**ANDROID APP DEVELOPMENT**

**Internship Report**

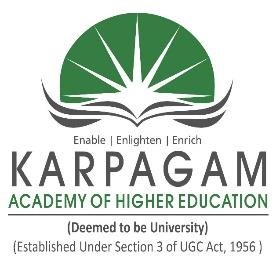
## Submitted to

**KARPAGAM ACADEMY OF HIGHER EDUCATIONAbout**

*in partial fulfillment of the requirements for the award of the degree of*

## BACHELOR OF SCIENCE IN COMPUTER SCIENCE

**Submitted by** Praveen S (22CSU045)



**Department of Computer Science**

# KARPAGAM ACADEMY OF HIGHER EDUCATION

## (Deemed to be University) (Established Under Section 3 of UGC Act, 1956)

**(Accredited with A+ Grade by NAAC in the Second Cycle) Pollachi Main Road, Eachanari Post,**

## Coimbatore – 641 021.

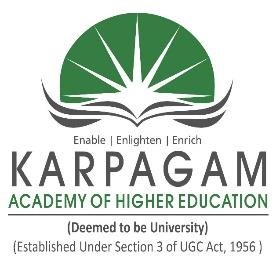
October 2023

# KARPAGAM ACADEMY OF HIGHER EDUCATION

## (Deemed to be University) (Established Under Section 3 of UGC Act, 1956)

(Accredited with A+ Grade by NAAC in the Second Cycle) Pollachi Main Road, Eachanari Post,

## Coimbatore – 641 021.



BACHELOR OF SCIENCE IN COMPUTER SCIENCE

# BONAFIDE CERTIFICATE

This is to certify that the Internship work entitled “*ANDROID APP DEVELOPMENT* ” is done by Praveen S (22CSU045), during the period August 31th , 2023 to October 21st, 2023 in partial fulfilment of the Degree of B.Sc., Computer Science.

Internal Examiner Head of the Department

# DECLARATION

I , hereby declare that the Internship entitled “ANDROID APP DEVELOPMENT”

submitted for the B.Sc., Computer Science Degree is my original work.

Place: Coimbatore Signature of the Student

Date:

(Praveen S)

# 

# 

# Abstract:

# The rapid proliferation of Android smartphones has created a dynamic market for innovative mobile applications. This project aims to embark on the journey of Android app development, focusing on creating user-centric and cutting-edge applications. Our development process involves meticulous planning, design, coding, testing, and deployment to ensure a seamless user experience.

# Key objectives of this project include:

# 1. Identifying and addressing user needs and pain points through in-depth market research.

# 2. Creating user-friendly interfaces with intuitive design and smooth navigation.

# 3. Implementing robust and efficient coding practices to ensure high performance and reliability.

# 4. Rigorous testing and quality assurance to guarantee bug-free operation.

# 5. Ongoing updates and improvements based on user feedback and emerging technologies.

# Through this Android app development project, we aim to deliver innovative solutions that enhance the lives of Android users while adhering to industry best practices and standard

# Table of contents

|  |  |  |  |
| --- | --- | --- | --- |
| S.NO | Index | | Page.no |
| 1 | Introduction | 5 | |
| 2 | About the project2.1 Module Description | 6 | |
|  | 2.2 Input/Output | 10 | |
|  | **2.3 Reports** | 45 | |
| 3 | Conclusion | 46 | |

# Introduction

*Patching Associates* was lending some very high-end, expensive industrial equipment to residents in Calgary to allow them to record the hum. However, they only had a limited number of equipment, and distributing them to a large number of homes in Calgary was not feasible. Developing a cell phone application to capture this data seemed to be the cheapest and best option because everybody has access to a smart-phone. The lowest frequency sound that can be recorded with a cell phone microphone is questionable. Hence, we could attach external microphones into our phone‘s earphone jack to pick up low frequency noise.

In Smith et.al describe a few acoustical metrics which can be easily run on cell phones with low computational power. It enables users to identify the key frequencies present in the Hum, which tells us whether there are multiple sources of the Hum. A network of cell-phones can be distributed across the community and used as a ―sound-source locator‖. Future plans also include setting up a cloud storage online and uploading data to it from our cell phone app. Features that are currently present and that can be added to the app in future are given in Noise cancellation using home-theater systems to generate an opposite phase signal which will cancel out the hum on superposition

2.1 Module description

Module 1

The application was initially developed by Adrien Gaspard, Mike Smith and Nicolas Lepine. A key feature of this application is the ability to capture and playback sound, so that the user knows that he is not imagining the Hum, but it actually exists. They went on to do some signal processing with the recorded signal, such as computing its Fourier Transform by doing an FFT to find out which frequencies are present in the signal. Adrien used the GraphView library to plot the time domain and frequency domain information of the captured sound signal. This version of the app is to be released soon. All the code and explanation required to build this app on your cell phone is listed in five parts in *Circuit Cellar Magazine* Articles to be released over the months of August to october, 2023

This version does the bare minimum to give us information about the recorded noise. However, some desired features are missing, the most important being the ability to store data, so that one can compare between two different recordings. The Hum frequency may be changing with time, and this feature is absolutely necessary to track all its changing signal content. Another disadvantage of this version is that it is slow. While dealing with GBs of recorded data, this becomes an undesirable feature. These issues are taken care of in version 2.

Module 2

This version has been developed by myself, Dr. Smith and Adrien Gaspard. It handles the issues of speed and storage associated with the previous version by storing all the data in a *SQLite* database . *SQLite* is the most widely deployed software library to implement databases on Android. *SQLite* also has the advantage of being embedded into every Android device, meaning that no setup procedure is required. A short tutorial on working with SQLite in android can be found in . The database stores data permanently. It will not be lost if the application is closed or if it crashes. It also makes data management faster. The last version was slow because we were passing all the data we wanted to plot in our graphs using an array. In this version, we just read it from the database, and the speed improves significantly.

Having our data in a database also allows us to go back and look at a recording from any day, and compare it with other recordings. We allow the user to look at the results of any captured sound from a list of all captured sounds, and compare it with others.

We do some more signal processing with the data. We implement the acoustical metrics defined in to identify the strongest frequencies in the signal after doing its FFT. The strongest peaks in the signal are termed as *Percentage Worse Case frequencies*. These signals stand out from the background noise. *Ratio*

*Background Noise frequencies* contains the frequencies which were overshadowed by the presence of a higher level of background noise, say during daytime. We then plot histograms of these metrics to find out which frequency occurs the most (likely to be the Hum frequency) using the same GraphView library. The code for calculating the metrics, plotting histograms, and building the UI is explained in the upcoming sections.

# About this Project

The IDE we have worked with for developing this app is *Android Studio*. Android Studio is the new official IDE for Android. Version 1 was developed in Eclipse. However, the Application Programming Interface (APIs) released in future will not be available for Eclipse. To get the latest updates, one must switch to Android Studio.

After installation has finished, you may be prompted to ―Start a New Android Project‖. Double click on it, a ―configure your new project‖ window appears. You are required to give the application a name, let‘s call it ‗Sound\_Record\_Analyse‘. Click on ―Next‖, a ―Target Android Devices‖ window appears. Leave the parameters as they are by default, which is the box ―Phone and Tablet‖ checked, with a minimum SDK of ―API 15: Android 4.0.3 (IceCreamSandwich). Click on ―Next‖ and when offered to ―Add an activity to Mobile, select ―Blank Activity‖ and click on ―Next‖. Customize the Activity by giving the Activity Name the name ―*MainActivity*‖, the layout Name ―*activity\_main*‖, the title ―*MainActivity*‖ and the Menu Resource Name ―*menu\_main*‖. You can finally press ―Finish‖.

Once installation of Android Studio is complete, you need to download and install the Software Development Kits (SDKs), which can be done by starting the SDK Manager in Android Studio (an icon present in the top right hand corner of the screen).



The list of SDKs to be installed. After the SDKs have finished installing, we are ready to create our application in Android Studio.

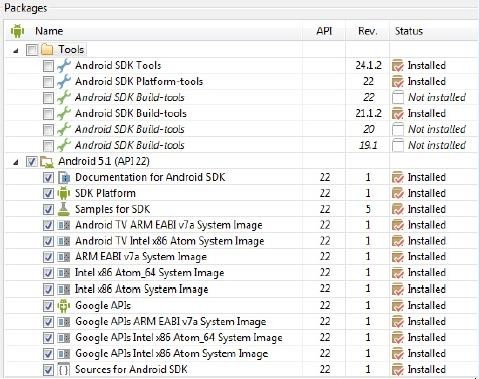


Fig 1 – List of SDKs to be installed

# CODE BEHIND THE APPLICATION

Each new screen that pops up in an android app is called an ‘. Each activity is associated with a

.xml file which defines the layout (the buttons and the text you see on the screen). The first activity which gets executed whenever we run the app is the *Main Activity* (much like the *main* method in a Java class). From this activity we usually start other activities. In this app we have three most important activities – *StartDSP* where we capture the sound and analyse it and store the result in a database, *DisplayGraph* where we plot the FFT graph and histograms, and *PickHistory* in which we create a list of all sounds recorded so far and let the user choose one so that we can plot its results. In two other activities, *MySQLiteDatabaseContract* and *MySQLiteDatabaseHelper* we create the database to store our results and write the functions associated with managing it. There are some other xml files in the *values* folder, such as *strings.xml* and *ints.xml* which contain the constants used in many of the activities.

**2.2 Input/Output**

## MySQLiteDatabaseContract

A database is usually composed of tables, which are in turn composed of many columns known as fields. This class contains the names of all columns and tables we want to put in our database. A class *TableEntry* implements an interface called ‗*BaseColumns‟.* In *TableEntry* we define the names of the tables and columns as String variables. The following is the structure of our database –

* A table called *analysis\_data* stores the result of the analysis – i.e, the *Percentage Worse Case* and *Ratio Background Noise* frequencies. It has the following fields – ―nameID‖, ―dateTime‖ which stores the date and time of recorded sound, ―comments‖ which stores any description entered by the user, ―maximum\_signal\_frequency‖ which stores the frequency with the maximum strength,

―percentage\_worse\_case‖ and ―ratio\_background\_noise‖ frequencies.

* Another table *fft\_data* stores the frequency domain information after doing FFT on the signal. It has the following fields – ―impulseno‖, which stores the FFT magnitude values of each impulse (we record sound as a number of impulses of 5s each), ―date‖ which stores the date and time of recorded sound, same as in *analysis\_data*, ―xvals‖ which has the frequency bins, ―yvals‖ which has the average of the magnitude spectrum of all impulses and ―comments\_fft‖ which also stores the description entered by the user, same as in *analysis\_data*.

**package** com.example.orchisamadas.sound\_record\_analyse;

**import** android.provider.BaseColumns;

**public final class** MySQLiteDatabaseContract{

**public** MySQLiteDatabaseContract(){}

**public static abstract class** TableEntry **implements** BaseColumns{

*//this table stores the analysis results*

**public static final** String ***TABLE\_NAME*** = **"analysis\_data"**; **public static final** String ***COLUMN\_NAME*** = **"nameID"**; **public static final** String ***COLUMN\_DATE*** = **"dateTime"**; **public static final** String ***COLUMN\_MAX\_SIGNAL*** =

**"maximum\_signal\_frequency"**;

**public static final** String ***COLUMN\_PERCENTAGE\_WORSE\_CASE*** =

**"percentage\_worse\_Case"**;

**public static final** String ***COLUMN\_RATIO\_BACKGROUND\_NOSE*** =

**"ratio\_background\_noise"**;

**public static final** String ***COLUMN\_COMMENT*** = **"comments"**;

*//this table stores the FFT results*

**public static final** String ***TABLE\_NAME\_FFT*** = **"fft\_data"**; **public static final** String ***COLUMN\_NAME\_DATE*** = **"date"**; **public static final** String ***COLUMN\_NAME\_XVALS*** = **"xvals"**; **public static final** String ***COLUMN\_NAME\_YVALS*** = **"yvals"**;

**public static final** String ***COLUMN\_NAME\_IMPULSE*** = **"impulseno"**; **public static final** String ***COLUMN\_NAME\_COMMENT*** = **"comments\_fft"**;

}

}

Listing 1 – MySQLiteDatabaseContract.java

## MySQLiteDatabaseHelper

This is a child class of *SQLiteOpenHelper* [7] which contains basic database management functions like *onCreate()*, *onUpgrade()* and *onOpen()* to create, upgrade and open the database. In this class, functions are written/overridden to create, delete and upgrade our database called ―*SoundAnalysisResults.db*‖.

