User's Guide:

AN EDUCATIONAL TOOL FOR HEARING AID COMPRESSION FITTING VIA A WEB-BASED ADJUSTED SMARTPHONE APP

N. ALAMDARI AND N. KEHTARNAVAZ
SIGNAL AND IMAGE PROCESSING LAB
UNIVERSITY OF TEXAS AT DALLAS

APRIL 2019

Table of Contents

Table of Figures

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 designed to be an educational tool to learn about how hearing aid compression fitting is carried out from a signal processing perspective and is designed to run in real-time with low-latency on smartphone platforms for hearing compression purposes as discussed in [1]. An interactive web-based program is developed based on the widely used DSL-v5 fitting rationale. This program can be accessed and used from any internet browser to generate the parameters of compression curves that correspond to the nine frequency bands used in the DSL-v5 prescriptive compression. These parameters are transferred to a smartphone in the form of a datafile which is then used by a compression app. The compression app can be viewed as a virtual hearing aid, running in real-time on both iOS and Android smartphones. This educational tool is easy-to-use as no programming knowledge is needed for adjusting gain across the frequency bands and subsequently running the corresponding compression curves in the smartphone app to appropriately compress input sound signals.

This user's guide is divided into three parts. The first part covers educational web-based compression fitting. The second part covers the iOS version of the compression app and the third part covers the Android version. Each part consists of five 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. Third section defines how to transfer a JSON datafile to a smartphone to be used by the compression app. Finally, the modularity and modification of the app are mentioned in the fifth section.

Devices used for running the compression app

- iPhone8 as iOS platform
- Google Pixel as Android platform

Compression 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:

- "Compression_App_Android" denoting the Android Studio project
- "Compression_App_iOS" denoting the iOS Xcode project

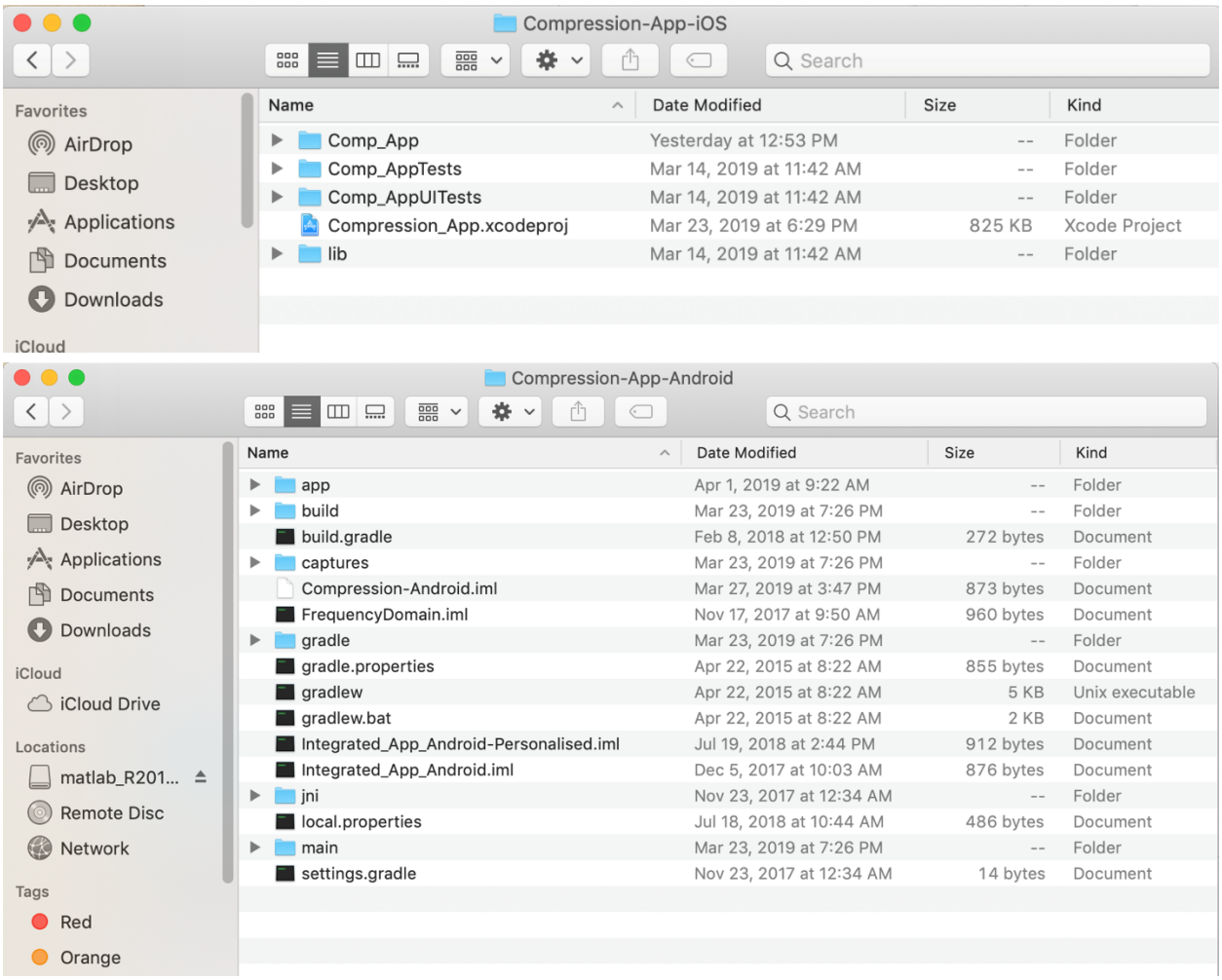


Fig. 1 - Compression app folder contents

Part 1

Web-based Compression Fitting

Figure 2 shows a block diagram of the components of the developed educational tool. The DSL-v5 compression fitting is carried out interactively on a browser. The languages used to write this interactive web-based program include HTML (Hypertext Markup Language), CSS (Cascading Style Sheets), and JavaScript.

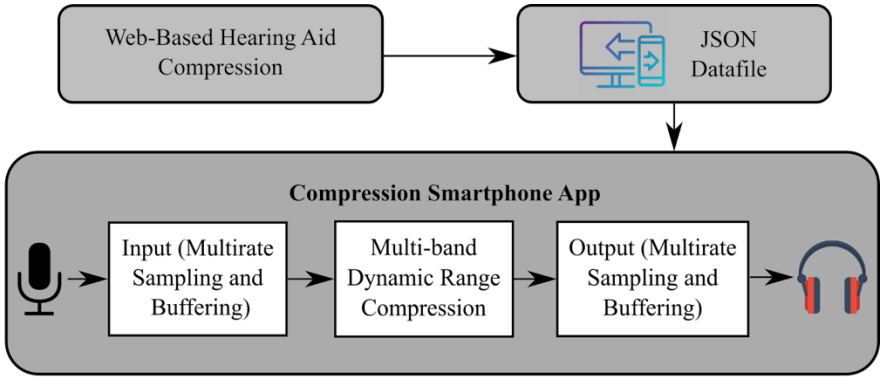


Fig. 2 - Block diagram of the developed educational tool

This web-based fitting is freely accessible and can be run on any browser via this link: http://www.utdallas.edu/~kehtar/WebBasedFitting.html.

Fig. 3 shows the graphical-user-interface (GUI) of the web-based compression fitting program. In this figure, the entries of the first table from the top correspond to:

- The age of a subject (more than 3 years old, 4-5 years old, 6 years old, or more than 6 years old)
- The type of hearing aid (CIC (Completely in Canal), ITC (In-The-Canal), ITE (In-The-Ear, or BTE Behind-The-Ear))
- The ear coupling of hearing aid (Earmolds or Eartips).

The second table in Fig. 3 incorporates a subject's audiometric thresholds for whom compression fitting is done. Audiometric thresholds (called audiogram) correspond to the softest level of sound a person can hear at different frequencies. The online utility in [2] is used by the program to obtain these measurements.

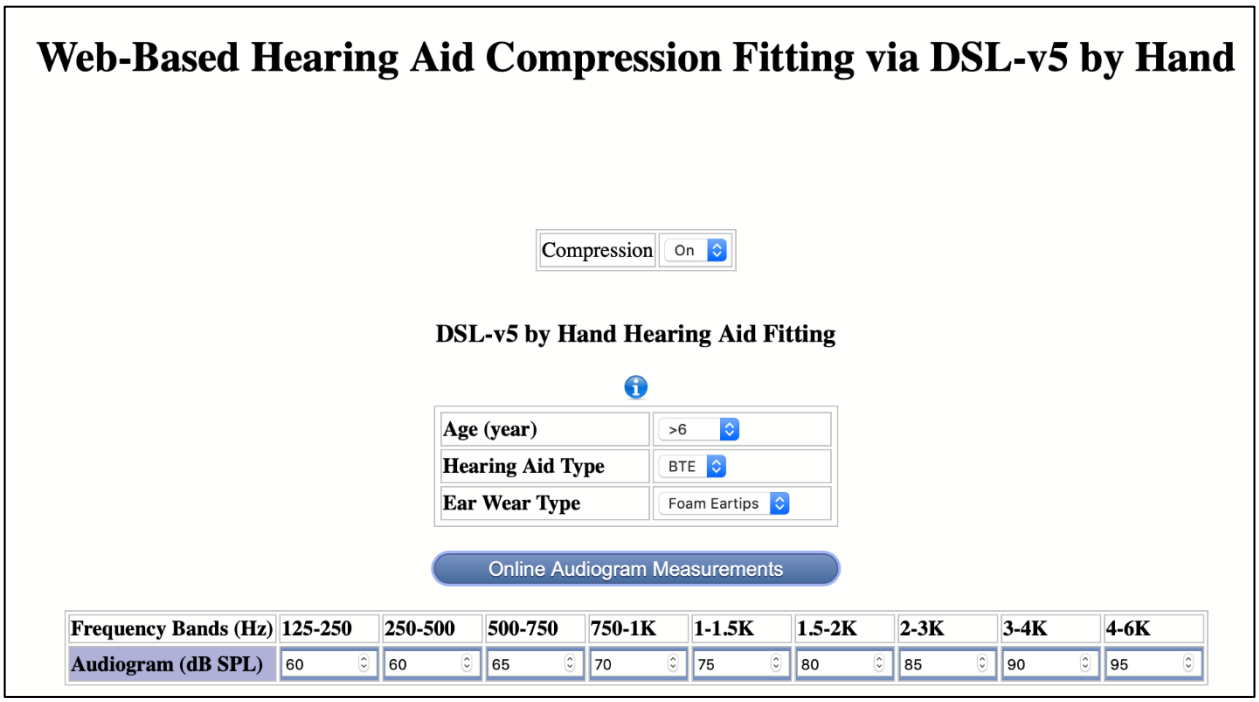


Fig. 3 - Graphical-user-interface (GUI) of the web-based compression fitting program (top part)

