Sound Ecology and Acoustic Health, Part 5
Frequency Domain Analysis

Adrien Gaspard and Mike Smith

Published in Circuit Cellar, Kick Media Corporation, U. S. A,
Issue 305, pp 18 – 31, 2015 -- http://circuitcellar.com/

Contact: Mike Smith:

Department of Electrical and Computer Engineering,
University of Calgary,
ICT533, 2500 University Drive, N.W.
Calgary, Alberta, Canada T2N 1N4

Email: Mike.Smith @ ucalgary.ca

Phone: +1-403-220-6142

Sound Ecology and Acoustic Health, Part 5
Frequency Domain Analysis

Last month we explained how finishing our mobile phone application for identifying community noise nuisance problem had suddenly become a "*must complete*" project for one of Mike's neighbour's kids at University. The eldest teenager had volunteered with a group of scholars from literature, art history, nursing, archaeology, religious studies, science and medicine wants to take research on ghosts seriously -- *sites.sas.upenn.edu/ghosts-healing*. The project involved working on analysing room acoustics as a possible source of "*that friendly spectral feeling*". Fig. 1 B

Last month we started working on fulfilling the promise from our first CC article to "*provide enough information to do some "real" digital signal processing (DSP) analysis"* We talked about grabbing the audio information and preparing it for display. This month we will handle the last part of our neighbour's frantic email *"Get Adrien to add some graphics' capability to display the frequency characteristics of the sounds in a room to make my term report more interesting!* The things we Canadians will do to stay on the right side of the neighbour's kids so they will shovel snow!

avoid generating their own community noise issue they quietly whisper to each other how to reliably excite a room resonance to be captured using the existing TGBN detector code.

They remain visibly shocked by the realization from last month that looking for small differences in the captured signals displayed as a function of time meant working (slowly) with a lot of data. They were obviously thrilled by the thought that soon there would be frequency information signal of captured signals analysed using discrete Fourier transform (DFT) algorithm code grabbed from the web.

Then there was the promise of possible heart pounding danger from their reckless decision to deliberately provoke room resonances, or sleeping ghosts, by outputting a Chirp, a sound burst sweeping from 50 to 1000 Hz. They fantasized that their frequency domain displays of angry ghosts, Fig. 2, would soon be selling for *K's of $K* on the soon-to-be-built CC Art Gallery website!

SPOILER ALERT: At the end of this article we have provided a picture of that Fig. 2 fleeing ghost which we have since persuaded to live behind a van der Vaal's force field.

THE STORY SO FAR (Play dramatic music recorded on the *.3GPP* app from Article 2):
Our intrepid programmers have made themselves safe from roaming ghosts by hiding behind a large 84" 4K resolution screen and using a wireless key-board (HGTG fans will recognize that's 2 x 42). To

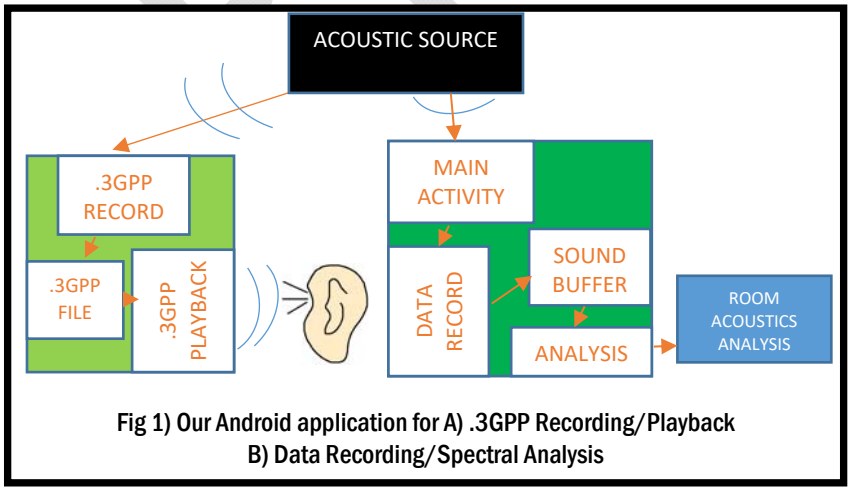

Fig 1) Our Android application for A) .3GPP Recording/Playback
B) Data Recording/Spectral Analysis

