User's Guide:
A Smartphone App Integrating Personalized Noise Reduction and Compression for Hearing Enhancement

N. ALAMDARI, S. BHARATHI, T. CHOWDHURY, A. SEHGAL, AND N. KEHTARNAVAZ

SIGNAL AND IMAGE PROCESSING LAB

UNIVERSITY OF TEXAS AT DALLAS

MARCH 2019

TABLE OF CONTENTS

TABLE OF FIGURES

Introduction

This user's guide covers how to use a smartphone app implementing a personalized noise reduction (NR) that is designed to run in real-time with low-latency on smartphone platforms for hearing enhancement purposes as discussed in [1]. The personalization is achieved by adding an unsupervised noise classifier [2] together with a personalized gain adjustment to the integrated app that is discussed in [3]. After applying a Wiener filtering noise reduction [4], gains in five frequency bands can be adjusted by the user to achieve personalized noise reduction depending on the noise environment identified by the classifier. The other signal processing modules of the app include voice activity detection [5] and compression [6].

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 personalized NR app

- iPhone8 as iOS platform
- Google Pixel as Android platform

Personalized NR 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:

- "PersonalizedNR_App_Android" denoting the Android Studio project
- "PersonalizedNR_App_iOS" denoting the iOS Xcode project

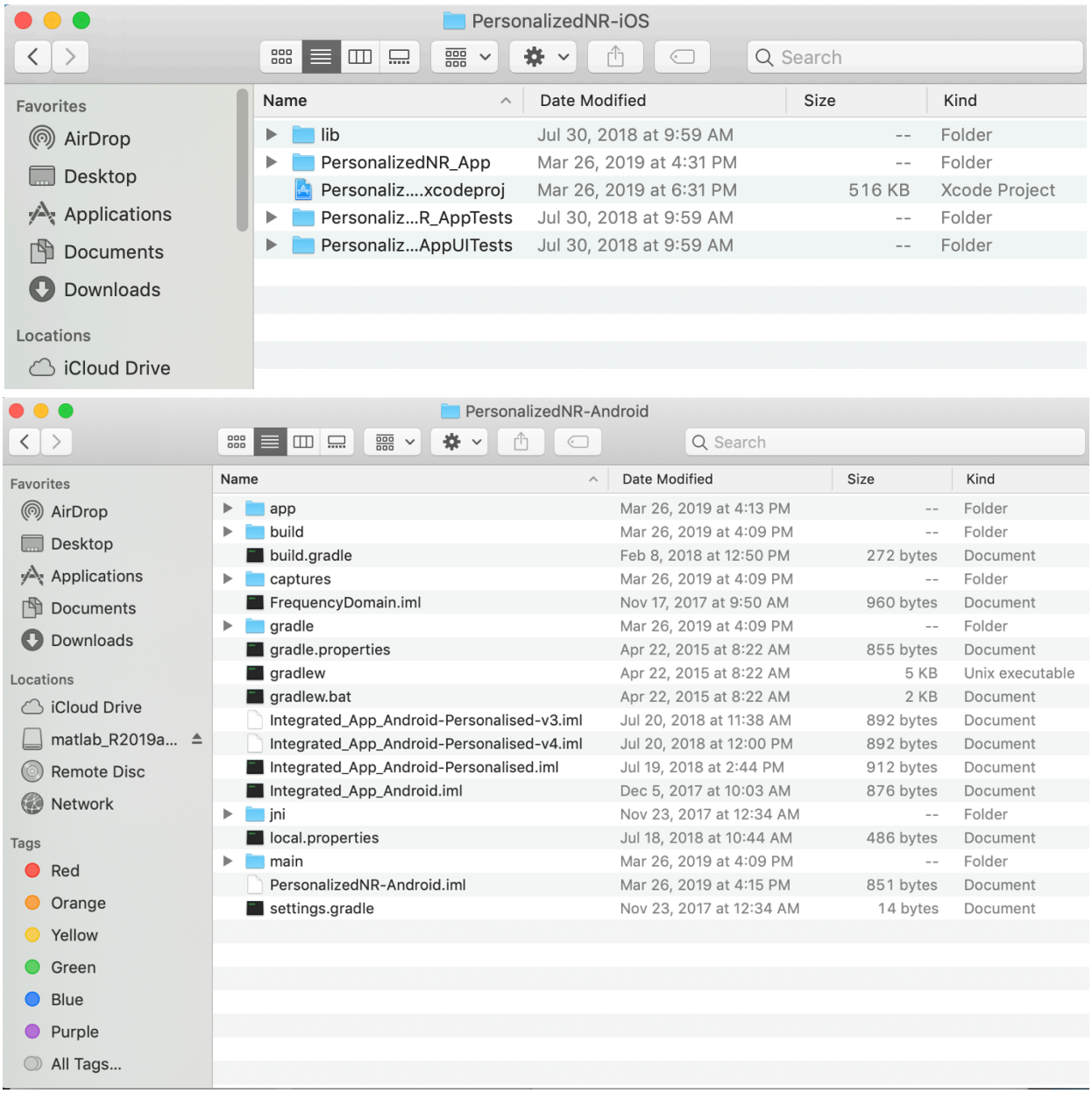


Fig. 1: Personalized NR app folder contents

Part 1

iOS

Section 1: Running the App

The iOS version of the personalized NR 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 [7] for more details.

To open the app project, double click on the "PersonalizedNR_App.xcodeproj" in the "PersonalizedNR_App_iOS" folder, refer to Fig. 2.

To run the project, enable developer mode in iPhone if required (see [8]), connect iPhone to a MAC computer using a USB cable, and then run the app by Command+R or simply click on the run button. The app gets installed. Make sure the Xcode is signed into using your Apple Developer ID, refer to Fig. 2.

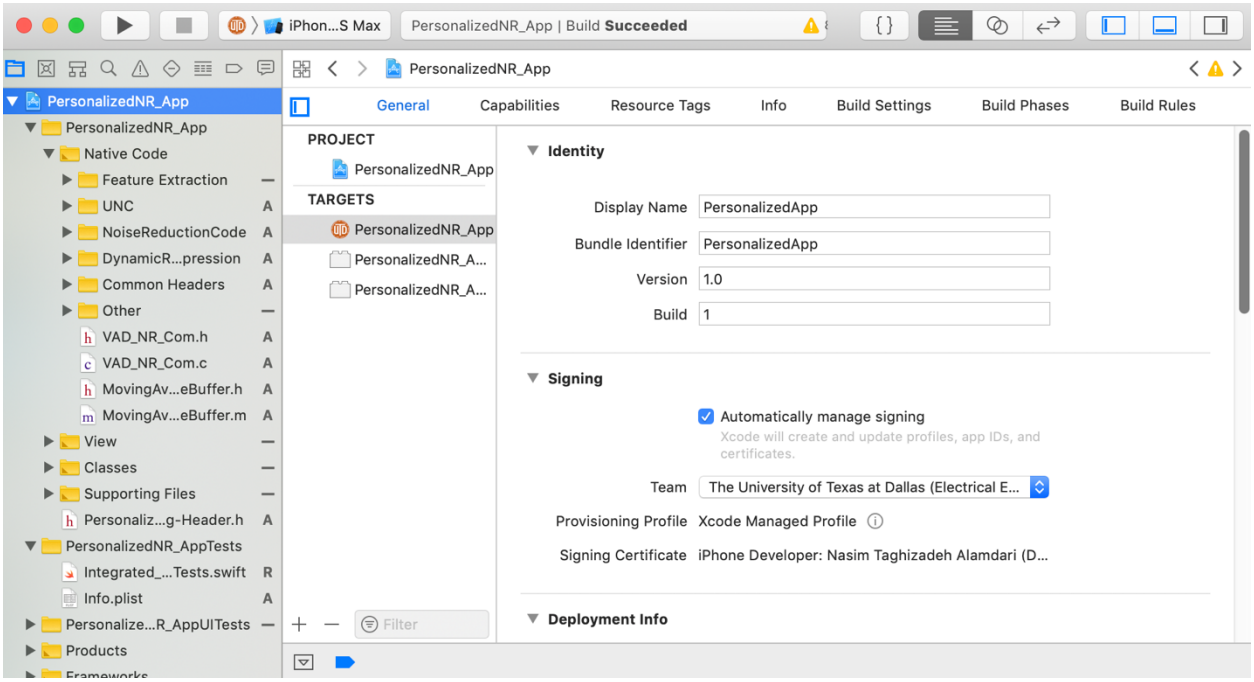


Fig. 2: Signing with Apple Developer ID

Section 2: iOS GUI

This section covers the GUI of the developed personalized NR app and its entries. The Graphical User Interface (GUI) consists of three views, see Fig. 3:

