Sound Ecology and Acoustic Health, Part 4
Time Domain Analysis

Adrien Gaspard and Mike Smith

Published in Circuit Cellar, Kick Media Corporation, U. S. A,
Issue 303, pp 32 - 43, 2017 -- 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 4

Time Domain Analysis

We have spent the last while working towards a mobile phone application to help identify a local noise nuisance problem. We joked with Mike's neighbour's kids that the record and play-back .3GPP file WAT_AN_APP application, Fig. 1A, was to impress them that "*Without Any Teenage Assistance Necessary, we could write an Android APP*" CC Issues ###. We then added just enough additional code (JEAC) to store an audio record for later analysis CC Issues ###. To continue the friendly tease in CC Issues ###, we pretended that the project code was actually designed to detect – "*Things that Go BOOm at Night+* - how many "*TGBN* ghosts" there are in the neighbourhood (Fig. 1B).

As they say "*Be careful what you wish for!*" Our neighbours got interested in the community noise issues we were really trying to measure. They had their teenagers explore the acoustic health of their home using our work-in-progress. Late yesterday, a knock on the door revealed our neighbours asking for help. Their eldest teenager had gone to the University of Pennsylvania. According to the Penn Arts and Sciences website *sites.sas.upenn.edu/ghosts-healing*, a group of scholars from literature, art history, nursing, archaeology, religious studies, science and medicine wants to take research on ghosts seriously. So our neighbour's kid decided to volunteer with this group. This turned into a term project -- working on analysing room acoustics as a possible source of "*that friendly spectral feeling*". Hence the frantic email message they wanted to pass on --

Term's nearly over! Could you please get Mike to hurry up and fulfill his promise in that first CC article of providing enough information to do some "real" digital signal processing (DSP) analysis? While he was at it – could he 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!

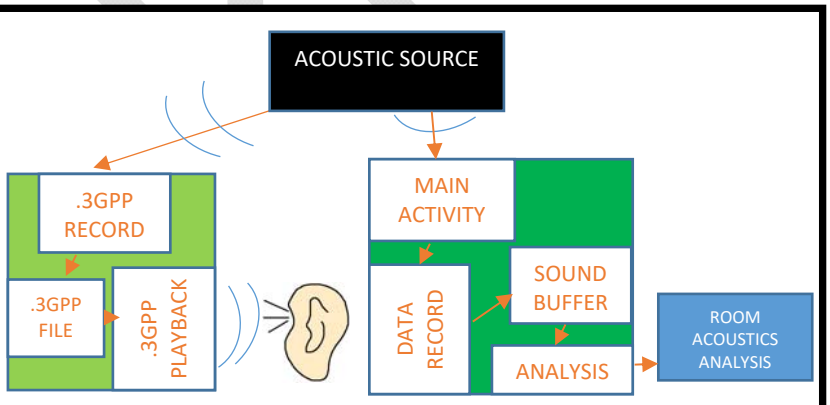


Fig 1) Our Android application for A) .3GPP Recording/Playback
B) Data Recording/Spectral Analysis

In Canada, it always good to keep on the right side of the neighbour's kids as they are a good (inexpensive) labour source for shovelling snow off sidewalks. So we decide to write a *RoomAcoustics Analysis Capability* addition. Actually we wanted to be able to say that we had *Penn*-ed some code (sorry for the pun (:-).).

First we will explain how to reliably excite a room resonance that can be captured by our existing TGBN detector code. We will graphically display the room audio signal to give us a first chance to compare resonance characteristics in different rooms. We found that looking for small differences in the captured signals displayed as a function of time meant working (slowly) with a lot of data. So we added a way to generate frequency information signal of captured signals using a discrete Fourier transform (DFT) algorithm code we grabbed from the web. Fig 2A shows the background noise recorded in our university lab. Having noticed a possible small 727 Hz ghost sleeping next to our desk, we tried to move around the room to better record its characteristic, Fig. 2B. The frequency characteristics of our two records look too similar for us to be sure that we have a non-snoring ghost close by.