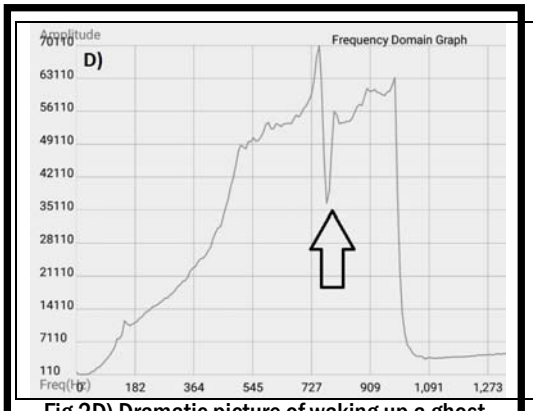

Fig 2D) Dramatic picture of waking up a ghost that flees the room. (From last month's article.)

```
        //Call the function that process the FFT
1100.   int error = doubleFFT(samples, numRecords, sampleBufferLength);
1101.   if (error == -1) {
1102.     if (!isCancelled())
1103.     publishProgress(-1, -1, -1, -1, -1, 0); //display error message
1104.     sampleBuffer = null;
1105.     return -1;
1106.   }

        // Grab first record for analysis and display
1107.   double[] toStorage_freq = new double[sampleBufferLength];
1108.   for (int n = 0; n < sampleBufferLength; n++) {
1109.     toStorage_freq[n] = samples[0][n] ] / REFSPL);;
1110.   }
1111.   if (isCancelled()) {return -1;}

        // reduce the size of our sample so the graph can load in a normal
1116.   int samplesPerPoint_freq = getResources().
          getInteger(R.integer.samples_per_bin_freq);
1117.   int width_freq = toStorage_freq.length / samplesPerPoint / 2;
1118.   double maxYval_freq = 0;
1119.   final double[] tempBuffer_freq = new double[width_freq];
1120.   for (int k = 0; k < tempBuffer_freq.length; k++) {
1121.     for (int n = 0; n < samplesPerPoint; n++)
1122.       tempBuffer_freq[k] += toStorage_freq[k * samplesPerPoint+ n];
1123.       tempBuffer_freq[k] /= (double) samplesPerPoint;
1124.     // Log.d("ADebugTag", "Value of tempBuffer: " +
              Double.toString(tempBuffer[k]));
1125.     if (maxYval_freq < tempBuffer_freq[k]) {maxYval_freq =
              tempBuffer_freq[k];}
1126.   }

        // Save X data
1130.   final double[] xVals_freq = new double[tempBuffer_freq.length];
1131.   for (int k = 0; k < xVals_freq.length; k++)
1132.     xVals_freq[k] = k * samplerate / (2 * xVals_freq.length);

1135.   button_graph_freq.setOnClickListener(new View.OnClickListener() {
1136.     public void onClick(View arg0) {
1137.       String which_button_pressed = "2";
1138.       Bundle extras_freq_values = new Bundle();
1139.       extras_freq_values.putDoubleArray("key_x_freq", xVals_freq);
1140.       extras_freq_values.putDoubleArray("key_y_freq",tempBuffer_freq);
1141.       extras_freq_values.putString("button_pressed",which_button_pressed);
1142.       Intent intent_graph_freq = new
              Intent(SoundAnalysis.this,DisplayGraph.class);
1143.       intent_graph_freq.putExtras(extras_freq_values);
1144.       intent_graph_freq.putExtras(extras_freq_values);
1145.       intent_graph_freq.putExtras(extras_freq_values);
1146.       startActivity(intent_graph_freq);
1147.     }
1148.   });
1149.   return 0;
1199. }// Continues in Listing 5
```

Listing 4B) Saving recorded data into the frequency domain using the *doInBackground* step from the *CaptureAudio* class

In the last article, we detailed how to set up the *doInBackground* code to handle the data processing in the time domain. We now tackle how to extend the analysis into the frequency domain, Listing 4B before describing the *DisplayGraph* activity needed to complete both the time and frequency domain analysis display.

We have already made some preparation for doing frequency domain analysis. Once again, if you have not done it yet, uncomment the call to nearestPow2Length() at Article 3 Listing 8 Line 956 as we now need it. By ensuring that the sample buffer length is a power of 2 we avoid needing to use a fancy version of the fast Fourier transform (FFT) algorithm

There are a lot of neat tricks needed to get the best out of an FFT algorithm which provide a discrete approximation of the continuous Fourier transform. We pass our time domain recorded data, *samples*, the number of records proceed, *numRecords* as well as the samples size, *sampleBufferLength* to *doubleFFT()* (see Listing 6A) at Line 1100. We respond to any FFT related error with an error message displayed on the screen, Lines 1101 to 1106.

The rest of code from Listing 4B is similar to the one used to process the recorded data in the time domain. The data from the sound record is grabbed for analysis and display, Lines 1107 to 1111. Frequency domain information is commonly displayed with the amplitude "y" values in the linear domain, Line 1109
.
We make sure that the function *isCancelled()* ends the task as soon as possible if it is cancelled before being completed normally, Line 1111. We then reduce the size of our data for the graph to load in a normal amount of time, Lines 1116 to 1126, before saving the "x" values into an array and pass them with the "y" values to the *DisplayGraph* activity using the *"Graph in Frequency domain"* button, an intent and bundles, Lines 1130 to 1148. We gracefully exit the modified *doInBackground* step with a "*return 0*" in Line 1149.
Last but not least, Listing 5, Lines 1200 to 1259 show the lasts steps needed to complete the *CaptureAudio* asynchronous class. Two remaining modifications are needed for the *CaptureAudio* class. Listing 5, Lines 1210 to 1216 are used in *onProgressUpdate()* to display the progress bar defined in the activity's layout file and translating the FFT calculations running in the background. Lines 1217 to 1220 display an error message on the phone's screen if an FFT related error happens. Method *onPostExecute()*, Lines 1250 to 1259, releases the recorder if that has not been done by that time, and warns the user if any errors have occurred

THE STORY CONTINUES: FREQUENCY DOMAIN ANALYSIS

Gentle reminder: Cutting and pasting the code from the CC electronic version and removing the line numbers (See Article 2) is more fun that typing the code in or grabbing it from the Circuit Cellar ftp site.

SOME NEEDED FINISHING TOUCHES

Listings 6 detail the different methods called by the *doInBackground* task. Notice that these methods have to be implemented out of the *CaptureAudio* asynchronous task.

The FFT is probably the commonly used algorithm in the world. It performs a discrete Fourier transform (DTF) that transforms a time domain signal into its frequency domain representation. We made use of *wikijava.org/wiki/The_Fast_Fourier_Transform_in_Java_%28part_1%29,* a java version available under the GNU FDL free doc, which we have slightly modified. Create an "*FFTbase*" activity in the *WAT_AN_APP* project and implement it using the code from Listing 7. See our previous articles, especially Article 1, for details concerning the creation of an activity in an Android project. No layout or menu need to be implemented for this activity

The *doubleFFT()* function that we have quickly mentioned earlier, from Listing 6A, calls the *FFTbase* activity. A frequency domain information has both amplitude and phase information and must be represented as a complex number. Line 1454 places our audio data into the real part of a complex array while Line 1455 sets the imaginary part to zero. After calling the FFT algorithm, Line 1456, the absolute value of the frequency domain signal is calculated, Lines 1458 and 1459. If an error has occurred the activity is stopped and an error message appears, Line 1457. Otherwise, we publish on the screen the progresses of the activity, using a progress bar, Line 1463.

