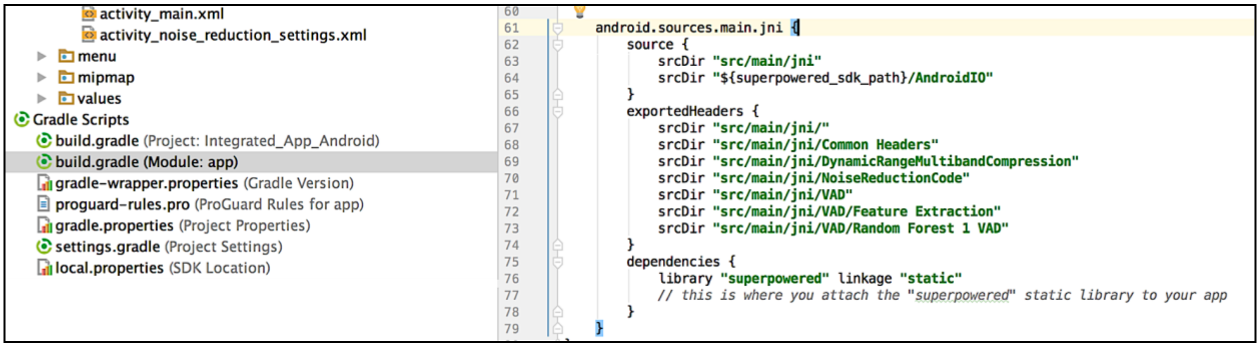
- **Available bandwidth:** The app is developed in such a way that the hearing aid modules used (i.e., VAD, Noise Reduction and Compression) can be replaced with other similar modules. It is also possible to add new modules if the available processing time bandwidth is not exceeded. The processing time 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 = 4ms. 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. 15. It is required to add the path of each folder and also its subfolders separately.


activity_main.xml
activity_noise_reduction_settings.xml
menu
mipmap
values
Gradle Scripts
build.gradle (Project: Integrated_App_Android)
build.gradle (Module: app)
gradle-wrapper.properties (Gradle Version)
proguard-rules.pro (ProGuard Rules for app)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)

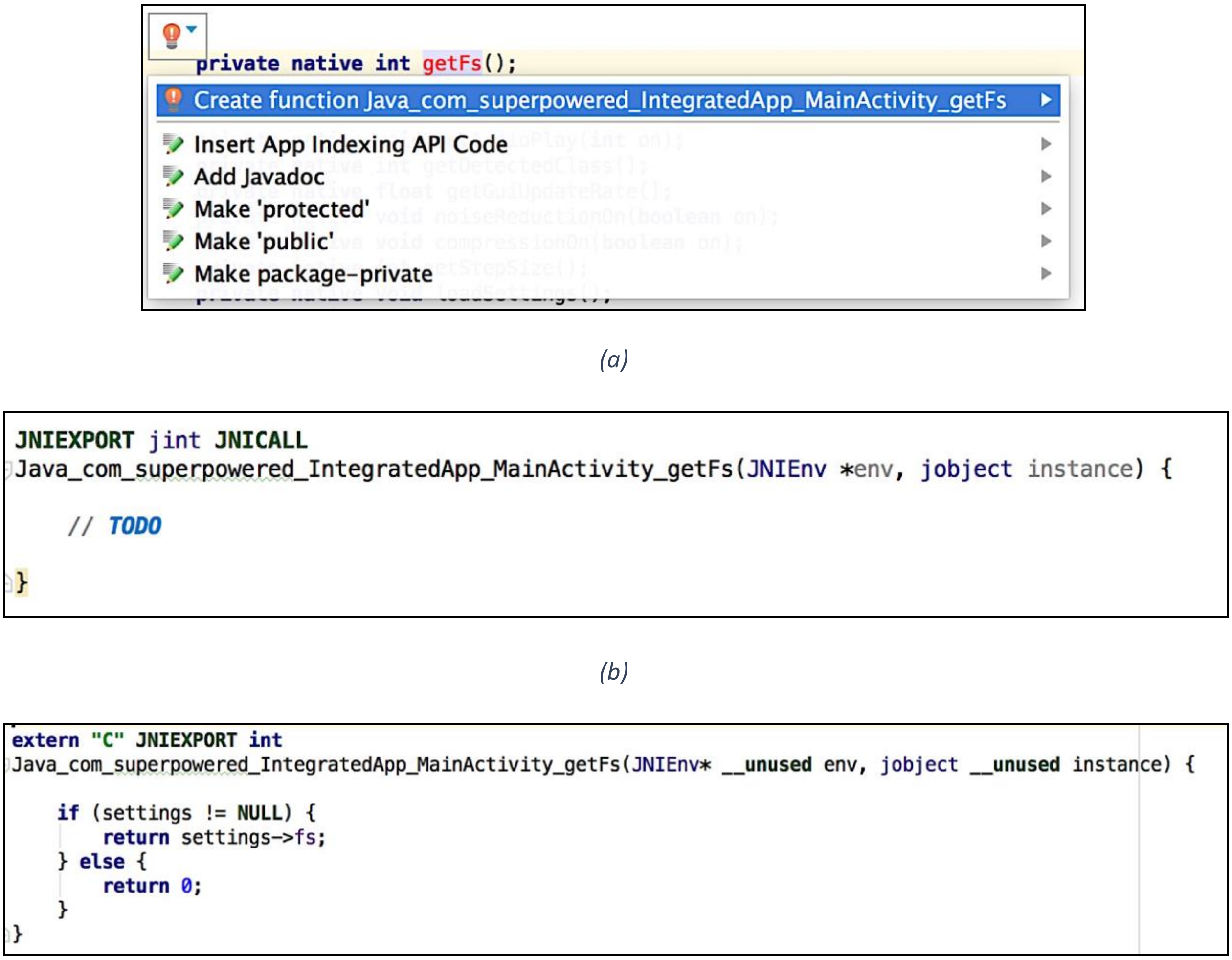
```
android.sources.main.jni {
    source {
        srcDir "src/main/jni"
        srcDir "${superpowered_sdk_path}/AndroidIO"
    }
    exportedHeaders {
        srcDir "src/main/jni/"
        srcDir "src/main/jni/Common Headers"
        srcDir "src/main/jni/DynamicRangeMultibandCompression"
        srcDir "src/main/jni/NoiseReductionCode"
        srcDir "src/main/jni/VAD"
        srcDir "src/main/jni/VAD/Feature Extraction"
        srcDir "src/main/jni/VAD/Random Forest 1 VAD"
    }
    dependencies {
        library "superpowered" linkage "static"
        // this is where you attach the "superpowered" static library to your app
    }
}
```