We decided to wake it up by outputting a three second Chirp, a sound burst from 50 to 1000 Hz. Fig. 2C show the frequency response of the Chirp signal, but there is not much there other than measuring the poor low frequency of our phone's speaker. However, we accidently got close enough that we woke up the sleeping ghost which significantly changed the frequency response of the room, Fig 2D.

GHOST PORTRAIT: Sorry, you'll have to wait to the end of Article 5 to see the picture of the ghost we persuaded to live behind a van der Vaal's force field.

EXCITING A ROOM RESONANCE

As our neighbours didn't feel like waking up sleeping ghosts, we explained how to excite a room resonance to our next door neighbours at the last BBQ before the snow fell. We lined up 10 glasses filled with different levels of water on a hard surface. You can generate a sound impulse if you clap your hands together. All of the glasses should have resonated, tinkled, as an impulse contains all possible frequencies in theory. However, persuading the BBQ group to sing "*Doh -- rah -- me ...*" at the glasses proved a better way of getting enough sound energy at a particular frequency. Once your group has found

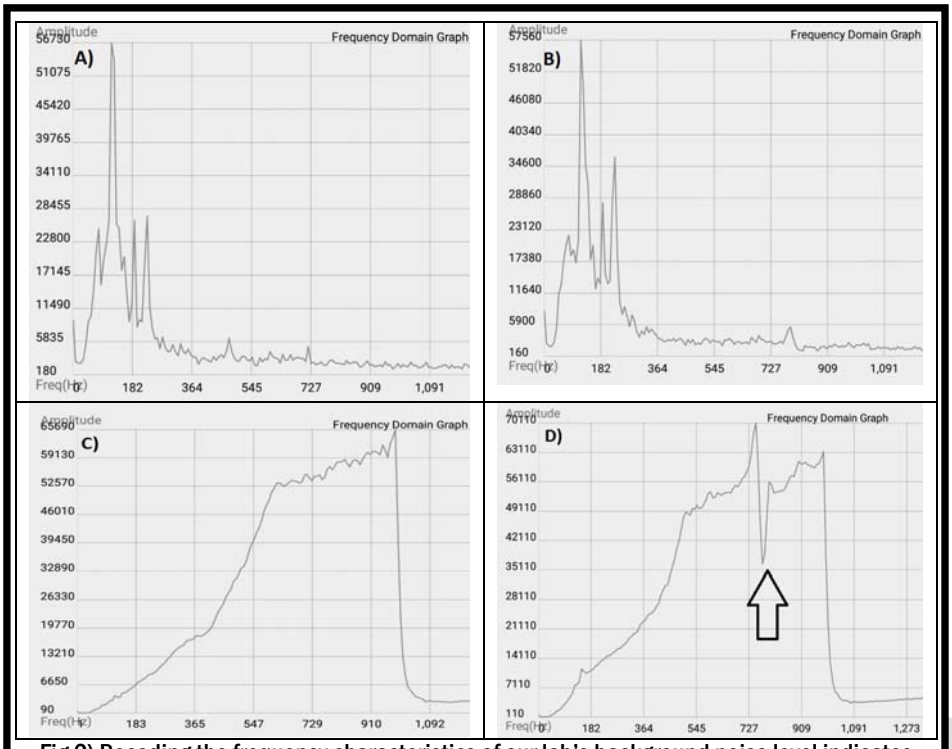


Fig 2) Recoding the frequency characteristics of our lab's background noise level indicates that we MIGHT be getting closer to a 727 Hz ghost after moving between positions (A) and (B). However the frequency characteristics of a Chirp sound burst, (C) really change when we wake up the ghost and it flees the room (D).

a note that starts a glass to sympathetically vibrate, then you can adjust their singing and get the "note" just right. We were not sure about suggesting that our neighbour's teenager persuade the Penn Choral Society members to join his term project and sing in each room while he recorded them with the TGBN part of our WAT_AN_APP! Instead we explained how to use a more systematic and controlled approach - generating a Chirp signal - a longer duration sound containing all frequencies.