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

2.1 Main View

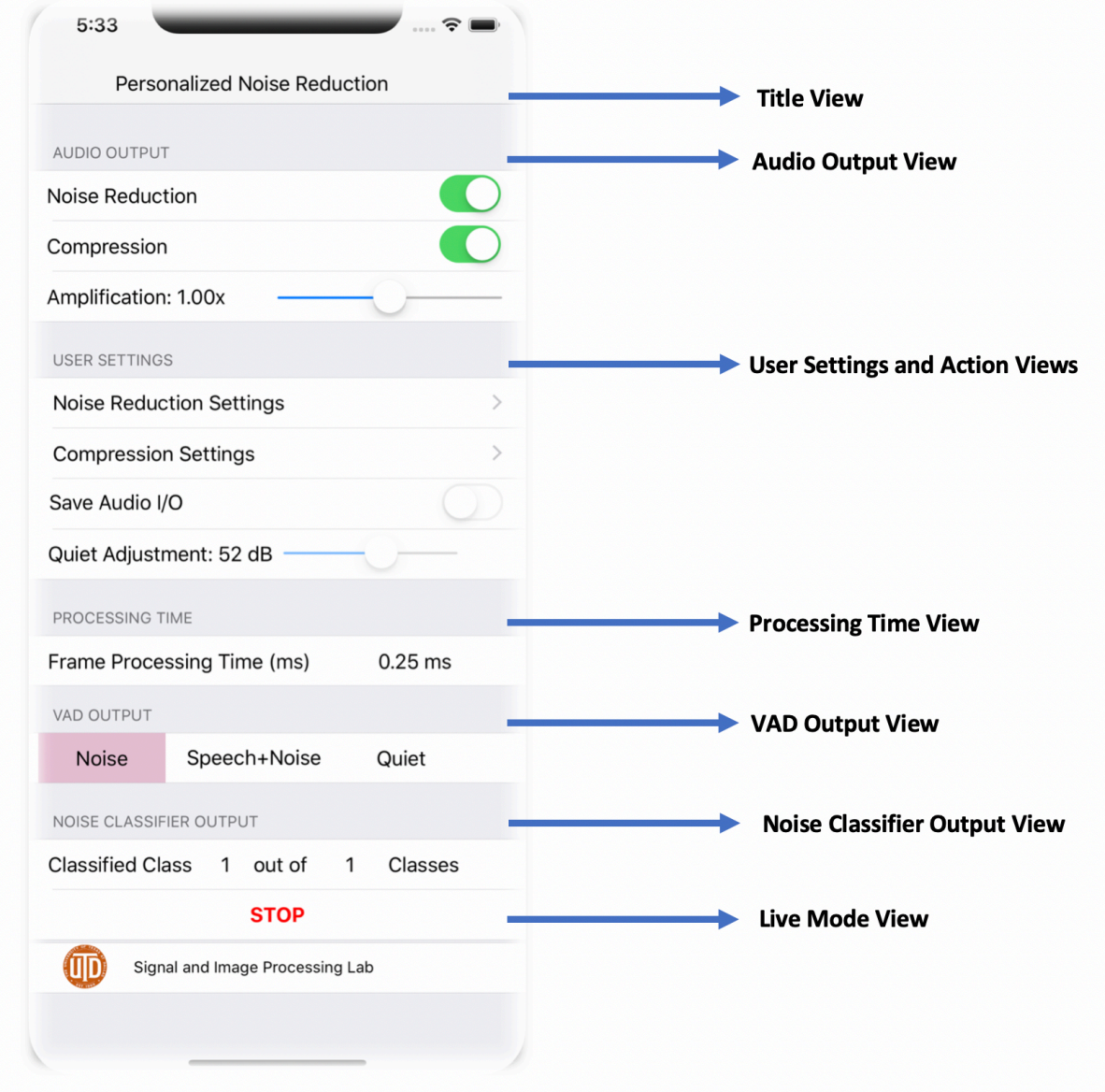


Fig. 3: Personalized NR iOS GUI: Main View

Main view consists of 7 segments, see Fig. 3.

- Title View
- Audio Output View
- User Settings and Actions View
- Processing Time View
- VAD Output View
- Noise Classifier Output View
- Live Mode View
- Main View
- Noise Reduction Settings View
- Compression Settings View

2.1.1 Title

The title displays the title of the app.

2.1.2 Audio Output

This part consists of:

- 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

2.1.3 User Settings

This part provides the following entries:

- Noise reduction (audio, and noise classifier) settings
- Compression settings
- An option to save input/output audio signals
- A slider to control Quiet Adjustment

2.1.4 Processing Time

This part shows the processing time in milliseconds taken by the personalized NR app per audio frame.

2.1.5 VAD Output

This part provides the VAD output, that is, noise, speech+noise, or quiet.

2.1.6 Noise Classifier Output

This part exhibits the unsupervised noise classifier output, that is, it shows the detected noise class out of the total number of previously identified classes.

2.1.7 Live Mode

This part includes a button for starting and stopping the Live mode of the app.

2.2 Noise Reduction Settings View

This view contains various settings for the noise reduction, audio processing, and unsupervised noise classifier. As shown in Fig. 4, the entries under this settings view are:

- **Sampling Frequency:** This field shows the sampling frequency. To obtain the lowest latency for 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 computationally more efficient.
- **Window Size (ms):** This field shows the window (frame) size in milliseconds that is passed through the signal processing modules of the app. Basically, this indicates how many data samples (corresponding to the specified window length) get processed by the personalized NR 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 between consecutive frames or windows in milliseconds. For 50% overlap, the content of a processing frame is updated at half of the window size time.

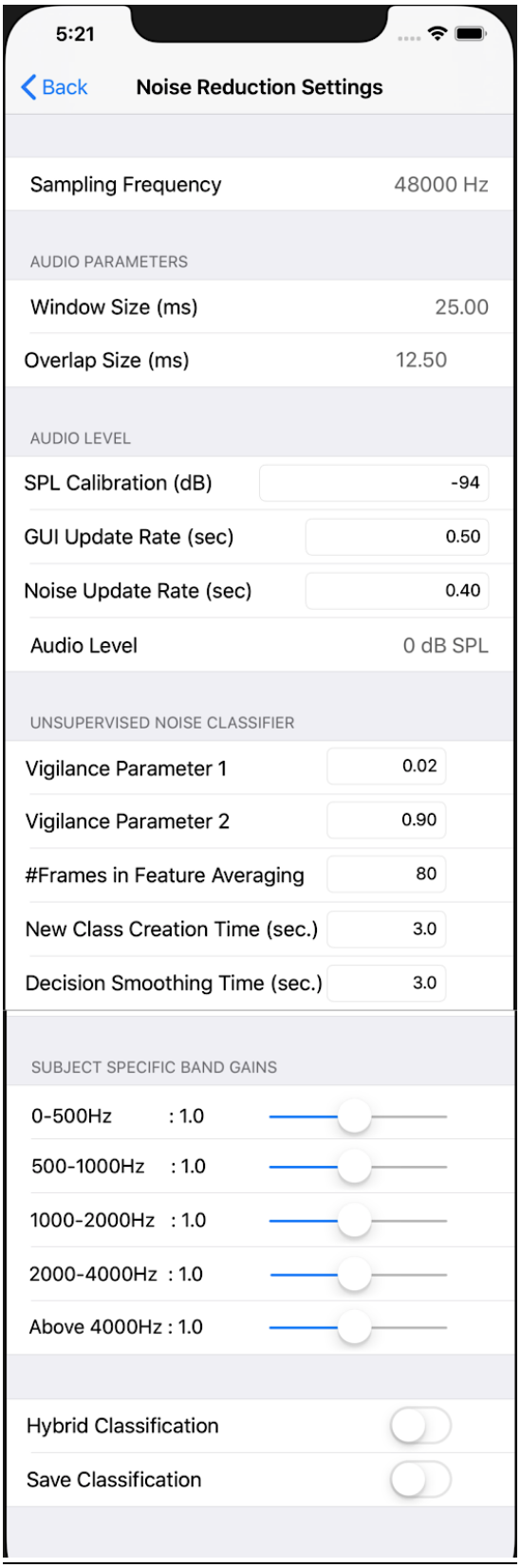


Fig. 4: Personalized NR app iOS GUI: Noise Reduction Settings View

- **SPL Calibration offset (dB):** This field allows the user to set or adjust a calibration constant which converts the audio level from dB FS (full scale) to dB SPL (sound pressure level).
- **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 performing noise reduction. This time can be adjusted by the user.
- **Audio Level (dB):** This field shows the measured sound pressure level (SPL) in dB of audio signals using the calibration constant.
- **Unsupervised Noise Classifier:** This field denotes settings for Noise Classification which includes following fields:

