

User's Guide: How to Run C/MATLAB Codes on Android Smartphones as Open Source and Portable Research Platforms for Hearing Improvement Studies

N. KEHTARNAVAZ AND A. SEHGAL
UNIVERSITY OF TEXAS AT DALLAS

MARCH 2017

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

Table of Contents

| | |
|---|-----------|
| INTRODUCTION | 3 |
| USER'S GUIDE ACCOMPANYING CODES | 3 |
| SECTION 1: ANDROID SOFTWARE TOOLS | 5 |
| 1.1 JAVA JDK | 5 |
| 1.2 ANDROID STUDIO WITH NATIVE DEVELOPMENT KIT | 5 |
| 1.3 ENVIRONMENT VARIABLE CONFIGURATION | 7 |
| 1.4 ANDROID STUDIO CONFIGURATION | 8 |
| SECTION 2: RUNNING C CODES AS ANDROID APPS | 11 |
| 2.1 PROGRAMMING LANGUAGE | 11 |
| 2.2 CREATING JAVA SHELL | 11 |
| 2.3 CREATING GUI | 13 |
| SECTION 3: CONVERTING MATLAB CODES TO C | 18 |
| 3.1 CREATING A MATLAB SCRIPT | 18 |
| 3.2 GENERATING C CODE USING MATLAB CODER | 18 |
| 3.3 RUNNING C CODE AS ANDROID APP | 22 |
| SECTION 4: I/O HARDWARE DEPENDENCIES | 27 |
| 4.1 USING HARDWARE PREFERRED SETTINGS | 27 |
| 4.2 MAINTAINING MICROPHONE CONSISTENCY | 27 |
| SECTION 5: REAL-TIME I/O IMPLEMENTATION | 28 |
| 5.1 CREATING AUDIO I/O APP | 28 |

Introduction

This user's guide covers the steps one needs to take in order to run C/MATLAB algorithms on Android mobile devices as open source and portable research platforms for hearing improvement studies.

The first section covers the software tools required for algorithm implementation on Android devices. In the second section, it is shown how to run C codes on Android devices. The third section discusses the use of the MATLAB Coder for converting MATLAB codes into C codes. The fourth section covers hardware dependencies that users will need to be aware of when implementing algorithms on different Android devices having different i/o characteristics. Finally, the fifth section describes frame-based processing for real-time operation.

User's Guide Accompanying Codes

The accompanying codes for this user's guide consist of the following (see Fig. 1):

- "codegen" – This folder includes the code generated by the MATLAB coder for the MATLAB script "fibonacci.m".
- "fibonacci_testbench.m" – This file is the testbench MATLAB script associated with "fibonacci.m".
- "FrequencyDomain" – This folder includes the code solution for section 5.
- "TestApp" – This folder includes the code solution for sections 2 and 3.

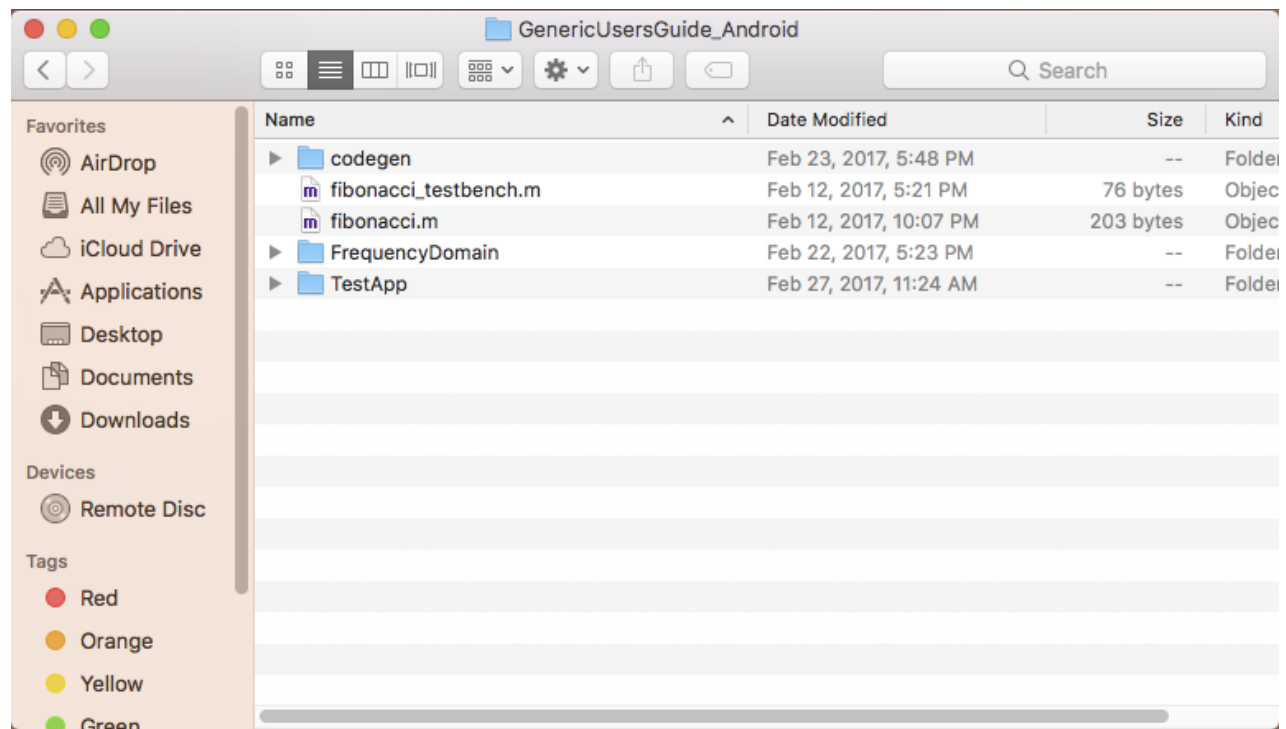


Fig. 1

Section 1: Android Software Tools

Android is an open-source operating system developed by Google for mobile phones and tablets. The Android apps are normally coded in Java. In this section, it is shown how to set up the Android Studio IDE (Integrated Development Environment) for developing Android apps.

1.1 Java JDK

Since Android apps require Java, the Java Development Kit (JDK) needs to be first installed on your computer. The latest version of JDK can be downloaded from this link:

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

The latest version of the JDK can be installed using the “Java Platform (JDK)”, shown in Fig. 2. Click download and select the appropriate downloaded package.

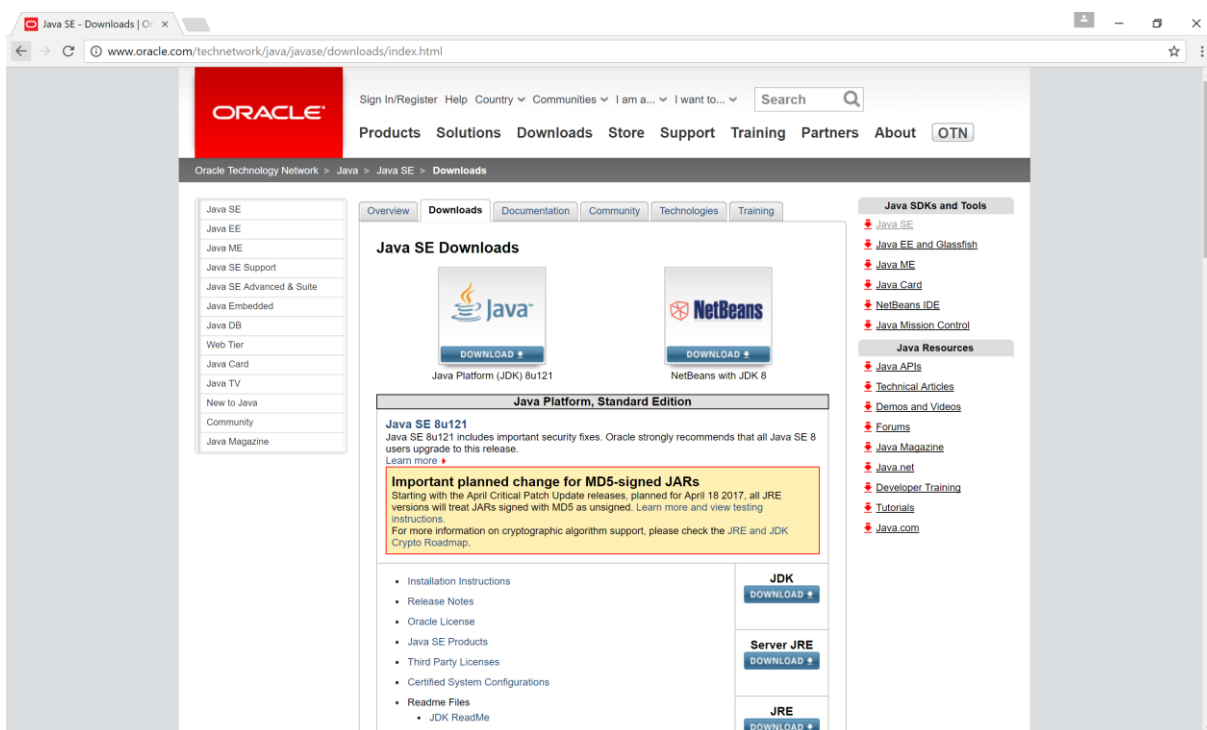


Fig. 2

1.2 Android Studio with Native Development Kit

It is recommended to create a common directory for installation of the Android software tools. Here, “C:\Android” is specified as the directory for the installation of all Android development related files. For Mac and Linux operating systems, a similar directory needs to get created and the same steps to be followed.

The recommended IDE for creating Android apps is Android Studio. It is available at the following link:

<https://developer.android.com/studio/index.html>

Along with Android Studio, the Native Development Kit (NDK) needs to be installed which allows incorporating C codes into the Android environment. The NDK can be downloaded from the following link:

<https://developer.android.com/ndk/downloads/index.html>

Depending on which operating system the above are installed, the corresponding Android Studio and NDK need to be placed into the directory created for the Android app development.

Install Android Studio first by running the installer. In the “Install Locations” page of the installation wizard, change the installation directory of Android Studio as shown in Fig. 3.