Fig. 15: Add native folder paths

- **<u>Data transaction between the GUI and Native Code:</u>** 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 [17-19]. 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. 16a.

- Click on the create option. It will create a function in FrequencyDomain.cpp, see Fig. 16b.
- Modify the function definition using extern “C” keyword, see Fig. 16c, since it is calling the “setting->fs” variable from a C file.



```
private native int getFs();
```

Create function Java_com_superpowered_IntegratedApp_MainActivity_getFs
Insert App Indexing API Code
Add Javadoc
Make 'protected'
Make 'public'
Make package-private

(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)

Fig. 16: Creating native linking function

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.

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 [8] which is available at [13]. Make sure to use the app with a proper window size and sampling frequency (decimated sampling frequency).

- **<u>AudioOutput controls:</u>**
  - "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->playAudo" 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. 16. 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.

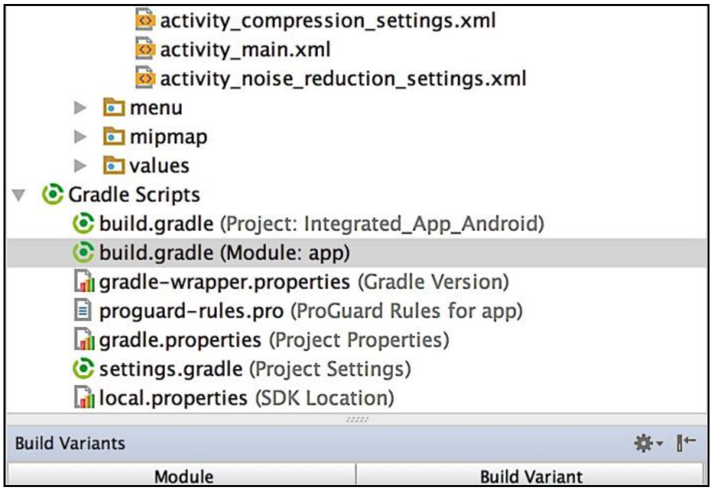
- **<u>Compression Controls:</u>** "CompressionSettingsController" provides a separate setting parameters for the compression module that are passed between the native code and the "CompressionSettings" activity. The compression parameters set by the user are saved in the "dataIn" array variable and 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. 16.

- **<u>Compiler optimization:</u>** 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.