|  |
| --- |
| **package** com.example.orchisamadas.sound\_record\_analyse;  **import** android.content.Context;  **import** android.database.sqlite.SQLiteDatabase;  **import** android.database.sqlite.SQLiteOpenHelper;  **import** com.example.orchisamadas.sound\_record\_analyse.MySQLiteDatabaseContract.TableEntry;  **public class** MySQLiteDatabaseHelper **extends** SQLiteOpenHelper{ **public static final** String ***NAME***=**"SoundAnalysisResults.db"**; **public static final int *VERSION***=1;  **public static** Context *mContext*;  **public** MySQLiteDatabaseHelper(Context context){  **super**(context,***NAME***,**null**,***VERSION***);*mContext*=context;  }  @Override  **public void** onCreate(SQLiteDatabase db) {  *//this table stores analysis results*  String create = **"CREATE TABLE IF NOT EXISTS "** + TableEntry.***TABLE\_NAME*** + **" ("** + + TableEntry.***\_ID*** + **" INTEGER PRIMARY KEY AUTOINCREMENT, "**  + TableEntry.***COLUMN\_NAME***+ **" TEXT, "**  + TableEntry.***COLUMN\_COMMENT*** + **" TEXT, "**  + TableEntry.***COLUMN\_DATE*** + **" TEXT, "**  + TableEntry.***COLUMN\_MAX\_SIGNAL*** + **" REAL, "**  + TableEntry.***COLUMN\_PERCENTAGE\_WORSE\_CASE*** + **" REAL, "**  + TableEntry.***COLUMN\_RATIO\_BACKGROUND\_NOSE*** + **" REAL)"**;  db.execSQL(create);  *//this table stores FFT results*  create = **"CREATE TABLE IF NOT EXISTS "** + TableEntry.***TABLE\_NAME\_FFT*** + **" ("**  + TableEntry.***\_ID*** + **" INTEGER PRIMARY KEY AUTOINCREMENT, "**  + TableEntry.***COLUMN\_NAME\_DATE*** + **" TEXT, "**  + TableEntry.***COLUMN\_NAME\_COMMENT***+ **" TEXT, "**  + TableEntry.***COLUMN\_NAME\_XVALS*** + **" BLOB, "**  + TableEntry.***COLUMN\_NAME\_YVALS*** + **" BLOB"**;  **int** numImpulses = *mContext*.getResources().getInteger(R.integer.***num\_impulses***);  **for**(**int** k = 0; k < numImpulses; k++)  create = create + **", "** + TableEntry.***COLUMN\_NAME\_IMPULSE*** + Integer.*toString*(k) + **" BLOB)"**; db.execSQL(create);  }  **public void** deleteAllEntries(SQLiteDatabase db,String tableName){ db.delete(tableName, **null**, **null**);}  **public void** deleteDatabase(){*mContext*.deleteDatabase(***NAME***);} @Override  **public void** onUpgrade(SQLiteDatabase db,**int** oldVersion,**int** newVersion) {  **if** (newVersion <= oldVersion)  **return**; deleteDatabase(); onCreate(db); **return**;  }  } Listing 2 – MySQLiteDatabaseHelper.java |

The *onCreate()* method is executed first. This creates the two tables in our database – *analysis\_data* and *fft\_data*. The String ‗create‘ contains the SQL command for creating a table (if it does not previously exist), with the given fields. *TableEntry.\_ID* has the unique row ID for each row of a column. It is auto- incremented. As expected, the other columns are of type ‗TEXT‘ (to store strings) and ‗REAL‘ (to store floating point numbers). Another data type ‗BLOB‘ is used to store the result for *fft\_data.* This is because SQLite databases cannot store arrays of double. Instead, they use Binary Large Object, BLOBs, which are a collection of binary data stored as a single entity. As mentioned before, sound is recorded in a number of impulses of 5s each. The FFT magnitude values of each impulse is stored in a separate column – called impulse0, impulse1, and so on. The column ‗yvals‘ stores the average of FFT magnitude values of all the impulses.

The other methods, *deleteAllEntries()* deletes a particular table of the database and *deleteDatabase()* deletes the entire database itself. *onUpgrade()* method is called whenever the database version is increased. We may want to do so if we want to change the structure of the database, i.e., add new tables or columns to existing tables. It simply deletes the old database and creates a new one.

## MainActivity

This is the first activity that gets executed when we launch the application. We must first write its layout. The layout is written in *activity\_main.xml* file (It is a simple layout that contains three buttons, placed one below another - *Record New Data, View Frequency Graphs* and *View Analysis Results.* (We refer to the buttons by their ids, the text to be written in them is given in the *strings.xm*l file, which I will add at the end of this document.) Pressing ‗*Record New Data‟* will start the activity **StartDSP** which captures sound and analyses it. ‗*View Frequency Graphs‟* will show us the FFT results (**gotoGraphFFT**) and ‗*View Analysis Results*

The MainActivity (**listing 3B**) contains calls to other activities using something called an *intent*. An Android *intent* is an abstract description of an operation to be performed. It can be used with *startActivity* to launch an Activity. We launch the activities, *StartDSP* or *DisplayGraph* depending on the button pressed by the user. We can use a *Bundle* to pass some values to the activity we are starting. In this case, *DisplayGraph* will either plot FFT or a histogram depending on the button pressed. Pressing

‗*View Frequency* display the FFT graph whereas pressing ‗*View Analysis Results*‘ will display the analysis histogram. The information regarding which button has been pressed needs to be conveyed to the *DisplayGraph* activity by the *MainActivity* by using a *bundle*.

|  |  |
| --- | --- |
| <**RelativeLayout xmlns:android=**[**"http://**](http://schemas.android.com/apk/res/android)**s**[**chemas.android.com/apk/res/android**](http://schemas.android.com/apk/res/android) **"**  **xmlns:tools=**[**"http://s**](http://schemas.android.com/tools)**c**[**hemas.android.com/tools**](http://schemas.android.com/tools)**" android:layout\_width ="fill\_parent" android:layout\_height ="fill\_parent"**>  <**Button android:id="@+id/btnStartDSP" android:layout\_width="wrap\_content"**  **android:layout\_height="wrap\_content" android:layout\_marginTop = "40dp" android:layout\_centerVertical="false" android:layout\_centerHorizontal="true" android:text="@string/start\_dsp" android:onClick="StartDSP"**/>  <**Button android:id="@+id/graph\_FFT"**  **android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:layout\_below="@+id/btnStartDSP" android:layout\_centerHorizontal="true" android:text="@string/graph\_FFT" android:onClick="gotoGraphFFT"** />  <**Button android:id="@+id/graph\_histogram" android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:layout\_below="@+id/graph\_FFT" android:layout\_centerHorizontal="true" android:text="@string/graph\_histogram" android:onClick="gotoHistogram"** />  </**RelativeLayout**>  Listing 3A – activity\_main.xml | Fig 2 – MainActivity screen |

|  |
| --- |
| **package** com.example.orchisamadas.sound\_record\_analyse;  **import** android.content.Intent;  **import** android.os.Bundle;  **import** android.support.v7.app.ActionBarActivity;  **import** android.view.View;  **public class** MainActivity **extends** ActionBarActivity { @Override  **protected void** onCreate(Bundle savedInstanceState) { **super**.onCreate(savedInstanceState); setContentView(R.layout.***activity\_main***);  }  **public void** StartDSP(View v){ |

|  |
| --- |
| Intent intent=**new** Intent(**this**,StartDSP.**class**); startActivity(intent);  }  */\*Starts the activity DisplayGraph to view previous graphs*  *We can either view previous FFT graphs or previous analysis histograms depending on which button is pressed \*/*  **public void** gotoGraphFFT(View v)  {  Bundle bundle = **new** Bundle(); bundle.putString(**"button\_pressed"**, **"1"**);  Intent intent = **new** Intent(**this**, DisplayGraph.**class**); intent.putExtras(bundle);  startActivity(intent);  }  **public void** gotoHistogram(View v)  {  Bundle bundle = **new** Bundle(); bundle.putString(**"button\_pressed"**, **"2"**);  Intent intent = **new** Intent(**this**, DisplayGraph.**class**); intent.putExtras(bundle);  startActivity(intent);  }  }  Listing 3B – MainActivity.java |

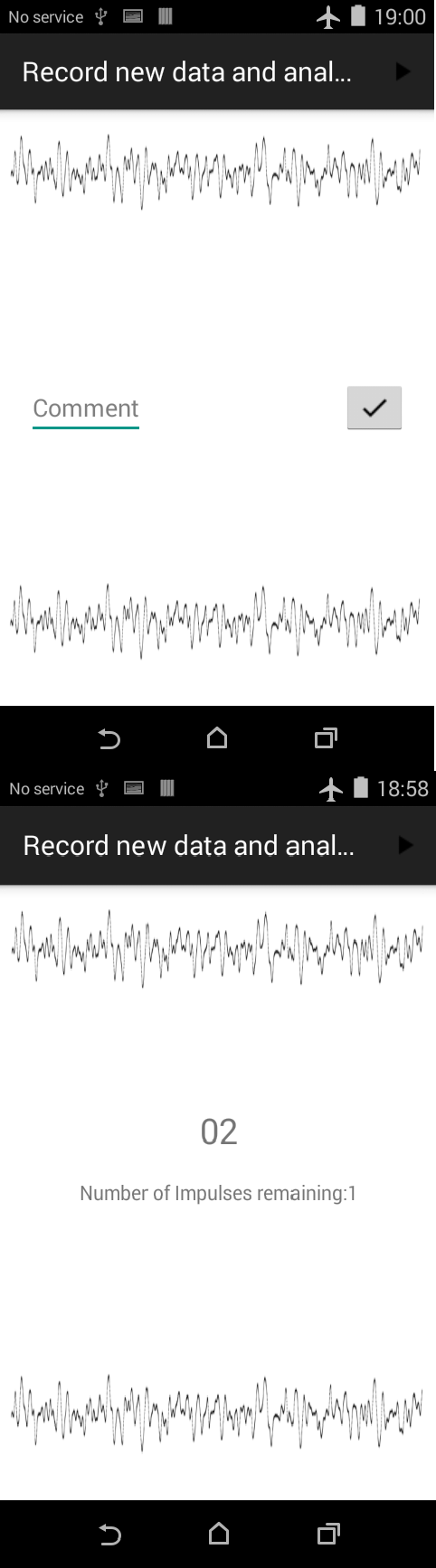
## StartDSP Activity

In this activity, we capture sound and do signal processing on it – obtain the frequency spectrum by doing an FFT [8] and then calculate the metrics described in. I will first describe sound capture, followed by the analysis method and storage in database. gives the layout for *StartDSP* activity.

In the *onCreate()* method, we create a media player object called ‗boo‘. It is basically a chirp that goes from 50Hz to 1000Hz in a duration of 5s (Chirp\_50\_1000Hz.wav). We let the user have the option of playing this chirp in case he wants to excite a room resonance. We must create a folder called ‗*raw*‘ in the

‗*res*‘ directory containing ‗*boo.wav*‘.

Next we open the database, ―*SoundAnlysisResults.db*‖ so that we can write data into it. *onOptionsSelectedMenu(*) lets us add a menu at the top right hand corner of the screen. I just want to add a play button here so that on pressing it, the user can play ‗boo.wav‘. The layout for this menu is given in *menu\_start\_dsp.xml* he method ‗*startPlaying()*‘ is executed when the user when user presses the play button on the menu. The chirp must be stored in the phone‘s memory/SD card before playing it (create ‗MySounds‘ folder in the phone‘s storage directory from your computer and paste Chirp\_50\_1000Hz.wav in it).