The history of Chirps is neat. In the early stages of AM radio, you could hear the interference of lightning strikes some distance away as a crackle on the loudspeaker. If you slightly mistuned the radio, the crackle changed into something else. What happened was that the lightning strike, an electro-magnetic impulse, generated a signal containing "all" electro-magnetic (EM) frequencies. Signals at different frequencies travelled at different speeds through the air because of an effect called EM dispersion, and arrived at the radio at different times. Analogue down-sampling within the AM radios caused the different EM frequencies to turn into different audio signals. So you heard what sounded like a bird whistle starting at low audio frequencies and going up to high; hence the name Chirp. You can generate an audio Chirp using a pedestrian underpass. If you clap your hands near the entrance, then you get a "*doo-WEE*" sound rather than a single clap echo. The *"doo"* sound is the reflection of low frequency sounds arriving back at your ears at an earlier time than the high frequency *"WEE"* reflections. Our preferred way of generating a 3 second Chirp sound

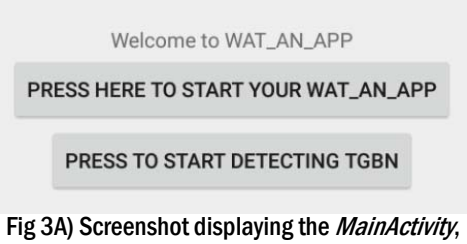


Fig 3A) Screenshot displaying the *MainActivity*, from which we can start the sound analysis

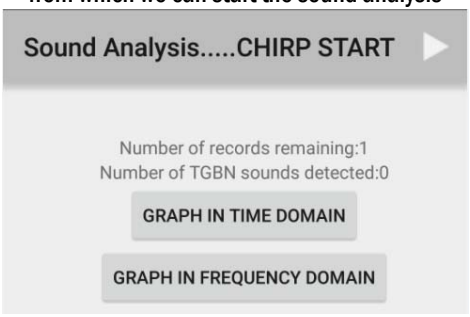


Fig 3B) Screenshot displaying the TGBN *SoundAnalysis* activity, from which we can output a Chirp, perform our calculations and call the activity that display graphs

burst is using *Audacity (audacity.sourceforge.net)* on a laptop.

In this article, we make an application capable of both outputting a Chirp and capturing the room sound ecology. However, it is not our favourite approach as generating a Chirp on a mobile has issues. The mobile audio electronics are not really designed to handle low frequency output or provide a lot of audio power without distortion (see Figs 2A and 2B). However plugging in an external speaker to boost the output cuts out the microphone input; a classic "CATCH-22" situation! We can fix that by capturing and then playing back the laptop Chirp as a .3GPP file on another Android phone using external speakers. By modifying the code in this article to output a *NADA.3GPP* file of "*the sound of silence*" rather than a stored Chirp you can have the best of both worlds!

Our plan for running the extended WAT_AN_APP is to press the *"Press To Start Detecting TGBN"* button from the *MainActivity* screen that pops up when the application starts, Fig.3A, to start the TGBN *SoundAnalysis* activity. To ensure that the sound capture starts immediately modify the *ints.xml* file from the "*WAT_AN_APP\res\values*" folder and set the threshold at which we start the recording from "10000" to "0" (Article 3 Listing 3 Line 2840). As will be explained later, the DSP frequency analysis code works best with a five second sound sampling so change the capture time from "7" to "5" in the *ints.xml* file.

As shown in Fig. 3B, we can press the play icon located in the action bar (top right corner of the screen) to output a three seconds Chirp signal. After the sound capture, you can display the audio information as either an audio time graph or an audio frequency graph.