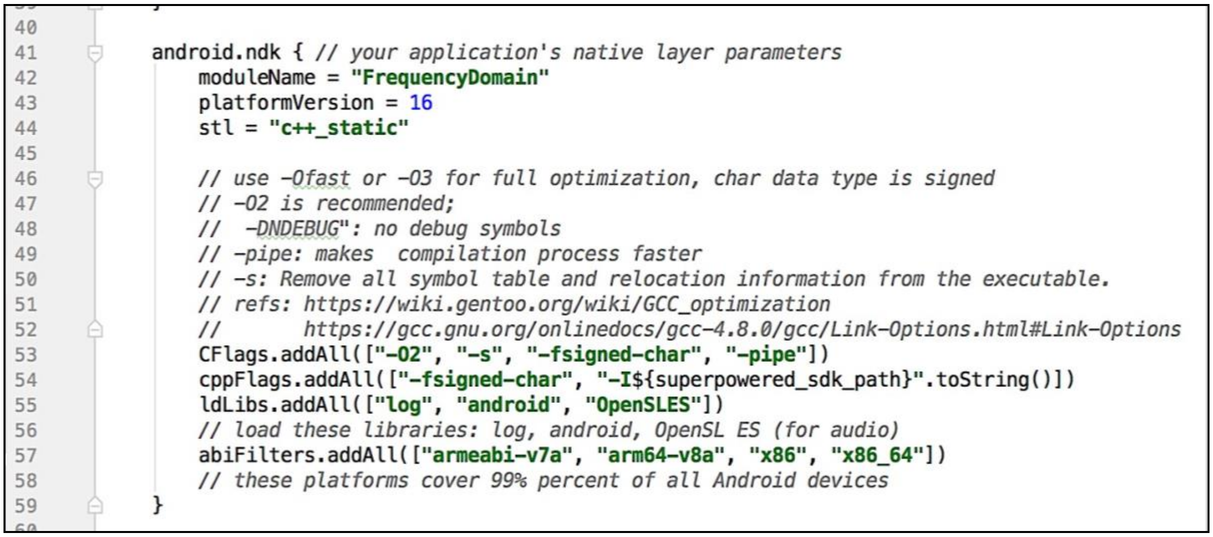
The details of the above are given in [14] and [15]. The user can change them according to specific requirements. To set these flags, follow the steps as noted in Fig. 17.

- Under "Android" of the project view explorer of Android Studio, select build.gradle (Module: app) under Gradle Scripts, see left side of Fig. 17.
- At the right side of Fig. 17, set the optimization flags under
  - model → android.ndk → CFlags.addAll();



activity_compression_settings.xml
activity_main.xml
activity_noise_reduction_settings.xml
▶ menu
▶ mipmap
▶ values
▼ Gradle Scripts
build.gradle (Project: Integrated_App_Android)
build.gradle (Module: app)
gradle-wrapper.properties (Gradle Version)
proguard-rules.pro (ProGuard Rules for app)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)

(left)



```
    android.ndk { // your application's native layer parameters
        moduleName = "FrequencyDomain"
        platformVersion = 16
        stl = "c++_static"

        // use -Ofast or -O3 for full optimization, char data type is signed
        // -O2 is recommended;
        //  -DNDEBUG": no debug symbols
        // -pipe: makes  compilation process faster
        // -s: Remove all symbol table and relocation information from the executable.
        // refs: https://wiki.gentoo.org/wiki/GCC_optimization
        //       https://gcc.gnu.org/onlinedocs/gcc-4.8.0/gcc/Link-Options.html#Link-Options
        CFlags.addAll(["-O2", "-s", "-fsigned-char", "-pipe"])
        cppFlags.addAll(["-fsigned-char", "-I${superpowered_sdk_path}".toString()])
        ldLibs.addAll(["log", "android", "OpenSLES"])
        // load these libraries: log, android, OpenSL ES (for audio)
        abiFilters.addAll(["armeabi-v7a", "arm64-v8a", "x86", "x86_64"])
        // these platforms cover 99% percent of all Android devices
    }
```

(right)

Fig. 17: Setting optimization level in Android Studio for the integrated app Android

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 [20] and for Android using the software package Superpowered SDK [16]. 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 1: Timing difference between the iOS and Android versions of the integrated app

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

Table 1 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 [21] hearing aid.

References:

[1] T. A. Chowdhury, A. Sehgal and N. Kehtarnavaz, "Integrating Signal Processing Modules of Hearing Aids into a Real-Time Smartphone App," 2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), Honolulu, HI, 2018, pp. 2837-2840.

[2] 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.

[3] 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.

[4] https://www.mathworks.com/help/audio/examples/multiband-dynamic-range-compression.html

[5] https://www.mathworks.com/help/audio/examples/multiband-dynamic-range-compression-for-ios-devices.html

[6] https://developer.apple.com/xcode/

[7] http://codewithchris.com/deploy-your-app-on-an-iphone/

[8] http://www.utdallas.edu/ssprl/files/Users-Guide-VAD.pdf

[9] http://www.utdallas.edu/ssprl/files/Users-Guide-Noise-Reduction.pdf

[10] https://www.mathworks.com/products/matlab-coder.html

[11] https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/index.html

[12] https://github.com/michaeltyson/TPCircularBuffer

[13] https://utdallas.app.box.com/v/SIP-NP-VAD1

[14] https://wiki.gentoo.org/wiki/GCC_optimization

[15] https://gcc.gnu.org/onlinedocs/gcc-4.8.0/gcc/Link-Options.html#Link-Options

[16] http://superpowered.com/

[17] N. Kehtarnavaz and F. Saki, *Anywhere-Anytime Signals and Systems Laboratory: From MATLAB to Smartphones*, Morgan and Claypool Publishers, 2016.

[18] N. Kehtarnavaz, S. Parris, A. Sehgal, *Smartphone-Based Real-Time Digital Signal Processing*, Morgan and Claypool Publishers, 2015.

[19] http://www.utdallas.edu/ssprl/files/Users-Guide-Android.pdf

[20] https://developer.apple.com/documentation/coreaudio

[21] https://www.starkey.com/hearing-aids/technologies/halo-2-made-for-iphone-hearing-aids