The next four tables that is shown in Fig. 4 indicate gain across 9 frequency bands for speech inputs at soft, moderate, loud, and maximum levels. The entry Target SPL Output in these tables is automatically determined based on the audiogram measurements and the pre-specified values in the DSL-v5 prescriptive tables. The other pre-specified entries of Input SPL, RECD (Real-Ear to Coupler Difference) and Mic Effect in DSL-v5 are automatically subtracted to generate desired gains in the frequency bands for soft (55dB), moderate (65dB), loud (75dB), and maximum output (90dB) levels.

Soft Speech (55 dB)

Frequency Bands (Hz)	125-250	250-500	500-750	750-1K	1-1.5K	1.5-2K	2-3K	3-4K	4-6K
Target SPL Output	84	82	81	83	87	93	96	99	102
Subtract Input SPL	**45**	**47**	**42**	**40**	**38**	**34**	**32**	**31**	**30**
Subtract RECD	3	5	5	6	8	6	2	3	8
Subtract Mic Effect	1	1	1	1	2	3	4	2	1
Target Gain	35	29	33	36	37	46	56	62	62

Moderate Speech (65 dB)

Frequency Bands (Hz)	125-250	250-500	500-750	750-1K	1-1.5K	1.5-2K	2-3K	3-4K	4-6K
Target SPL Output	89	88	88	91	97	103	106	109	112
Subtract Input SPL	**55**	**57**	**52**	**50**	**48**	**44**	**42**	**41**	**40**
Subtract RECD	3	5	5	6	8	6	2	3	8
Subtract Mic Effect	1	1	1	1	2	3	4	2	1
Target Gain	30	25	30	34	39	50	58	63	63

Loud Speech (75 dB)

Frequency Bands (Hz)	125-250	250-500	500-750	750-1K	1-1.5K	1.5-2K	2-3K	3-4K	4-6K
Target SPL Output	90	92	94	97	104	110	115	117	117
Subtract Input SPL	**57**	**65**	**66**	**64**	**65**	**61**	**57**	**56**	**50**
Subtract RECD	3	5	5	6	8	6	2	3	8
Subtract Mic Effect	1	1	1	1	2	3	4	2	1
Target Gain	29	21	22	26	29	40	52	56	58

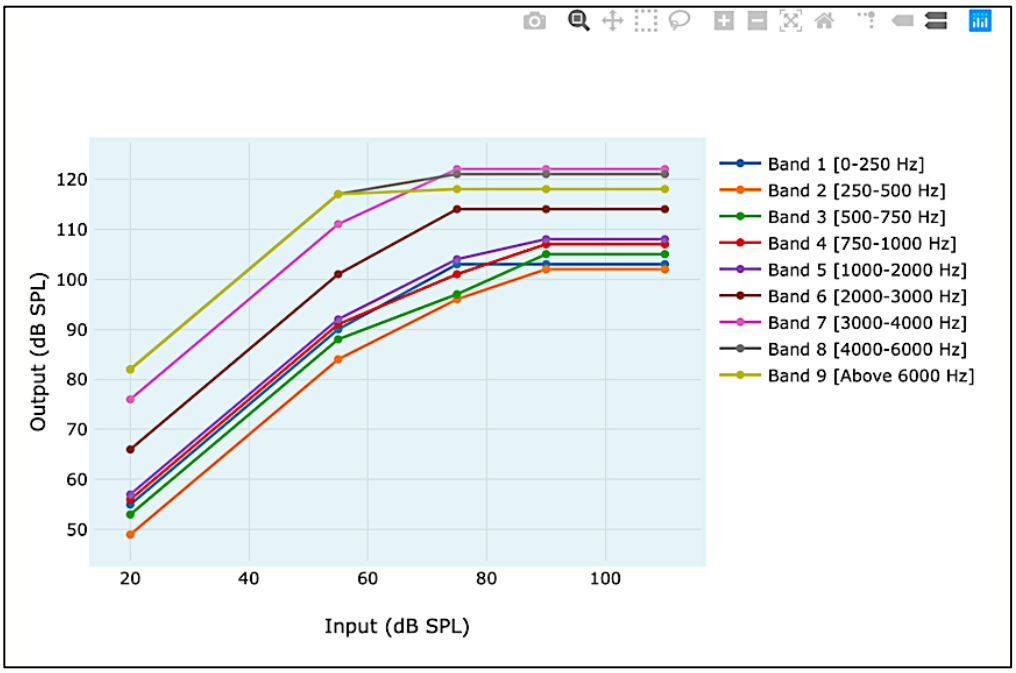
Max Output (90 dB)

Frequency Bands (Hz)	125-250	250-500	500-750	750-1K	1-1.5K	1.5-2K	2-3K	3-4K	4-6K
Target SPL Max Output	106	107	110	113	116	120	124	124	126
Subtract RECD	3	5	5	6	8	6	2	3	8
Target OSPL-90	103	102	105	107	108	114	122	121	118

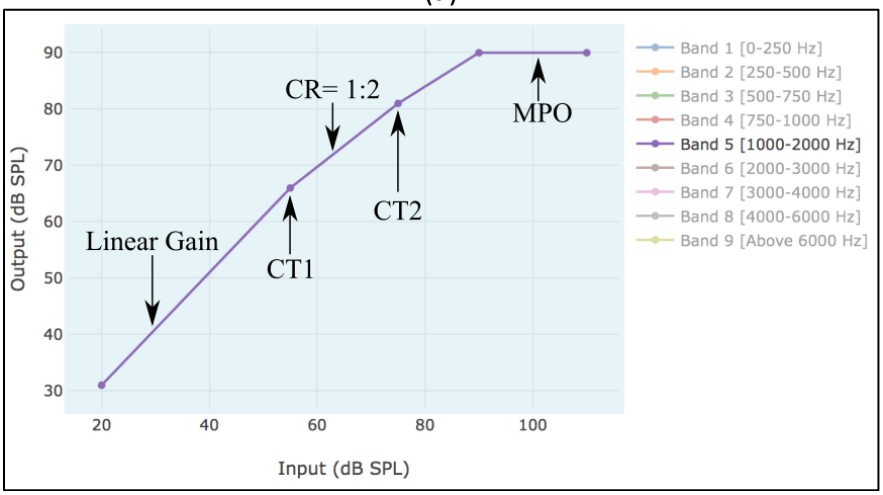
Update Tables

Fig. 4 - Gain across 9 frequency bands according to DSL-v5 prescriptive settings

By pressing “Update Tables” button in Fig. 4, gain from the DSL-v5 tables is then converted into compression curves corresponding to the 9 frequency bands as shown in Fig. 5.:



(a)



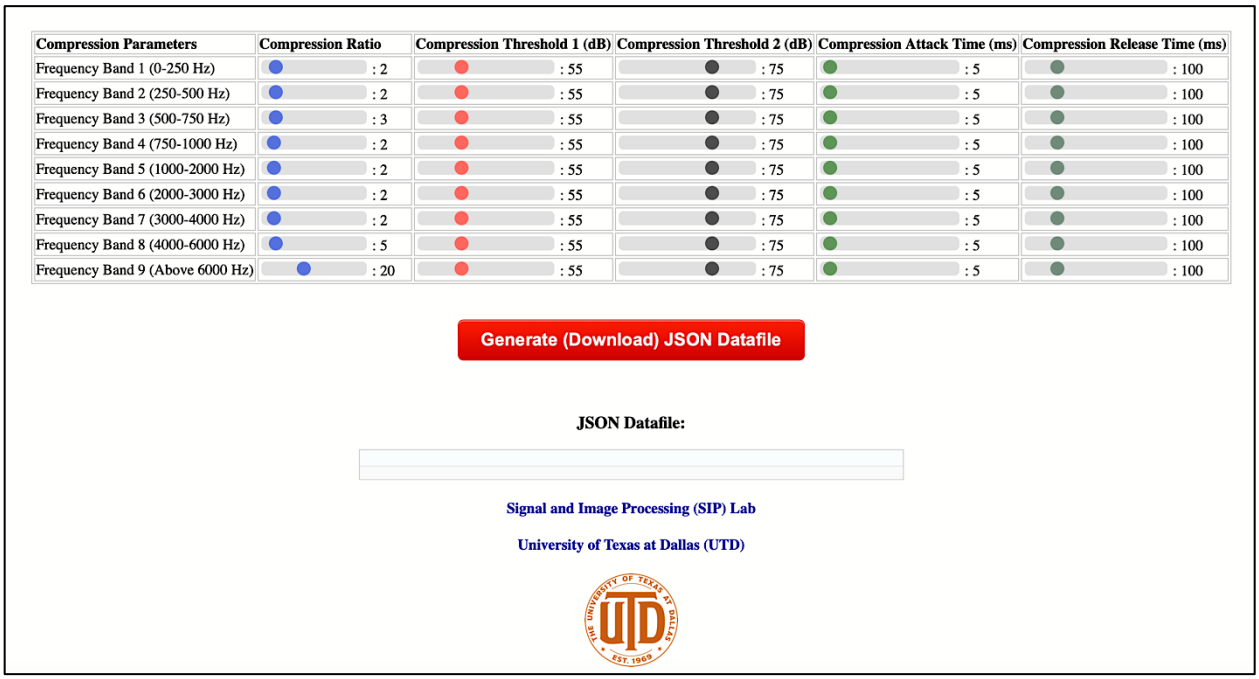
(b)