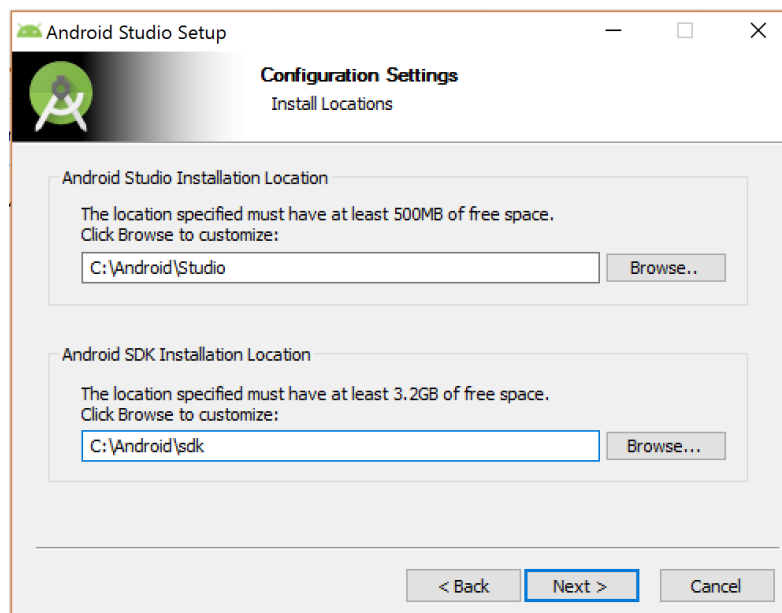


Fig. 3

After the installation is finished, deselect “Start Android Studio”. Next, extract the NDK in the “C:\Android” folder. After the extraction is complete, rename the “android-ndk-<version>” file to ndk. Finally, make sure “C:\Android” has both a sdk and an ndk folder.

1.3 Environment Variable Configuration

Before running Android Studio for the first time, the system environment needs to be set up by adding the SDK platform-tools folder to the system path variable and setting the variables to define the Android Virtual Device (AVD) storage location as well as the locations for the Android SDK and NDK. The steps listed here are for the Windows 10 operating system. Similar steps need to be followed for other operating systems.

On your desktop, right click on the Computer icon and select Properties. Next, open the Advanced system settings menu and click on Environment Variables at the bottom of the Advanced tab, see Fig. 4. Then, create new system variables by clicking the “New...” button below the System variables section, shown in Fig. 5. There are three new system variables that need to be set: “ANDROID_SDK_HOME” with the value: “C:\Android”, “ANDROID_SDK_ROOT” with the value: “%ANDROID_SDK_HOME%\sdk”, and “ANDROID_NDK_HOME” with the value “%ANDROID_SDK_HOME%\ndk”.

Then, add the following text as a new edit to your system path variable “%ANDROID_SDK_ROOT%\platform-tools” as shown in Fig. 6. The modifications are now complete and the settings menus can be closed.

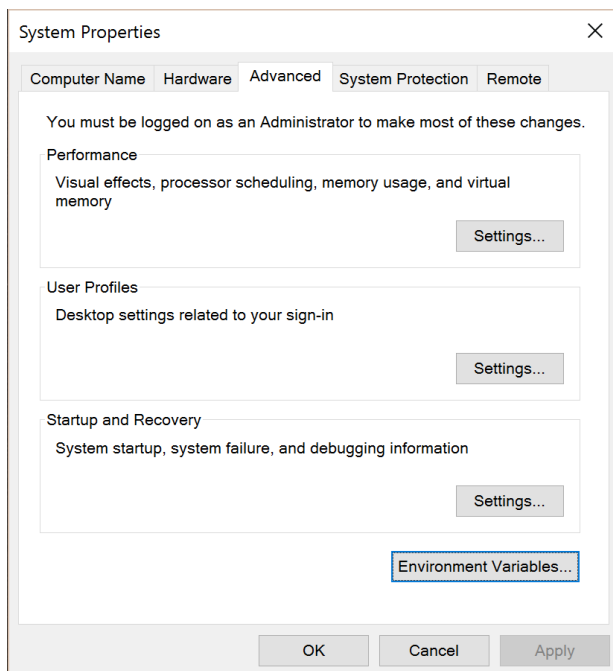


Fig. 4

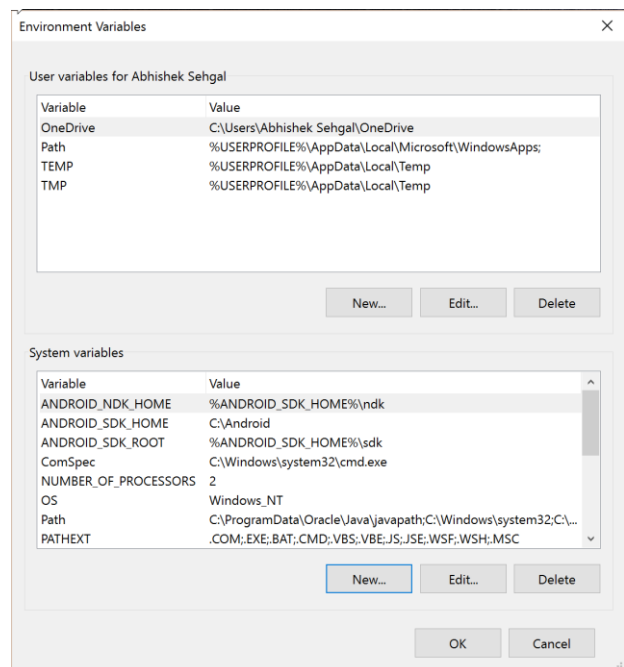


Fig. 5

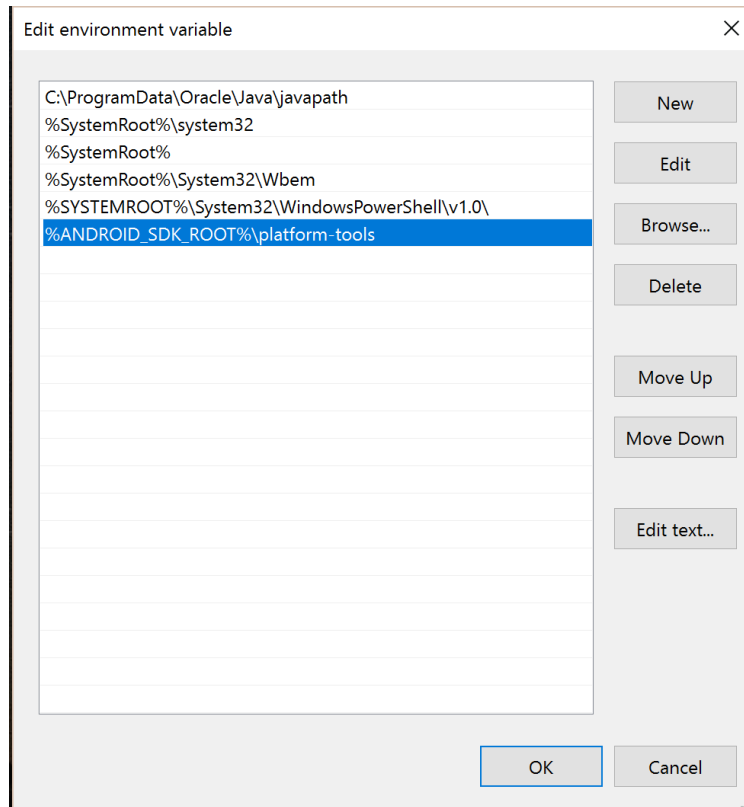
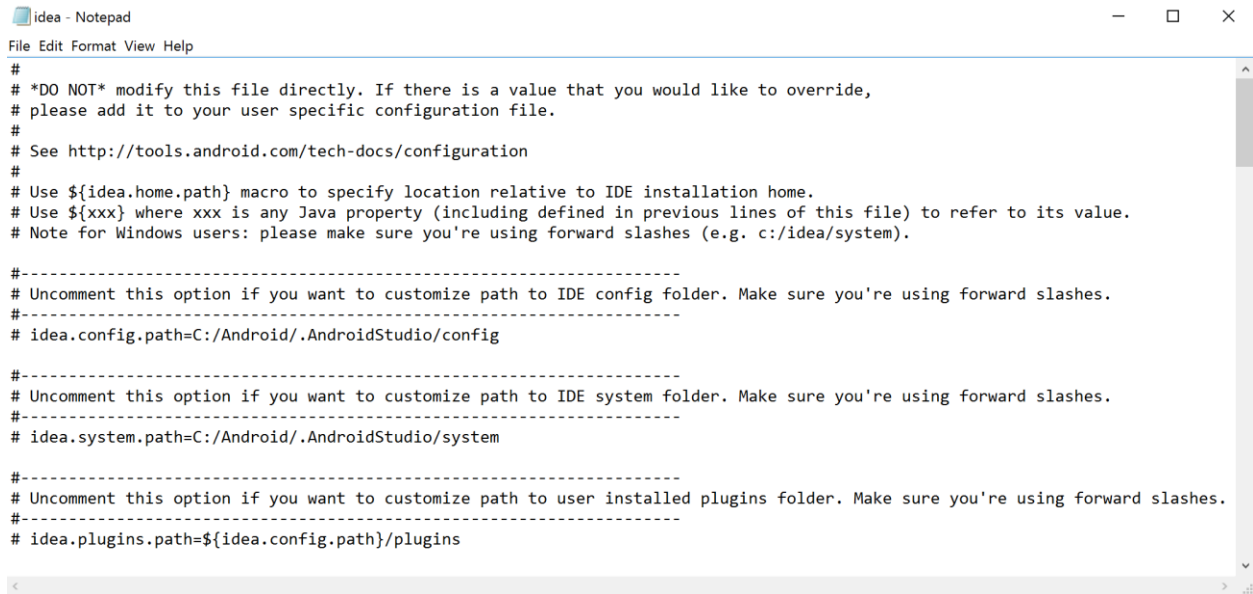


Fig. 6

1.4 Android Studio Configuration

Navigate to the “C:\Android\Studio\bin” directory and edit the file “idea.properties” with a text editor. Uncomment the lines “idea.config.path” and “idea.system.path” and replace `${idea.home}` with “C:/Android” for both of these lines. The final lines need to appear as shown in Fig. 7. At this time, also set up a shortcut to either `studio.exe` or `studio64.exe` on your desktop, as this is the main executable for Android Studio.