<**RelativeLayout xmlns:android=**[**"http://**](http://schemas.android.com/apk/res/android)**s**[**chemas.android.com/apk/res/android”**](http://schemas.android.com/apk/res/android)

**xmlns:tools=**[**"http://sche**](http://schemas.android.com/tools)**m**[**as.android.com/tools**](http://schemas.android.com/tools)**"**

**android:background = "@drawable/dsp" android:layout\_width="fill\_parent" android:layout\_height="fill\_parent" tools:context="${relativePackage}.${activityClass}"** >

<**TextView**

**android:id="@+id/remaining\_impulses" android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:layout\_centerHorizontal = "true" android:layout\_centerVertical="true"**

/>

<**TextView android:id="@+id/textViewTime" android:layout\_width="wrap\_content" android:layout\_height="wrap\_content"**

**android:layout\_above ="@id/remaining\_impulses" android:layout\_marginBottom="17dp" android:textSize="25sp" android:layout\_centerHorizontal="true" android:layout\_centerVertical="true"**

/>

<**ImageButton**

**android:id = "@+id/playback" android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:layout\_centerHorizontal = "true" android:layout\_centerVertical="true" android:layout\_above = "@id/textViewTime" android:src = "@drawable/ic\_play\_arrow\_black\_24dp" android:visibility = "invisible"**/>

<**EditText android:id="@+id/Addcomment" android:layout\_centerHorizontal="true" android:layout\_centerVertical ="true" android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:layout\_alignParentStart = "true" android:layout\_alignParentLeft = "true" android:hint = "Comment" android:layout\_marginLeft="20dp"** />

<**ImageButton android:id="@+id/Enter"**

**android:layout\_centerHorizontal="true" android:layout\_centerVertical ="true" android:layout\_width="wrap\_content" android:layout\_height="wrap\_content"**

|  |  |
| --- | --- |
| **android:layout\_alignParentRight = "true" android:layout\_alignParentEnd = "true" android:layout\_marginRight="20dp"**  **android:src ="@drawable/ic\_done\_black\_24dp"** />  <**ProgressBar**  **android:id="@+id/computation\_progress" style="?android:attr/progressBarStyleHorizontal" android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:layout\_centerVertical ="true" android:layout\_centerHorizontal="true" android:indeterminate="false" android:max="100"**  **android:progress="0" android:visibility="invisible"**  />  <**TextView**  **android:id = "@+id/analysing" android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:layout\_centerHorizontal="true" android:layout\_centerVertical="true" android:layout\_above = "@id/computation\_progress" android:visibility="invisible"**  />  <**Button android:id="@+id/btnDisplayGraph" android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:layout\_centerVertical="true" android:layout\_centerHorizontal="true"**  **android:layout\_below = "@id/computation\_progress" android:text="@string/view\_fft\_result" android:onClick="DisplayGraph" android:visibility="invisible"**/>  <**Button android:id="@+id/btnDisplayHistogram" android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:layout\_centerVertical="true" android:layout\_centerHorizontal="true"**  **android:layout\_below = "@id/btnDisplayGraph" android:text="@string/view\_analysis\_result" android:onClick="DisplayGraph" android:visibility="invisible"**/>  </**RelativeLayout**>  Listing 4 – activity\_start\_dsp.xml | Fig 3 – StartDSP layouts |

|  |
| --- |
| **package** com.example.orchisamadas.sound\_record\_analyse;  **import** java.io.BufferedInputStream; **import** java.io.BufferedOutputStream; **import** java.io.DataInputStream; **import** java.io.DataOutputStream; **import** java.io.File;  **import** java.io.FileInputStream; **import** java.io.FileOutputStream; **import** java.io.IOException; **import** java.io.InputStream; **import** java.io.OutputStream; **import** java.nio.ByteBuffer; **import** java.util.ArrayList; **import** java.util.Collections; **import** java.util.Date;  **import** java.util.HashSet; **import** java.util.List; **import** java.util.Set;  **import** java.util.concurrent.TimeUnit; **import** android.content.ContentValues; **import** android.content.Context; **import** android.content.Intent;  **import** android.database.sqlite.SQLiteDatabase;  **import** android.media.AudioFormat; **import** android.media.AudioManager; **import** android.media.AudioRecord; **import** android.media.AudioTrack; **import** android.media.MediaPlayer;  **import** android.media.MediaRecorder.AudioSource;  **import** android.os.AsyncTask;  **import** android.os.Bundle;  **import** android.os.CountDownTimer;  **import** android.os.Environment;  **import** android.support.v7.app.ActionBarActivity;  **import** android.text.format.DateFormat;  **import** android.util.Log; **import** android.view.MenuItem; **import** android.view.View;  **import** android.view.inputmethod.InputMethodManager;  **import** android.widget.Button; **import** android.widget.EditText; **import** android.widget.ImageButton; **import** android.widget.ProgressBar; **import** android.widget.TextView; **import** android.widget.Toast;  **import** com.example.orchisamadas.sound\_record\_analyse.MySQLiteDatabaseContract.TableEntry;  **public class** StartDSP **extends** ActionBarActivity { TextView **TextHandleRemainingImpulses**; AudioRecord **recorder**;  CaptureAudio **captureAudio**; TextView **textViewTime**; String **title**;  EditText **comment**; |

|  |
| --- |
| ImageButton **done**;  **final** CounterClass **timer** = **new** CounterClass(5000, 1000);  **private static final double *REFSPL*** = 0.00002;  **private** MediaPlayer **mPlayer** = **null**; **private** MediaPlayer **mediaPlayer**;  @Override  **protected void** onCreate(Bundle savedInstanceState) {  **super**.onCreate(savedInstanceState);  **mediaPlayer** = MediaPlayer.*create*(**this**, R.raw.***boo***); setContentView(R.layout.***activity\_start\_dsp***);  MySQLiteDatabaseHelper databaseHelper = **new** MySQLiteDatabaseHelper(StartDSP.**this**);  *//open or create database*  SQLiteDatabase db = openOrCreateDatabase(Environment.*getExternalStorageDirectory*() + File.***separator*** + databaseHelper.***NAME***, ***MODE\_PRIVATE***, **null**);  databaseHelper.onCreate(db);  }  *//play chirp when play button is pressed.*  **public boolean** onOptionsItemSelected(MenuItem item){  *//Handle presses on the action bar items*  **switch**(item.getItemId()){  **case** R.id.***GenerateChirp***:startPlaying();  **return true**; **default**:  **return super**.onOptionsItemSelected(item);  }  }  **public void** startPlaying(){ **mPlayer**=**new** MediaPlayer(); **try**{  **mPlayer**.setDataSource(Environment.*getExternalStorageDirectory*().getAbsolutePath()+ **"/MySounds/Chirp\_50\_1000Hz.wav"**);  **mPlayer**.prepare(); **mPlayer**.start();  }  **catch**(IOException e){}  }  @Override  **protected void** onStart(){  **super**.onStart();  *//allow user to enter title*  **comment** = (EditText) findViewById(R.id.***Addcomment***);  **done** = (ImageButton) findViewById(R.id.***Enter***);  Toast.*makeText*(StartDSP.**this**, **"Add a small description of the noise you're hearing"**, Toast.***LENGTH\_SHORT***).show();  **done**.setOnClickListener(**new** View.OnClickListener() { @Override  **public void** onClick(View v) {  **title** = **comment**.getText().toString(); **if**(**title** == **null**)  **title** = **" "**;  *//close virtual keyboard*  InputMethodManager inputManager = (InputMethodManager) getSystemService(Context.***INPUT\_METHOD\_SERVICE***);  inputManager.hideSoftInputFromWindow(getCurrentFocus().getWindowToken(), |

|  |
| --- |
| InputMethodManager.***HIDE\_NOT\_ALWAYS***);  Toast.*makeText*(StartDSP.**this**, **"Description saved"**, Toast.***LENGTH\_SHORT***).show();  **comment**.setVisibility(View.***INVISIBLE***); **done**.setVisibility(View.***INVISIBLE***);  **TextHandleRemainingImpulses** = (TextView) findViewById(R.id.***remaining\_impulses***);  **TextHandleRemainingImpulses**.setText(getResources().getString (R.string.***remaining\_impulse\_leadtext***) + Integer.*toString*(getResources().getInteger(R.integer.***num\_impulses***)));  **textViewTime** = (TextView)findViewById(R.id.***textViewTime***);  **captureAudio** = **new** CaptureAudio(); **captureAudio**.execute();  }  });  }  @Override  **protected void** onPause(){  **if**(**captureAudio** != **null**) **captureAudio**.cancel(**false**);  **super**.onPause(); finish();}  *//countdown timer to show recording time remaining*  **public class** CounterClass **extends** CountDownTimer {  **public** CounterClass(**long** millisInFuture, **long** countDownInterval) {  **super**(millisInFuture, countDownInterval);} @Override  **public void** onFinish() {  **textViewTime**.setText(**"Captured"**);} @Override  **public void** onTick(**long** millisUntilFinished) {  **long** millis = millisUntilFinished;  String hms = String.*format*(**"%02d"**, TimeUnit.***MILLISECONDS***.toSeconds(millis) – TimeUnit.***MINUTES***.toSeconds(TimeUnit.***MILLISECONDS***.toMinutes(millis)));  System.***out***.println(hms);  **textViewTime**.setText(hms);  }  }  Listing 5A – StartDSP.java setting up sound recording |

|  |
| --- |
| <**menu xmlns:android=**[**"http://sche**](http://schemas.android.com/apk/res/android)**m**[**as.android.com/apk/res/android"**](http://schemas.android.com/apk/res/android) **xmlns:app="**[**http://schemas.android.com/apk/res-auto**](http://schemas.android.com/apk/res-auto)**" xmlns:tools=**[**"http://sche**](http://schemas.android.com/tools)**m**[**as.android.com/tools**](http://schemas.android.com/tools)**" tools:context="com.example.orchisamadas.sound\_record\_analyse.StartDSP"**>  <**item**  **android:id="@+id/GenerateChirp" android:icon="@drawable/ic\_play\_arrow\_black\_24dp" app:showAsAction="always" android:title="@string/audio\_play"**/>  </**menu**>  Listing 5B – menu\_start\_dsp.xml |

The *onStart()* method is executed first. We let the user input a comment about the sound he is about to capture, which we save in a String called ‗title‘. To do so, we add an EditText widget beside a TextView widget, as given in. Once the user finishes inserting the description, we display the information about recording time and number of impulses remaining. We then call the CaptureAudio to start recording.

CounterClass generates a timer to countdown from 5s whenever we are recording an impulse. In the layout file *activity\_start\_dsp.xml* we add a TextView widget, ‗textViewTime‘ which shows the recording time remaining. We modify ‗textViewTime‘ in CounterClass every second to show the countdown. On finishing recording all impluses, ‗textViewTime‘ shows the text ‗Captured‘.

In**,** I describe the method to capture audio and save it as an array of short values. The class CaptureAudio extends ‗*AsyncTask*‘. A process can run on multiple threads within the Android system. When the application first runs, it will use the User Interface (UI) thread to control everything we see on the screen. While doing shorter operations on this thread is acceptable, doing longer operations may cause the system to stop responding to user interaction, causing the user to think that the program is running slowly or has stopped running. To fix this, Android uses *AsyncTask* class so that you can shift longer operations to different threads, and keep the main UI thread running smoothly. An asynchronous task is defined on Android using three types, Async<Params, Progress, Result> and four steps: *onPreExecute*, *doInBackground*, *onProgressUpdate* and *onPostExecute* in Article 3 .

*onPreExecute* is the first step to be invoked and sets up our *StartDSP* activity. This task initializes the AudioRecorder using the following format: *AudioRecord (int audioSource, int sampleRateInHz, int channelConfig, int audioFormat, int bufferSizeInBytes)*. Our audio source is the device‘s microphone. The sample rate and number of channels have been configured in the *ints.xml* file given later. The audio format ―ENCODING\_PCM\_16BIT‖ means that our audio buffer will be filled with signed integer values ranging from the maximum value of -327637 to a minimum value of 32768. If initialisation fails, a warning toast message is displayed and the recorder is released.

|  |
| --- |
| **private class** CaptureAudio **extends** AsyncTask<Void, Integer, Integer>{  **protected void** onPreExecute(){  **int** bufferSize=2\*AudioRecord.*getMinBufferSize*(getResources().getInteger(R.integer.***sample\_rate***), getResources().getInteger(R.integer.***num\_channels***),AudioFormat.***ENCODING\_PCM\_16BIT***); **recorder**= **new** AudioRecord(AudioSource.***MIC***,getResources().getInteger(R.integer.***sample\_rate***),  getResources().getInteger(R.integer.***num\_channels***),AudioFormat.***ENCODING\_PCM\_16BIT***,bufferSize);  **if**(**recorder**.getState()!=AudioRecord.***STATE\_INITIALIZED***){ Toast.*makeText*(StartDSP.**this**, getResources().getString(R.string.***recorder\_init\_fail***),Toast.***LENGTH\_LONG***).show(); **recorder**.release();  **recorder**=**null**; **return**;}  }  **protected** Integer doInBackground(Void ... params) {  **if** (**recorder** == **null**) {  **return** -1;} |

|  |
| --- |
| **int** remainingImpulses = getResources().getInteger(R.integer.***num\_impulses***);  *//length = sampleRate \* recordTime*  **int** detectBufferLength = getResources().getInteger(R.integer.***detect\_buffer\_length***);  **int** sampleBufferLength = getResources().getInteger(R.integer.***sample\_rate***) \*  getResources().getInteger(R.integer.***capture\_time***);  sampleBufferLength =  *nearestPow2Length*(sampleBufferLength);  **short**[] detectBuffer = **new short**[detectBufferLength];  **short**[][] sampleBuffer = **new short**[remainingImpulses][sampleBufferLength]; **recorder**.startRecording();  **while** (remainingImpulses > 0) { publishProgress(-1, -1, -1, -1); **int** samplesRead = 0;  **while** (samplesRead < detectBufferLength)  samplesRead += **recorder**.read(detectBuffer, samplesRead, detectBufferLength - samplesRead);  **if** (detectImpulse(detectBuffer)) { remainingImpulses--;  publishProgress(0, remainingImpulses, -1, -1);  System.*arraycopy*(detectBuffer, 0, sampleBuffer[remainingImpulses], 0, detectBufferLength); samplesRead = detectBufferLength;  **while** (samplesRead < sampleBufferLength)  samplesRead += **recorder**.read(sampleBuffer[remainingImpulses], samplesRead, sampleBufferLength  - samplesRead);}  **if** (isCancelled()) { detectBuffer = **null**; sampleBuffer = **null**; **return** -1;}  }*//end while(remainingImpulses > 0)*  detectBuffer = **null**;  **if** (**recorder** != **null**) { **recorder**.release(); **recorder** = **null**;}  **if** (!isCancelled()) { publishProgress(-1, -1, 0, -1);}  *//save recorded audio to an external file in memory card to enable playback option*  saveRecord(sampleBuffer, sampleBufferLength);  Listing 5C – StartDSP.java - capture audio |

In theory, all we need to do now is to start recording the noise. The buffer, which is internal to the AudioRecord instance, will be filled up with data. While recording, we need a number of operations to update the user interface. Once again, these operations don‘t have to be done in parallel with the recording task, which takes the most time. The *doInBackground* method is called to operate the background computation, which can take time. We use it to initialize the number of records and the buffers before starting the record using the previously initialized MediaRecorder *recorder*. Until the number of records remaining reaches 0, we save the recorded data into a buffer and make sure that this buffer is full before computing a result.