Details of the *nearestPow2Length()* to ensure that our radix 2 FFT algorithm will have no problems with the buffer length is given in Listing 6B. Listing 6C, Lines 1600 to 1603, detail an intent to call the *DisplayGraph* activity to display the recorded data in the time and frequency domains. We are finally ready to investigate graphing on Android!

```
1200.  protected void onProgressUpdate(Integer ... data){
       //NEW CODE – INSERT AFTER Article 3, Listing 9
          //Lines 1201 to 1209
1210.  if (data[4]!=-1){
1211.   TextHandleNumberRecords.setVisibility
           (TextView.INVISIBLE);
1212.    TextHandleNumberTGBN.setVisibility
           (TextView.INVISIBLE);
1213.    ProgressBar temp = (ProgressBar)
           findViewById(R.id.computation_progress);
1214.    temp.setVisibility(View.VISIBLE);
1215.    temp.setProgress(data[4]);
1216.  }
       // computation error
1217.  if (data[5] != -1){
1218.    Toast.makeText(SoundAnalysis.this,getResources().
           getString(R.string.error),
              Toast.LENGTH_LONG).show();
1219.  }
1220.}

1250.  protected void onPostExecute(Integer data){
       //NEW CODE – INSERT AFTER Article 3, Listing 9
          //Lines 1251 to 1254
       // check if an error occurred
1255.  if (data == -1) {
1256.    Toast.makeText(SoundAnalysis.this,getResources().
           getString(R.string.error),
              Toast.LENGTH_LONG).show();
1257.    return;
1258.  }
1259.}

    //INSERT BEFORE Article 3, Listing 10 Lines 1300 to 1406
```

Listing 5) Last steps of the *CaptureAudio* asynchronous class, implemented in SoundAnalysis.java from the *RoomAcousticssrc* folder**

DISPLAYING GRAPHICS ON ANDROID - ADDING THE *GraphView* LIBRARY

An external library is needed as displaying graphics is not something you can natively do in an Android project. In this article we made use of *GraphView*, an open source library for Android licensed under the GNU license (*android-graphview.org*). *GraphView* is a custom *View* used to plot charts in Android applications. As it extends Android's *View* class, we can use it to simply create basic graphs in the layout xml file, or create and program more complex ones in the activity's java code.

Given some compatibility issues we found with the latest GraphView, 4.0, we decided to stay with version 3.1. Please refer back the Yoga finger exercises in Articles 2 and 3 before trying to *Click* enough times to add GraphView 3.1 to Eclipse.

Click on *https://github.com/jjoe64/GraphView/raw/master/public/graphview-3.1.jar* to download the *graphview-3.1.jar* library file. *Click* on "Keep" to start the download after the warning message "*This type of file can harm your computer. Do you want to keep graphview-31.jar anyway?* "appears.

```
        //INSERT AFTER Article 3, Listing 10 Lines 1300 to 1406
1450. protected int doubleFFT(double[][][] samples, int numRecords, int sampleSize){
1451.   double[] real = new double[sampleSize];
1452.   double[] imag = new double[sampleSize];

1453.   for(int k = 0; k < numRecords; k++){
1454.     System.arraycopy(samples[k], 0, real, 0, sampleSize);
1455.     for(int n = 0; n < sampleSize; n++) {imag[n] = 0;}
1456.     int error = FFTbase.fft(real, imag, true);
1457.     if(error == -1) {return -1;}
1458.     for(int n = 0; n < sampleSize; n++)
1459.       samples[k][n] = Math.sqrt(real[n]*real[n] + imag[n]*imag[n]);
1460.     if(isCancelled()) {return -1;}
1461.     else{
1462.       int prog = (int) 100* (k+1) / numRecords;
1463.       publishProgress(-1, -1, prog, -1);
1464.     }
1465.   }
1466.   return 0;
1467. }
1468. }// Continues in Listing 6B
```

Listing 6A) SoundAnalysis.java from the *RoomAcoustics* *src* folder, doubleFFT function

```
1500. public static int nearestPow2Length(int length) {
1501.   int temp = (int) (Math.log(length) / Math.log(2.0) + 0.5);
1502.   length = 1;
1503.   for (int n = 1; n <= temp; n++) {length = length * 2;}
1504.   return length;
1505. }// Continues in Listing 6C
```

Listing 6B) SoundAnalysis.java from the RoomAcoustics \src folder, nearestPow2Length function

```
1600. public void DisplayGraph(View v) {
1601.   Intent beginSoundAnalysis = new Intent(this, DisplayGraph.class);
1602.   startActivity(beginSoundAnalysis);
1603. }
1699.
```