    - **Vigilance Parameter 1:** This field allows the user to set the vigilance parameter for the first ART2 unsupervised noise classifier.
    - **Vigilance Parameter 2:** This field allows the user to set the vigilance parameter for the second ART2 unsupervised noise classifier.
    - **#Frames in Feature Averaging:** This field allows the user to set the number of frames over which the feature vector is averaged before it is passed onto the classifier. One second corresponds to 80 frames.
    - **New Class Creation Time (sec.):** This field allows the user to set the time for the new noise class creation. That is, the noise classifier waits for this amount of time before it creates a new class to avoid frequent switching of noise classes.
    - **Decision Smoothing Time (sec.):** This field allows the user to set the time for the classification decision to be conducted in a smooth manner.
- **Subject Specific Band Gains:** This section allows the user to set the band gains across five frequency bands. Gain values at each band can be varied by the user between 0.1 to 2.0 in steps of 0.1. These gains can be adjusted for each noise class and saved for next time the app is run. This view consists of five sliding bars enabling the adjustment of gains for the following five bands:

    - **0 – 500Hz**: A seek bar (in Android, sliding bar is named a seek bar) to control the band gain for the frequency band 0-500Hz (default set to 1.0)
    - **500 – 1000Hz:** A seek bar to control the band gain for the frequency band 500-100Hz (default set to 1.0).

- **1000 – 2000Hz**: A seek bar to control the band gain for the frequency band 1000-2000Hz (default set to 1.0).
- **2000 – 4000Hz:** A seek bar to control the band gain for the frequency band 2000-4000Hz (default set to 1.0).
- **Above 4000Hz:** A seek bar to control the band gain for the frequency band above 4000Hz (default set to 1.0).

- **Save/Hybrid Mode:** This section consists of:
  - A switch to turn on and off the Hybrid Mode for classification and subject specific band gains.
  - A switch to turn on and off the Save Classification and Band Gains parameters.

2.3 Compression Settings View

This view shows various settings for the compression module for the five frequency bands, 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 low to a high value.
- **Release 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.

The following 5 frequency bands are considered in the compression:

- 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 of the frequency bands. The app uses a scrolling option to have a large view of the compression settings, see Fig. 5.

Fig. 5: Personalized NR 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 "PersonalizedNR_App.xcodeproj" in the folder "PersonalizedNR-App" as shown in Fig. 1. The code is divided into 3 parts, see Fig. 6:

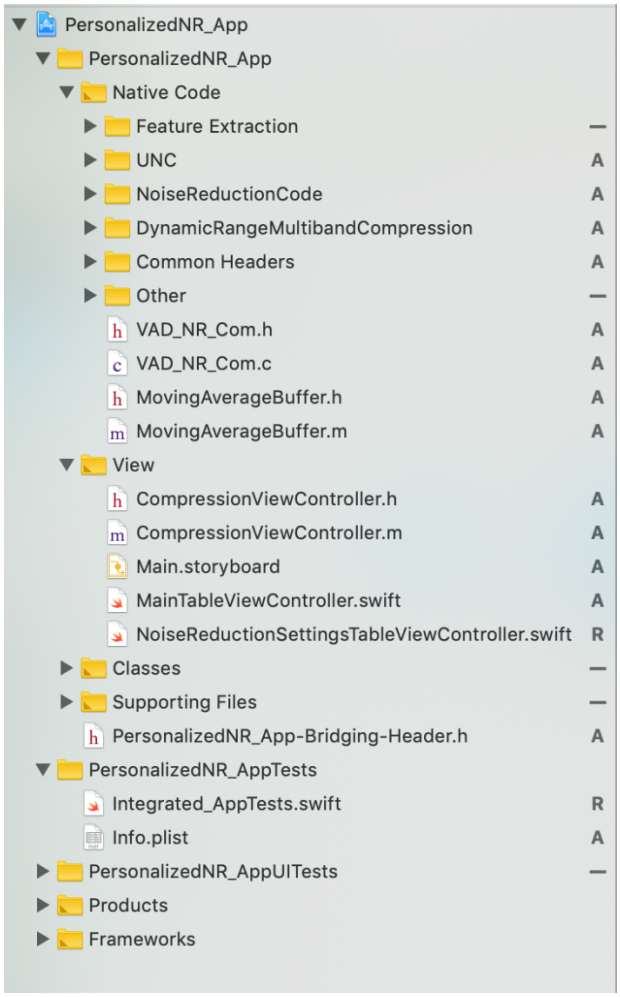


Fig. 6: Personalized NR app iOS version code flow

- Native Code
- View
- Classes

3.1 Native Code

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

- **VAD_NR_Com:** This denotes 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].
- **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.
- **Feature Extraction:** This corresponds to codes as described in [2] to extract subband and Mel Frequency Spectrum coefficients features per audio frame.
- **UNC:** This corresponds to the codes as described in [2] to perform ART2 Unsupervised Noise Classification (UNC). This includes the following codes:
    - **Art2FusionClassifier:** This corresponds to the codes described in [2] for the noise classifier module, which is developed in C/C++. This classifier uses Mel Frequency Spectrum coefficients and subband features for the classification.
    - **Equalizer:** This corresponds to the equalizer code to obtain a smooth interpolating curve from the five subject specific gains set by the user. This code is developed in C.
- **NoiseReductionCode:** This corresponds to the codes as described in [10] for the noise reduction module, which is developed in MATLAB and then converted into C using the MATLAB Coder [11].

- **DynamicRangeMultibandCompression:** This corresponds to the codes for the compression module, which is developed in MATLAB and then converted into C using the MATLAB Coder [11].

- **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.

- **Other:** This includes the following codes:

    - **filterCoefficients.h**: This component includes the filter coefficients for the FIR filter.

- **FIRFilter:** FIR filtering is done to lowpass filter 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).
- **Timer.h:** This component is for computing processing time of each frame, including time for extracting features and classification.

The breakdown of the three native code modules along with the common header, Feature Extraction, and Other folder is shown in Fig. 7.

Fig. 7: Further breakdown of native code modules in iOS version of the personalized NR app

3.2 View:

This section provides the setup for the GUI of the personalized NR 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, audio settings, and unsupervised noise classifier. 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.
- **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.
- **CNN_VAD_Setup:** This sets up CNN-based VAD.
- **TensorFlowInterference:** This calls CNN-based VAD functions.

3.4 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 [13], 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.

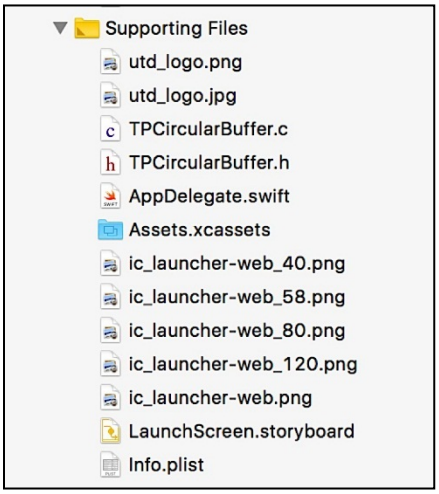


Fig. 8: Supporting files

Section 4: Modularity and Modification

This section covers the modularity of the personalized NR 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 considered here (i.e., Noise Reduction, Unsupervised Noise Classifier 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 Personalized NR app is straightforward and includes:
    - Detection of speech or noise activity of incoming audio frames by the VAD.
    - Noise classification using an ART2 fusion solution based on the VAD decision.
    - Computation of the interpolated gain curve based on the user specified band gains and the noise classifier output.
    - 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 appear in the "VAD_NR_Com" source files (.h and .c).