If an impulse above the background sound level is detected (*detectImpulse*) we start to collect data, decrease the number of records remaining, update the UI thread and fill the buffer *sampleBuffer* with the current data plus the data that has just been captured. An impulse is detected only if the noise level is beyond a certain threshold. In case you want to detect all sounds, change the value of threshold in *ints.xml* to 0. To save the sound that we have just captured, we call the method *saveRecord*

On the recorded audio, we first do an FFT, and store the results in our database . To do so, we first normalize our recorded data to keep the maximum and minimum values between +1 and -1. We then apply a smoothing effect on our data to smooth its edges. This is also known as windowing. The need for windowing is explained in [9]. The *applyBasicWindow()* function is described in. Once the signal has been windowed, we do a DFT on it by calling the function doubleFFT, which in turn calls the class *FFTBase*. The radix 2 FFT algorithm executes a lot faster if the number of samples in the signal is of a power of 2. So, we write the function *nearestPow2Length()*to adjust the signal length to the nearest power of 2 before computing its FFT.

After *doubleFFT()* is called, it returns an array of FFT magnitude values in the matrix *sample*. Another array *toStorage* stores the average magnitude values of FFT of all the impulses. We take a moving average filter of 32 samples to smooth our spectrum. The individual smoothed magnitude spectrum values of each impulse is stored in *tempImpBuffer[ ]* and the average smoothed spectrum of all impulses is stored in *tempBuffer[ ]*. The corresponding frequency values are stored in *xVals*.

The next part is storing our arrays into the table *fft\_data*. We use something called *ContentValues* to add the data to the appropriate columns. We create an object of *SQLiteDatabaseHelper* and the command *getWritableDatabase*() allows us to write into the database. To store data as BLOBs, we use *ByteBuffe*r arrays which store numbers as bytes.

The *insert* command inserts *tempBuffer[ ]* into the column ―yvals‖, *tempImpBuffer[N]* in ―impulsenoN‖ (N stands for the nth impulse), *xVals[ ]* into ―xvals‖, the String ‗title‘ we obtained from the user into

―comments\_fft‖ and the current date and time into ―dateTime‖.

|  |
| --- |
| *//doing FFT*  **final int** numImpulses = getResources().getInteger(R.integer.***num\_impulses***);  **double**[][] samples = **new double**[numImpulses][sampleBufferLength];  *//normalizing time domain data*  **for** (**int** k = 0; k < numImpulses; k++) {  **double** max = 0;  **for** (**int** n = 0; n < sampleBufferLength; n++) { samples[k][n] = (**double**) sampleBuffer[k][n]; **if** (max < samples[k][n]) {  max = samples[k][n];  }  }  **for** (**int** h = 0; h < sampleBufferLength; h++) { samples[k][h] /= max;  }  } |

|  |
| --- |
| sampleBuffer = **null**;  *//we apply a slight smoothing effect to the edge of the sample to improve*  *//our result*  applyBasicWindow(samples, numImpulses, sampleBufferLength);  *//do FFT*  **int** error = doubleFFT(samples, numImpulses, sampleBufferLength);  **if** (error == -1) {  **if** (!isCancelled()) { publishProgress(-1, -1, -1, 0);  }  sampleBuffer = **null**; **return** -1;  }  */\*Store the FFT results into table fft\_data.*  *Here we average all the samples to compute the averaged data set\*/*  **double**[] toStorage = **new double**[sampleBufferLength]; **for**(**int** k = 0; k < numImpulses; k++)  {  **for**(**int** n = 0; n < sampleBufferLength; n++) toStorage[n] += samples[k][n]/***REFSPL***;  }  **for**(**int** n = 0; n < sampleBufferLength; n++) toStorage[n] /= numImpulses;  **if**(isCancelled()) **return** -1;  *//reduce the size of our sample*  **int** samplesPerPoint = getResources().getInteger(R.integer.***samples\_per\_bin***);  **int** width = toStorage.**length** / samplesPerPoint / 2;  **double** maxYval = 0;  **double**[] tempBuffer = **new double**[width]; **for**(**int** k = 0; k < tempBuffer.**length**; k++)  {  **for**(**int** n = 0; n < samplesPerPoint; n++)  tempBuffer[k] += toStorage[k\*samplesPerPoint + n]; tempBuffer[k] /= (**double**)samplesPerPoint;  **if**(maxYval < tempBuffer[k]) maxYval = tempBuffer[k];  }  ContentValues vals = **new** ContentValues(); ContentValues values = **new** ContentValues(); MySQLiteDatabaseHelper databaseHelper = **new**  MySQLiteDatabaseHelper(StartDSP.**this**); SQLiteDatabase db = databaseHelper.getWritableDatabase();  *//we're going to save every single impulse separately*  **for**(**int** i = 0; i < numImpulses; i++)  {  **double** maxTemp = 0;  **double**[] tempImpBuffer = **new double**[width]; **for**(**int** k = 0; k < tempImpBuffer.**length**; k++)  { |

|  |
| --- |
| **for**(**int** n = 0; n < samplesPerPoint; n++)  tempImpBuffer[k] += (samples[i][k\*samplesPerPoint + n] / ***REFSPL***); tempImpBuffer[k] /= (**double**)samplesPerPoint;  **if**(maxTemp < tempImpBuffer[k]) maxTemp = tempImpBuffer[k];  }  ByteBuffer byteImpBuffer = ByteBuffer.*allocate*(width\*8);  **for**(**int** k = 0; k < width; k++) byteImpBuffer.putDouble(tempImpBuffer[k]);  vals.put(TableEntry.***COLUMN\_NAME\_IMPULSE*** + Integer.*toString*(i), byteImpBuffer.array());  }  **double**[] xVals = **new double**[tempBuffer.**length**];  **double** sampleRate = getResources().getInteger(R.integer.***sample\_rate***);  **for**(**int** k = 0; k < xVals.**length**; k++)  xVals[k] = k\* sampleRate / (2\*xVals.**length**);  ByteBuffer byteBufferY = ByteBuffer.*allocate*(tempBuffer.**length**\*8);  **for**(**int** k = 0; k < tempBuffer.**length**; k++) byteBufferY.putDouble(tempBuffer[k]);  vals.put(TableEntry.***COLUMN\_NAME\_YVALS***, byteBufferY.array()); ByteBuffer byteBufferX = ByteBuffer.*allocate*(xVals.**length**\*8); **for**(**int** k = 0; k < xVals.**length**; k++)  byteBufferX.putDouble(xVals[k]); vals.put(TableEntry.***COLUMN\_NAME\_XVALS***, byteBufferX.array());  String date = DateFormat.*format*(**"LLL dd, yyyy HH:mm"**, **new** Date()).toString(); vals.put(TableEntry.***COLUMN\_NAME\_DATE***, date); vals.put(TableEntry.***COLUMN\_NAME\_COMMENT***, **" - "** + **title**); db.insert(TableEntry.***TABLE\_NAME\_FFT***, **null**, vals);  Listing 5D – Do FFT and store in database |

The next bit is where we do some real analysis on the data to figure out what frequencies are most prominent in the signal As mentioned before, we want to calculate two metrics – *Percentage Worse Case Frequencies* and *Ratio Background Noise Frequencies*. To do so, we first pass a moving average filter of width 1Hz over the entire frequency range. Since we are interested in low frequency noise analysis, we only keep frequency domain information till 300Hz and discard the rest. Then, we break the frequency range into 32 overlapping frequency bands of 15Hz each, with upper and lower limits of each band given by:

𝐿𝑜𝑤𝑒𝑟 𝑙i𝑚i𝑡 = 2 + 5𝑛 𝐻𝑧 𝑈𝑝𝑝𝑒𝑟 𝐿i𝑚i𝑡 = 17 + 5𝑛 𝐻𝑧 ; 0 ≤ 𝑛 ≤ 31

So, the final frequency range is 2 Hz to 172 Hz.

To calculate *Percentage Worse Case frequencies*, i.e., the most prominent peaks in the magnitude spectrum, we find out the strongest signal in the entire frequency range. A noise nuisance is said to have occurred if the power of a particular frequency in a frequency band was greater than a specified percentage of the strongest signal power. We would expect such signals to stand out from the background noise. We calculate the ratio of strength of signal at particular frequency to the strongest signal strength.

If this ratio is greater than a certain *threshold*, we consider it to be a *Percentage Worse Case Frequency*. The *threshold* is given by the formula:

𝑡ℎ𝑟𝑒𝑠ℎ𝑜𝑙𝑑 = 1 − 0.5 × 𝑚𝑒𝑎𝑛 𝑑𝑒𝑣i𝑎𝑡i𝑜𝑛 𝑜ƒ 𝑠i𝑔𝑛𝑎𝑙 ƒ𝑟𝑜𝑚 𝑠𝑡𝑟𝑜𝑛𝑔𝑒𝑠𝑡 𝑠i𝑔𝑛𝑎𝑙 𝑠𝑡𝑟𝑒𝑛𝑔𝑡ℎ

𝑠𝑡𝑟𝑜𝑛𝑔𝑒𝑠𝑡 𝑠i𝑔𝑛𝑎𝑙 𝑠𝑡𝑟𝑒𝑛𝑔𝑡ℎ

*Ratio Background Noise frequencies* are the frequencies that are consistently present in the signal but have been overshadowed by the background noise. They are significantly stronger than the average strength of the weakest frequency band. So we first find out the weakest band by finding the frequency band which has the maximum number of weak peaks. A signal is said to be weak if its ratio to the average band strength is lesser than a certain value called *weakThreshold*.