Listing 6C) SoundAnalysis.java from the *RoomAcoustics* *src* folder, calling the DisplayGraph activity

```
1800. package com.wat_an_app;
1801. public class FFTbase {
1802.   public static int fft(final double[] inputReal, double[]
          inputImag, boolean DIRECT) {
1803.     int n = inputReal.length;
1804.     double ld = Math.log(n) / Math.log(2.0);
1805.     if (((int) ld) - ld != 0) {return -1;}
1806.     int nu = (int) ld; int n2 = n / 2; int nu1 = nu - 1;
1807.     double tReal, tImag, p, arg, c, s;

          // Here I check if I'm going to do the direct transform or
            //the inverse transform.
1810.     double constant;
1811.     if (DIRECT) {constant = -2 * Math.PI;}
1812.     else {constant = 2 * Math.PI;}

          // First phase - calculation
1820.     int k = 0;
1821.     for (int l = 1; l <= nu; l++) {
1822.       while (k < n) {
1823.         for (int i = 1; i <= n2; i++) {
1824.           p = bitreverseReference(k >> nu1, nu);
1825.           arg = constant * p / n;
1826.           c = Math.cos(arg); s = Math.sin(arg);
1827.           tReal = inputReal[k + n2] * c + inputImag[k + n2] * s;
1828.           tImag = inputImag[k + n2] * c - inputReal[k + n2] * s;
1829.           inputReal[k + n2] = inputReal[k] - tReal;
1830.           inputImag[k + n2] = inputImag[k] - tImag;
1831.           inputReal[k] += tReal; inputImag[k] += tImag;
1832.           k++;
1833.         }
1834.         k += n2;
1835.       }
1836.       k = 0; nu1--; n2 /= 2;
1837.     }
          // Second phase - recombination
1840.     k = 0; int r;
1841.     while (k < n) {
1842.       r = bitreverseReference(k, nu);
1843.       if (r > k) {
1844.         tReal = inputReal[k]; tImag = inputImag[k];
1845.         inputReal[k] = inputReal[r]; inputImag[k] =
        inputImag[r];
1846.         inputReal[r] = tReal; inputImag[r] = tImag;
1847.       }
1848.       k++;
1849.     }
1850.     double radice = 1 / Math.sqrt(n);
1851.     for (int i = 0; i < inputReal.length; i++) {
1852.       inputReal[i] = inputReal[i] * radice;
1853.       inputImag[i] = inputImag[i] * radice;
1854.     }
1855.     return 0;
1856.   }

          //The reference bitreverse function
1860. private static int bitreverseReference(int j, int nu) {
1861.   int j2; int j1 = j; int k = 0;
1862.   for (int i = 1; i <= nu; i++) {j2 = j1 / 2; k = 2 * k + j1 - 2 *
          j2; j1 = j2; }
1863.   return k;
1864. }
1865. }
```

Listing 7) FFTbase.java from the *RoomAcoustics* *src* folder

```
2000. package com.wat_an_app;
2001. import android.app.Activity;
2002. import android.graphics.Color;
2003. import android.os.Bundle;
2004. import android.view.View;
2005. import android.widget.RelativeLayout;
2006. import android.widget.TextView;
2007. import com.jjoe64.graphview.GraphView;
2008. import com.jjoe64.graphview.GraphView.GraphViewData;
2009. import com.jjoe64.graphview.GraphViewSeries;
2010. import com.jjoe64.graphview.LineGraphView;

2020. public class DisplayGraph extends Activity {
2021.   double[] ReceivedXvalues_Time; double[] ReceivedYvalues_Time;
2022.   double[] ReceivedXvalues_Freq; double[] ReceivedYvalues_Freq;
2023.   String which_button_pressed;
2024.   TextView x_axis_time, x_axis_freq;
2025.   TextView y_axis_time, y_axis_freq;

2030.   @Override protected void onCreate(Bundle savedInstanceState) {
2031.     super.onCreate(savedInstanceState);
2032.     setContentView(R.layout.activity_display_graph);

          // The following 4 lines link to the information in the layout file
             activity_display_graph.xml
2033.     x_axis_time=(TextView) findViewById(R.id.x_axis_time);
2034.     x_axis_freq=(TextView) findViewById(R.id.x_axis_freq);
2035.     y_axis_time=(TextView) findViewById(R.id.y_axis_time);
2036.     y_axis_freq=(TextView) findViewById(R.id.y_axis_freq);

2040.     Bundle BundleReceivedButton= getIntent().getExtras();
2041.     which_button_pressed =
            BundleReceivedButton.getString("button_pressed");
2042.     int which_button_pressed_int = Integer.parseInt(which_button_pressed);

2045.     Bundle BundleReceivedXvalues_Time = getIntent().getExtras();
2046.     ReceivedXvalues_Time =
            BundleReceivedXvalues_Time.getDoubleArray("key_x_time");
2047.     Bundle BundleReceivedYvalues_Time = getIntent().getExtras();
2048.     ReceivedYvalues_Time =
            BundleReceivedYvalues_Time.getDoubleArray("key_y_time");

2050.     Bundle BundleReceivedXvalues_freq = getIntent().getExtras();
2051.     ReceivedXvalues_Freq =
            BundleReceivedXvalues_freq.getDoubleArray("key_x_freq");
2052.     Bundle BundleReceivedYvalues_freq = getIntent().getExtras();
2053.     ReceivedYvalues_Freq =
            BundleReceivedYvalues_freq.getDoubleArray("key_y_freq");

2054.   if (which_button_pressed_int == 1)
2055.       loadGraph(ReceivedXvalues_Time, ReceivedYvalues_Time,
              which_button_pressed_int);
2056.     if (which_button_pressed_int == 2)
2057.       loadGraph(ReceivedXvalues_Freq,
              ReceivedYvalues_Freq,which_button_pressed_int);
2058.   } // Continues in Listing 11
```

Listing 8) The prologue of the *DisplayGraph.java* file (*WAT_AN_APP\src* folder) sets up the UI and catches back the x and y values from the *SoundAnalysis* activity in the time and frequency domain

Once the .jar file has downloaded, create a new "*libs*" folder in the Eclipse/Android project by right *Clicking* on "*WAT_AN_APP*" then "*New*" and "*Folder*". Right *Click* on the brand new created "*libs*" folder and *Click* on "*Import*". An "*Import*" window appears, select "*General*" then "*File System*" and *Click* on "*Next*" to cause the window "*From directory:*" to appear.