Android Studio may now be started for the first time using the shortcut just created. The tool will prompt to import settings and check for the correct Java JDK. At the “Install Type” screen, select the “Custom” option and click Next. On the “SDK Components Setup” screen, verify that the Android SDK Location is properly detected as “C:\Android\sdk”. If it is correct, click “Finish”, which will cause checking for any available updates to Android Studio. When this is done, the Android Studio home screen should appear as shown in Fig. 8.



```
#
# *DO NOT* modify this file directly. If there is a value that you would like to override,
# please add it to your user specific configuration file.
# See http://tools.android.com/tech-docs/configuration
#
# Use ${idea.home.path} macro to specify location relative to IDE installation home.
# Use ${xxx} where xxx is any Java property (including defined in previous lines of this file) to refer to its value.
# Note for Windows users: please make sure you're using forward slashes (e.g. c:/idea/system).

#-----
# Uncomment this option if you want to customize path to IDE config folder. Make sure you're using forward slashes.
#-----
# idea.config.path=C:/Android/.AndroidStudio/config
#-----
# Uncomment this option if you want to customize path to IDE system folder. Make sure you're using forward slashes.
#-----
# idea.system.path=C:/Android/.AndroidStudio/system
#-----
# Uncomment this option if you want to customize path to user installed plugins folder. Make sure you're using forward slashes.
#-----
# idea.plugins.path=${idea.config.path}/plugins
```

Fig. 7

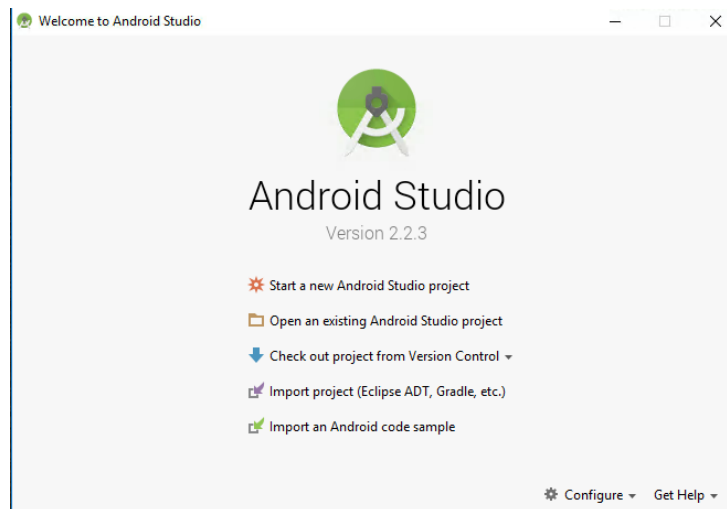


Fig. 8

Now run the “SDK Manager”, whose entry can be found by clicking on the “Configure” option. The “SDK Manager” will automatically select any components of the Android SDK that need updating, as illustrated in Fig. 9. From this menu, additional system images for emulation and API (Application Program Interface) packages for future Android versions can be added. Click the “Install” option and allow the update process to complete.

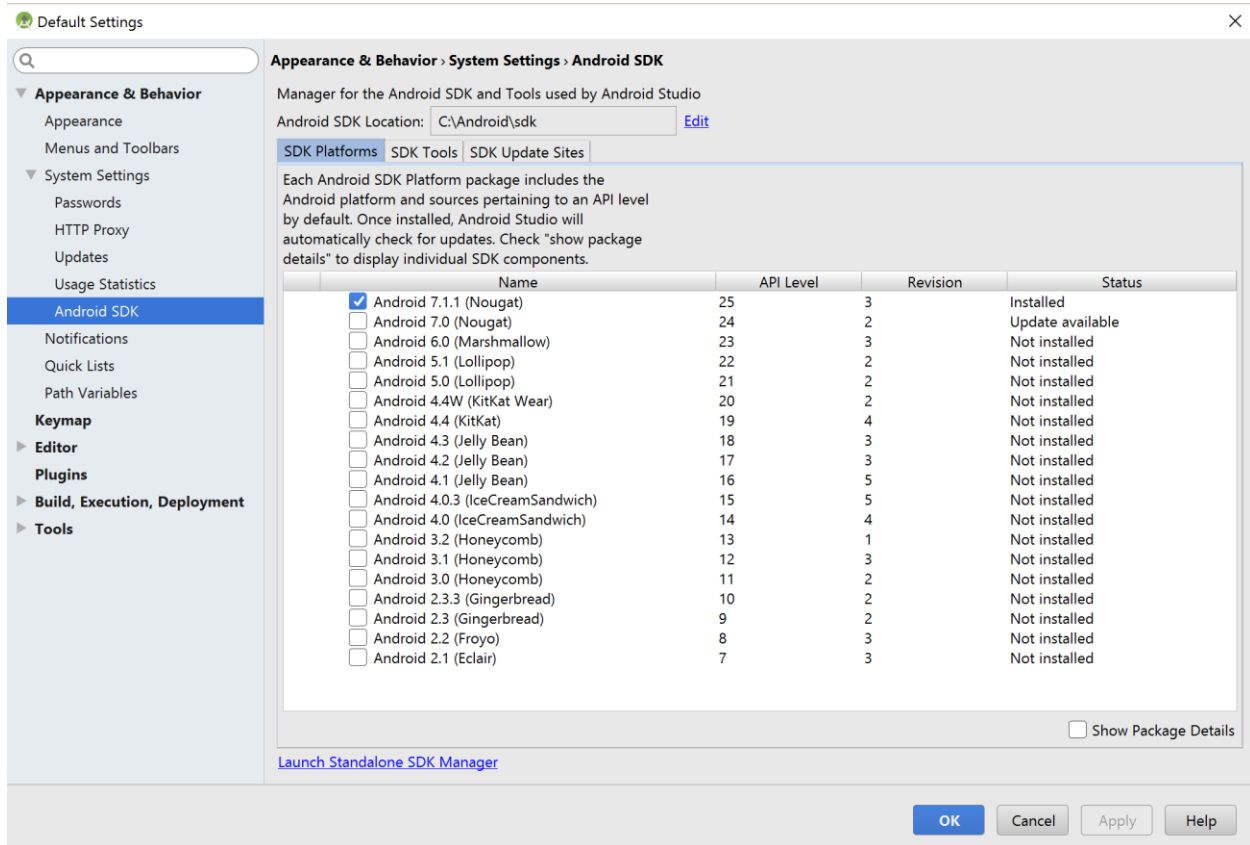


Fig. 9

For help with the installation of Android Studio, one may refer to Chapter 2 of the book “Smartphone-Based Real-Time Digital Signal Processing”, which can be acquired from this link:

<http://www.morganclaypool.com/doi/abs/10.2200/S00666ED1V01Y201508SPR013>

Section 2: Running C Codes as Android Apps

2.1 Programming Language

For creating Android apps, Java is used to create a shell for inserting and running C codes. To allow C codes to be called by Java, the Java Native Interface (JNI) framework is utilized. JNI acts as a translator that allows native code like C, C++ or assembly to be used when the app engine is not written in Java.

2.2 Creating Java Shell

The creation of a Java shell starts by creating a GUI (Graphical User Interface) to link data to a C code. The steps needed for creating a basic shell are listed below:

- Open Android Studio and select “Start a new Android Studio project” on the startup splash screen, see Fig. 10.

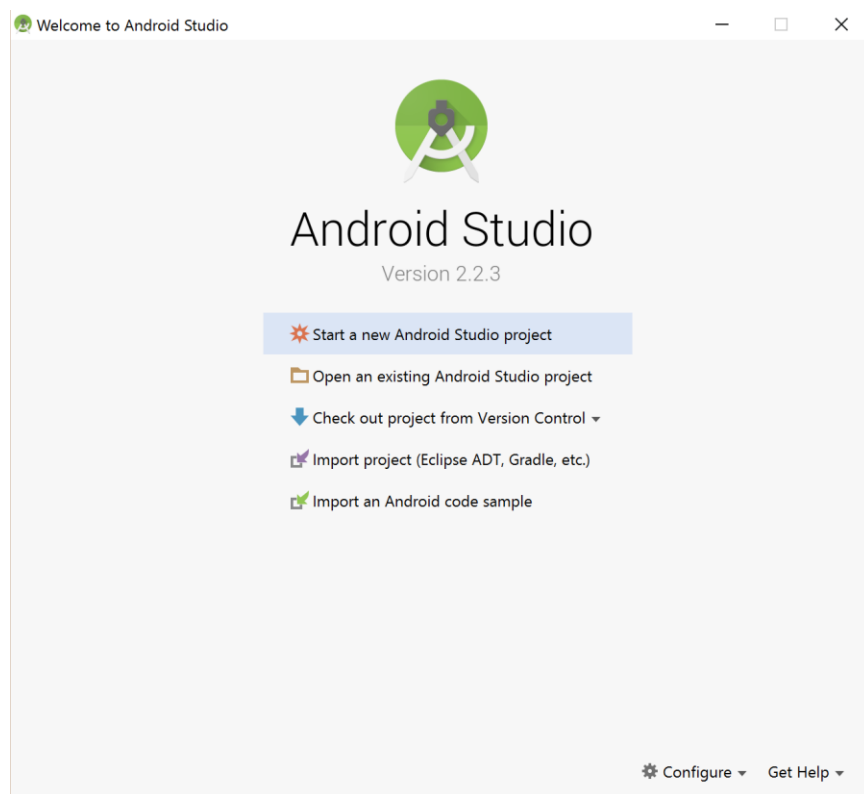


Fig. 10

- On the page that comes up, shown in Fig. 11, set the application name as TestApp and the company domain as dsp.com. This is important as it affects the naming of the native methods. Do not select “Include C++ Support”.