- **Replace or Add modules:** To replace or add module(s), include/remove:
    - Corresponding headers in "VAD_NR_Com.h"
    - Set variables in the "VADNoiseReductionCompression" structure (same header).
    - Initialize variables inside the "initVAD_NoiseReduction_Compression" function as defined in "VAD_NR_Com.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 "PersonalizedNR_App-Bridging-Header.h" which is included inside the native code folder (refer to Fig 6). 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.

    - **Unsupervised Noise Classification and Subject Specific Band Gains Adjustment:**
        - "settings->userBandGains" saves the linear band gains value from the seek bar.

- "settings->saveData" saves the switch status for Save Classification.
- "settings->hybridMode" saves the switch status for Hybrid Classification.
- "settings->vigilance1" and "settings->vigilance1" saves the Vigilance Parameter 1 and 2, respectively.
- "settings->FeatAvgBufferLength" saves the #frames in feature averaging.
- "settings->NewClusterCreationBufferTime" saves the time for the purpose of creating a new cluster.
- "settings->DecisionSmoothingBufferTime" saves the decision smoothing time.

  - **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 [15] and [16]. 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.

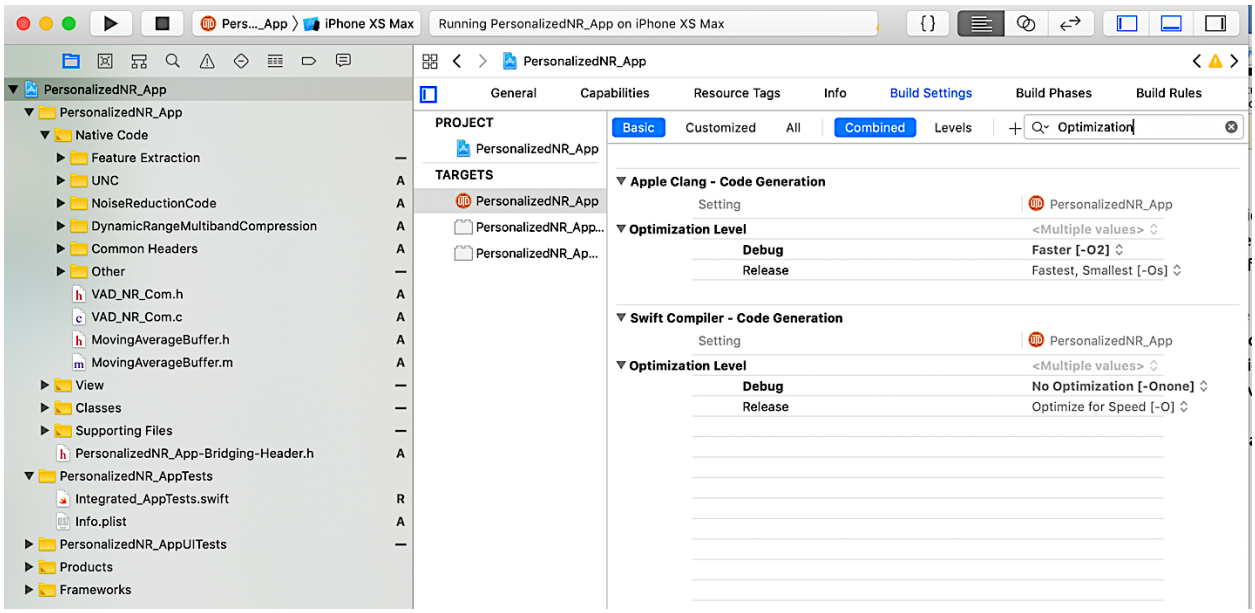


Fig. 9: Settings optimization level in Xcode for the personalized NR app iOS

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 Personalized NR app, it is necessary to have Superpowered SDK which can be obtained from the link at [17]. 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.

Section 2: Android GUI

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

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

2.1 Main View

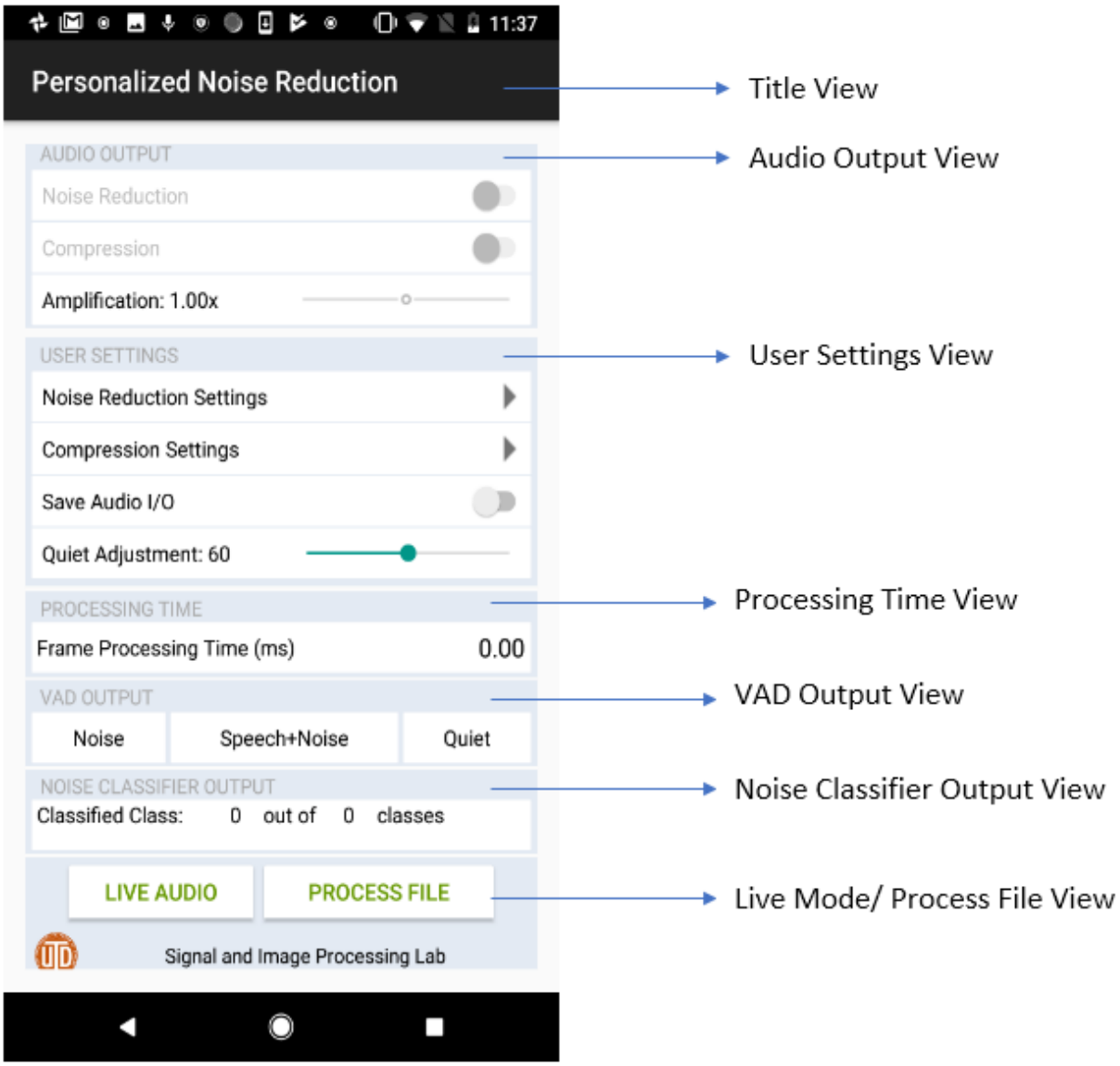


Fig. 10: Personalized NR Android GUI: Main view

Main view consists of 10 parts, see Fig. 10:

- Title
- Audio Output
- User Settings
- Processing Time
- VAD Output
- Noise Classifier Output
- Live Mode/Process File

2.1.1 Title

The title displays the title of the app.

2.1.2 Audio Output

This part 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.