Fig. 5 - Compression curves

Then, each compression curve is expressed in terms of the following four parameters [3]:

- *Compression Threshold (CT)* – This parameter indicates the point after which the compression is applied.
- *Compression Ratio (CR)* - This parameter indicates the amount of compression.
- *Attack Time (AT)* – 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* - 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 default values of the attack and release times are set to 5ms and 100ms, respectively. The compression ratio of each frequency band for the compression thresholds of 55 dB and 75 dB SPL is computed based on the desired gains (see Fig. 6).



Compression Parameters	Compression Ratio	Compression Threshold 1 (dB)	Compression Threshold 2 (dB)	Compression Attack Time (ms)	Compression Release Time (ms)
Frequency Band 1 (0-250 Hz)	: 2	: 55	: 75	: 5	: 100
Frequency Band 2 (250-500 Hz)	: 2	: 55	: 75	: 5	: 100
Frequency Band 3 (500-750 Hz)	: 3	: 55	: 75	: 5	: 100
Frequency Band 4 (750-1000 Hz)	: 2	: 55	: 75	: 5	: 100
Frequency Band 5 (1000-2000 Hz)	: 2	: 55	: 75	: 5	: 100
Frequency Band 6 (2000-3000 Hz)	: 2	: 55	: 75	: 5	: 100
Frequency Band 7 (3000-4000 Hz)	: 2	: 55	: 75	: 5	: 100
Frequency Band 8 (4000-6000 Hz)	: 5	: 55	: 75	: 5	: 100
Frequency Band 9 (Above 6000 Hz)	: 20	: 55	: 75	: 5	: 100

Fig. 6 - Compression parameters in 9 frequency bands of DSL-v5

The button at the end of the web-based program as shown in Fig. 6 generates a JSON (JavaScript Object Notation) datafile which includes the parameters of the compression curves in a readable form (see Fig. 7). JSON is a popular web programming approach for transmitting data to other platforms.

The created JSON datafile is then transferred to the smartphone and placed inside the compression app folder to be used by it. Sections 2.4 and 3.4 provide more information regarding how to transfer the JSON datafile to the compression app for both Android and iOS smartphones.

JSON Datafile:


```
{
  "compressionSwitch": 1,
  "CompNumParams": 5,
  "compressionSettings": {
    "Band1": [
      2,
      55,
      75,
      5,
      100
    ],
    "Band2": [
      2,
      55,
      75,
      5,
      100
    ],
    "Band3": [
      3,
      55,
      75,
      5,
      100
    ],
    "Band4": [
      2,
      55,
      75,
      5,
      100
    ],
    "Band5": [
      2,
      55,
      75,
      5,
      100
    ],
    "Band6": [
      2,
      55,
      75,
      5,
      100
    ],
    "Band7": [
      2,
      55,
      75,
      5,
      100
    ],
    "Band8": [
      5,
      55,
      75,
      5,
      100
    ],
    "Band9": [
      20,
      55,
      75,
      5,
      100
    ]
  }
}
```


Fig. 7- Generated JSON datafile

Part 2

iOS

Section 1: Running the App

The iOS version of the compression 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 [5] for more details.

To open the app project, double click on the "Compression_App.xcodeproj" project in the "Compression_App_iOS" folder, refer to Fig. 8. To run the project, enable developer mode in iPhone if required (see [6]), 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. 8.

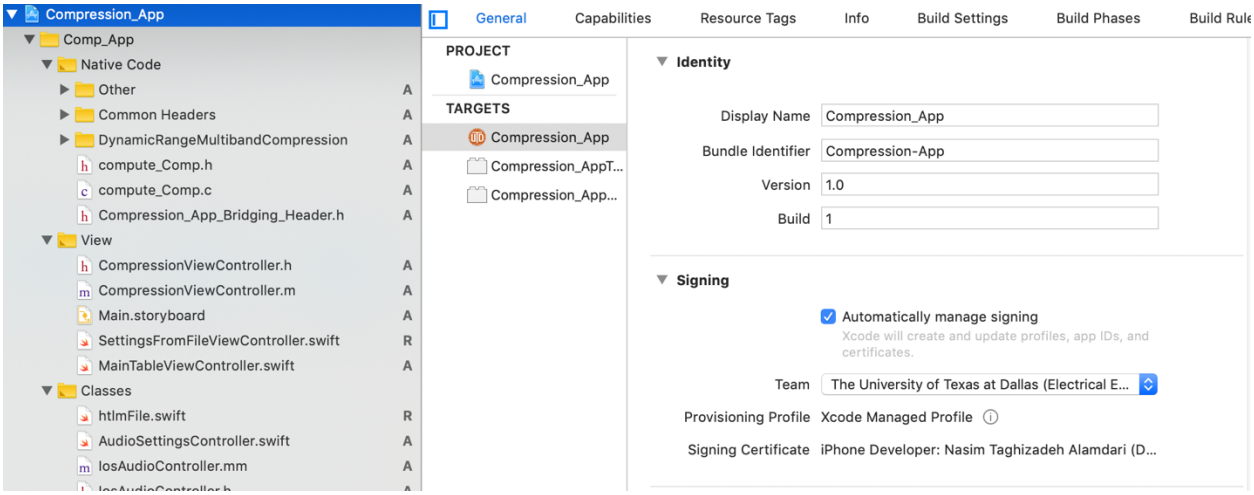


Fig. 8 - Signing with Apple Developer ID

Section 2: iOS GUI

This section covers the GUI of the developed compression app and its entries. The Graphical User Interface (GUI) consists of three views, see Fig. 9:

- Main View
- Set Settings from a File View
- Compression Settings View

2.1 Main View

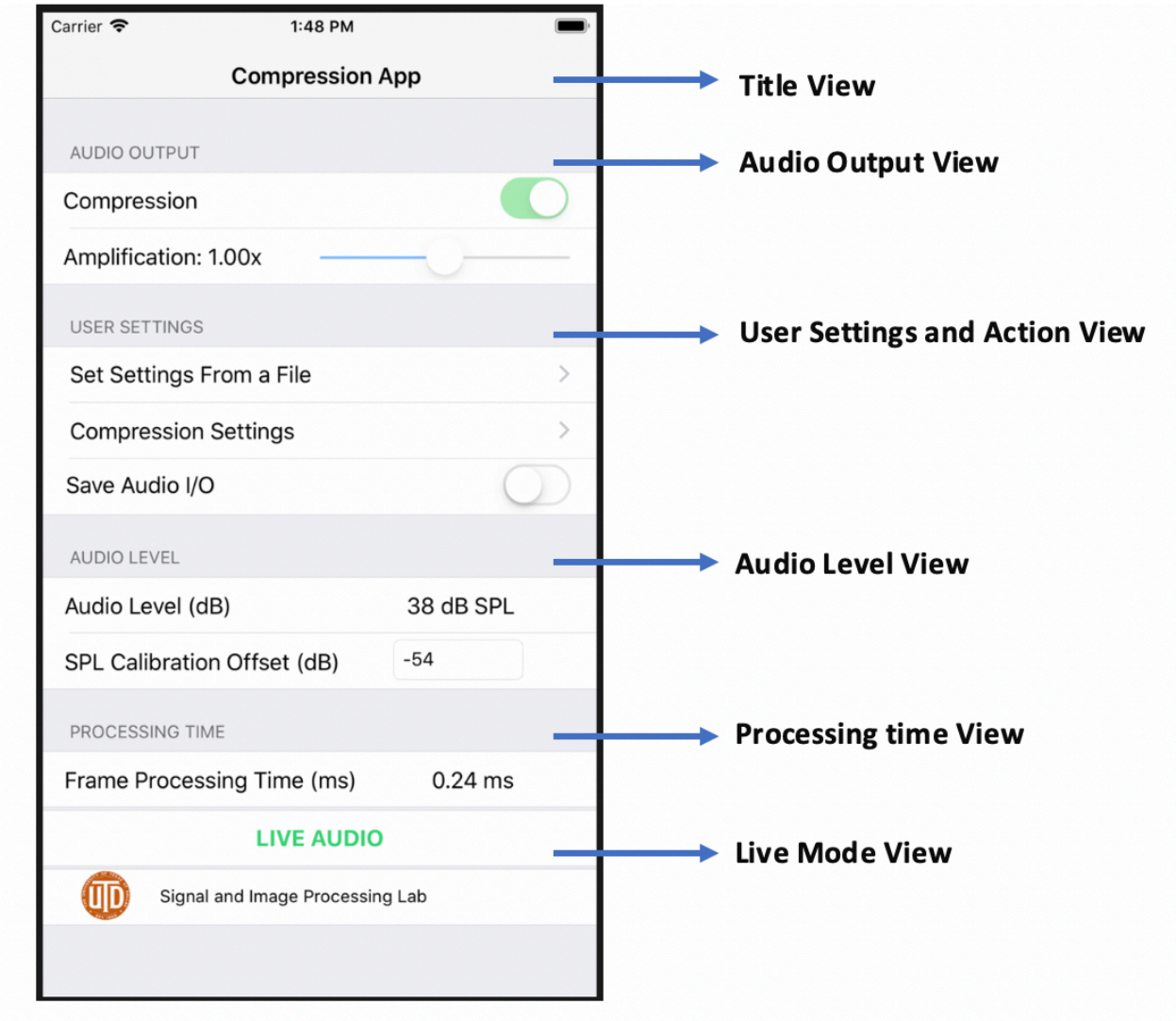


Fig. 9 - Compression iOS GUI: Main View

The Main view consists of 6 segments, see Fig. 9.