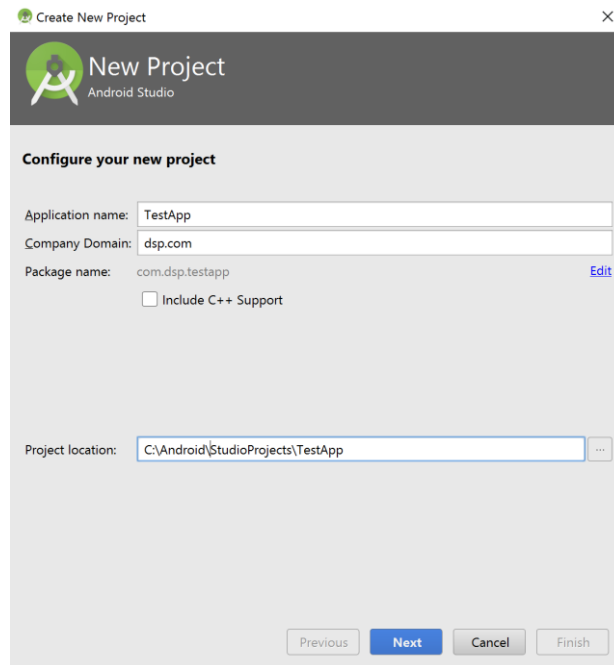


Fig. 11

- In the next screen, select the platform as “Phone and Tablet” and set the minimum SDK as “API 15” as shown in Fig. 12.

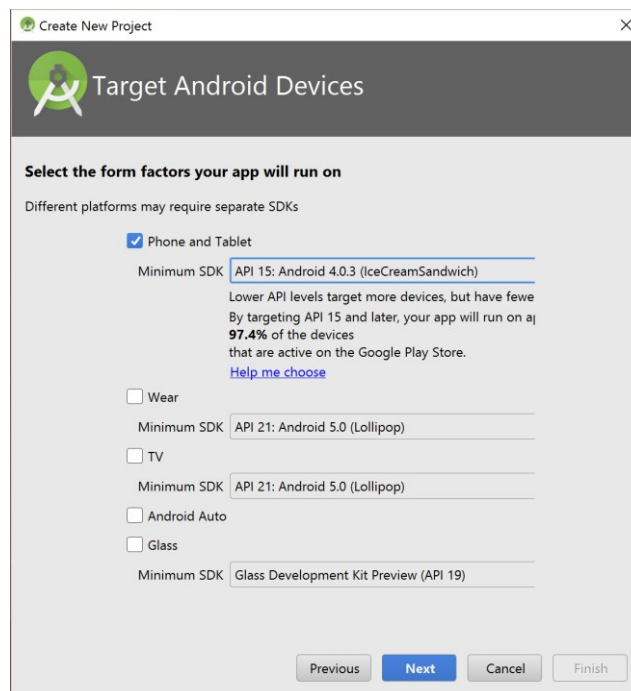


Fig. 12

- On the next screen, select “Empty Activity”.
- Leave the default naming and click “Finish”. The new app project is now created.

2.3 Creating GUI

Navigate to the Java directory of the app in the Project window and open the “MainActivity.java” file under “com.dsp.helloworld”.

The entity that typically defines an Android app is called an “Activity”. Activities are generally used to define user interface elements. An Android app has activities containing various sections that users might interact with, such as the main app window. Activities can also be used to construct and display other activities—such as if a settings window is needed. Whenever an Android app is opened, the “onCreate” function or method is called. This method can be regarded as the “main” (C terminology) of an activity. Other methods may also be called during various portions of the app lifecycle as detailed at the following website:

<http://developer.android.com/training/basics/activity-lifecycle/starting.html>

In the default code created by the SDK, “setContentView(R.layout.activity_main)” exhibits the GUI. The layout is described in the file “res/layout/activity_main.xml” in the Package Explorer window. Open this file to preview the user interface. Layouts can be modified using the WYSIWYG editor, which is built into Android Studio. For now, the basic GUI suits our purposes with one minor modification noted below.

- Open the XML text of the layout, see Fig. 13, by double clicking on the “Hello world!” text or by clicking on the activity_main.xml tab next to the Graphical Layout tab.
- Add the line android:id="@+id/Log" within the “<TextView/>” section on a new line and save the changes. This gives a name to the TextView UI element.

TextView in the GUI acts similar to a console window. It displays text. Additional text can be appended to it. By adding the “android:id” directive to TextView, interfacing can be done within the code.

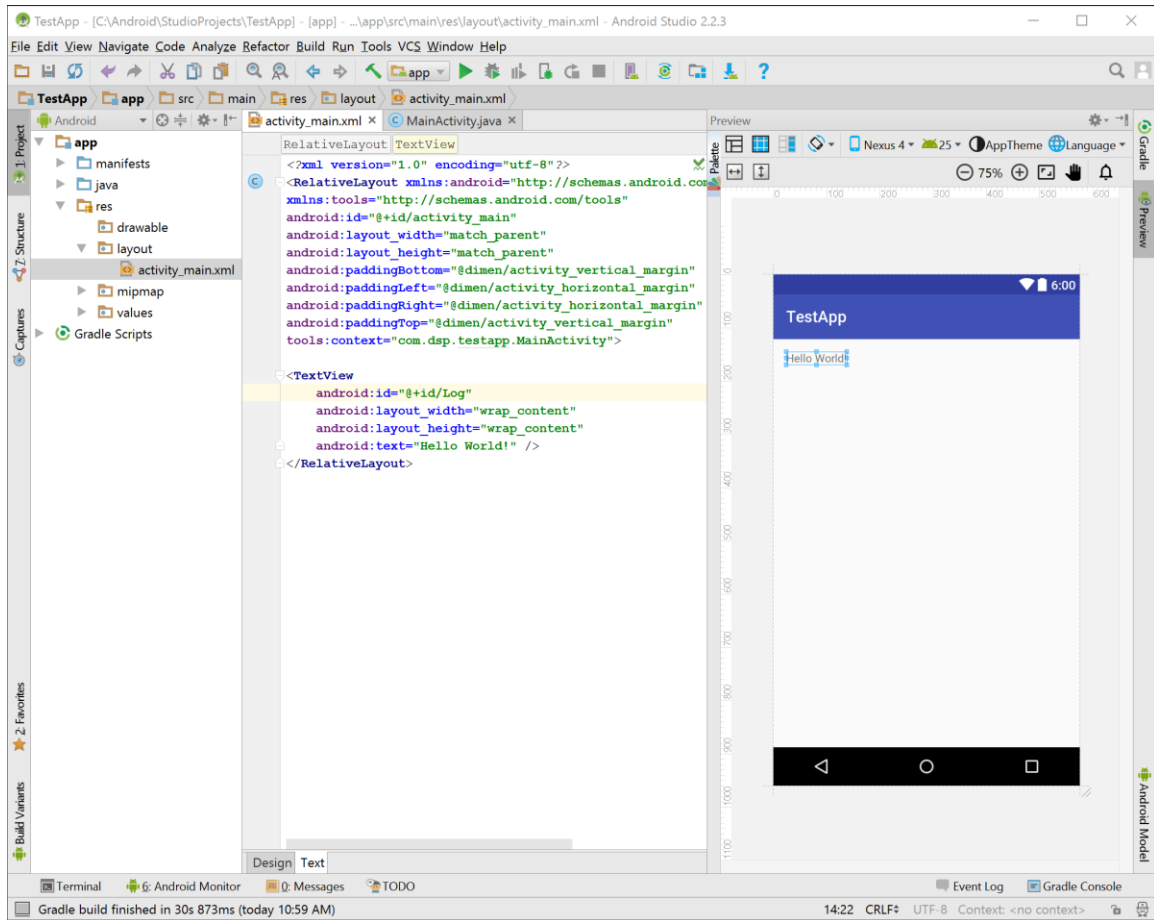


Fig. 13

To allow Android Studio to be able to build C code, it is needed to configure settings related to the NDK. In the project navigator, add a JNI folder by right clicking on the “app” folder and navigating to “New->Folder->JNI” as shown in Fig. 14. Click “Finish” on the next screen without editing the location of the folder.

Now right click on the newly created JNI folder, labelled “cpp”. Add a new file by going to “New->file”. Name the file “CmakeLists.txt” and save it. Inside the file, add the following code:

```
cmake_minimum_required(VERSION 3.4.1)

add_library(Algorithm SHARED
    Algorithm.c
)
target_link_libraries(Algorithm
    android
    log)
```

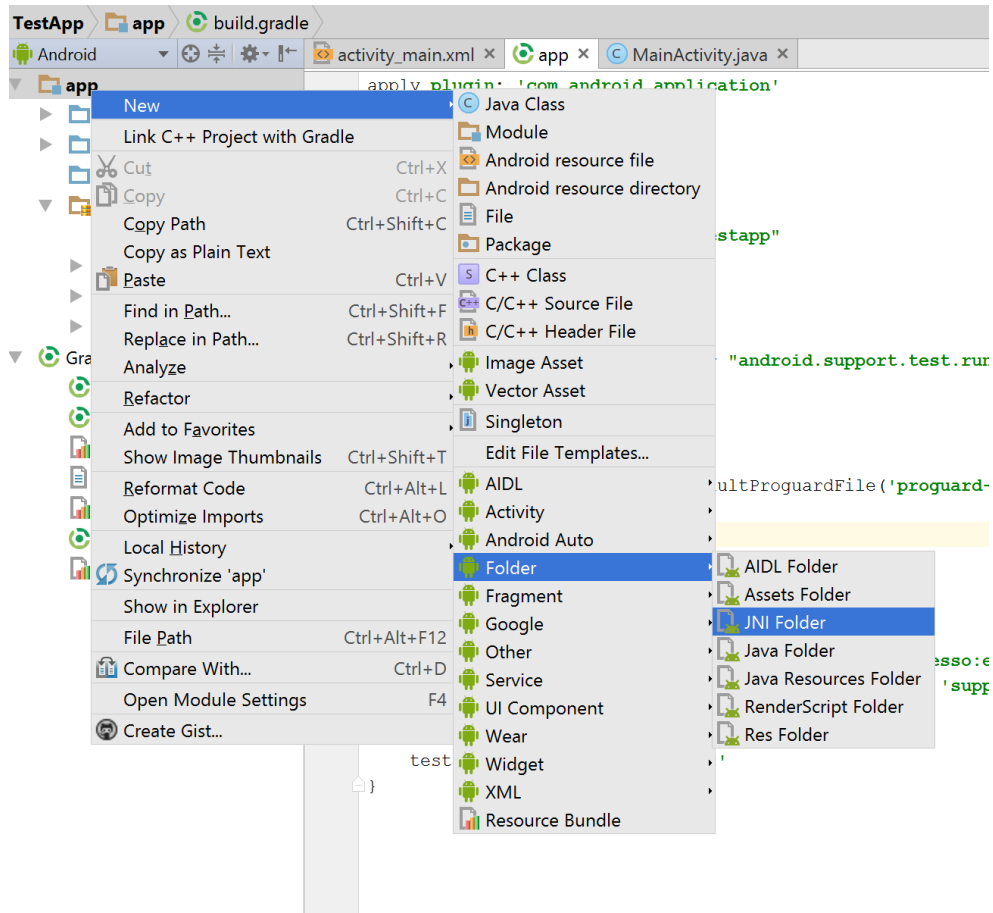


Fig. 14

Now open “local.properties” file in the project navigator. Add the following line of code after the sdk definition. This is assuming that the ndk was stored in “C:\Android”. This points Android Studio to the ndk directory.

```
ndk.dir=C:\Android\ndk
```