2.1.3 User Settings

This part provides the following entries:

- Noise reduction (and audio) settings
- Compression settings
- An option to save input/output audio signals
- A seek bar (in Android, slider is named a seek bar) to control Quiet Adjustment.

2.1.4 Processing Time

This part shows the processing time in milliseconds taken by the personalized NR app per audio frame.

2.1.5 VAD Output

This part provides the VAD output, that is, noise, speech+noise, or quiet.

2.1.6 Noise Classifier Output

This part exhibits the unsupervised noise classifier output, that is, it shows the detected noise class out of the total number of previously identified classes.

2.1.7 Live Mode/Process File

This part includes two buttons:

- A button for starting and stopping the Live mode of the app.
- A button for processing the selected file.

2.2 Noise Reduction Settings View

This view contains various settings for the noise reduction and audio processing. As shown in Fig. 11, the entries under 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 that is passed through the signal processing modules of the app. Basically, this indicates how many data samples (corresponding to the specified 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 between consecutive frames or windows in milliseconds. For 50% overlap, the content of a processing frame is updated at half of the window size time.

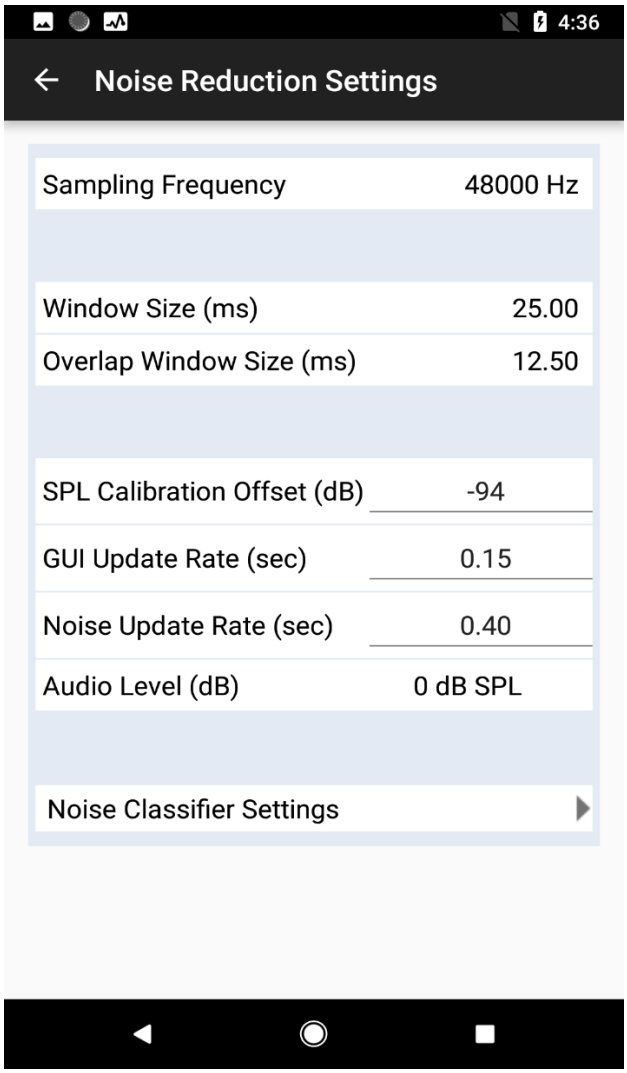


Fig. 11: Personalized NR app Android GUI: Noise Reduction Settings View

- **SPL Calibration offset (dB):** This field allows the user to set or adjust a calibration constant which converts the audio level from dB FS (full scale) to dB SPL (sound pressure level).
- **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 performing noise reduction. This time can be adjusted by the user.
- **Audio Level (dB):** This field shows the measured sound pressure level (SPL) in dB of audio signals using the calibration constant.
- **Noise Classifier Settings:** This field denotes a shortcut for Noise Classification Settings View.

2.3 Noise Classification Settings View

This view contains various settings for the unsupervised noise classification and subject specific band gains. As shown in Fig. 12, the settings consist of 3 views:

- Unsupervised Noise Classifier
- Subject Specific Band Gains
- Save/Hybrid Mode

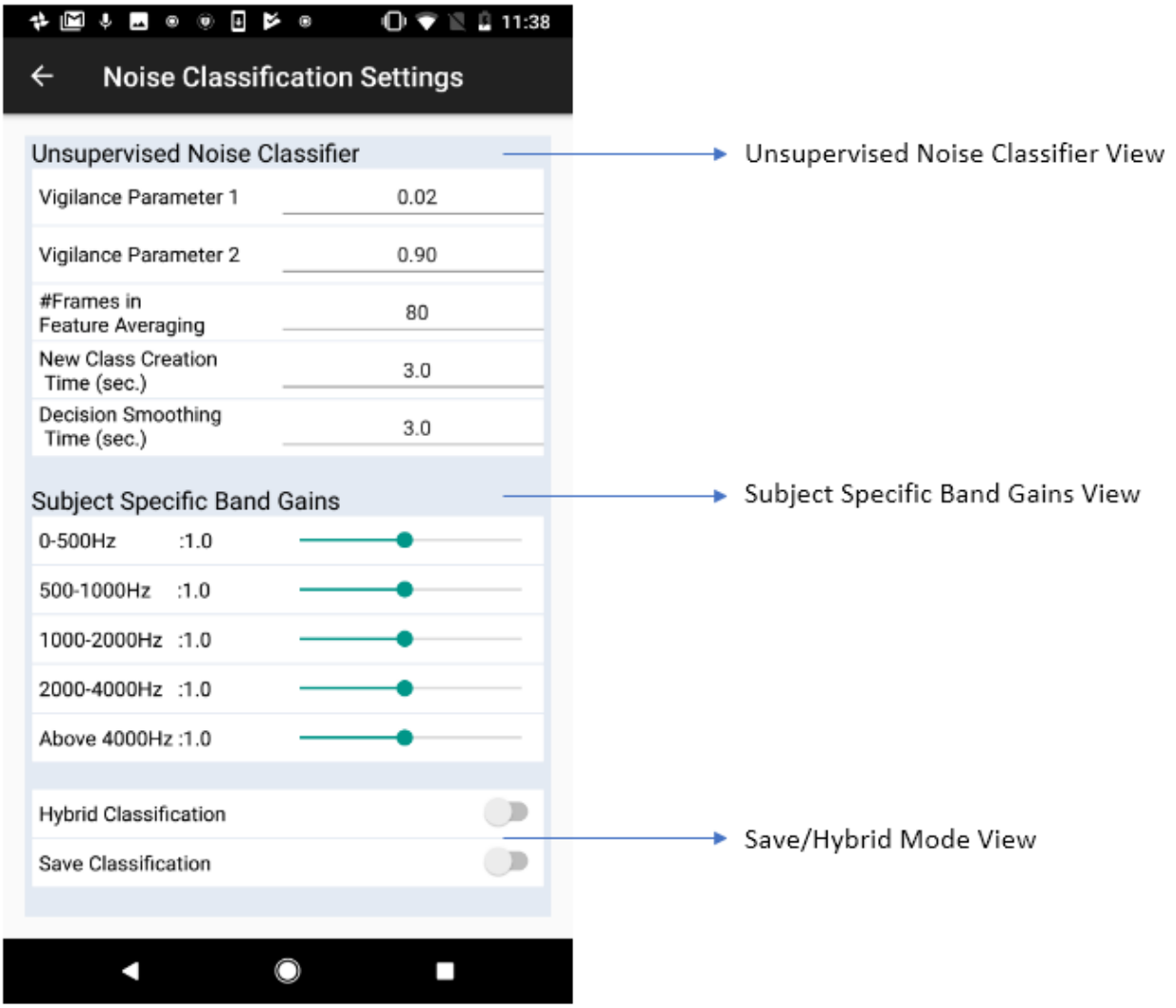


Fig. 12: Personalized NR app GUI: Noise Classification Settings View

2.3.1 Unsupervised Noise Classifier View

This view consists of:

- **Vigilance Parameter 1:** This field allows the user to set the vigilance parameter for the first ART2 unsupervised noise classifier.
- **Vigilance Parameter 2:** This field allows the user to set the vigilance parameter for the second ART2 unsupervised noise classifier.
- **#Frames in Feature Averaging:** This field allows the user to set the number of frames over which the feature vector is averaged before it is passed onto the classifier. One second corresponds to 80 frames.
- **New Class Creation Time (sec.):** This field allows the user to set the time for the new noise class creation. That is, the noise classifier waits for this amount of time before it creates a new class to avoid frequent switching of noise classes.
- **Decision Smoothing Time (sec.):** This field allows the user to set the time for the classification decision to be conducted in a smooth manner.