𝑤𝑒𝑎𝑘𝑇ℎ𝑟𝑒𝑠ℎ𝑜𝑙𝑑 = 1 − 1.5×𝜎𝑏𝑎𝑛𝑑

𝜇𝑏𝑎𝑛𝑑

𝑤ℎ𝑒𝑟𝑒 𝜎𝑏𝑎𝑛𝑑= 𝑠𝑡𝑎𝑛𝑑𝑎𝑟𝑑 𝑑𝑒𝑣i𝑎𝑡i𝑜𝑛 𝑜ƒ 𝑏𝑎𝑛𝑑 𝑠𝑡𝑟𝑒𝑛𝑔𝑡ℎ, 𝜇𝑏𝑎𝑛𝑑 = 𝑚𝑒𝑎𝑛 𝑏𝑎𝑛𝑑 𝑠𝑡𝑟𝑒𝑛𝑔𝑡ℎ

Once we find out the weakest band, we calculate the ratio of signal strength to average strength of weakest band. If it is greater than a certain limit, we consider the frequency to be a *Ratio Background Noise Frequency.* The limit is given by:

𝑙i𝑚i𝑡 = 1 + 1.5 × 𝑚𝑒𝑎𝑛 𝑑𝑒𝑣i𝑎𝑡i𝑜𝑛 𝑜ƒ 𝑠i𝑔𝑛𝑎𝑙 ƒ𝑟𝑜𝑚 𝑎𝑣𝑔. 𝑠𝑡𝑟𝑒𝑛𝑔𝑡ℎ 𝑜ƒ 𝑤𝑒𝑎𝑘𝑒𝑠𝑡 𝑏𝑎𝑛𝑑

𝑚𝑒𝑎𝑛 𝑠𝑡𝑟𝑒𝑛𝑔𝑡ℎ 𝑜ƒ 𝑤𝑒𝑎𝑘𝑒𝑠𝑡 𝑏𝑎𝑛𝑑

After doing all this calculation, we sort and store the *Percentage Worse Case frequencies* and *Ratio Background Noise Frequencies* in an *ArrayLists*, which we save as ‗REAL‘ values in our table *analysis\_data* in columns ―percentage\_worse\_case‖ and ―ratio\_background\_noise‖*,* along with the String

‗title‘ in ―comments‖ and the date and time in ―dateTime‖*.*

|  |
| --- |
| *//Do DSP here*  **for** (**int** i = 0; i < numImpulses; i++) {  *//Generating average over 1 Hz*  **double** averageOver = 1 / (**double**) getResources().getInteger(R.integer.***averageOverDenominator***);  *//sampleBufferLength = numPts in Matlab =32768*  **double** freqDeltaF = (**double**) (sampleRate) / sampleBufferLength;  **int** ptsAverageOverFreq = (**int**) Math.*floor*(averageOver / freqDeltaF);  **int** numPtsAfterAverage = (**int**) Math.*floor*(sampleBufferLength / ptsAverageOverFreq);  *//we only want to keep values till 300Hz for our analysis*  **int** upperLimitFreq = 300;  **double** freqDeltaFAfterAverage = (**double**) (sampleRate) / numPtsAfterAverage;  **int** ptsTillUpperLimit = (**int**) Math.*floor*((**double**) (upperLimitFreq) / freqDeltaFAfterAverage);  **double**[] arrayOfFFTAverages = **new double**[ptsTillUpperLimit]; **double**[] arrayOfFreqAverages = **new double**[ptsTillUpperLimit]; **for** (**int** n = 0; n < ptsTillUpperLimit; n++) {  **for** (**int** k = 0; k < ptsAverageOverFreq; k++) { arrayOfFFTAverages[n] += samples[i][n \* ptsAverageOverFreq + k];  }  arrayOfFFTAverages[n] /= ptsAverageOverFreq;  }  **for** (**int** k = 0; k < ptsTillUpperLimit; k++) {  arrayOfFreqAverages[k] = ((**double**) (sampleRate) / (numPtsAfterAverage)) \* k; |

|  |
| --- |
| }  *//breaking into frequency bands*  **int** numPtsInEachBand = (**int**) Math.*floor*(15 / freqDeltaFAfterAverage); **double**[][] freqBandYvals = **new double**[32][numPtsInEachBand]; **double**[][] freqBandXvals = **new double**[32][numPtsInEachBand];  **for** (**int** n = 0; n <= 31; n++) {  **int** startFreq = 2 + (5 \* n);  **int** startingPt = (**int**) Math.*floor*(startFreq / freqDeltaFAfterAverage);  **for** (**int** k = 0; k < numPtsInEachBand; k++) {  freqBandYvals[n][k] = (arrayOfFFTAverages[startingPt + k] / ***REFSPL***); freqBandXvals[n][k] = arrayOfFreqAverages[startingPt + k];  }  }  *//identify strongest signal and average band power* **double**[] avgBandPower = **new double**[32]; **double** strongestSignal = 0;  **double** strongestSignalFreq = 0;  **for** (**int** n = 0; n <= 31; n++) {  **for** (**int** k = 0; k < numPtsInEachBand; k++) {  **if** (freqBandYvals[n][k] > strongestSignal) { strongestSignal = freqBandYvals[n][k]; strongestSignalFreq = freqBandXvals[n][k];  }  avgBandPower[n] += freqBandYvals[n][k];  }  avgBandPower[n] /= numPtsInEachBand;  }  *//calculating percentage worse case -- these are the frequenices which have maximum power.*  **double** dev = 0;  **for** (**int** n = 0; n <= 31; n++) {  **for** (**int** k = 0; k < numPtsInEachBand; k++)  dev += Math.*pow*(freqBandYvals[n][k] - strongestSignal,2);  }  dev /= (32\*numPtsInEachBand); dev = Math.*sqrt*(dev);  **double** threshold = 1 -(0.5 \* dev)/strongestSignal;  List<Double> percentageWorseCase = **new** ArrayList<Double>();  **for** (**int** n = 0; n <= 31; n++) {  **for** (**int** k = 0; k < numPtsInEachBand; k++) {  **if** (freqBandYvals[n][k] / strongestSignal >= threshold) percentageWorseCase.add(freqBandXvals[n][k]);  }  }  *//removing repeated frequencies and sorting arrayList* Set<Double> unique = **new** HashSet<Double>(); unique.addAll(percentageWorseCase); percentageWorseCase.clear(); percentageWorseCase.addAll(unique); Collections.*sort*(percentageWorseCase);  *//calculating Ratio Background Noise* **double** [] std = **new double**[32]; **double** weakThreshold =0;  **for** (**int** n = 0; n <= 31; n++) {  **for** (**int** k = 0; k < numPtsInEachBand; k++)  std[n] += Math.*pow*(freqBandYvals[n][k] - avgBandPower[n], 2); |

|  |
| --- |
| std[n] /= numPtsInEachBand; std[n] = Math.*sqrt*(std[n]);  }  **int**[] numberWeakPeaks = **new int**[32]; **for** (**int** n = 0; n <= 31; n++) {  weakThreshold = 1 - (1.5 \* std[n])/avgBandPower[n];  **for** (**int** k = 0; k < numPtsInEachBand; k++) {  **if** (freqBandYvals[n][k] / avgBandPower[n] <= weakThreshold) numberWeakPeaks[n]++; }  }  *//calculating frequency band with least power*  **int** lowestPeakBand = 0;  **double** avgPowerLowestPeakBands = 0;  **int** min = numberWeakPeaks[0];  **for** (**int** n = 0; n <= 31; n++) {  **if** (numberWeakPeaks[n] <= min) { min = numberWeakPeaks[n]; lowestPeakBand = n; }  }  *//calculating average power of lowestPeakBand*  **for** (**int** k = 0; k < numPtsInEachBand; k++) avgPowerLowestPeakBands += freqBandYvals[lowestPeakBand][k];  avgPowerLowestPeakBands /= numPtsInEachBand; dev = 0;  **for** (**int** n = 0; n <= 31; n++) {  **for** (**int** k = 0; k < numPtsInEachBand; k++)  dev += Math.*pow*(freqBandYvals[n][k] - avgPowerLowestPeakBands,2);} dev /= (32\*numPtsInEachBand);  dev = Math.*sqrt*(dev);  **double** limit = 1 + (dev)/avgPowerLowestPeakBands; List<Double> ratioBackgroundNoise = **new** ArrayList<Double>(); **for** (**int** n = 0; n <= 31; n++) {  **for** (**int** k = 0; k < numPtsInEachBand; k++) {  **if** (freqBandYvals[n][k] / avgPowerLowestPeakBands >= limit) ratioBackgroundNoise.add(freqBandXvals[n][k]); }  }  Set<Double> uniqueElements = **new** HashSet<Double>(); uniqueElements.addAll(ratioBackgroundNoise); ratioBackgroundNoise.clear(); ratioBackgroundNoise.addAll(uniqueElements); Collections.*sort*(ratioBackgroundNoise);  *//inserting into database*  db = databaseHelper.getWritableDatabase();  **int** minimum;  **int** maximum;  **if** (percentageWorseCase.size() > ratioBackgroundNoise.size()) { maximum = percentageWorseCase.size();  minimum = ratioBackgroundNoise.size();  } **else** {  maximum = ratioBackgroundNoise.size(); minimum = percentageWorseCase.size();}  **for** (**int** n = 0; n < minimum; n++) {  values.put(TableEntry.***COLUMN\_NAME***, **"IMPULSE"** + Integer.*toString*(i)); values.put(TableEntry.***COLUMN\_DATE***, date); values.put(TableEntry.***COLUMN\_COMMENT***, **" - "** + **title**); values.put(TableEntry.***COLUMN\_MAX\_SIGNAL***, strongestSignalFreq); |

|  |
| --- |
| values.put(TableEntry.***COLUMN\_PERCENTAGE\_WORSE\_CASE***, percentageWorseCase.get(n)); values.put(TableEntry.***COLUMN\_RATIO\_BACKGROUND\_NOSE***, ratioBackgroundNoise.get(n)); db.insert(TableEntry.***TABLE\_NAME***, **null**, values);  }  **for** (**int** n = minimum; n < maximum; n++) { values.put(TableEntry.***COLUMN\_NAME***, **"IMPULSE"** + Integer.*toString*(i)); values.put(TableEntry.***COLUMN\_DATE***, date); values.put(TableEntry.***COLUMN\_COMMENT***, **" - "** +**title**); values.put(TableEntry.***COLUMN\_MAX\_SIGNAL***, strongestSignalFreq);  **if** (maximum == ratioBackgroundNoise.size()) { values.put(TableEntry.***COLUMN\_PERCENTAGE\_WORSE\_CASE***, 0.0); values.put(TableEntry.***COLUMN\_RATIO\_BACKGROUND\_NOSE***, ratioBackgroundNoise.get(n));  } **else** {  values.put(TableEntry.***COLUMN\_PERCENTAGE\_WORSE\_CASE***, percentageWorseCase.get(n)); values.put(TableEntry.***COLUMN\_RATIO\_BACKGROUND\_NOSE***, 0.0); }  db.insert(TableEntry.***TABLE\_NAME***, **null**, values);  }  **int** prog = (**int**) 100 \* (i + 1) / numImpulses; publishProgress(-1, -1, prog, -1);  }  db.close();  **if** (isCancelled())  **return** -1;  **else**  **return** 0;  }  – Data analysis and storage |

In the last part of this activity, , we implement the *onProgressUpdate()* method. o*nProgressUpdate* displays the task progress on the UI while the background task is still executing. This method works in parallel with *publishProgress().*The method *publishProgress()* loads the ProgressBar widget depending on the progress of the computation. The values passed in parameters in *publishProgress* are stored in a data array that is given to the *onProgressUpdate* method. *data* starts the countdown timer whenever a new impulse is detected. *data[1]* decreases the value of the TextView widget ‗*remaining\_impulses*‘ by one each time 5s of sound is recorded, *data* updates the ProgressBar and displays the TextWidget ‗Analysing…‘ below the Progress Bar and *data* shows a computation error Toast if the FFT computation fails.

The *onPostExecute()* method is executed once *doInBackground()* has finished. This displays an error if something goes wrong while computing and enables the ‗*View Frequency Graphs*‘ and ‗*View Analysis Results‟* buttons. Pressing on either starts the ‗*DisplayGraph*‘ Activity. It also enables the audio playback button.

The method *detectImpluse()* is used to detect an impulse if the sound exceeds a certain threshold. *doubleFFT(*) calls the *FFTBase* class to get the magnitude spectrum of the signal. The method *applyBasicWindow()* smooths the signal by multiplying the first few and the last few samples with a gradually decaying series of fractions lesser than 1. *nearestPow2Length()* adjusts the signal length to the nearest power of 2. *saveRecord()* saves the buffer of captured data to an external file called

―recordedSound.wav‖ and playbackAudio() uses *AudioTrack* to read and playback values from

―recordedSound.wav‖ if user presses the ―playback‖ button.