Browse in the file system to find the directory where the library was downloaded ("*C:\Users\ajfgaspa\Downloads*" by default) and *Click* "*OK*". *Click* on the directory name ("*Download*") in the left panel, without checking the checkbox located next to it, then check the "*graphview-3.1.jar*" file that has appeared on the right panel. Check that you see the message "*Into folder: WAT_AN_APP/libs*",

We have not yet *Clicked* 42 times in this article, so we know more are needed. *Click* on "*Finish*" and expect to get a configuration error as you will have to convert the downloaded .jar file for use on Android. To handle that *Click* on the "*WAT_AN_APP*" project, and choose "*Build*". Under the "*Libraries*" tab, *Click* on "*Add JARs…*" and select "*WAT_AN_APP\libs\graphview-3.1.jar*". *Click* on "*OK*" twice to have now the .jar file included into the project with its definitions available to Eclipse!

DISPLAYING GRAPHICS ON ANDROID – THE *DISPLAYGRAPH* ACTIVITY

After having implemented the required imports, Listing 8 Lines 2000 to 2010, and included the GraphView classes, Lines 2007 to 2010, we can start developing the *DisplayGraph* activity. We first declare the variables we are going to use to receive the arrays containing the "x" and "y" values in the time and frequency domains from the *SoundAnalysis* activity, Lines 2021 and 2022. We do not forget that we need to know which button the user has pressed, the "*Graph in Time Domain*" or "*Graph in Frequency Domain*" one, Line 2023, to display the corresponding graph.

By default *GraphView 3.1* does not offer the possibility to add titles to the axis. To do so, we have to add some *TextView* variables, Lines 2024 and 2025, which requires the activity's layout, *activity_graph_display.xml*, given in Listing 9. The layout file is composed of a relative layout for the time domain graphic, which overlap another relative layout used for the frequency domain graphic. This trick allows setting the *DisplayGraph* activity with a unique layout file, composed of a layout for the time domain graphic, which will display the "x" values in seconds, as well as a layout for the frequency domain graphic, which will display the "x" values in Hz.

We make use of a *dimens.xml* file, Listing 10, to set up the layout's margins in the activity's layout file, Listing 9 Lines 2309 to 2312. The *dimens.xml* file must be manually created: right click on the

```
2100.  private void loadGraph(double[] ReceivedXvalues,double[]
         ReceivedtempBuffer, int which_button_pressed) {
2101.    String GraphTitle = null;
2102.    if(which_button_pressed==1)
2103.       GraphTitle="Time Domain Graph";
2104.    if(which_button_pressed==2)
2105.       GraphTitle="Frequency Domain Graph";

2106.    GraphView graphView = new LineGraphView(this,
           GraphTitle);

2110.    int[] minmax = new int[2];
2111.    int min = Integer.MAX_VALUE, max =
           Integer.MIN_VALUE;
2112.    minmax[0] = min; minmax[1] = max;
2113.    min = (int) minmax[0]; max = (int) minmax[1];

2114.    final int numRecords =
           getResources().getInteger(R.integer.num_records);

2115.    RelativeLayout layout_time_graph = (RelativeLayout)
           findViewById(R.id.graphic_layout_time);
2116.    RelativeLayout layout_freq_graph = (RelativeLayout)
           findViewById(R.id.graphic_layout_freq);

2120.    if(which_button_pressed==1){
2121.       layout_time_graph.addView(graphView);

           //change 4.096 to 8.192 if recording for 9 sec
2122.       graphView.setViewPort(0, 4.096);
2123.       x_axis_freq.setVisibility(View.INVISIBLE);
2124.       y_axis_freq.setVisibility(View.INVISIBLE);
2125.    }

2130.    if(which_button_pressed==2){
2131.       layout_freq_graph.addView(graphView);
2132.       graphView.setViewPort(0,2000);
2133.       x_axis_time.setVisibility(View.INVISIBLE);
2134.       y_axis_time.setVisibility(View.INVISIBLE);
2135.    }

2136.    graphView.setScrollable(true);
2137.    graphView.setScalable(true);
2138.    graphView.setBackgroundColor(Color.WHITE);
2139.    graphView.getGraphViewStyle().setGridColor(Color.GRAY);
2140.    graphView.getGraphViewStyle().setHorizontal
           LabelsColor(Color.BLACK);
2141.    graphView.getGraphViewStyle().setVertical
           LabelsColor(Color.BLACK);

2145.    if (which_button_pressed == 1) {
2146.    graphView.addSeries(generateGraphSeries(ReceivedXvalues,
           ReceivedtempBuffer, minmax, numRecords));
2147.   graphView.setVerticalLabels(GenerateVerticalLabels(minmax));
2148.    graphView.setManualYAxisBounds(minmax[1], minmax[0]);
2149.    }
2150.    if (which_button_pressed == 2) {
2151.    graphView.addSeries(generateGraphSeries(ReceivedXvalues,
           ReceivedtempBuffer, minmax, numRecords));
2152.   graphView.setVerticalLabels(GenerateVerticalLabels(minmax));
2153.    graphView.setManualYAxisBounds(minmax[1], minmax[0]);
2154.    }
2155.  } // Continues in Listing 11B
```

Listing 11A) DisplayGraph.java from the *RoomAcoustics \src* folder, *loadGraph* function that sets up the graphic and add our recorded data to it.

"*WAT_AN_APP\res\values*" folder then "*New*", "*File*", and implement it as shown on Listing 10.