After this step, open “build.gradle” belonging to the app module and add the following code to the “android” section of the file. This tells Android Studio the location of the “CMakeLists.txt” file.

```
externalNativeBuild {
    cmake {
        path 'src/main/jni/CMakeLists.txt'
    }
}
```

Now create a new C file by right-clicking on the “cpp” folder and navigating to “New->C/C++ Source File”. Select the type as “.c” and name the file “Algorithm”. After the file has been created, enter the following code in the file:

```
#import <jni.h>

JNIEXPORT jstring JNICALL Java_com_dsp_testapp_MainActivity_getString(JNIEnv *env, jobject thiz) {
    return (*env)->NewStringUTF(env, "Hello World from C!");
}
```

After entering the code to be called from “MainActivity”, take the following steps:

- Add a TextView by importing it in “MainActivity” using the following code line:

```
import android.widget.TextView;
```

- Load the C functions by adding the following code in the public class definition:

```
static{
    System.loadLibrary("Algorithm");
}
public native String getString();
```

- In the “onCreate” function, add the following lines to change the text of TextView to the string received from the C code:

```
TextView log = (TextView)findViewById(R.id.Log);
log.setText(getString());
```

- Next, sync the gradle. The CMakeLists.txt file should now appear under ‘External Build Files’ in the project navigator.

As a result, the Android app accepts a string from a C code, and then prints it to the main activity of the app and displays it to the user. The app can be run on the Android emulator or an actual Android smartphone or tablet by clicking on the play button in the taskbar or navigating to “Run-> Run app”. The output of the app should appear as shown in Fig. 15.

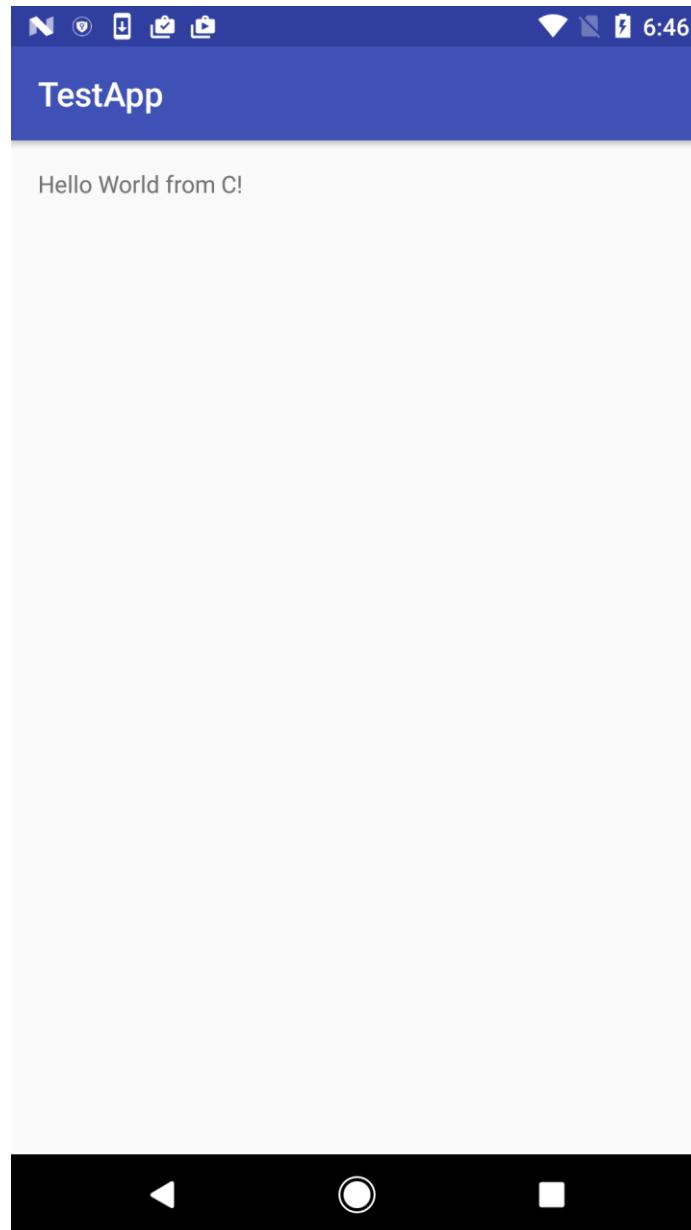


Fig. 15

Sample codes can also be obtained from the Google GitHub repository at the following link:

<https://github.com/googlesamples/android-ndk/tree/master/hello-jni>

Section 3: Converting MATLAB Codes to C

This section looks at deploying MATLAB codes on an Android device by using the MATLAB Coder utility. A simple example is shown here indicating the generation of C code from a MATLAB code which can be placed into “TestApp” covered in Section 2.

3.1 Creating a MATLAB Script

The MATLAB Coder requires a function of interest to be coded in MATLAB and also a Test Bench script to be coded in MATLAB for verifying the outputs of the function and also to see whether the function runs without any errors. The example shown here involves the creation of the Fibonacci sequence.

- Let us name a function “fibonacci” in MATLAB. Enter the following code. Keep the order of the parameters as noted below, otherwise the output will not appear correctly.

```
function fib_sequence = fibonacci(n,t1,t2)
fib_sequence = zeros(1,n);
fib_sequence(1) = t1;
fib_sequence(2) = t2;
for i = 3:n
    fib_sequence(i) = fib_sequence(i-1) + fib_sequence(i-2);
end
```

- Next, create a test bench script to verify the output of the function. The test bench script for the above function is displayed below:

```
clear all;
clc;
n = 10;
t1 = 0;
t2 = 1;
fib_sequence = fibonacci(n, t1, t2);
```

This script allows generating the Fibonacci sequence for 10 numbers with the initial elements being 0 and 1.

3.2 Generating C Code Using MATLAB Coder

After the function and the test bench script are created, an equivalent C code can be generated by using the MATLAB Coder. The MATLAB Coder can be found under the APPS tab in

the MATLAB toolbar. Before starting the MATLAB Coder, make sure you are in the directory which contains the function and the test bench script.

- In the MATLAB Coder splash screen, shown in Fig. 16, select the Numeric Conversion as “Convert to single precision”. In the Generate code for function, browse the function name. After you select both the options, the splash screen will display the location where the MATLAB Coder will create the project. Click Next.

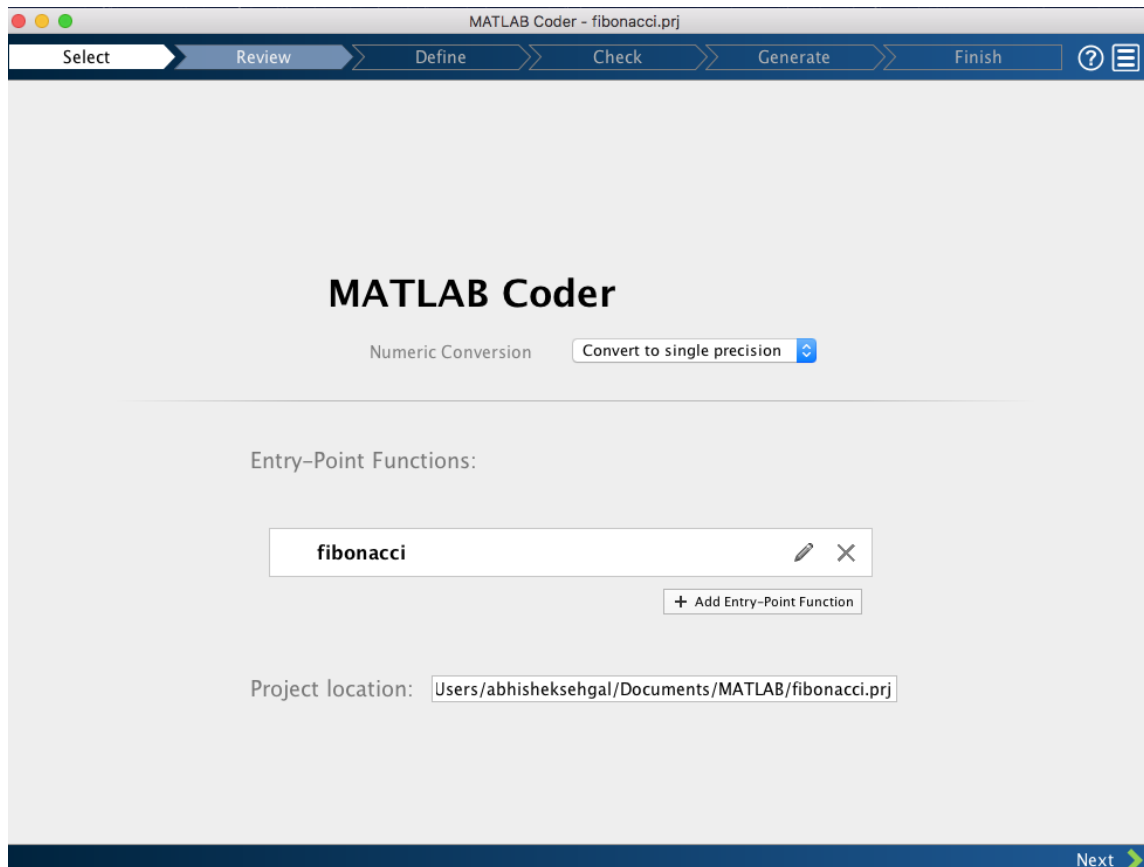


Fig. 16

- In the screen that comes up, shown in Fig. 17, select the test bench and click “Autodefine input Types”. Make sure the option “Does this code use global variables?” is set to No. This will populate the splash screen with all the inputs to the function. Verify all the three inputs are present, namely n, t1 and t2. Then, click Next.