|  |
| --- |
| **protected void** onProgressUpdate(Integer ... data){  **if**(data[0] == 0) {**timer**.start();}  **if**(data[1] != -1) **TextHandleRemainingImpulses**.setText(getResources().getString (R.string.***remaining\_impulse\_leadtext***) + Integer.*toString*(data[1]));  **if**(data[2] != -1){ **TextHandleRemainingImpulses**.setVisibility(TextView.***INVISIBLE***); ProgressBar temp = (ProgressBar) findViewById(R.id.***computation\_progress***); temp.setVisibility(View.***VISIBLE***); temp.setProgress(data[2]);  TextView showProgress = (TextView) findViewById(R.id.***analysing***); showProgress.setText(**"Analysing..."**); showProgress.setVisibility(View.***VISIBLE***);}  **if**(data[3] != -1)  Toast.*makeText*(StartDSP.**this**, getResources().getString(R.string.***computation\_error***), Toast.***LENGTH\_LONG***).show();}  **protected void** onPostExecute(Integer data){  **if**(**recorder** != **null**){ **recorder**.release(); **recorder** = **null**;} **if**(data == -1){  Toast.*makeText*(StartDSP.**this**, getResources().getString(R.string.***error***), Toast.***LENGTH\_LONG***).show();}  **else** {  *//allowing user to playback on pressing a button*  ImageButton playback = (ImageButton) findViewById(R.id.***playback***); playback.setVisibility(View.***VISIBLE***); playback.setOnClickListener(**new** View.OnClickListener() {  @Override  **public void** onClick(View v) { playbackAudio();  }  });  TextView showProgress = (TextView) findViewById(R.id.***analysing***); showProgress.setText(**"Analysis Complete"**);  */ Start the DisplayGraph activity on click of a button. Button 1 displays FFT graph* Button FFTbutton = (Button) findViewById(R.id.***btnDisplayGraph***); FFTbutton.setVisibility(View.***VISIBLE***);  FFTbutton.setOnClickListener(**new** View.OnClickListener() {  **public void** onClick(View arg0) { String which\_button\_pressed = **"1"**; Bundle bundle = **new** Bundle();  bundle.putString(**"button\_pressed"**, which\_button\_pressed); Intent intent = **new** Intent(StartDSP.**this**, DisplayGraph.**class**); intent.putExtras(bundle);  startActivity(intent);  }  });  *//Button 2 displays Analysis Histogram*  Button Histbutton = (Button) findViewById(R.id.***btnDisplayHistogram***); Histbutton.setVisibility(View.***VISIBLE***); Histbutton.setOnClickListener(**new** View.OnClickListener() {  **public void** onClick(View arg0) {  String which\_button\_pressed = **"2"**; Bundle bundle = **new** Bundle(); |

|  |
| --- |
| bundle.putString(**"button\_pressed"**, which\_button\_pressed); Intent intent = **new** Intent(StartDSP.**this**, DisplayGraph.**class**); intent.putExtras(bundle);  startActivity(intent);  }  });  **return**;  }}  **protected void** onCancelled(){  **if**(**recorder** != **null**){**recorder**.release();**recorder** = **null**;}}  **protected boolean** detectImpulse(**short**[] samples){  **int** threshold = getResources().getInteger(R.integer.***detect\_threshold***);  **for**(**int** k = 0; k < samples.**length**; k++){  **if**(samples[k] >= threshold){**return true**;}} **return false**;  }  **protected int** doubleFFT(**double**[][] samples, **int** numImpulses, **int** sampleSize){ **double**[] real = **new double**[sampleSize]; **double**[] imag = **new double**[sampleSize]; **for**(**int** k = 0; k < numImpulses; k++){  System.*arraycopy*(samples[k], 0, real, 0, sampleSize);  **for**(**int** n = 0; n < sampleSize; n++)  imag[n] = 0; **int** error = FFTbase.*fft*(real, imag, **true**); **if**(error == -1) {**return** -1;}  **for**(**int** n = 0; n < sampleSize; n++)  samples[k][n] = Math.*sqrt*(real[n]\*real[n] + imag[n]\*imag[n]);  **if**(isCancelled()) {**return** -1;}  }  **return** 0;}}  **protected void** applyBasicWindow(**double**[][] samples, **int** numImpulses, **int** sampleLength)  {  **for**(**int** k = 0; k < numImpulses; k++)  {  samples[k][0] \*= 0.0625;  samples[k][1] \*= 0.125;  samples[k][2] \*= 0.25;  samples[k][3] \*= 0.5;  samples[k][4] \*= 0.75;  samples[k][5] \*= 0.875;  samples[k][6] \*= 0.9375;  samples[k][sampleLength - 7] \*= 0.9375;  samples[k][sampleLength - 6] \*= 0.875;  samples[k][sampleLength - 5] \*= 0.75;  samples[k][sampleLength - 4] \*= 0.5;  samples[k][sampleLength - 3] \*= 0.25;  samples[k][sampleLength - 2] \*= 0.125;  samples[k][sampleLength - 1] \*= 0.0625;  }  **return**;  }  **public static int** nearestPow2Length(**int** length){  **int** temp = (**int**) (Math.*log*(length) / Math.*log*(2.0) + 0.5);length = 1;  **for**(**int** n = 1; n <= temp; n++) {length = length \* 2;}  **return** length;}  *//save recorded data in an external file to enable user to playback*  **public void** saveRecord(**short** sampleBuffer[][], **int** sampleBufferLength){ |

|  |
| --- |
| **final int** numImpulses = getResources().getInteger(R.integer.***num\_impulses***);  File file = **new** File(Environment.*getExternalStorageDirectory*(), **"recordedSound.wav"**); **if** (file.exists())  file.delete();  **try** {  file.createNewFile();  } **catch** (IOException e) {}  **try** {  OutputStream os = **new** FileOutputStream(file); BufferedOutputStream bos = **new** BufferedOutputStream(os); DataOutputStream dos = **new** DataOutputStream(bos);  **for** (**int** k = numImpulses -1; k >= 0; k--) {  **for** (**int** n = 0; n < sampleBufferLength; n++) dos.writeShort(sampleBuffer[k][n]);  }}  **catch**(IOException e){}  }  *//playback record*  **public void** playbackAudio(){  File file = **new** File(Environment.*getExternalStorageDirectory*(), **"recordedSound.wav"**);  *// Get the length of the audio stored in the file (16 bit so 2 bytes per short)*  **int** audioLength = (**int**)(file.length()/2); **short** [] audio = **new short**[audioLength]; **try** {  InputStream is = **new** FileInputStream(file); BufferedInputStream bis = **new** BufferedInputStream(is); DataInputStream dis = **new** DataInputStream(bis);  **int** n = 0;  **while** (dis.available() > 0) { audio[n] = dis.readShort(); n++;}  dis.close();}  **catch**(IOException e){}  *// Create a new AudioTrack object using the same parameters as the AudioRecord*  AudioTrack audioTrack = **new** AudioTrack(AudioManager.***STREAM\_MUSIC***, 8000,  AudioFormat.***CHANNEL\_CONFIGURATION\_MONO***,  AudioFormat.***ENCODING\_PCM\_16BIT***, audioLength, AudioTrack.***MODE\_STREAM***);  *// Start playback*  audioTrack.play();  *// Write the audio buffer to the AudioTrack object*  audioTrack.write(audio, 0, audioLength);  }  }  Listing 5F – After analysis and storage |

|  |
| --- |
| **package** com.example.orchisamadas.sound\_record\_analyse;  **public class** FFTbase {  **public static int** fft(**final double**[] inputReal, **double**[]inputImag, **boolean** DIRECT) {  **int** n = inputReal.**length**;  **double** ld = Math.*log*(n) / Math.*log*(2.0);  **if** (((**int**) ld) - ld != 0) {**return** -1;}  **int** nu = (**int**) ld; **int** n2 = n / 2; **int** nu1 = nu - 1;  **double** tReal, tImag, p, arg, c, s;*// check if direct transform or//the inverse transform.*  **double** constant;  **if** (DIRECT) {constant = -2 \* Math.***PI***;}  **else** {constant = 2 \* Math.***PI***;}*// First phase - calculation*  **int** k = 0;  **for** (**int** l = 1; l <= nu; l++) {  **while** (k < n) {  **for** (**int** i = 1; i <= n2; i++) {  p = *bitreverseReference*(k >> nu1, nu); arg = constant \* p / n;  c = Math.*cos*(arg); s = Math.*sin*(arg);  tReal = inputReal[k + n2] \* c + inputImag[k + n2] \* s; tImag = inputImag[k + n2] \* c - inputReal[k + n2] \* s; inputReal[k + n2] = inputReal[k] - tReal;  inputImag[k + n2] = inputImag[k] - tImag; inputReal[k] += tReal; inputImag[k] += tImag; k++;  }  k += n2;  }  k = 0; nu1--; n2 /= 2;  }*// Second phase - recombination*  k = 0; **int** r;  **while** (k < n) {  r = *bitreverseReference*(k, nu);  **if** (r > k) {  tReal = inputReal[k]; tImag = inputImag[k];  inputReal[k] = inputReal[r]; inputImag[k] =inputImag[r]; inputReal[r] = tReal; inputImag[r] = tImag;  } k++;  }  **double** radice = 1 / Math.*sqrt*(n);  **for** (**int** i = 0; i < inputReal.**length**; i++) { inputReal[i] = inputReal[i] \* radice; inputImag[i] = inputImag[i] \* radice;  }  **return** 0;  }*//The reference bitreverse function*  **private static int** bitreverseReference(**int** j, **int** nu) {  **int** j2; **int** j1 = j; **int** k = 0;  **for** (**int** i = 1; i <= nu; i++) {j2 = j1 / 2; k = 2 \* k + j1 - 2 \*j2; j1 = j2; }  **return** k;  }  }  Listing 6 – FFTBase.java |

## 

* *DisplayGraph*

In this activity, I use some graphics to plot the FFT spectrum and histograms of the

*Percentage Worse Case Frequencies* and *Ratio Background Noise Frequencies*. Download the GraphView

*jar* file from and paste it in the *libs*

folder of your project to use it.

The screen for *DisplayGraph* comes with a drop-down menu, with options to switch from FFT plot to histogram and vice versa, compare between two histograms, zoom in and out, record new data, delete all data etc. Let us first look at the menu items in the *menu\_display\_graph.xml* file.

|  |
| --- |
| <**menu xmlns:android=**[**"http://sche**](http://schemas.android.com/apk/res/android)**m**[**as.android.com/apk/res/android"**](http://schemas.android.com/apk/res/android) **xmlns:app="**[**http://schemas.android.com/apk/res-auto**](http://schemas.android.com/apk/res-auto)**" xmlns:tools=**[**"http://sche**](http://schemas.android.com/tools)**m**[**as.android.com/tools**](http://schemas.android.com/tools)**" tools:context="com.example.orchisamadas.analyse\_plot.DisplayHistogram"** >  <**item**  **android:id="@+id/delete\_database" android:orderInCategory="99" android:title="@string/delete\_database\_string" app:showAsAction="never"**/>  <**item**  **android:id="@+id/about" android:orderInCategory="100" android:title="@string/about" app:showAsAction="never"** />  <**item**  **android:id="@+id/history" android:orderInCategory="52" android:title="@string/history" app:showAsAction="ifRoom"** />  <**item**  **android:id = "@+id/zoomIn" android:orderInCategory="51"**  **android:icon = "@drawable/ic\_zoom\_in\_black\_24dp" android:title="Zoom in" app:showAsAction="ifRoom"** />  <**item**  **android:id = "@+id/zoomOut" android:orderInCategory="50"**  **android:icon = "@drawable/ic\_zoom\_out\_black\_24dp" android:title="Zoom Out" app:showAsAction="ifRoom"** />  <**item**  **android:id="@+id/record\_data" android:orderInCategory="98"**  **android:title="@string/start\_dsp" app:showAsAction="never"** /> |

|  |
| --- |
| <**item**  **android:id="@+id/show\_all" android:orderInCategory="97" android:title="Show total" app:showAsAction="never"** />  <**item**  **android:id="@+id/show\_FFT" android:orderInCategory="96" android:title="Show FFT" app:showAsAction="never"** />  <**item**  **android:id="@+id/show\_hist" android:orderInCategory="95" android:title="Show Histogram" app:showAsAction="never"** />  <**item**  **android:id="@+id/compare" android:orderInCategory="94" android:title="Compare with" app:showAsAction="never"** />  </**menu**>  Listing 7A – menu\_display\_graph.xml |
| Fig 4 – MenuInflater menu in DisplayGraph |

tells us about the items in the options menu. Each item is given a unique id and an orderInCategory. Lower the order of the item, higher its position on the drop-down menu. If showAsAction is set to ―ifRoom‖, then the item is shown on the top bar of the screen instead of the drop-down menu.

describes what action is to be performed when one of the items in the menu is selected. Before that, we declare some global variables used throughout the activity.

As mentioned before, we can compare results of two different recordings or see the result of any recording based on the date and time at which it was recorded. We give the user a list of the date and time of recording of all captured sounds along with their comments in the *PickHistory* Activity (explained later) and allow them to choose one. We can look up the table in our database according to the date-time selected by the user, and get values for all the other fields.