2.3.2 Subject Specific Band Gains View

This view allows the user to set the band gains across five frequency bands. Gain values at each band can be varied by the user between 0.1 to 2.0 in steps of 0.1. These gains can be adjusted for each noise class and saved for next time the app is run. This view consists of five sliding bars enabling the adjustment of gains for the following five bands:

- **0 – 500Hz**: A seek bar (in Android, sliding bar is named a seek bar) to control the band gain for the frequency band 0-500Hz (default set to 1.0)
- **500 – 1000Hz:** A seek bar to control the band gain for the frequency band 500-100Hz (default set to 1.0).
- **1000 – 2000Hz**: A seek bar to control the band gain for the frequency band 1000-2000Hz (default set to 1.0).
- **2000 – 4000Hz:** A seek bar to control the band gain for the frequency band 2000-4000Hz (default set to 1.0).
- **Above 4000Hz:** A seek bar to control the band gain for the frequency band above 4000Hz (default set to 1.0).

2.3.3 Save/Hybrid Mode View

This view consists of:

- A switch to turn on and off the Hybrid Mode for classification and subject specific band gains.
- A switch to turn on and off the Save Classification and Band Gains parameters.

2.4 Compression Settings View

This view shows various settings for the compression module for the five frequency bands, see Fig 13. 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 following 5 frequency bands are considered in the compression:

- 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 of the frequency band. The app uses a scrolling option to have a large view of the compression settings, see Fig. 13.

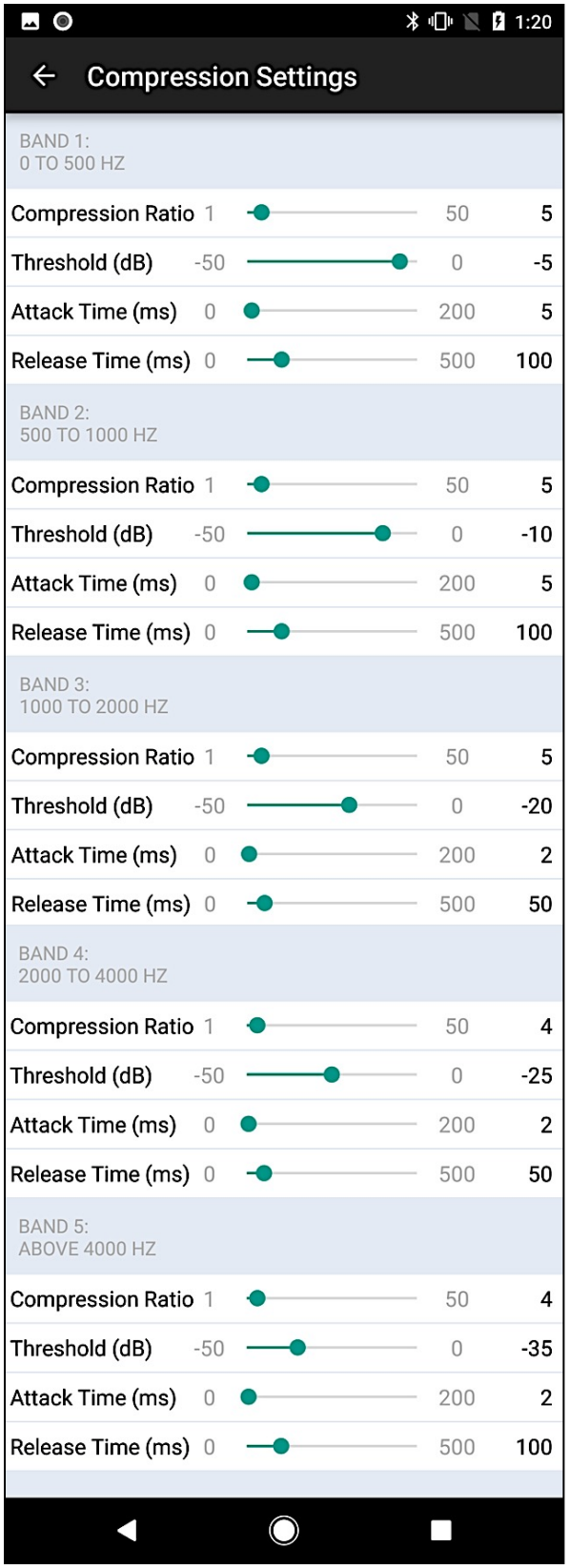


Fig. 13: Personalized NR app Android GUI: Compression Settings view.

Section 3: Code Flow

This section covers the personalized NR app code flow. The folder organization of the app appears as displayed in Fig. 14.

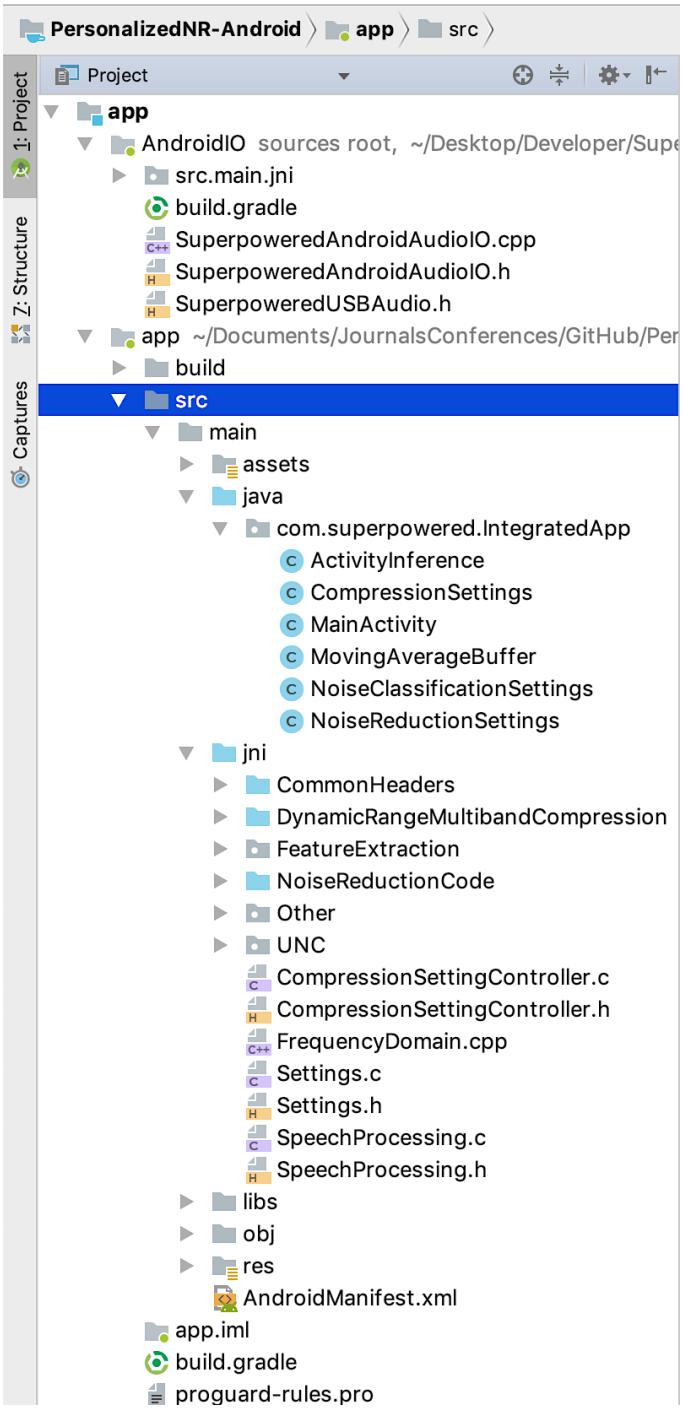


Fig. 14: Personalized NR app Android version: project organization

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

- **AndroidIO:** This folder provides the Superpowered source root files to control the audio i/o interface of the app.
- **src/main/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.
    - **ActivityInference:** This contains the TensorFlow implementation of VAD using CNN.
    - **NoiseClassificationSettings:** This contains the GUI elements and controls the Noise Classification and Subject Specific Band Gains view.
- **src/main/jni:** This folder contains all the function calls to start the audio i/o and the C codes of the implemented hearing aid modules.
- **src/res/layout:** The layout subfolder of the recourse folder contains the GUI layouts of the personalized NR app appearing in these three .xml files:
    - **activity_main.xml:** This file provides the layout for the Main view.
    - **activity_noise_reduction_settings.xml:** This file provides the layout for the Noise Reduction Settings view.
    - **activity_compression_settings.xml:** This file provides the layout for the Compression Settings view.
    - **activity_classification_settings.xml:** This file provides the layout for the Noise Classification Settings view.