We display the time domain axis legends, which are the text "*Time(sec)*" on the bottom left of the "x" axis using the *TextView*, Listing 9 Lines 2315 to 2325, and the text "*Amplitude*" on the top left corner of the "y" axis using Lines 2330 to 2340. The reasonment is identical to display "*Freq(Hz)*" and "*Amplitude*" on the frequency domain axis, Lines 2355 to 2381, using a new relative layout "*graphic_layout_freq*", Line 2351.

Back to Listing 8 after this short break, we call the *onCreate()* method, Lines 2030 to 2058 to initialize the activity. We start by calling *setContentView()*, Line 2032, to set up the layout resource defining our User Interface. Note how we can interact with the *TextView* using their ids stored in *activity_display_graph.xml* by using *findViewById()*, Lines 2033 to 2036, before using bundles and the previously declared arrays to catch back the values sent from *SoundAnalysis*.

After all the *Clicking*, we are not done with physical activity yet. Poltergeists will often try getting your attention by booing or *throwing* things at you. It is therefore reasonable that we catch our activity's attention by *throwing* the key "*button_pressed*" from *SoundAnalysis* to *DisplayGraph*. This key contains the value associated to the button the user has pressed to start the *DisplayGraph* activity. We *catch* this key back using the bundle *BundleReceivedButton* that gets the string associated with the button and *pass* it to *which_button_pressed* (Lines 2040 and 2041). We *transform* this string value into an int to be able to work with it, Line 2042. We have now fully managed to pass a value from one activity to the other and… Touchdown! The reasoning is identical to passing the "x" values in the time (Lines 2045 and 2046) and frequency (Lines 2050 and 2051) domain, as well as the "y" values (Lines 2047 and 2048 and Lines 2052 and 2053).

Once all the data passed, we load the graph by calling the *loadGraph()* method. If the button pressed has a value equal to "1", the user has pressed the "*Graph in Time Domain*" button, we then pass in the *loadGraph* parameters the time domain values (Lines 2054 and 2055). Otherwise, we pass it the frequency domain values (Lines 2056 and 2057).

```
2300. <?xml version="1.0" encoding="utf-8"?>
2301. <!DOCTYPE project>
2302. <RelativeLayout
2303.  xmlns:android="http://schemas.android.com/apk/res/android"
2304.  xmlns:tools="http://schemas.android.com/tools"
2305.  android:id="@+id/graphic_layout_time"
2306.  android:layout_width="wrap_content"
2307.  android:layout_height="wrap_content"
2308.  android:orientation="vertical"
2309.  android:paddingBottom="@dimen/activity_vertical_margin"
2310.  android:paddingLeft="@dimen/activity_horizontal_margin"
2311.  android:paddingRight="@dimen/activity_horizontal_margin"
2312.  android:paddingTop="@dimen/activity_vertical_margin"
2313.  tools:context=".MainActivity" >

2315.  <TextView
2316.    android:id="@+id/x_axis_time"
2317.    android:layout_width="75dp"
2318.    android:layout_height="15dp"
2319.    android:layout_alignParentBottom="true"
2320.    android:layout_alignParentLeft="true"
2321.    android:layout_marginBottom="5dp"
2322.    android:layout_marginLeft="0dp"
2323.    android:textSize="11dp"
2324.    android:text="@string/graph_x_time"
2325.  />

2330.  <TextView
2331.    android:id="@+id/y_axis_time"
2332.    android:layout_width="75dp"
2333.    android:layout_height="15dp"
2334.    android:layout_alignParentLeft="true"
2335.    android:layout_alignParentTop="true"
2336.    android:layout_marginLeft="0dp"
2337.    android:layout_marginTop="-2dp"
2338.    android:textSize="11dp"
2339.    android:text="@string/graph_y_amplitude"
2340.  />

2350.  <RelativeLayout
2351.    android:id="@+id/graphic_layout_freq"
2352.    android:layout_width="wrap_content"
2353.    android:layout_height="wrap_content" >

2355.    <TextView
2356.      android:id="@+id/x_axis_freq"
2357.      android:layout_width="75dp"
2358.      android:layout_height="15dp"
2359.      <!-- COPY FROM Lines 2319 to 2323-->
2364.     android:text="@string/graph_x_frequency"
2365.    />

2370.    <TextView
2371.      android:id="@+id/y_axis_freq"
2372.      android:layout_width="75dp"
2373.      android:layout_height="15dp"
2374.      <!-- COPY FROM Lines 2334 to 2338-->
2380.      android:text="@string/graph_y_amplitude"
2381.    />
2382.  </RelativeLayout>
2399. </RelativeLayout>
```

Listing 9) ***activity_display_graph.xml.*** **from the** ***RoomAcoustics\res\layout*** **folder**

```
2400. <resources>
2401.  <!-- Default screen margins, per the Android
       Design guidelines. -->
2402.  <dimen name="activity_horizontal_margin">
         16dp</dimen>
2403.  <dimen name="activity_vertical_margin">
         16dp</dimen>
2404. </resources>
```

Listing 10) ***dimens.xml.*** **from the** ***RoomAcoustics\res\value*** **folder**

Listing 11A Lines 2100 to 2155 describes the *loadGraph()* function, which first give the graph a title depending on the graph to display, Lines 2101 to 2105 before activating a *LineGraphView,* Line 2106. To make sure the graph will fit onto the screen, we need to identify the minimum and maximum values to display, Lines 2110 to 2113.

This activity's layout is composed of two relative layouts, Lines 2115 and 2116. If the time domain button is pressed, we add to the layout the time domain graph, set its view port to originally display the "x" values within a given range and configure the frequency domain information to be invisible on the screen, Lines 2120 to 2125. The same operations are done for the frequency domain graph, Lines 2130 to 2135.