- Title View
- Audio Output View
- User Settings and Actions View
- Audio Level View
- Processing Time View
- Live Mode 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 compression module
- A slider to control the final amplification

2.1.3 User Settings

This part provides the following entries:

- Reading compression parameters from a JSON file
- Compression settings
- An option to save input/output audio signals

2.1.4 Audio Level

This part provides the following entries:

- **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).
- **Audio Level (dB):** This field shows the measured sound pressure level (SPL) in dB of audio signals using the calibration constant.

2.1.5 Processing Time

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

2.1.6 Live Mode

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

2.2 Set Settings from a file View

This part lists all the JSON files, see Fig. 10.

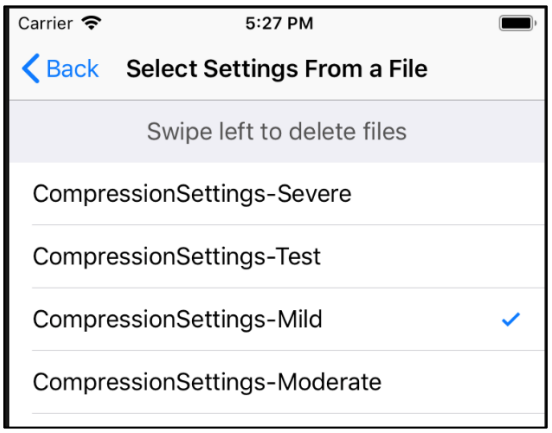


Fig. 10 - Select settings from a file

2.3 Compression Settings View

This view shows various settings for the compression app for the five frequency bands, see Fig. 11. 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 app:

- 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 complete view of the compression settings, see Fig. 11.

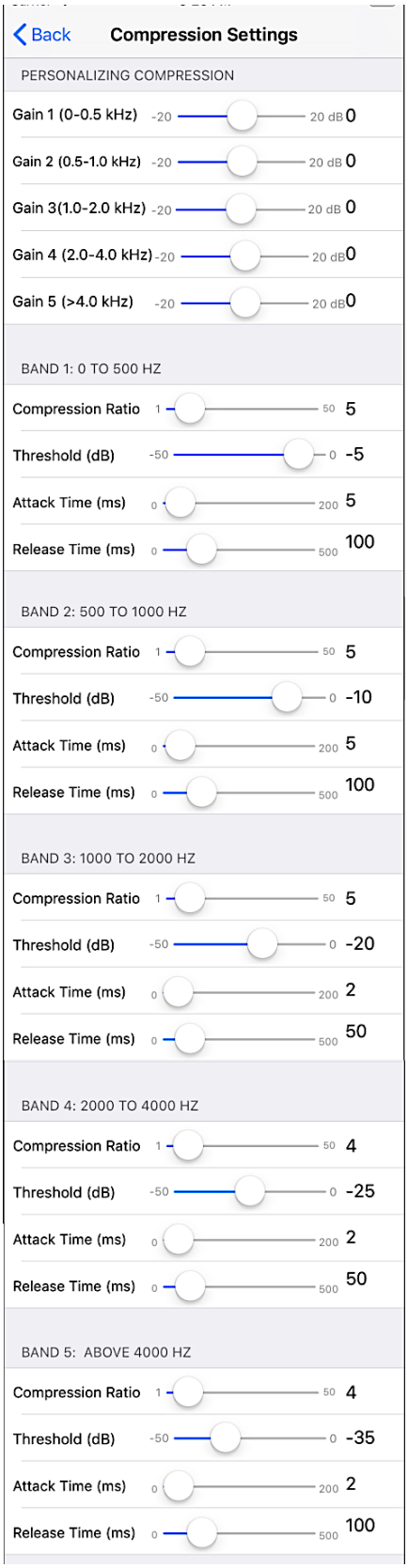


Fig. 11 - Compression app iOS GUI: Compression Settings view

Section 3: Code Flow

This section states the app code flow. The user can view the code by running "Compression_App.xcodeproj" in the folder "Compression-App" as shown in Fig. 1. The code is divided into 3 parts, see Fig. 12:

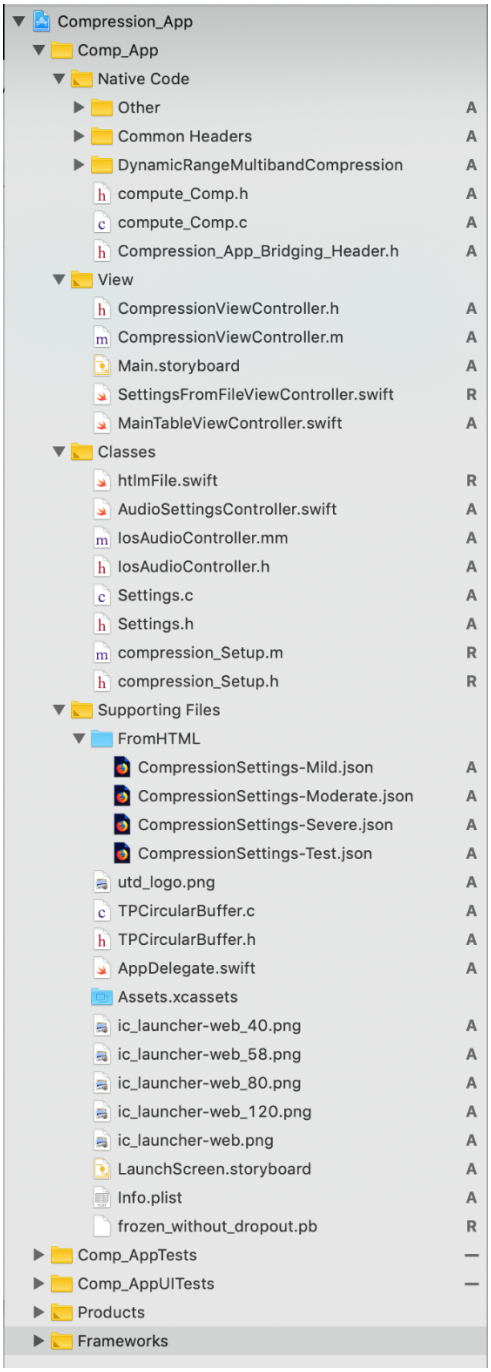


Fig. 12 - Compression app iOS version code flow

- 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:

- **compute_Comp:** This denotes the entry point of the native codes of the hearing aid modules. It initializes all the settings for compression module and then processes the incoming audio signal according to the signal processing pipeline described in [1].

- **DynamicRangeMultibandCompression:** This corresponds to the codes for the compression module, which is developed in MATLAB and then converted into C using the MATLAB Coder [7].

- **Common Headers:** This provides some common headers for compression module. Note that these files are generated by the MATLAB Coder by converting MATLAB codes into C codes.

- **Other:** This includes the following:
    - **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 2.1.
    - **filterCoefficients.h** -> This component includes filter coefficients for the FIR filter.
    - **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).
    - **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. 13.