3.1 Native Code and Settings

This native code section states the implementation aspects of the hearing aid modules and their 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 denotes 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].
- **Settings:** This provides the parameters for the audio control settings. The native codes use the parameters here in a structure for audio processing. "MainActivity" initially loads the settings structure when the app is loaded. The corresponding parameters are updated through FrequencyDomain appearing in the GUI.
- **CompressionSettingController:** This provides a separate 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 five bands is used by the native code for compression.
- **Feature Extraction:** This corresponds to codes as described in [2] to extract subband and Mel Frequency Spectrum coefficients features per audio frame.
- **UNC:** This corresponds to the codes as described in [2] to perform ART2 Unsupervised Noise Classification (UNC). This includes following codes:
    - **Art2FusionClassifier:** This corresponds to the codes described in [2] for the noise classifier module, which is developed in C/C++. This classifier uses Mel Frequency Spectrum coefficients and subband features for the classification.
    - **Equalizer:** This corresponds to the equalization code to obtain a smooth interpolating curve from the five subject specific gains set by the user. This code is developed in C.
- **NoiseReductionCode:** This corresponds to the codes as described in [10] for the noise reduction module, which is developed in MATLAB and then converted into C using the MATLAB Coder [11].

- **DynamicRangeMultibandCompression:** This corresponds to the codes for the compression module, which is developed in MATLAB and then converted into C using the MATLAB Coder [11].

- **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.

- **Other:** This includes the following codes:

    - **filterCoefficients.h**: This component includes filter coefficients for the FIR filter.
    - **FIRFilter:** FIR filtering is done to lowpass filter 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).
    - **Timer.h:** This component is for computing processing time of each frame, including time for extracting features and classification.

The breakdown of the three native code modules along with the *common header*, *Feature Extraction*, and *Other* folders are shown in Fig. 15.

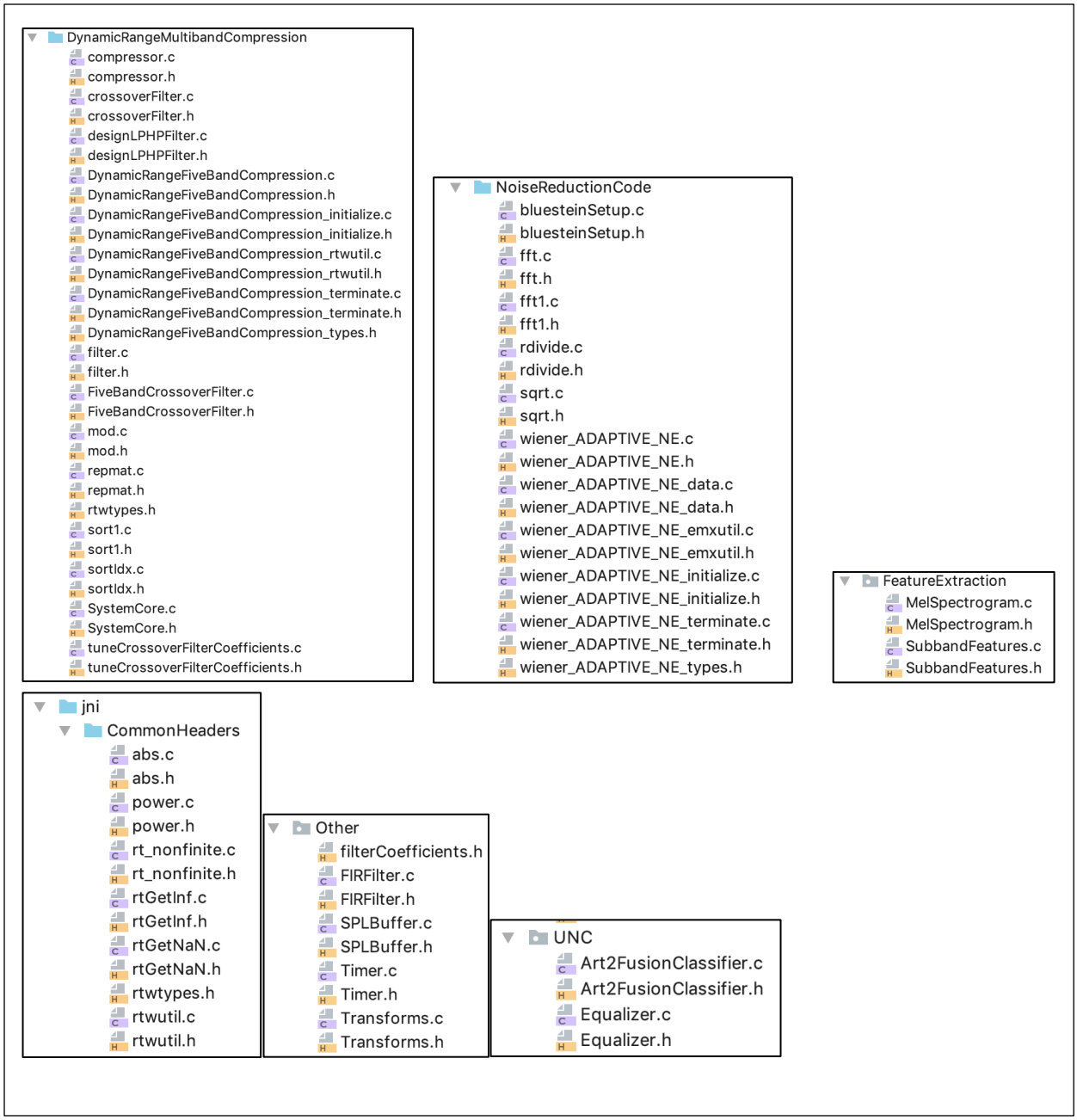


Fig. 15: Further breakdown of native code modules in Android

Section 4: Modularity and Modification

This section covers the modularity of the personalized NR 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 considered here (i.e., Noise Reduction, Unsupervised Noise Classifier, 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 used. To have the lowest latency, many Android smartphones process 192 samples of an incoming audio signal at 48kHz sampling frequency (refer to [18] for more information about frame samples for each Android device). 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 personalized NR app is straightforward and is as follows:
    - Detection of speech or noise activity of incoming audio frames by the VAD.
    - Noise classification using an ART2 fusion solution based on the VAD decision.
    - Computation of the interpolated gain curve based on the user specified band gains and the noise classifier output.
    - Noise estimation and reduction of incoming audio frames based on the VAD decision and the interpolated gain curve.
    - 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), the following need to be included or removed:
    - Corresponding headers in "SpeechProcessing.h"
    - 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. 16. It is required to add the path of each folder as well as the subfolders separately.