The viewport defines that part of the graphic that is currently visible. By default, the viewport synchronizes automatically with the data, meaning that the all data are plotted. As we have a large set of data but only want to show a part of it, the user having the possibility to easily navigate to the part that is not displayed by default, we define the explicit bounds of the viewport, using *setViewPort(),* Lines 2122 and 2132. You will need to modify the value of the *ViewPort* from *4.096* to *8.192* if you increase the recording time to 9 seconds.

We now configure the graphic to be scalable and scrollable by your finger, Line 2136 and 2137, arrange its background color to be white, Line 2138, set the grid color to be gray, Line 2139, and finally set the horizontal and vertical labels to be black, Lines 2140 and 2141. Depending on the graph we want to output, the time domain (Lines 2145 to 2149) or frequency domain (Line 2150 to 2154) one, we configure the *generateGraphSeries()* and *setVerticalLabels()* functions, used to generate the graph and make sure that the values on the "y" axis are integer values.

Listing 11B details the *generateGraphSeries()* method which allows generating our graphic by passing several parameters: the "x" data for the frequency values, the "y" data for the amplitude, minmax for the bounds of the axis and the number of records, to be able to distinguish the different records with several colors, Lines 2202 to 2209. If an error occurs, the graph is plotted with a red color, Line 2203. The code that follows, Lines 2204 to 2210, display the data on the graph with different color, depending of the number of records processed. As we

```
2200.  private GraphViewSeries generateGraphSeries(double[] xData,
         double[] yData, int[] minmax, int recNum) {
2201.    int color = 0;
2202.    switch (recNum) {
2203.      case -1: color = Color.RED; break;
2204.      case 0: color = Color.BLUE; break;
2205.      case 1: color = Color.GRAY; break;
2206.      case 2: color = Color.CYAN; break;
2207.      case 3: color = Color.GREEN; break;
2208.      case 4: color = Color.MAGENTA; break;
2209.      default: color = Color.YELLOW; break;
2210.    }
2211.    GraphViewSeries.GraphViewSeriesStyle style = new
           GraphViewSeries.GraphViewSeriesStyle(color, 3);
2212.    GraphViewSeries graphSeries = new GraphViewSeries(null, style,
           new GraphViewData[] { new GraphViewData(0, 0d) });

2213.    for (int k = 0; k < xData.length; k++) {
2214.    graphSeries.appendData(new GraphViewData(xData[k], yData[k]),
           true,xData.length);
2215.    if (minmax[1] < yData[k]) {
2216.       minmax[1] = (int) Math.ceil(yData[k]);}
2217.    if (minmax[0] > yData[k]) {
2218.     minmax[0] = (int) Math.floor(yData[k]);}
2219.    }
2220.    return graphSeries;
2221.  } // Continues in Listing 11C
```

Listing 11B) DisplayGraph.java from the *RoomAcoustics \src* folder, *generateGraphSeries* function that draws the data passed in parameters

work with one record, the data are displayed in gray, Line 2205. A style is created, Line 2111, to give the graph the color we have just configured, which depend on the number of record. In our case, the variable dealing with the number of record, *recNum* is assigned the number "1", corresponding to a gray plot on the graph. This color is applied to the style line 2211, as well as a thickness value of "3". This style is applied to a new GraphViewSeries, *graphSeries*, used to generate the graph's "x" and "y" values. The "x" and "y" data are added to the series Line 2213 and 2214. The "y" axis is then adjusted, Lines 2115 to 2119, using the floor and ceiling functions that map a real number to the largest previous or the smallest following integer.

Listing 11C describes a function, *GenerateVerticalLabels()*, which makes use of the *minmax* array to generate integer values on the y axis. We truncate the last digit of every number, and make all the values integer values, which are easier to read than floating point values. Lines 2251 to 2261 are used to make all the values a multiple of 10. We then use a stride, Line 2265, which we make a multiple of 5 Line 2270. We finally apply and return our new labels, Line 2275 and 2276, so that they can be used by our graph.

Finally, add the code from Listing 12 Lines 240 to 247 to update the different activities' strings

```
2250. private String[] GenerateVerticalLabels(int[] minmax) {
2251.   if (minmax[0] >= 0) {
2252.     minmax[0] = minmax[0] / 10; minmax[0] = minmax[0] *
2253.       10;}
2254.   else {
2255.     minmax[0] = minmax[0] / 10;
2256.     minmax[0] = (minmax[0] - 1) * 10;}
2257.     if (minmax[1] >= 0) {
2258.       minmax[1] = minmax[1] / 10;
2259.       minmax[1] = (minmax[1] + 1) * 10;}
2260.   else {
2261.       minmax[1] = minmax[1] / 10; minmax[1] = minmax[1] *
            10;}

2265.   int numIntervals = 0; int stride = 0;

2266.   if ((minmax[1] - minmax[0]) <= 100) {
2267.     numIntervals = (minmax[1] - minmax[0]) / 10 + 1;
          stride =10;}
2268.   else {
2269.     numIntervals = 11; stride = (minmax[1] - minmax[0]) / 10;
        // make stride a multiple of 5
2270.     stride = stride / 5; stride = (stride + 1) * 5;
        // max must therefore be slightly larger than before
2271.     minmax[1] = minmax[0] + stride * (numIntervals - 1);
2272.   }

2273.   String[] labels = new String[numIntervals];
2274.   for (int k = 0; k < numIntervals; k++)
2275.     labels[k] = Integer.toString(minmax[0] + (numIntervals - k –
            1)* stride);
2276.     return labels;
2277.   }
2278. } // HIP-HIP HURRAY -- END OF  THE DISPLAY GRAPH
      ACTIVITY
```