*CURRENT\_DATE* is initialised to null by default because we want to show the result of the last captured audio in case no date is picked by the user. When the user wants to compare between two captured sounds, or wants to display the result of an old captured sound, the *PickHistory* Activity is called which returns the date and time picked by the user in *RECEIVED\_DATE*. *request = 1* means that we always request a result from the *PickHistory* Activity. *which\_button\_pressed* determines which plot to load – FFT graph or the histogram. If *SHOW\_ALL* is ―yes‖ then all the impulses are shown separately in the FFT graph; if ―no‖ then their average is shown. If *SHOW\_TOTAL* is ―yes‖, *Percentage Worse Case frequencies* and *Ratio Background Noise frequencies* of all the recorded audio so far are plotted in the histogram. *COMPARE* determines whether comparison of two different captured sounds is to be done or not. *xLabels[]* is the array of x-axis labels for our graphs.

|  |
| --- |
| **package** com.example.orchisamadas.sound\_record\_analyse;  **import** java.nio.ByteBuffer; **import** java.util.ArrayList; **import** java.util.Collections; **import** java.util.List;  **import** com.example.orchisamadas.sound\_record\_analyse.  MySQLiteDatabaseContract.TableEntry;  **import** android.content.Intent;  **import** android.database.Cursor;  **import** android.database.sqlite.SQLiteDatabase;  **import** android.graphics.Color;  **import** android.os.Bundle;  **import** android.support.v7.app.ActionBarActivity;  **import** android.util.Log; **import** android.view.Menu; **import** android.view.MenuItem; **import** android.widget.Toast;  **import** com.jjoe64.graphview.GraphView;  **import** com.jjoe64.graphview.LegendRenderer;  **import** com.jjoe64.graphview.helper.StaticLabelsFormatter; **import** com.jjoe64.graphview.series.BarGraphSeries; **import** com.jjoe64.graphview.series.DataPoint;  **import** com.jjoe64.graphview.series.LineGraphSeries;  **public class** DisplayGraph **extends** ActionBarActivity {  *//if received date is null, current/previous graph is loaded*  **public static** String *RECEIVED\_DATE* = **null**; **public static** String *CURRENT\_DATE* =**null**; **public static** String *which\_button\_pressed* = **null**; **public static final int *request*** = 1;  *//Shows results of all recordings* |

|  |
| --- |
| **public static** String *SHOW\_ALL* = **"NO"**; **public static** String *SHOW\_TOTAL* = **"NO"**; **public static** String *COMPARE* = **"NO"**;  *//setting x axis labels to allow zoom in and out in histograms* **public static** String [] *xLabels* = **new** String [12]; @Override  **protected void** onCreate(Bundle savedInstanceState) {  **super**.onCreate(savedInstanceState);  *//default xaxis labels*  **for**(**int** n = 0;n < *xLabels*.**length**;n++) *xLabels*[n] = Integer.*toString*(20 + (20\*n));  *//RECEIVED\_DATE should always point to last recorded/current data unless History is selected RECEIVED\_DATE* = **null**;  Bundle bundle = getIntent().getExtras(); *which\_button\_pressed* = bundle.getString(**"button\_pressed"**); **if** (*which\_button\_pressed*.equals(**"1"**))  loadFFT(*RECEIVED\_DATE*);  **else if** (*which\_button\_pressed*.equals(**"2"**)) loadHistogram(*RECEIVED\_DATE*);  }  *//Options menu start*  @Override  **public boolean** onCreateOptionsMenu(Menu menu) {  *// Inflate the menu; this adds items to the action bar if it is present.* getMenuInflater().inflate(R.menu.***menu\_display\_graph***, menu); **return true**;  }  *//disabling compare and Show FFT options for FFT graph. Disabling Show Histogram option for*  *//Histogram graph. Enabling Show Histogram and Show FFT options for compare Histogram graph.*  @Override  **public boolean** onPrepareOptionsMenu (Menu menu) {  **if** (*which\_button\_pressed*.equals(**"1"**)) { menu.findItem(R.id.***compare***).setEnabled(**false**); menu.findItem(R.id.***show\_hist***).setEnabled(**true**); menu.findItem(R.id.***show\_FFT***).setEnabled(**false**);  }  **else** {  **if** (*COMPARE*.equals(**"YES"**)) menu.findItem(R.id.***show\_hist***).setEnabled(**true**);  **else**  menu.findItem(R.id.***show\_hist***).setEnabled(**false**); menu.findItem(R.id.***compare***).setEnabled(**true**); menu.findItem(R.id.***show\_FFT***).setEnabled(**true**);  }  **return true**;  }  @Override  **public boolean** onOptionsItemSelected(MenuItem item) {  **int** id = item.getItemId();  **if** (id == R.id.***delete\_database***) {  *//delete all entries*  MySQLiteDatabaseHelper mydb = **new** MySQLiteDatabaseHelper(**this**); SQLiteDatabase db = mydb.getWritableDatabase(); mydb.deleteAllEntries(db, TableEntry.***TABLE\_NAME\_FFT***); mydb.deleteAllEntries(db, TableEntry.***TABLE\_NAME***); |

|  |
| --- |
| *//reload the graph so it displays the "no data" screen*  **if**(*which\_button\_pressed*.equals(**"1"**)) loadFFT(*RECEIVED\_DATE*);  **else**  loadHistogram(*RECEIVED\_DATE*);  db.close(); db = **null**; mydb = **null**;  *//Confirm that all entries were deleted.*  Toast.*makeText*(**this**, getResources().getString(R.string.***deleted\_database***), Toast.***LENGTH\_LONG***)  .show();  **return true**;}  **if**(id == R.id.***about***){  **if**(*which\_button\_pressed*.equals(**"1"**))  Toast.*makeText*(DisplayGraph.**this**, getResources().getString(R.string.***about\_fft***), Toast.***LENGTH\_LONG***).show(); .  **else**  Toast.*makeText*(DisplayGraph.**this**, getResources().getString(R.string.***about\_hist***), Toast.***LENGTH\_LONG***) .show();  **return true**;}  **if**(id == R.id.***history***) {  *//start the activity which displays a list of previous entries*  *//and allows the user to choose one to display*  *//disable show-total if enabled SHOW\_TOTAL* = **"NO"**;  Intent intent = **new** Intent(**this**, PickHistory.**class**); Bundle bundle = **new** Bundle();  bundle.putString(**"button\_pressed"**, *which\_button\_pressed*); intent.putExtras(bundle);  startActivityForResult(intent, ***request***);}  **if**(id == R.id.***record\_data***){  *//starts the StartDSP activity to record more data* Intent intent = **new** Intent(**this**, StartDSP.**class**); startActivity(intent);  finish();  **return true**;}  **if**(id == R.id.***show\_all***)  {  *//show all the impulses of current recording*  **if**(*which\_button\_pressed*.equals(**"1"**)) {  **if** (*SHOW\_ALL* == **"YES"**)  *SHOW\_ALL* = **"NO"**;  **else**  *SHOW\_ALL* = **"YES"**; loadFFT(*RECEIVED\_DATE*);  }  *//show histogram of all data recorded so far*  **else** {  *SHOW\_TOTAL* = **"YES"**;  loadHistogram(*RECEIVED\_DATE*);}  } |

|  |
| --- |
| **if**(id == R.id.***show\_FFT***) {  *//on choosing this option, the FFT graph is displayed*  loadFFT(*RECEIVED\_DATE*);  *which\_button\_pressed* = **"1"**;}  **if**(id == R.id.***show\_hist***) {  *//on choosing this option, the histogram is displayed*  *//disable comparison histogram COMPARE* = **"NO"**;  *//disable show-total SHOW\_TOTAL* = **"NO"**;  loadHistogram(*RECEIVED\_DATE*);  *which\_button\_pressed* = **"2"**;}  **if**(id == R.id.***compare***){  */\*compares current histogram with a previous histogram as selected by user from PickHistory Activity \*/ COMPARE* = **"YES"**;  Intent intent = **new** Intent(**this**, PickHistory.**class**); Bundle bundle = **new** Bundle();  bundle.putString(**"button\_pressed"**, *which\_button\_pressed*); intent.putExtras(bundle);  startActivityForResult(intent, ***request***);}  *//to zoom in and out just change the xaxis labels of graph*  **if**(id == R.id.***zoomIn***){  **for**(**int** n = 0;n < *xLabels*.**length** ;n++)  *xLabels*[n] = Integer.*toString*(50 + (10\*n));  **if**(*which\_button\_pressed*.equals(**"1"**)) loadFFT(*RECEIVED\_DATE*);  **else**{  **if**(*COMPARE* == **"YES"**)  compareHistogram(*CURRENT\_DATE*, *RECEIVED\_DATE*);  **else**  loadHistogram(*RECEIVED\_DATE*);}  }  **if**(id == R.id.***zoomOut***){  **for**(**int** n = 0;n < *xLabels*.**length** ;n++)  *xLabels*[n] = Integer.*toString*(20 + (20\*n));  **if**(*which\_button\_pressed*.equals(**"1"**)) loadFFT(*RECEIVED\_DATE*);  **else**{  **if**(*COMPARE* == **"YES"**)  compareHistogram(*CURRENT\_DATE*, *RECEIVED\_DATE*);  **else**  loadHistogram(*RECEIVED\_DATE*);}  }  **return super**.onOptionsItemSelected(item);  }  *//result from PickHistory activity which allows user to choose a date*  @Override  **protected void** onActivityResult(**int** requestCode, **int** resultCode, Intent data){  **if** (requestCode == ***request***) {  **if** (resultCode == ***RESULT\_OK***)  *RECEIVED\_DATE* = data.getStringExtra(**"RESULT\_STRING"**); |

|  |
| --- |
| **if** (*which\_button\_pressed*.equals(**"1"**)) loadFFT(*RECEIVED\_DATE*);  **else** {  **if** (*COMPARE*.equals(**"NO"**)) { loadHistogram(*RECEIVED\_DATE*);  */\*CURRENT\_DATE is the reference date with respect to which comparisons with other dates are made\*/*  *CURRENT\_DATE* = *RECEIVED\_DATE*;}  **else**  compareHistogram(*CURRENT\_DATE*, *RECEIVED\_DATE*);  }  }  **else**  **return**;  }  Listing 7B – MenuInflater options in DisplayGraph |

The *onCreate()* method sets x-axis labels and loads the last FFT graph or histogram depending on which button was pressed. *onPrepareOptionsMenu()* enables/disables certain menu items depending on which graph is loaded on the screen. In *onOptionSelectedMenu()* we determine which item has been selected *(item.getId)* and perform a certain task accordingly. Let‘s go through the to-do tasks associated with all the items one by one.

* *delete\_database* – Deletes both the tables in the database (*deleteAllEntries* in) and loads the graphs to display that no data is there to be plotted.
* *about* – shows a Toast on screen which gives a description of the graphs we are plotting.
* *history* – If user picks this item, we go the *PickHistory* Activity and show a list of the date and time of recording of all captured sounds, from which the user picks one, and accordingly a graph is loaded.
* *record\_data* – Selecting this item starts the *StartDSP* activity which records and analyses new data.
* *show\_all* – toggles between showing average FFT graph and FFT graph of all impulses separately if

*which\_button\_pressed* equals 1. If *which\_button\_pressed* is 2, then *SHOW\_TOTAL* will be set to

―yes‖.

* *show\_FFT* – this option is enabled only when the screen displays histogram. On pressing this button, the FFT graph is loaded.
* *show\_hist* – this option is enabled only when the screen displays the FFT graph. On pressing this button, the analysis histogram is loaded.
* *compare* – enabled only when screen displays histograms. Calls the function *compareHistogram*() and passes *RECEIVED\_DATE* and *CURRENT\_DATE* as parameters to plot histograms of two different captured sounds.
* *zoom in, zoom out* – these change the value of *xlabels[]* to expand/contract the x-axis.

*onActivityResult()* is needed because the activity *PickHistory* returns a result (*RECEIVED\_DATE*). According to the date and time returned, the *loadHistogram*(), *compareHistogram*() or *loadFFT*() functions are called to display the result of a sound recorded at a particular date and tim

gives the *loadFFT()* function which plots the frequency spectrum we computed and stored in our table *fft\_data* in *StartDSP*. It receives as a parameter a particular date and time, ‗*received\_date*‘, and searches the ―date‖ column of *fft\_data* for *„received\_date*‘ and loads the values from other fields,‖xvals‖,‖yvals‖ and

―impulseno‖ corresponding to this date and time.

The function *getReadableDatabase*() lets us read from our database. If we do not want to show all impulses separately, but load just the average, we need to load the column ―yvals‖. If we want to show all impulses separately, we load all values from columns ―impulse0‖, ―impulse1‖… ―impulseN‖. We also want to load columns ―xvals‖, ―date‖ and ―comments\_fft‖ (for the title of the graph). We save the names of the fields we want to load in a String array called *projection[ ]*. We use something called a SQL *query* to get data from our table, and a *cursor* to get read-write access to the result set returned by the query. We do not want to load all rows but the specific ones which are same as ‗*received\_date‟*. The String variable ‗*where*‘ points to the rowID from which we want to load our data. If ‗*received\_date*‘ is null, then we want to load the last saved data. In that case, ‗*where‟* = null, and *cursor.moveToLast()* will take the cursor to the last saved data which can be accessed. If ‗*received\_date*‘ is a particular date and time, then ‗*where*‘ points to the row in column ―date‖ which is same as ‗*received\_date*‘.

Once the cursor points to the correct position from which we want to load our data, we use ByteBuffer arrays and functions *cursor.getBlob()*, *cursor.getString()* to retrieve individual fields. The arrays *xData[]* and *yData*[] contain the frequency values and the magnitude values retrieved from ―xvals‖, ―yvals‖/ ―impluseno‖ respectively. We set the title of the graph as the date and time of the recording along with the comment entered by the user before recording.