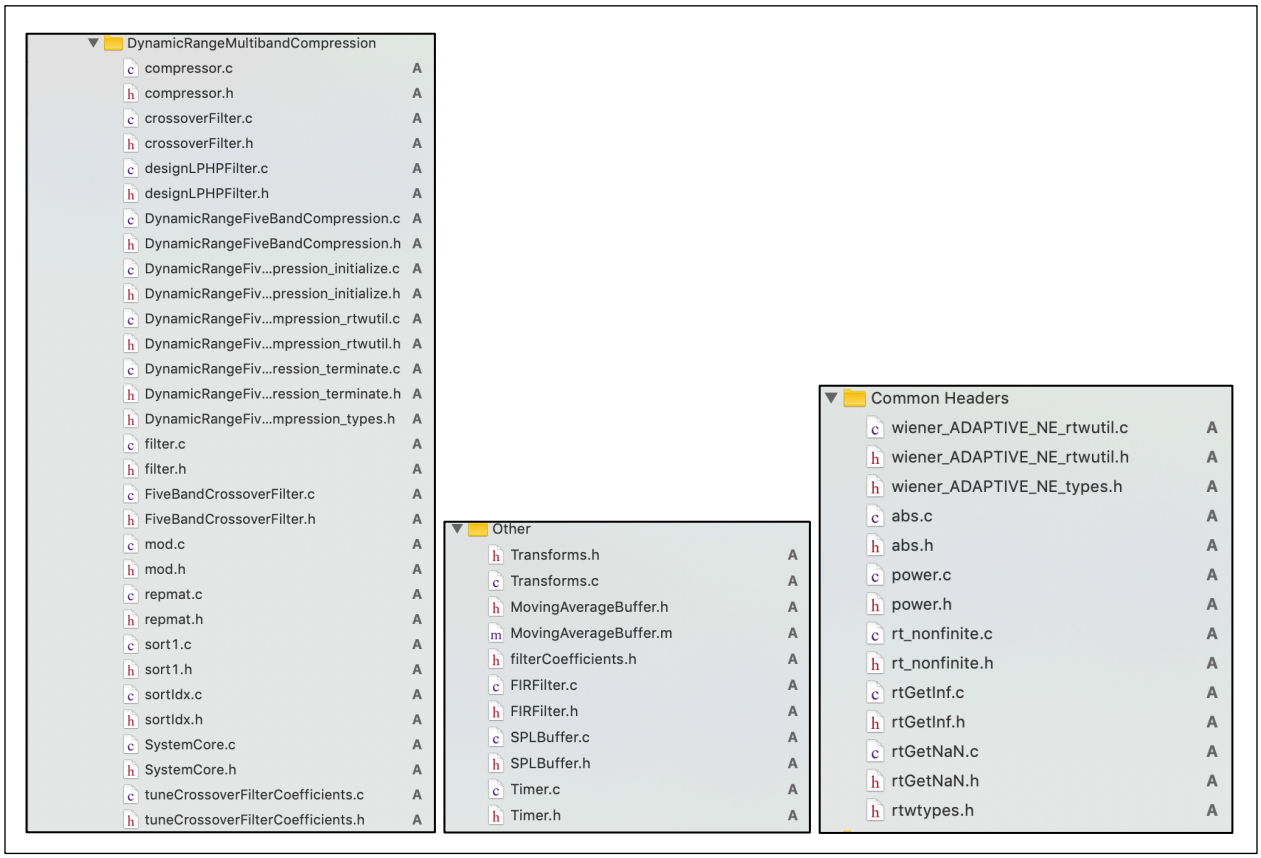


Fig. 13 - Further breakdown of native code modules in iOS version of the app

3.2 View:

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

- **Main.storyboard:** This provides the layout of the compression app GUI.
- **MainTableViewController:** This provides the GUI elements and actions of the main GUI view. This is developed using Swift [8].
- **SettingsFromFilesTableViewController:** This provides the GUI elements and actions to load, show the existing JSON data files and recognize selection of a JSON file. 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.
- **html_File:** This sets up reading, deleting and importing JSON files and load compression settings from a JSON file. This is written in Swift.

3.4 Supporting Files:

There are supporting files as shown in Fig. 14. Along with the app logo and launcher images, there are the following:

- **TPCircularBuffer:** This is an implementation of the circular buffer in [9], which is used in the compression app to obtain a desirable frame size for audio processing.
- **AppDelegate:** This is called when the app is launched and initializes AudioSettingController.
- **FromHTML:** This folder contains all the imported JSON files.

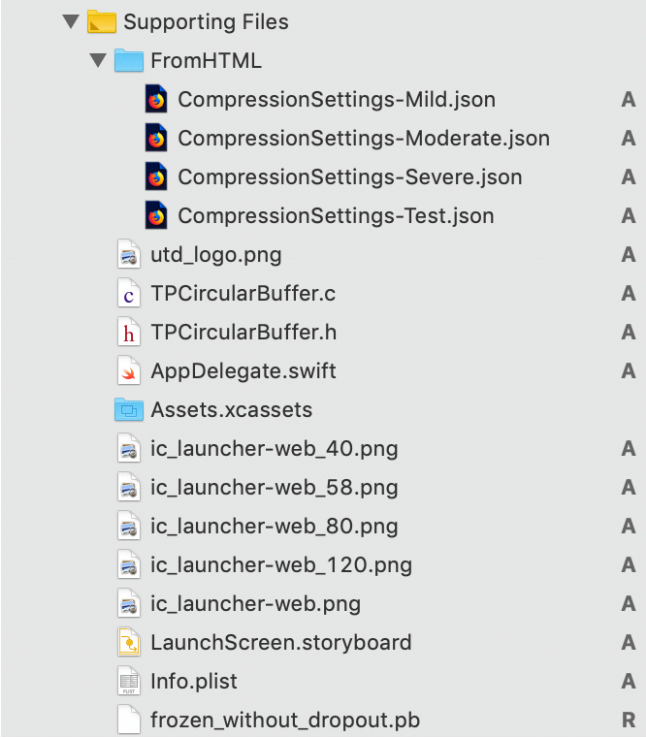


Fig. 14 - Supporting files

Section 4: Insert settings to iOS App

This section discusses how compression parameters files are handled in the app.

4.1 Gain File Format

A compression parameters file is a JavaScript Object Notation (JSON) text file that stores information about the compression parameters. The main JSON object in a compression file is in the format shown in Fig. 15 below. Note that the app expects the number of frequency bands to be exactly 9, and the length of each frequency band to be 5. Later, these 9 frequencies are converted to 5 to gain computational efficiency. The five elements in the array correspond to the compression parameters for the 9 frequency bands.

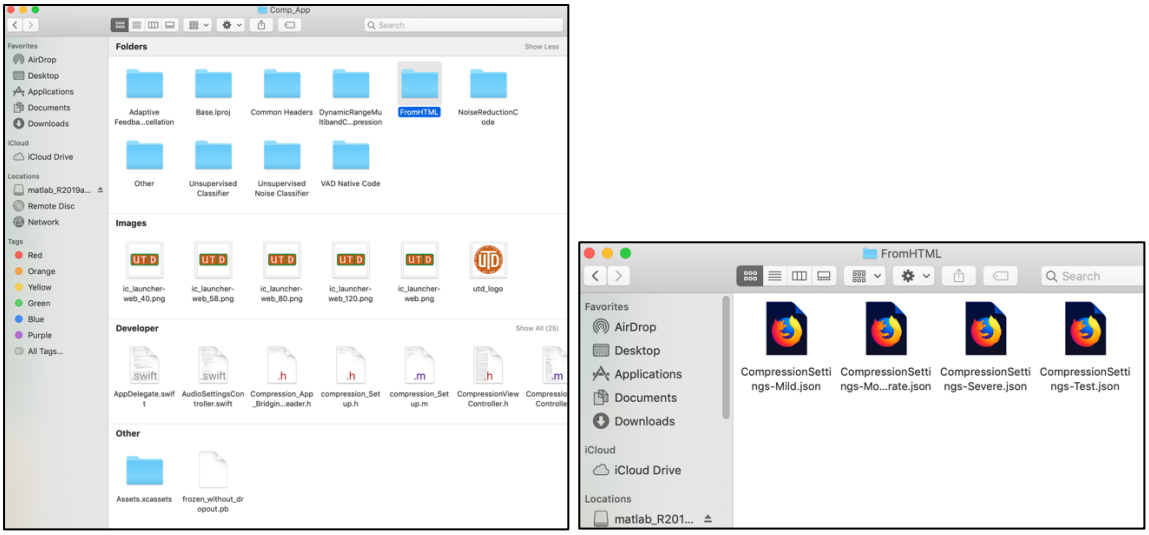
CompressionSettings-Mild.json

```
{
  "compressionSwitch": 1,
  "CompNumParams": 5,
  "compressionSettings": {
    "Band1": [
      2,
      55,
      75,
      5,
      100
    ],
    "Band2": [
      2,
      55,
      75,
      5,
      100
    ],
    "Band3": [
      2,
      55,
      75,
      5,
      100
    ],
    "Band4": [
      2,
      55,
      75,
      5,
      100
    ],
    "Band5": [
      1,
      55,
      75,
      5,
      100
    ],
    "Band6": [
      2,
      55,
      75,
      5,
      100
    ],
    "Band7": [
      1,
      55,
      75,
      5,
      100
    ],
    "Band8": [
      2,
      55,
      75,
      5,
      100
    ],
    "Band9": [
      1,
      55,
      75,
      5,
      100
    ]
  }
}
```

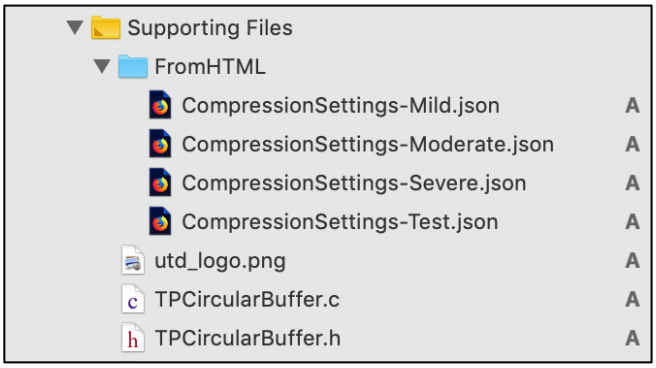
Fig. 15 - Gains file format

4.2 Importing a JSON File

One needs to transfer the desired JSON file to the “FromHTML” folder (see Fig. 16(a) and 16(b). Open the code in Xcode and “clean” and “build” the code. When running the app on a smartphone device, new JSON files will get appeared.



(a) (b)



(c)

Fig. 16. Importing a JSON file to iOS Compression app.