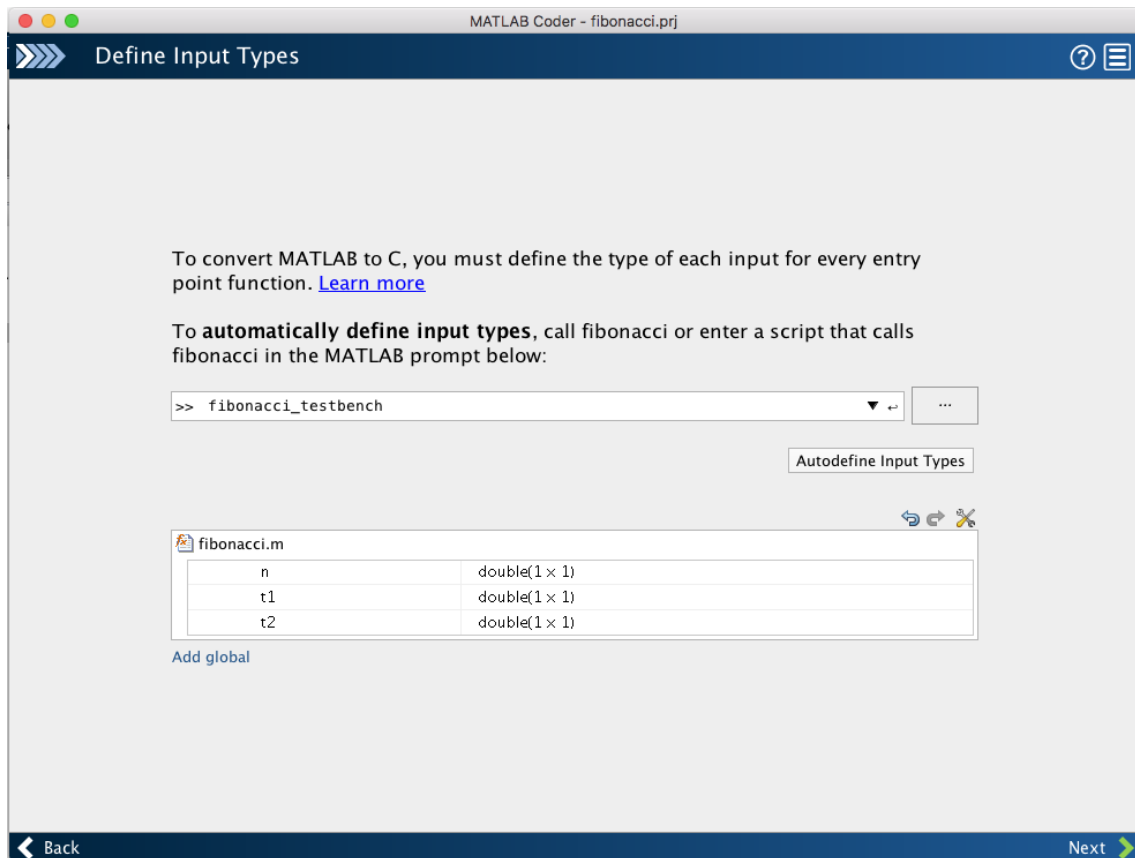


Fig. 17

- In the next splash screen, shown in Fig. 18, click “Check for Issues”. If any errors are displayed at this point, rectify them. Click Next when the screen shows “No issues detected”.

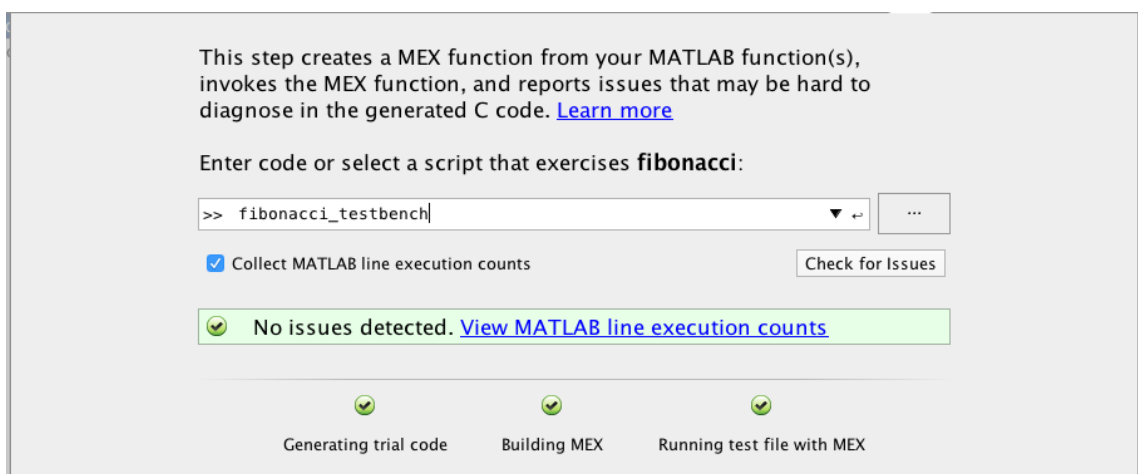


Fig. 18

- In the next screen that comes up, shown in Fig. 19, check to see that the options appear as

- Build Type: Source Code
- Language: C

If these options are correct, then click “Generate”.

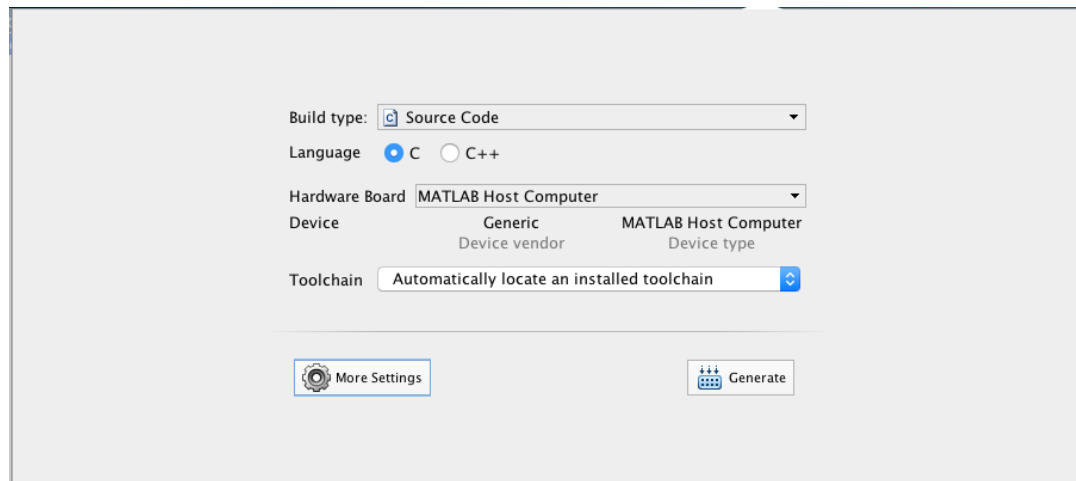


Fig. 19

- After the C code is generated, it will display all the .c and .h files associated with the generated C code as shown in Fig. 20. Click Next.
- The next page will display the project summary along with the locations of the generated output.

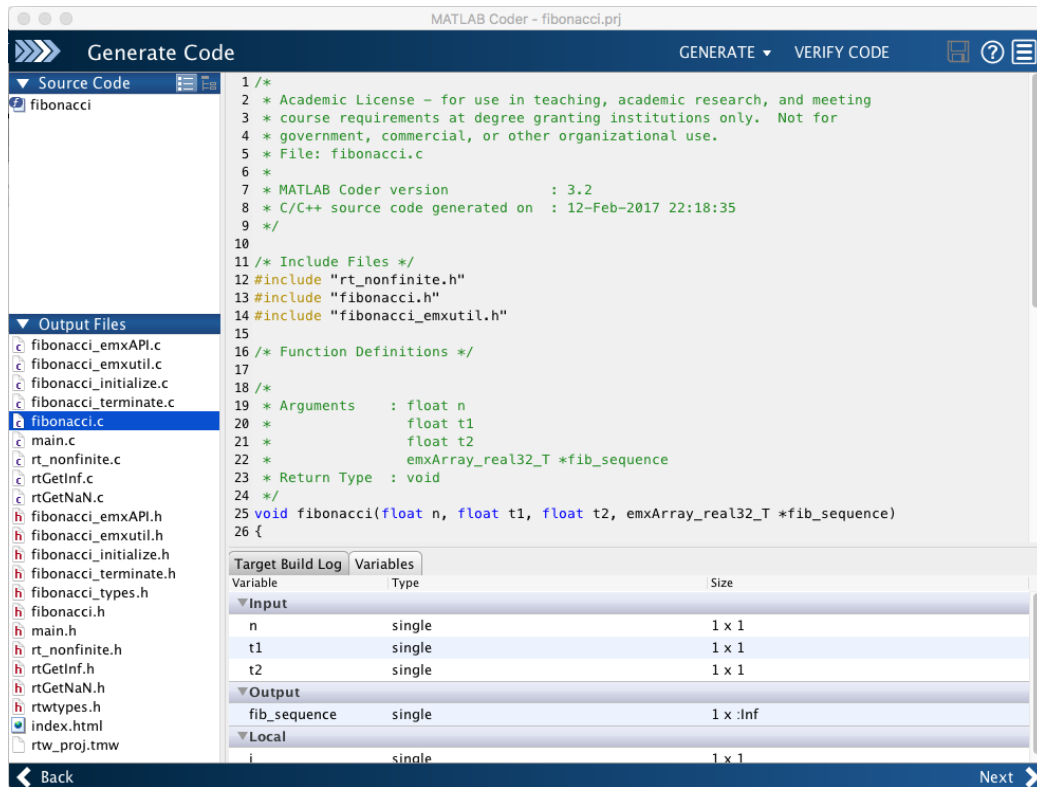


Fig. 20

3.3 Running C Code as Android App

- First make the following changes to the "CMakeLists.txt" file.

```
cmake_minimum_required(VERSION 3.4.1)

file(GLOB C_FILES "*.c")

add_library(
    Algorithm SHARED
    Algorithm.c
    ${C_FILES}
)

target_link_libraries(
    Algorithm
    android
    log)
```

- Then, navigate to the folder with the C source code and add all the .c and .h files by copying and pasting them in the JNI folder, as shown in Fig. 21 and Fig. 22. Sync the gradle and all the .c and .h files should appear in the cpp folder.