The next part is all about plotting using the GraphView library. We have included a GraphView graph widget in *activity\_display\_graph.xml* We find that widget by its id and link it to a GraphView object

‗*graph*‘.We must first put the x and y axis coordinates into a *DataPoint* array. We then use a *LineGraphSeries* series called *„data‟* to save the DataPoint array. *graph.addSeries(data)* adds the *DataPoint* series to our line graph.

In case *SHOW\_ALL* = ―yes‖, then we consider each column ―impluseno0‖,‖impulseno1‖… ―impulsenoN‖ as a separate series and plot them with different colors. We use the function *GenerateVerticalLabels[ ]* to generate y-axis labels which are all multiples of 10. *StaticLabelsFormatter* is used to generate labels for both axes. The title is set with *setTitle*. We set the range for the axes with *setMinx, setMaxX, setMinY* and *setMaxY*. *setScrollable* allows the user to scroll through the graph and *setScalable* allows zooming in.

|  |
| --- |
| **private void** loadFFT(String received\_date) { setContentView(R.layout.***activity\_display\_graph***);  MySQLiteDatabaseHelper databaseHelper = **new** MySQLiteDatabaseHelper(**this**); SQLiteDatabase db = databaseHelper.getReadableDatabase();  **final int** numImpulses = getResources().getInteger(R.integer.***num\_impulses***); String[] projection = **null**;  GraphView graph = **null**;  **if** (*SHOW\_ALL*.equals(**"NO"**)) {  *//If we're displaying the average, we only want the average Y vals.*  projection = **new** String[]{ TableEntry.***COLUMN\_NAME\_XVALS***, TableEntry.***COLUMN\_NAME\_YVALS***, TableEntry.***COLUMN\_NAME\_DATE***, |

|  |
| --- |
| TableEntry.***COLUMN\_NAME\_COMMENT***};  }  **else** {  *//If we're displaying all impulses, we need to load all the impulses.*  projection = **new** String[3 + numImpulses];  **for** (**int** k = 0; k < numImpulses; k++)  projection[k] = TableEntry.***COLUMN\_NAME\_IMPULSE*** + Integer.*toString*(k); projection[numImpulses] = TableEntry.***COLUMN\_NAME\_XVALS***; projection[numImpulses + 1] = TableEntry.***COLUMN\_NAME\_DATE***; projection[numImpulses + 2] = TableEntry.***COLUMN\_NAME\_COMMENT***;  }  Cursor c = **null**;  *//handle the case where we want to load a specific data set. we do this*  *//using the given received\_date.*  **if** (received\_date != **null**) {  **final** String where = TableEntry.***COLUMN\_NAME\_DATE*** + **" = '"** + received\_date + **"'"**; c = db.query(TableEntry.***TABLE\_NAME\_FFT***,  projection, where, **null**,  **null**, **null**, **null**, **null**);  }  *//handle the case where we want to load the latest data*  *//or where our previous code fails for some reason*  **if** (received\_date == **null** || c.getCount() == 0) {  c = db.query(TableEntry.***TABLE\_NAME\_FFT***, projection,  **null**, **null**, **null**, **null**, **null**, **null**);  }  **if** (c.getCount() == 0) {  Toast.*makeText*(DisplayGraph.**this**, getResources().getString(R.string.***no\_entries***), Toast.***LENGTH\_LONG***).show();  **return**;  }  *//find the length of the BLOB*  **int** numBytes = getResources().getInteger(R.integer.***sample\_rate***) \* getResources().getInteger(R.integer.***capture\_time***);  *//check to make sure that the sampled length buffer is a power of two*  numBytes = StartDSP.*nearestPow2Length*(numBytes);  numBytes = numBytes \* 4 / getResources().getInteger(R.integer.***samples\_per\_bin***);  **byte**[] tempByte = **new byte**[numBytes]; **double**[] xData = **new double**[numBytes / 8]; **double**[] yData = **new double**[numBytes / 8];  */\*I'm using an array for the min and max values so that it can be passed between*  *\* functions without losing data. \*/*  **int**[] minmax = **new int**[2];  **int** min = Integer.***MAX\_VALUE***, max = Integer.***MIN\_VALUE***;  *//get the latest data set based on our search criteria* |

|  |
| --- |
| c.moveToLast();  tempByte = c.getBlob(c.getColumnIndex(TableEntry.***COLUMN\_NAME\_XVALS***)); ByteBuffer byteBuffer = ByteBuffer.*allocate*(tempByte.**length**); byteBuffer.put(tempByte);  byteBuffer.rewind();  **for** (**int** k = 0; k < xData.**length**; k++) xData[k] = byteBuffer.getDouble();  String title = c.getString(c.getColumnIndex(TableEntry.***COLUMN\_NAME\_DATE***)) + c.getString(c.getColumnIndex(TableEntry.***COLUMN\_NAME\_COMMENT***));  graph = (GraphView) findViewById(R.id.***FFTgraph***);  **if** (*SHOW\_ALL*.equals(**"NO"**)) {  *//grab the average blob, and convert it back into it's original form (double)* tempByte = c.getBlob(c.getColumnIndex(TableEntry.***COLUMN\_NAME\_YVALS***)); byteBuffer.clear();  byteBuffer.rewind(); byteBuffer.put(tempByte); byteBuffer.rewind();  **for** (**int** k = 0; k < yData.**length**; k++) { yData[k] = byteBuffer.getDouble(); **if**(minmax[1] < yData[k])  minmax[1] = (**int**) Math.*ceil*(yData[k]);  **if**(minmax[0] > yData[k])  minmax[0] = (**int**) Math.*floor*(yData[k]);  }  *//add the dataset to the graph*  DataPoint[] values = **new** DataPoint[yData.**length**];  **for** (**int** n = 0; n < yData.**length**; n++) {  DataPoint v = **new** DataPoint(xData[n], yData[n]); values[n] = v;  }  LineGraphSeries<DataPoint> data = **new** LineGraphSeries<DataPoint>(values); graph.addSeries(data);  min = (**int**) minmax[0]; max = (**int**) minmax[1];  }  **else** {  *//we load each set of data separately*  **for** (**int** i = 0; i < numImpulses; i++) {  *//grab the associated impulse blob and convert it back into doubles*  tempByte = c.getBlob(c.getColumnIndex(TableEntry.***COLUMN\_NAME\_IMPULSE*** + Integer.*toString*(i)));  byteBuffer.clear(); byteBuffer.rewind(); byteBuffer.put(tempByte); byteBuffer.rewind();  **for** (**int** k = 0; k < yData.**length**; k++) { yData[k] = byteBuffer.getDouble(); **if** (minmax[1] < yData[k])  minmax[1] = (**int**) Math.*ceil*(yData[k]);  **if** (minmax[0] > yData[k])  minmax[0] = (**int**) Math.*floor*(yData[k]);  }  *//add the impulse data to the series*  DataPoint[] values = **new** DataPoint[yData.**length**];  **for** (**int** n = 0; n < yData.**length**; n++) {  DataPoint v = **new** DataPoint(xData[n], yData[n]); |

|  |
| --- |
| values[n] = v;  }  LineGraphSeries<DataPoint> data = **new** LineGraphSeries<DataPoint>(values); graph.addSeries(data);  **if** (min > minmax[0])  min = (**int**) minmax[0];  **if** (max < minmax[1]) max = (**int**) minmax[1];  **int** color = 0;  */\* The colors are in the form of ALPHA RED GREEN BLUE, meaning the first*   * *byte is the alpha value (opacity or see-through-ness and in this* * *case is always 0xff), the second byte is red and so forth.\*/*   **switch** (i) {  **case** -1:  color = Color.***BLUE***; **break**; **case** 0:  color = 0xff0099cc; **break**; **case** 1:  color = 0xff9933cc; **break**; **case** 2:  color = 0xff669900; **break**; **case** 3:  color = 0xffff8800; **break**; **case** 4:  color = 0xffcc0000; **break** ;  **case** 5:  color = 0xff33b5e5; **break**; **case** 6:  color = 0xffaa66cc; **break**; **case** 7:  color = 0xff99cc00; **break**; **case** 8:  color = 0xffffbb33; **break**; **case** 9:  color = 0xff4444; **break**; **default**:  color = Color.***BLUE***; **break**;  }  data.setColor(color);  }  }  db.close(); db = **null**;  minmax[0] = min; minmax[1] = max;  *//set up graph so it displays the way we want*  StaticLabelsFormatter staticLabelsFormatter = **new** StaticLabelsFormatter(graph); staticLabelsFormatter.setVerticalLabels(GenerateVerticalLabels(minmax)); staticLabelsFormatter.setHorizontalLabels(**null**);  graph.setTitle(title); graph.getViewport().setScrollable(**true**); graph.getViewport().setScalable(**true**); graph.getViewport().setMinY(minmax[0]); graph.getViewport().setMaxY(minmax[1]); graph.getViewport().setMinX(0);  graph.getViewport().setMaxX(1000); |

|  |
| --- |
| graph.getGridLabelRenderer().setVerticalAxisTitle(**null**);  }  **private** String[] GenerateVerticalLabels(**int**[] minmax)  {  *//we need to truncate the last digit so everything is a nice multiple of 10*  **if**(minmax[0] >= 0){  minmax[0] = minmax[0] / 10; minmax[0] = minmax[0] \* 10;}  **else{**  minmax[0] = minmax[0] / 10; minmax[0] = (minmax[0] - 1) \* 10;}  **if**(minmax[1] >= 0){  minmax[1] = minmax[1] / 10; minmax[1] = (minmax[1] + 1) \* 10;}  **else{**  minmax[1] = minmax[1] / 10; minmax[1] = minmax[1] \* 10;}  **int** numIntervals = 0;  **int** stride = 0;  **if**( (minmax[1] - minmax[0]) <= 100){  numIntervals = (minmax[1] - minmax[0]) / 10 + 1; stride = 10;}  **else{**  numIntervals = 11;  stride = (minmax[1] - minmax[0]) / 10;  *//make stride a multiple of 5*  stride = stride / 5; stride = (stride + 1) \* 5;  *//max must therefore be slightly larger than before*  minmax[1] = minmax[0] + stride \* (numIntervals - 1);  }  String[] labels = **new** String[numIntervals];  **for**(**int** k =0; k < numIntervals; k++)  labels[k] = Integer.*toString*(minmax[0] + (numIntervals - k - 1) \* stride);  **return** labels;  }  Listing 7C - loadFFT() in DisplayGraph |
| *<?***xml version="1.0" encoding="utf-8"***?>*  <**LinearLayout xmlns:tools=**[**"http://sche**](http://schemas.android.com/tools)**m**[**as.android.com/tools**](http://schemas.android.com/tools)**" xmlns:android="**[**http://schemas.android.com/apk/res/android"**](http://schemas.android.com/apk/res/android) **android:layout\_width="fill\_parent" android:layout\_height="fill\_parent" android:screenOrientation="landscape" tools:context="${relativePackage}.${activityClass}"**>  <**android:com.jjoe64.graphview.GraphView android:layout\_width="wrap\_content" android:layout\_height="wrap\_content" android:id="@+id/FFTgraph" xmlns:android=**[**"http://**](http://schemas.android.com/apk/res/android)**s**[**chemas.android.com/apk/res/android"**](http://schemas.android.com/apk/res/android) />  </**LinearLayout**>  Listing 7D – activity0\_display\_graph.xml |

2.3 Reports

**1-10: Idea Validation and Market Research**

* Understand the app’s purpose and target audience.
* Conduct market research to validate the app idea.
* Identify key competitors and analyze their features.

**Day 11-20: Planning and Design**

* Define the app’s core functionalities and features.
* Sketch the app and create a wireframe.
* Design the app’s user interface and user experience.

**Day 21-30: Setting Up the Development Environment and Starting Development**

* Install Android Studio and set up the Android SDK.
* Create an Android Virtual Device for testing.
* Start developing the app’s basic functionality.

**Day 31-45: Developing Advanced Features**

* Implement the app’s advanced features.
* Ensure the app’s performance is optimized.
* Make sure the app is secure and respects user privacy.

**Day 46-55: Testing**

* Test the app thoroughly to identify any bugs or issues.
* Perform unit testing, integration testing, and system testing.
* Conduct usability testing to ensure the app provides a good user experience.

**Day 56-60: Deployment and Marketing**

* Prepare the app for deployment to the Google Play Store.
* Create a marketing plan to promote the app.
* Launch the app and monitor its performance.

**Conclusion**

Android app development is a dynamic and ever-evolving field. It involves a variety of components and concepts that work together to create a functional and intuitive mobile app. Understanding the basics of layouts and views is crucial for creating a visually appealing user interface, while activities and intents enable users to navigate between different screens and pass data between them.

The market size of Android app development in India is rapidly growing, and it is expected to continue to do so in the coming years. The future for Android Developers is very bright. Given the increasing usage of Android, it’s clear that Android is definitely on the rise.

So, whether you’re a seasoned developer or just starting out, there’s never been a better time to dive into Android app development.