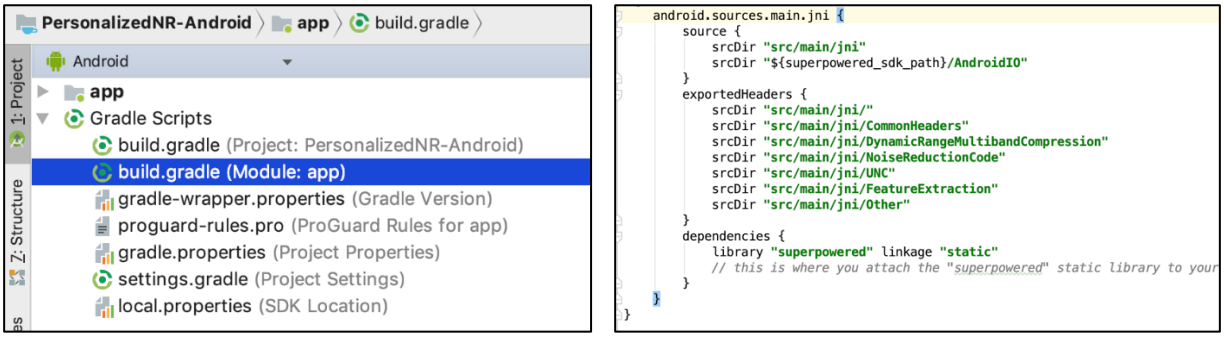
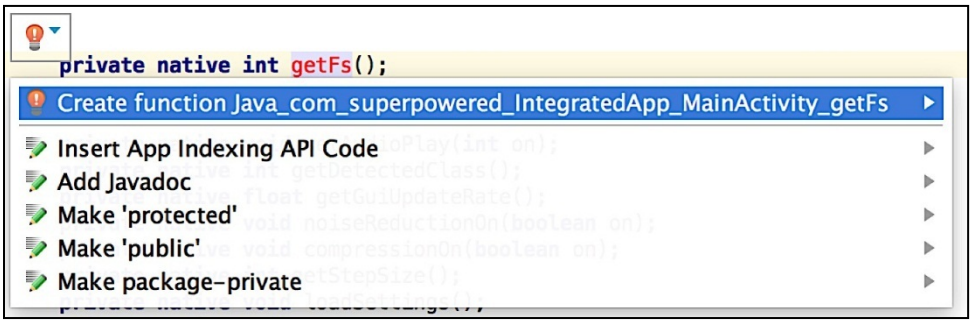


Fig. 16: Add native folder paths

- **<u>Data transaction between the GUI and Native Code:</u>** For the Android version of the developed personalized NR app, Java Native Interface (JNI) is used to interface with native codes in the Java environment. This approach is described in [19-21]. The following procedure needs to be followed:
  - To call any C function or update the settings parameters, declare a linking function in the Java Activity file in which it will be utilized (MainActivity, NoiseReductionSettings, NoiseClassificationSettings 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. 17a.

  - Click on the create option. It will create a function in FrequencyDomain.cpp, see Fig. 17b.
  - Modify the function definition using extern “C” keyword, see Fig. 17c, noting that it is calling the “setting->fs” variable from a C file.



(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. 17: Creating native linking function

Follow the steps for all the native calls. The entry point to the Native Portion of the app from Java is obtained through the "**private native void** FrequencyDomain()" function after performing 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 the settings is bridged through “FrequencyDomain.cpp”. 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 the 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 using “SPLBuffer” over the “guiUpdateInterval” time.
  - “settings->noiseEstimateTime” saves the time over which noise parameters are estimated and averaged.

  This information is passed to the GUI through “FrequencyDomain.cpp”. 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.

- **Unsupervised Noise Classification and Subject Specific Band Gains Adjustment:**
  - “settings->userBandGains” saves the linear band gains value from the seek bar.
  - “settings->saveData” saves the switch status for Save Classification.
  - “settings->hybridMode” saves the switch status for Hybrid Classification.
  - “settings->vigilance1” and “settings->vigilance1” saves the Vigilance Parameter 1 and 2 respectively.
  - “settings->FeatAvgBufferLength” saves the #frames in feature averaging.
  - “settings->NewClusterCreationBufferTime” saves the time for the purpose of creating a new cluster.
  - “settings->DecisionSmoothingBufferTime” saves the decision smoothing time.

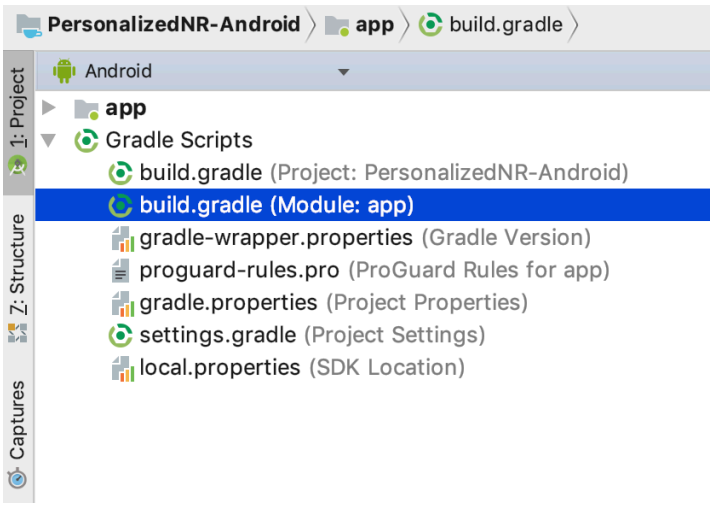
These data are passed from and to the GUI through “FrequencyDomain.cpp”.

- **Compression Controls:** “CompressionSettingsController” provides a separate setting parameter 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.

- **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 [15] and [16]. The user can change them according to specific requirements. To set these flags, follow the steps as noted in Fig. 18.

- Under “Android” of the project view explorer of Android Studio, select build.gradle (Module: app) under Gradle Scripts, see left side of Fig. 18.
- At the right side of Fig. 18, set the optimization flags under
  - model → android.ndk → CFlags.addAll();



(left)

```
40
41      android.ndk { // your application's native layer parameters
42          moduleName = "FrequencyDomain"
43          platformVersion = 16
44          stl = "c++_static"
45
46          // use -Ofast or -O3 for full optimization, char data type is signed
47          // -O2 is recommended;
48          //  -DNDEBUG": no debug symbols
49          // -pipe: makes  compilation process faster
50          // -s: Remove all symbol table and relocation information from the executable.
51          // refs: https://wiki.gentoo.org/wiki/GCC_optimization
52          //       https://gcc.gnu.org/onlinedocs/gcc-4.8.0/gcc/Link-Options.html#Link-Options
53          CFlags.addAll(["-O2", "-s", "-fsigned-char", "-pipe"])
54          cppFlags.addAll(["-fsigned-char", "-I${superpowered_sdk_path}".toString()])
55          ldLibs.addAll(["log", "android", "OpenSLES"])
56          // load these libraries: log, android, OpenSL ES (for audio)
57          abiFilters.addAll(["armeabi-v7a", "arm64-v8a", "x86", "x86_64"])
58          // these platforms cover 99% percent of all Android devices
59      }
```

(right)

Fig. 18: Settings optimization level in Android Studio for the personalized NR app Android.

Timing difference between iOS and Android versions of Personalized NR app

The low-latency audio i/o setup for iOS is done using the software package CoreAudio API [22] and for Android using the software package Superpowered SDK [17]. 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 personalized NR app.

Version of the Personalized NR 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	1.4	0.3	1.1

For Android smartphones, it is worth mentioning that in order to measure the correct CPU utilization, the *sustained performance mode* of the Superpowered package is required to be changed from the default mode of *active* to *de-active* as otherwise an erroneous CPU utilization would be obtained. This mode can be changed in FrequencyDomain.c by modifying “true”:

```
SuperpoweredCPU::setSustainedPerformanceMode(true);
```

To “false”:

```
SuperpoweredCPU::setSustainedPerformanceMode(false);
```

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.

References:

[1] N. Alamdari, S. Yaraganalu, and N. Kehtarnavaz, "A Real-Time Personalized Noise Reduction Smartphone App for Hearing Enhancement", *Proceedings of IEEE Signal Processing in Medicine and Biology Symposium (SPMB)*, Philadelphia, PA, Dec 2018.

[2] N. Alamdari, and N. Kehtarnavaz, "A Real-Time Smartphone App for Unsupervised Noise Classification in Realistic Audio Environments," *Proceedings of IEEE International Conference on Consumer Electronics*, Las Vegas, NV, Jan 2019.

[3] T. Chowdhury, A. Sehgal and N. Kehtarnavaz, "Integrating Signal Processing Modules of Hearing Aids into a Real-Time Low-Latency Smartphone App," *Proceedings of IEEE Conference on Engineering in Medicine and Biology,* Honolulu, HI, , 2018.

[4] 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.

[5] A. Sehgal, and N. Kehtarnavaz, "A Convolutional Neural Network Smartphone App for Real-Time Voice Activity Detection" *IEEE Access*, vol. 6, pp. 9017-9026, 2018.

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

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

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

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

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

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

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

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

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

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

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

[17] http://superpowered.com/

[18] https://superpowered.com/latency

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

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

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

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