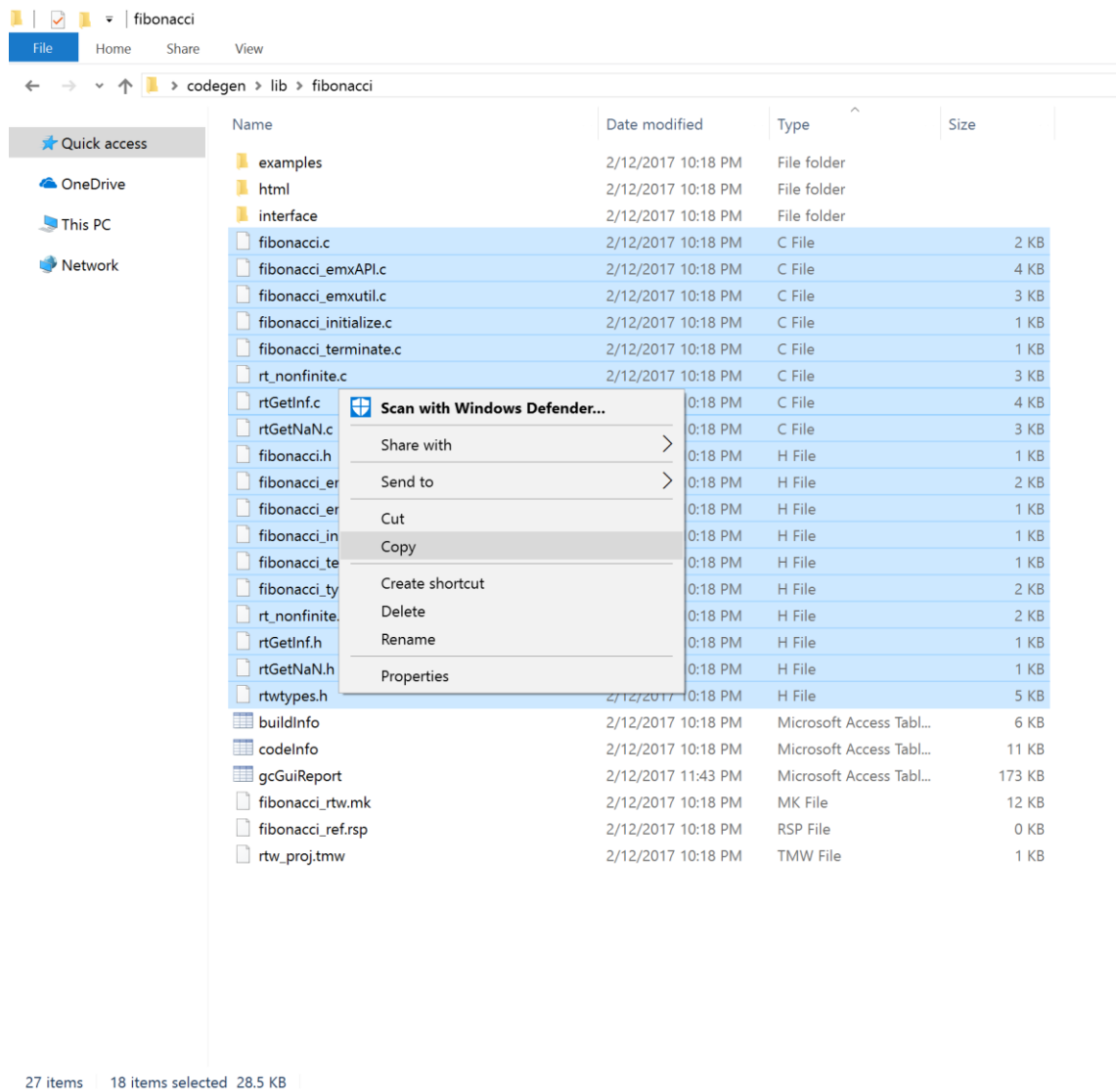


Fig. 21

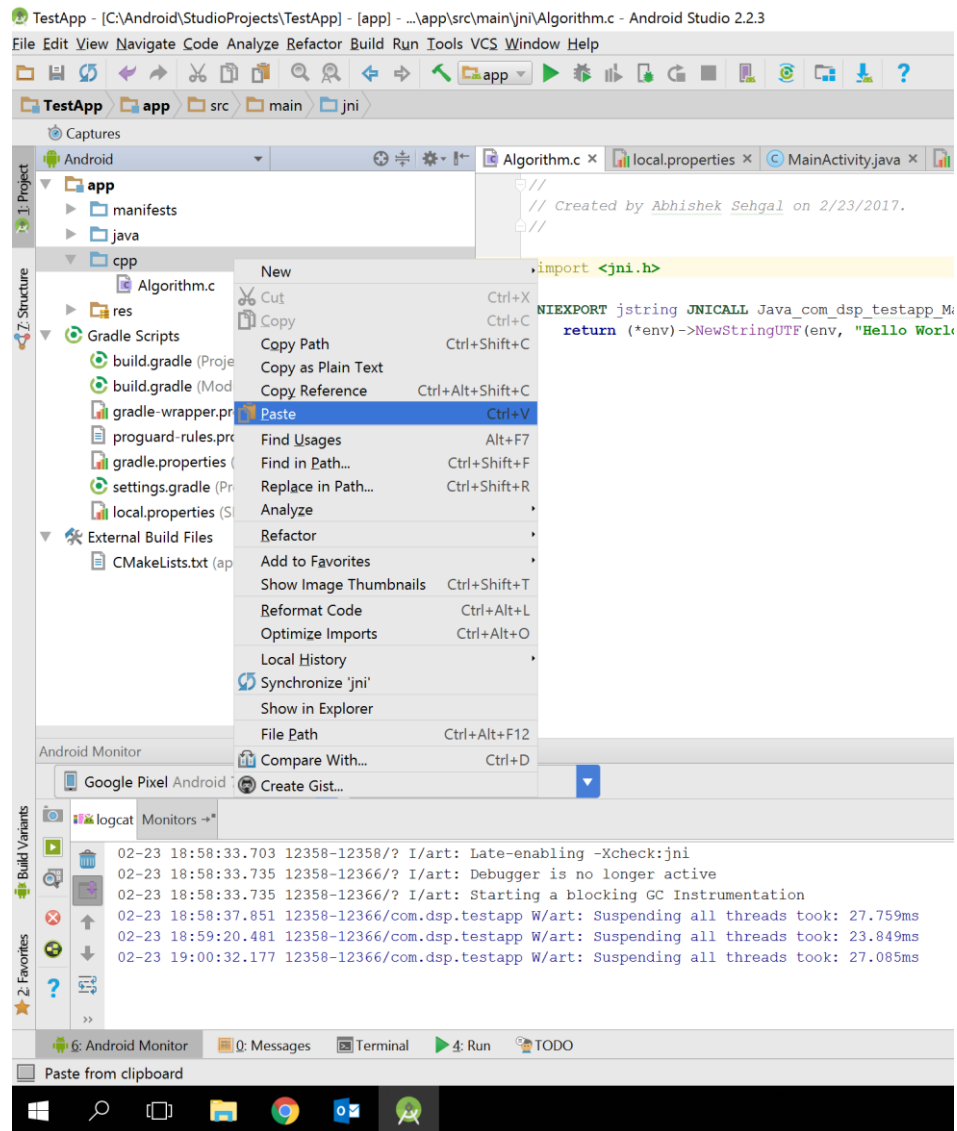


Fig. 22

- In the “Algorithm.c” file, add the following directives. These directives are used to call the MATLAB Coder functions in the C code.

```
#include "rt_nonfinite.h"
#include "fibonacci.h"
#include "fibonacci_initialize.h"
#include "fibonacci_terminate.h"
#include "fibonacci_emxAPI.h"
#include <android/log.h>
```


- In the getString function in “Algorithm.c”, add the following code before the return function. Since MATLAB has its own data types, this code allows memory to be allocated and then de-allocated when data types have finished their purpose.

```
// Inputs to the MATLAB Function
float n = 10;
float t1 = 0;
float t2 = 1;

// MATLAB array data type definition
emxArray_real32_T *fib_sequence;

// Allocating memory to the MATLAB array
// 2 represents the number of dimensions
emxInitArray_real32_T(&fib_sequence, 2);

// MATLAB function call
// Inputs are entered first in order, followed by the outputs
fibonacci(n, t1, t2, fib_sequence);

// Printing the elements from the Fibonacci Sequence
int i;
for (i = 0; i < n; i++) {
    __android_log_print(ANDROID_LOG_DEBUG, "Fibonacci Output", "Fibonacci Sequence %d - %0.0f\n", i + 1, fib_sequence->data[i]);
}

// Deallocate the MATLAB data array
emxDestroyArray_real32_T(fib_sequence);
```

- Run the app. When the button is pressed in the app, the console will be populated with the Fibonacci sequence as shown in Fig. 23.