Section 5: Modularity and Modification

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

- **Available bandwidth:** The app is developed in such a way that the hearing aid compression module used 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 audio 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 compression app is straightforward and includes:
    - Compression of the incoming audio frames.

The declarations and definitions of initializations and destructions as well as the main function that call the module appear in the "compute_Comp" source files (.h and .c).

- **Replace or Add modules:** To replace or add module(s), include/remove:
    - Corresponding headers in "compute_Comp.h".
    - Set variables in the "CompressionVars" structure (same header).
    - Initialize variables inside the "init_Compression" function as defined in "compute_Comp.c".
    - Destruct variables inside the function named "destroy_Compression" for optimized memory allocation and usage.
    - Function(s) that calls an updated module at the proper place inside the main function is named "perform_Compression".

- **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:
    - **AudioOutput controls:**
        - "settings->compressionOutputType" saves the switch status for compression.
        - "settings->amplification" saves the 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->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.

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 [10] and [11]. The user can change them according to specific requirements. To set these flags, follow the steps noted below, see Fig. 17.
  - 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.

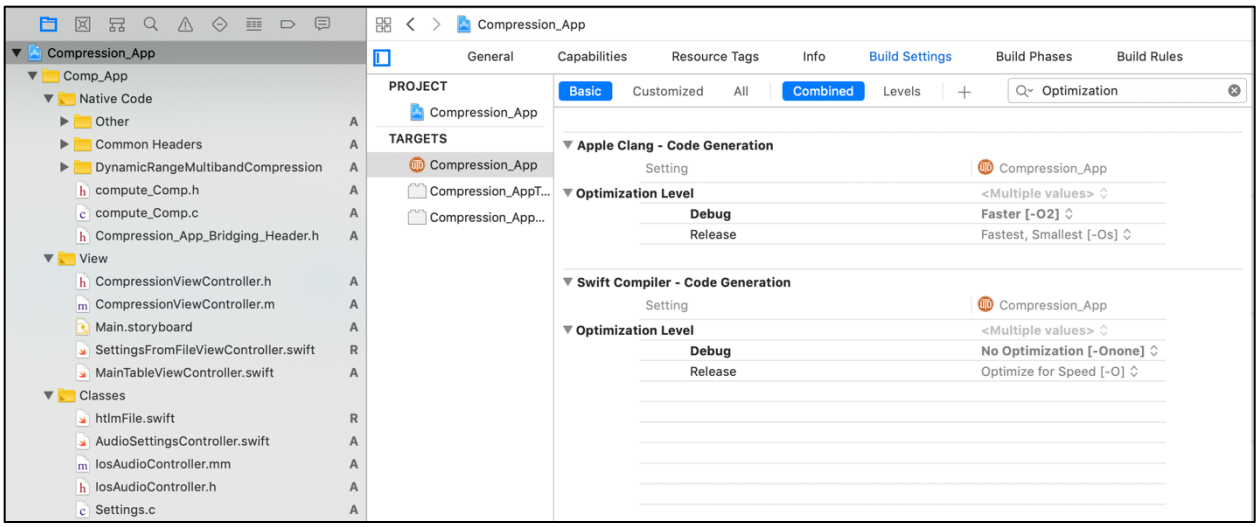


Fig. 17 - Settings optimization level in Xcode for the compression app iOS version

Part 3

Android

Section 1: Running the App

The Android version of the compression app was developed using Android Studio (Version 3.1). To run the Android version of the Compression app, it is necessary to have Superpowered SDK which can be obtained from the link at [12]. 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 compression app and its entries. The GUI consists of three views:

- Main View
- Set Settings from File View
- Compression Settings View

2.1 Main View

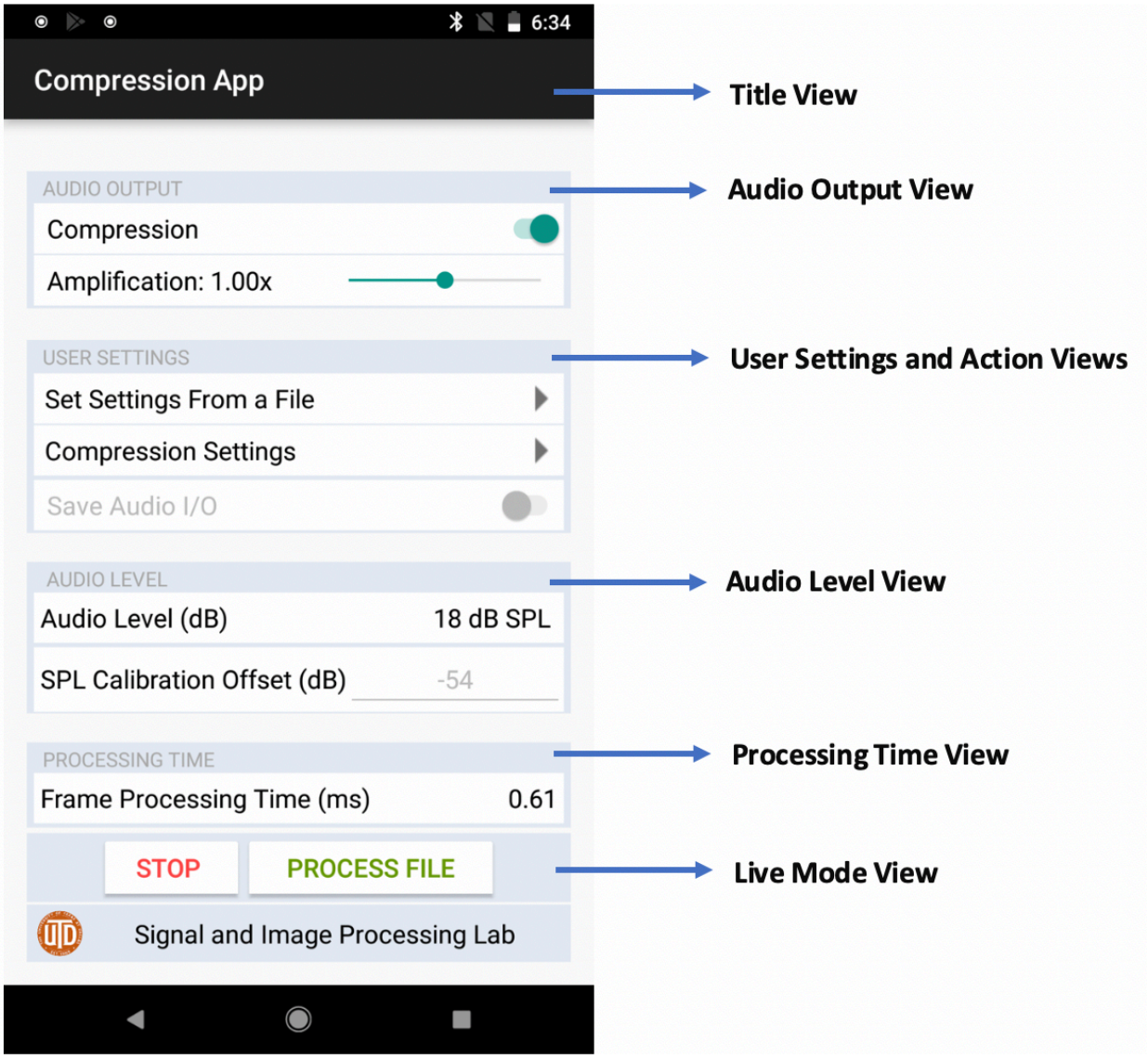


Fig. 18 - Compression iOS GUI: Main view

The Main view consists of 6 segments, see Fig. 18.

- Title View
- Audio Output View
- User Settings and Actions View
- Audio Level View
- Processing Time View
- Live Mode 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 compression module
- A slider to control final amplification

2.1.3 User Settings

This part provides the following entries:

- Reading compression parameters from a JSON file
- Compression settings
- An option to save input/output audio signals

2.1.4 Audio Level

This part provides the following entries:

- **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).
- **Audio Level (dB):** This field shows the measured sound pressure level (SPL) in dB of audio signals using the calibration constant.

2.1.5 Processing Time

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

2.1.6 Live Mode

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

2.2 Set Settings from a file View

This part lists all the JSON files, see Fig. 19.

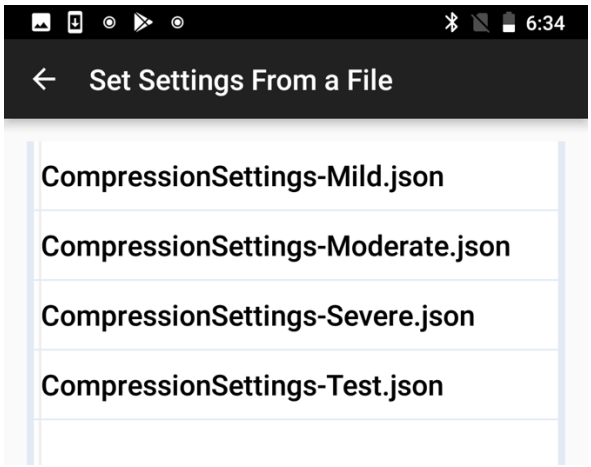


Fig. 19 - Select settings from a JSON file

2.3 Compression Settings View

This view shows various settings for the compression module for the five frequency bands, see Fig. 20. 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 app:

- 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 complete view of the compression settings, see Fig. 20.