Listing 11C) DisplayGraph.java from the *RoomAcoustics \src* folder, *GenerateVerticalLabels function* that generate integer values on the "y" amplitude axis

```
200. <?xml version="1.0" encoding="utf-8"?>
201. <resources>
205.   <!-- SAME AS Article 1 Listing 3, Lines 205 -- 214 -->
220.   <!-- SAME AS Article 2 Listing 4, Lines 220 -- 224 -->
230.   <!-- SAME AS Article 3 Listing 11, Lines 220 -- 237-->
238.   <string name="title_activity_sound_analysis">
         SoundAnalysis….CHIRP START</string>
       <!-- String required for the third part of the application, record
       and output Boo sound if  TGBN sound is detected -->
       <!-- ADDED ARTICLE 4 -->
240.   <string name="audio_play">Sound Analysis</string>
241.   <string name="title_activity_fftbase">FFTbase</string>
242.   <string name="title_activity_display_graph">
         DisplayGraph</string>
243.   <string name="button_start_graph_time">Graph in Time
         Domain</string>
244.   <string name="button_start_graph_freq">Graph in Frequency
         Domain</string>
245.   <string name="graph_y_amplitude">Amplitude</string>
246.   <string name="graph_x_time">Time(sec)</string>
247.   <string name="graph_x_frequency">Freq(Hz)</string>
249. </resources>
```

Listing 12) Preset string values must be set in *strings.xml* from *WAT_AN_APP\res\values*

in the *strings.xml* file, located in the folder *WAT_AN_APP\res\values*. Note that we have updated a string from Article 3 Listing 11 Line 238, to add the text "*CHIRP START"* next to the icon the user has to press to start outputting the 5 sec Chirp.

So after all that – we hope Mike's neighbour's kid can have some fun on his term project. We know he will acknowledge us -- at the end of his project report and, at next summer's first BBQ, with some of those fancy sausages my neighbours persuade their butcher to make!

CONCLUSION

Step by step, we have gained the knowledge to learn how to create an application on an Android device to record and display the frequency characteristics of a sound in a room. Having started by getting familiar to the development environment, we have developed an audio recorder/player application from which we have learnt that, to record data into a buffer for analysis, we have to use an *AudioRecord* class and not a *MediaRecorder* one. After starting recording once a sound is detected, we have first processed to a simple analysis, which has consisted in comparing the recorded data to a threshold, and outputting a sound if the data were greater than this threshold. We have then pushed the analysis further, by applying a FFT algorithm on our data and graph both the time and frequency domain results. Some updates will follow on the papers to come, especially the possibility to easily navigate from the time domain graph to the frequency one. The use of a database to keep track of all the recordings done by the user is also a feature to come. This will allow us to graph our records separately, or to display their average.

D.I.Y. GHOSTS: The frequency domain, Fig.2A to 2D, show the acoustic ecology of our Lab at the University of Calgary – before and after trying to wake up a ghost we thought we had spotted with a Chirp sound burst.

We have to confess, the ghost we found was not really spooky, please see Fig.4. We actually got a stronger resonance by outputting a Chirp using the mobile phone's speaker placed over a plastic bottle nearly filled with water. Changing the water level gives you different ghost sounds.

You can play "Hunt the ghost" with your younger children at their next birthday BBQ party by hiding bottles at different places around your garden. Inside the house you risk upsetting the bottles and making a water mess.

We had the most BBQ fun when we pointed the phone's speaker towards our opened mouths and output the Chirp. The first competition was to see who can get the strongest resonance, and for a greater challenge, who could generate the highest frequency resonance.

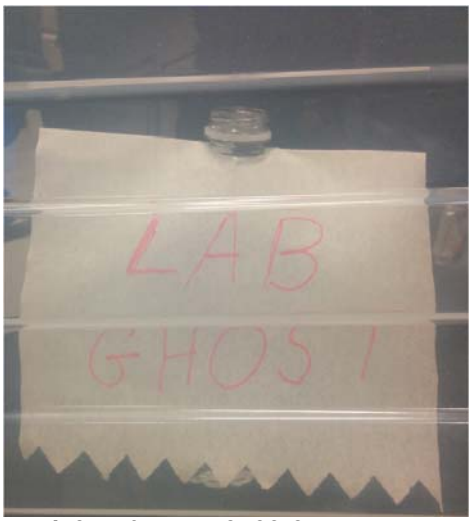

Fig 4) One of our Lab GHOSTS living protected behind a van der Waals force field

For our work we decided to force the graphing activity to pop up in the screen landscape orientation mode as the graphs looked better that way. Have a look at the quick help guide at the end of this article to learn how to do this operation.

BIOGRAPHY

Adrien Gaspard received his Master of Engineering from CPE Lyon, France in February 2015. He tackled his final practicum as an exchange student in Electrical and Computer Engineering at the University of Calgary. He undertook self-directed term projects directed towards the possible use of noise cancelling to solve the community noise problem in Calgary community of Ranchlands. His long term career goal is in embedded systems and wireless telecommunications. He can be contacted at gasp.adrien@gmail.com

Mike Smith has been contributing to Circuit Cellar magazine since the '80s. He is a professor in computer engineering at the University of Calgary, Canada. When not singing with Calgary's Adult Recreational choir *Up2Something,* Mike's main interests are in developing new biomedical engineering algorithms and moving them onto multi-core and multiple-processor embedded systems in a systematic and reliable fashion. Mike has recently become a convert to the application of Agile Methodologies in the embedded environment. He is Analog Devices University Ambassador (2001 – 2015). He can be contacted at Mike.Smith@ucalgary.ca