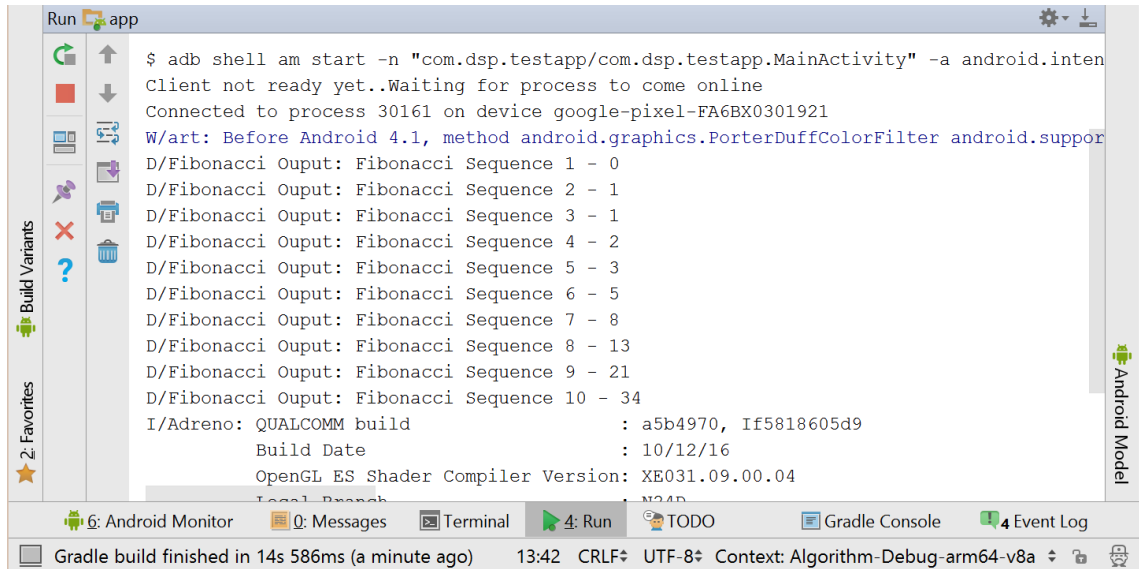


Fig. 23

This simple example showcases how to generate a C code from a MATLAB code. More details of the conversion from MATLAB to C codes are provided in the book “Anywhere-Anytime Signals and Systems Laboratory: From MATLAB to Smartphones”, which can be acquired from this link:

<http://www.morganclaypool.com/doi/abs/10.2200/S00727ED1V01Y201608SPR014>

Section 4: I/O Hardware Dependencies

This section discusses the i/o hardware dependency issues associated with different smartphones or mobile devices and a solution to cope with them.

4.1 Using Hardware Preferred Settings

The microphone of a particular smartphone is designed to work with the lowest latency at a certain sampling frequency and with a specific input frame size. To achieve low latency apps, such as noise classification and speech enhancement, the preferred settings by the manufacturer need to be used. Using non-preferred i/o settings increases the i/o latency due to resampling or data rearrangement. Furthermore, care must be taken by not choosing the frame size to be too small as this can lead to frames getting skipped due to a lack of adequate processing time.

The following link provides a listing of optimum frame sizes and sampling frequency for different smartphones and the i/o delays associated with them:

<http://superpowered.com/latency>

Generally, the preferred sampling frequency for Android devices is 48 kHz and the minimum input frame size or length varies from device to device.

4.2 Maintaining Microphone Consistency

Since different microphones are used in different smartphones, the input frequency characteristics may vary from smartphone to smartphone. For example, some manufacturers favor flat response microphones and some favor microphones with low frequency emphasis for speech processing. Other factors that may affect sound data captured by a smartphone is the location of the smartphone microphone from which audio data are captured, and whether a smartphone cover is placed around the microphone or not.

The difference in microphones and i/o hardware leads to inconsistency among data collected from different smartphones, which would affect the outcome of signal processing algorithms. As a solution, the training and testing of a signal processing algorithm ought to be carried out by the microphone of the same smartphone to decouple the effect of different microphone characteristics and the performance of the algorithm.

Section 5: Real-Time I/O Implementation

For real-time low-latency implementation on Android devices, it is advisable to use Superpowered utility, which is an audio API developed for mobile devices. On Android devices, Superpowered uses OpenSL to allow processing of audio data in a low-latency manner. More information on these APIs are available at the following links:

- Superpowered: <http://superpowered.com>
- OpenSL ES: <https://www.khronos.org/opensles/>

In what follows, a simple audio I/O path is implemented via Superpowered and it is shown how to implement an algorithm as an Android app.

5.1 Creating Audio I/O App

- Go to <http://superpowered.com> and download the SuperpoweredSDK. Store it in “C:\Android”.
- Open Android Studio. In the splash screen, select “Open an existing Android Studio Project”.
- Navigate to the SuperpoweredSDK folder. In the “Android” subfolder, open the “FrequencyDomain” example in Android Studio, as shown in Fig. 24.

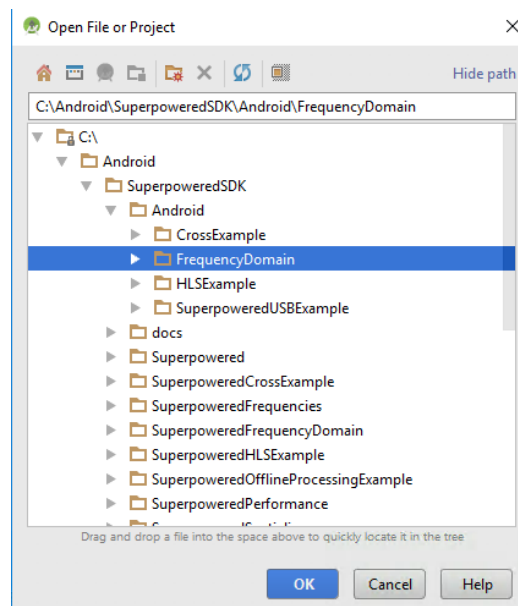


Fig. 24

- If the project shows errors on opening, that may be due to an incorrect path to the Superpowered directory. Go to “local.properties” of the project in the project navigator window and change the path of the Superpowered directory to “C:\Android\SuperpoweredSDK\Superpowered” and sync the gradle again.
- Open “MainActivity”. In that, the JNI native function “FrequencyDomain” can be seen, which is the C++ function that calls the function to start audio I/O with the supplied sampling rate and input buffer size. As it gets called inside the “onCreate” method, the audio path is created as soon as the app is loaded.
- Now go to the “FrequencyDomain.cpp” file inside “cpp->jni” in the project navigator. In that two methods can be seen:
 - FrequencyDomain: This is the JNI method which is called from “MainActivity”. This function is used to create a Superpowered Audio I/O session.
 - audioProcessing: This is the callback linked with the Audio I/O session which is repeatedly called when the input data are available for processing.
- To build a simple unprocessed audio I/O path, let us add a simple C code to process the incoming audio. Right-Click on the JNI folder and go to “New->C/C++ Source File”. Enter the name of the file as “File”, set the type as “.c” and click on “create an associated header”. The files may not appear in the Project Navigator. Add the following codes to the “CMakeLists.txt” file.

- Add the following line of code:

```
file(GLOB C_FILES "*.c")
```

- In the “add_library” section, add the following line of code:

```
${C_FILES}
```

After this, when you sync the gradle, the two files should appear in the “jni” folder.

- In the “FIR.c” file, add the following code:

```
void FIR(float* input, float* output, int nSamples) {
    int i = 0;

    static float endSamples[2] = {0,0};

    for (i = nSamples - 1; i > 1; i--) {
        output[i] = (input[i] + input[i - 1] + input[i - 2])/3;
    }
}
```

```

        output[1] = (input[1] + input[0] + endSamples[1])/3;
        output[0] = (input[0] + endSamples[1] + endSamples[0])/3;

        endSamples[1] = input[nSamples - 1];
        endSamples[0] = input[nSamples - 2];

    }

```

And in the FIR.h file, add the following code:

```
void FIR(float* input, float* output, int nSamples);
```

- After the algorithm has been coded in C, it can be called in the main file. In “FrequencyDomain.cpp”, replace the existing code with the following code:

```

#include <jni.h>
#include <stdlib.h>
#include <SuperpoweredFrequencyDomain.h>
#include <AndroidIO/SuperpoweredAndroidAudioIO.h>
#include <SuperpoweredSimple.h>
#include <SuperpoweredCPU.h>
#include <SLES/OpenSLES.h>
#include <SLES/OpenSLES_AndroidConfiguration.h>

// To Call C functions
extern "C" {
#include "FIR.h"
}

// Globally declared audio buffers
static float *inputBufferFloat, *leftInputBuffer, *rightInputBuffer, *leftOutputBuffer, *rightOutputBuffer;

// This is called periodically by the media server.
static bool audioProcessing(void * __unused clientdata, short int *audioInputOutput, int numberOfSamples, int
__unused samplerate) {

    SuperpoweredShortIntToFloat(audioInputOutput, inputBufferFloat, numberOfSamples, 2);
    SuperpoweredDeInterleave(inputBufferFloat, leftInputBuffer, rightInputBuffer, numberOfSamples);
    FIR(leftInputBuffer, leftOutputBuffer, numberOfSamples);
    FIR(rightInputBuffer, rightOutputBuffer, numberOfSamples);
    SuperpoweredFloatToShortIntInterleave(leftOutputBuffer, rightOutputBuffer, audioInputOutput,
numberOfSamples);

    return true;
}

extern "C" JNIEXPORT void
Java_com_superpowered_frequencydomain_MainActivity_FrequencyDomain(JNIEnv * __unused
javaEnvironment, jobject __unused obj, jint samplerate, jint buffersize) {

    inputBufferFloat = (float *)malloc(buffersize * sizeof(float) * 2 + 128);
    leftInputBuffer = (float *)malloc(buffersize * sizeof(float) + 128);
    rightInputBuffer = (float *)malloc(buffersize * sizeof(float) + 128);
    leftOutputBuffer = (float *)malloc(buffersize * sizeof(float) + 128);
    rightOutputBuffer = (float *)malloc(buffersize * sizeof(float) + 128);

    SuperpoweredCPU::setSustainedPerformanceMode(true);
    new SuperpoweredAndroidAudioIO(samplerate, buffersize, true, true, audioProcessing, NULL, -1,
SL_ANDROID_STREAM_MEDIA, buffersize * 2); // Start audio input/output.
}

```

In the above code, the `SuperpoweredAudioIO` session is initialized and then in `"audioProcessing"` the audio data are processed. The audio received is interleaved and it is deinterleaved before processing. After the processing is complete, the audio is interleaved again and stored back in the original buffer `"audioInputOutput"`. When `true` is returned in the method, it sends the buffer to the speaker.

You can now run the app and listen to the audio path. Similar to `"FIR.c"`, one can add other audio processing algorithms to process audio data.