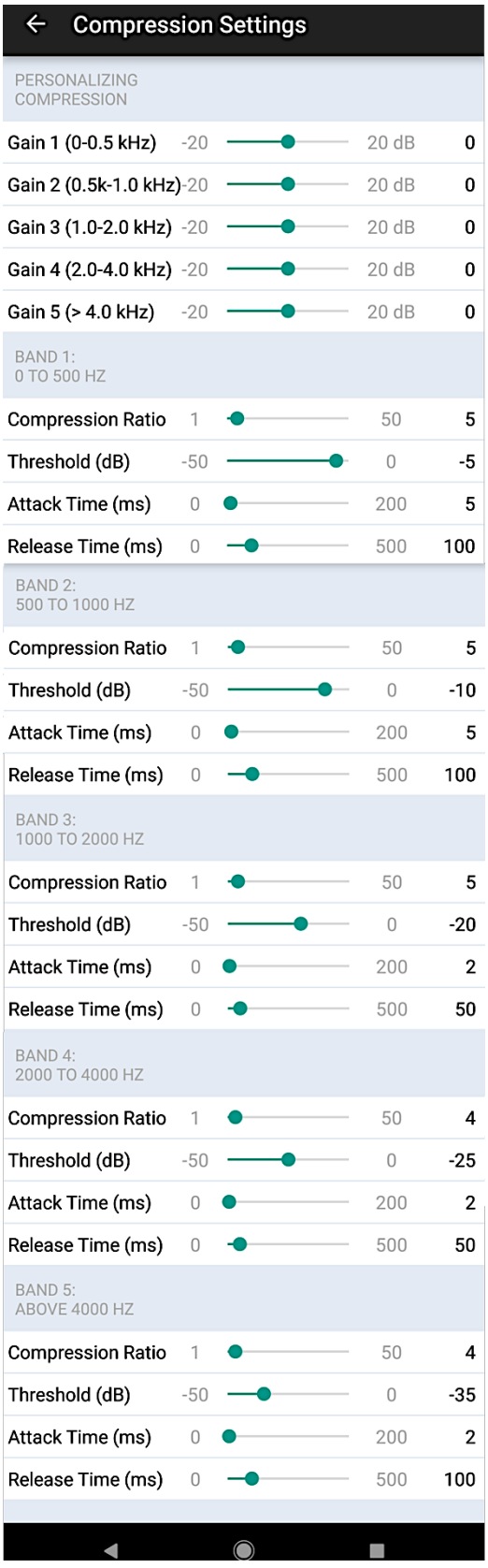


Fig. 20 - Compression app Android GUI: compression Settings view

Section 3: Code Flow

This section covers the compression app code flow. The folder organization of the app appears as shown in Fig. 21.

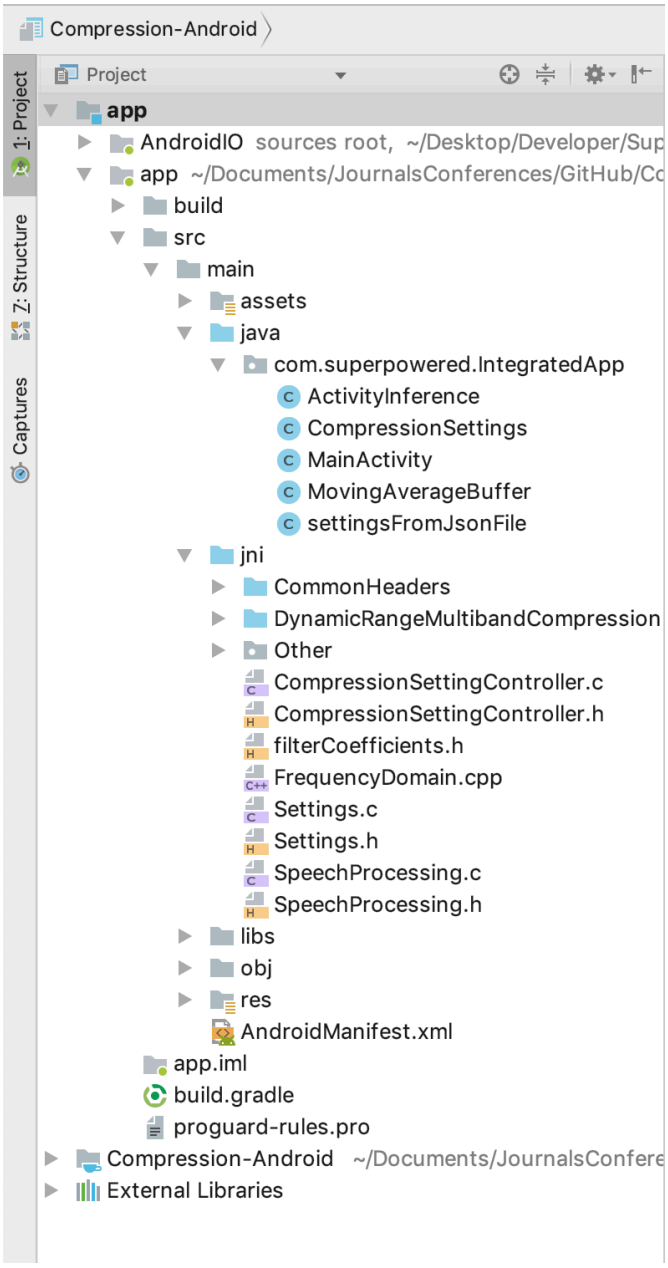


Fig. 21 - Compression app Android version: project organization

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:** This java folder contains three ".java" files, which handle all the operations of the app and provides the link between the GUI and the native code.
    - **MainActivity:** This contains the GUI elements and controls the Main view of the app.
    - **SettingsFromJSONFile:** This contains the GUI elements and controls the loading, showing, and selection existing JSON files.
    - **CompressionSettings:** This contains the GUI elements and controls the Compression Settings 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 compression app appearing in these three .xml files:
    - **activity_main.xml:** This file provides the layout for the Main view.
    - **activity_compression_settings.xml:** This file provides the layout for the Compression Settings view.
    - **activity_settings_from_json_file.xml:** This file provides the layout for the SettingsFromJSONFile 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 compression module 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.

- **DynamicRangeMultibandCompression:** This corresponds to the codes for the compression module, which is developed in MATLAB and then converted into C using the MATLAB Coder [7].

- **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
    - **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 compression native code module along with the *common header*, *Feature Extraction*, and *Other* folders are shown in Fig. 22.

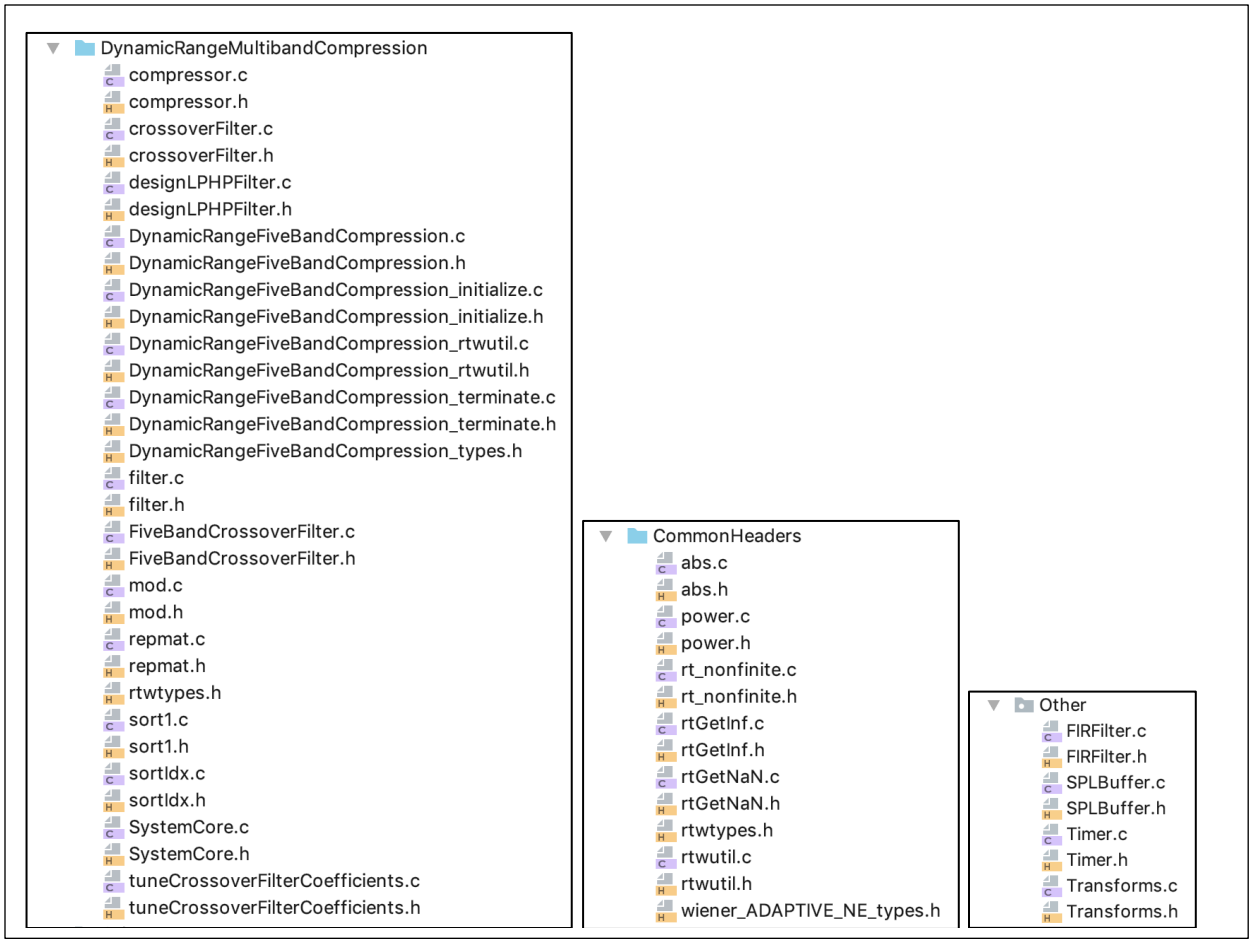


Fig. 22 - Further breakdown of native code modules in Android

Section 4: Insert settings to Android App

This file can be transferred to the Android version of the compression app via email or by connecting the smartphone device to a desktop/laptop with cable.

- **Via an Email**: Move the JSON file to Settings ->Storage -> Files -> CompressionApp_Android folder on the smartphone.
- **Via a Cable**

Make sure that the smartphone is set to allow computer access to the smartphone by following these steps noted below:

Settings -> Connected Device -> USB -> select “Transfer files”.

- **Via Cable from Windows**: Open the smartphone’s folder from “This PC” and put the JSON file in “CompressionApp_Android” folder as shown in Fig. 23.

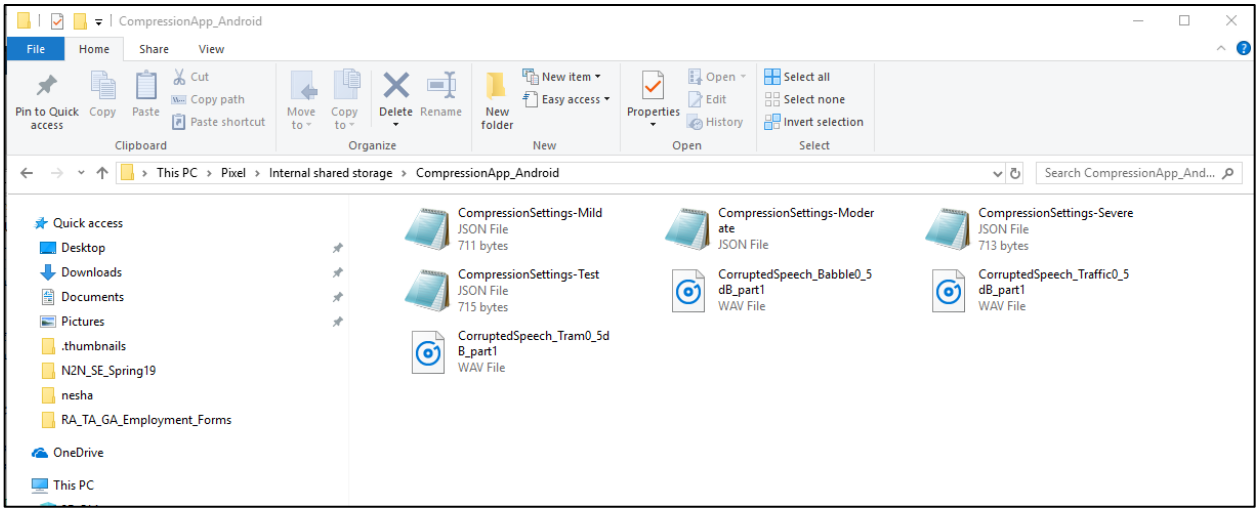


Fig. 23 - Steps to import a JSON file to Android version of the compression app via a cable

- **Via Cable from MacOS**:
  - Install “Android File Transfer” software on macOS.
  - Connect the smartphone to macOS via a cable.
  - Open “Android File Transfer”.
  - Transfer a desired JSON file to “CompressionApp_Android” folder.

Section 5: Modularity and Modification

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

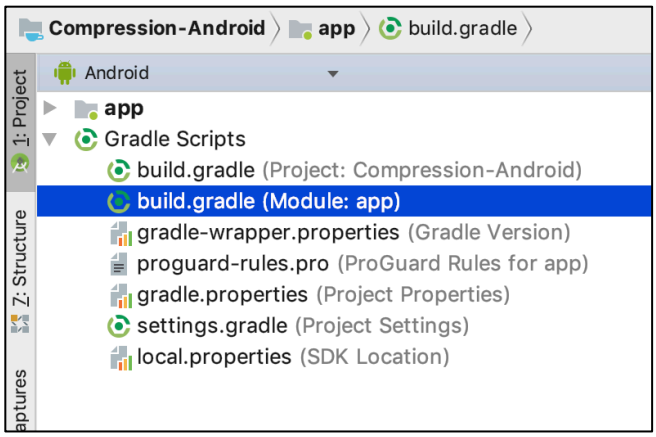
- **Available bandwidth:** The app is developed in such a way that the hearing aid compression fitting module used can be replaced with other similar modules. It is also possible to add new modules if the available processing time bandwidth is not exceeded. The processing time bandwidth can be obtained from the sampling frequency and audio i/o buffer size of the Android smartphone used. To have the lowest audio latency, many Android smartphones process 192 samples of an incoming audio signal at 48kHz sampling frequency (refer to [13] for more information on 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 compression app is straightforward and is as follows:
    - Applying compression onto the incoming audio frames

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

- **Replace or Add modules:** To replace or add module(s), the following need to be done:
    - Remove corresponding headers in "SpeechProcessing.h".
    - Set variables in the "CompressionVars" structure (same header).
    - Initialize variables inside the "init_Compression" function as defined in "SpeechProcessing.c".
    - Destruct variables inside the function named "destroy_Compression" for optimized memory allocation and usage.
    - Function(s) that calls an updated module at the proper place inside the main function is named "perform_Compression".

All the native codes and headers folder path need to be included in the "build.gradle" file inside "android.sources.main.jni { exportedHeaders {….} } as shown in Fig. 24. It is required to add the path of each folder as well as the subfolders separately.




```
    android.sources.main.jni {
        source {
            srcDir "src/main/jni"
            srcDir "${superpowered_sdk_path}/AndroidIO"
        }
        exportedHeaders {
            srcDir "src/main/jni/"
            srcDir "src/main/jni/CommonHeaders"
            srcDir "src/main/jni/DynamicRangeMultibandCompression"
            srcDir "src/main/jni/Other"

        }
        dependencies {
            library "superpowered" linkage "static"
            // this is where you attach the "superpowered" static library to your app
        }
    }
}
```

Fig. 24 - Add native folder paths

- **Data transaction between the GUI and Native Code:** For the Android version of the developed compression app, Java Native Interface (JNI) is used to interface with native codes in the Java environment. This approach is described in [14-16]. 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, SettingsFromJSONFile, 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.

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

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:

- **AudioOutput controls:**
  - "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->dbpower" provides the dB SPL power computed in "Transform" and averaged using "SPLBuffer" over the "guiUpdateInterval" time.

  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.

- **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.

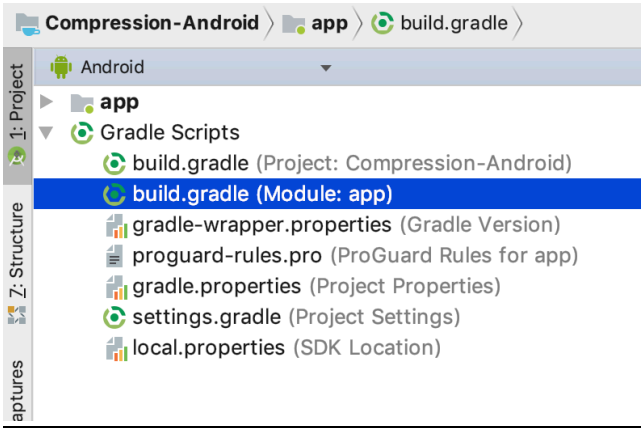
- **Compiler optimization:** Using compiler optimizations improves the code performance and/or size depending on which optimization level is used. For the Android version of the

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 [10] and [11]. The user can change them according to specific requirements. To set these flags, follow the steps as illustrated in Fig. 25.

- Under "Android" of the project view explorer of Android Studio, select build.gradle (Module: app) under Gradle Scripts, see left side of Fig. 25.
- At the right side of Fig. 25, 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. 25 - Settings optimization level in Android Studio for the compression app Android version

Timing difference between iOS and Android versions of the compression app

The low-latency audio i/o setup for iOS is done using the software package CoreAudio API [17] and for Android using the software package Superpowered SDK [12]. 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 shown below:

Table 1: Timing difference between the iOS and Android versions of the compression app

Version of the Compression 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

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

from “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 for Bluetooth connection than the Android platform.

References:

[1] N. Alamdari, E. Lobarinas, and N. Kehtarnavaz, "An Educational Tool for Hearing Aid Compression Fitting via a Web-Based Adjusted Smartphone App", *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Brighton, United Kingdom, May 2019.

[2] S. Pigeon, https://hearingtest.online/, 2017.

[3] Starkey, Compression Handbook, https://starkeypro.com/pdfs/The_Compression_Handbook.pdf, 2017.

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

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

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

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

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

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

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

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

[12] http://superpowered.com/

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

[14] N. Kehtarnavaz, F. Saki, and A. Duran, *Anywhere-Anytime Signals and Systems Laboratory: From MATLAB to Smartphones, 2nd Edition*, Morgan and Claypool Publishers, 2018.

[15] N. Kehtarnavaz, A. Sehgal, and S. Parris, *Smartphone-Based Real-Time Digital Signal Processing, 2nd Edition* Morgan and Claypool Publishers, 2018.

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

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