GETTING STARTED

This Article is an extension of our last article in CC ###, so you will see a lot of similar file names and line numbers. The code for the *MainActivity* is identical to the one from Article 3 Listings 1A and 1B. As a reminder, the *setContentView()* makes use of the layout file *activity_main.xml* to generate a screen with a message welcoming the user in the application, and two buttons to start the audio record/playback and audio analysis activities. When we called the *SoundAnalysis* activity in Article 3, we initialized the recorder, detected a sound and recorded it into a local array. Then we did some simple DSP analysis – check to see if the recorded data was above a certain threshold. This time, we are going to manipulate the recorded data using some DSP program before graphing the results using the *GraphView* library.

THE SOUND ANALYSIS ACTIVITY

SoundAnalysis uses an asynchronous task which offloads the computations to a worker thread. This is necessary to not stall out the user interface (UI) thread which could cause the UI to stop responding. The SoundAnalysis activity's layout, *activity_sound_analysis.xml* is described in Listing 1. The *TextView* displaying the number of records remaining and the number of TGBN sounds detected, Lines 1710 to 1739, are identical to Article 3 Listing 2. We simply add a progress bar on the screen, translating the background FFT calculations, Lines 1740 to 1751, as well as two buttons, to start displaying graphs in the time (Lines 1760 to 1768) or frequency domain (Lines 1770 to 1778).

Listing 2 shows code to add the time and frequency display buttons, Lines 559, 605 and 606. Overviews of all the methods we need to develop are given at the end of Listing 2. The *SoundAnalysis* activity covers a lot of code. To avoid typing. cut-and-paste the line from the listings in the CC electronic version, or *visit the CC website at XXXXX*. The three methods in Listing 3A set up an action bar so that we can press an icon that output a Chirp sound which the application will record. The run-once *onCreateOptionsMenu()* method, Line 610, is responsible for handling the content of the activity's menu that appears on the action bar. As this is our first action bar, we need to set up a "menu" folder in "*WAT_AN_APP\res|menu*". Add the "*menu_sound_analysis.xml"* file to this folder and insert the code given in Listing 3B Lines 2500 to 2005. In order that Listing 3B Line 2502 can display a *play* icon we need to add an "*ic_action_play.png*" file in the "*WAT_AN_APP\res\drawable-mdpi*" folder. This icon can be obtained from a *"…\Action Bar Icons\holo_dark\09_media_play\drawable-mdpi"* folder, following the quick help guide section of Article 2 to have more information on adding an icon.

Lines 700 to 708, Listing 3A shows the *onOptionsItemSelected()*, hook called whenever an item in the option menu is selected. Line 703 activates the *startPlaying()* method which outputs a Chirp to play a sound from 50 Hz to 1000 Hz. The

```
    <!--Used by SoundAnalysis activity -->
1700. <RelativeLayout
1701. xmlns:android="http://schemas.android.com
        /apk/res/android"
1702. <!-- COPY FROM Article 1, Listing 2 Lines 102 to105
        -->
1710.  <TextView
1711.    <!-- COPY FROM Article 3, Listing 2 Lines 1711–
           1716 -->
1720.  <TextView
1721.    <!-- COPY FROM Article 3, Listing 2 Lines 1721–
           1727 -->
1730.  <TextView
1731.    <!-- COPY FROM Article 3, Listing 2 Lines 1730–
           1739 -->
1740.  <ProgressBar
1741.    android:id="@+id/computation_progress"
1742.    style="?android:attr/progressBarStyleHorizontal"
1743.    android:layout_width="wrap_content"
1744.    android:layout_height="wrap_content"
1745.    android:layout_centerHorizontal="true"
1746.    android:layout_centerVertical="true"
1747.    android:indeterminate="false"
1748.    android:max="100"
1749.    android:progress="0"
1750.    android:visibility="invisible"
1751.   />

1760.  <Button
1761.    android:id="@+id/start_graph_time"
1762.    android:layout_width="wrap_content"
1763.    android:layout_height="wrap_content"
1764.    android:layout_below="@id/number_tgbn_sounds"
1765.    android:layout_centerHorizontal="true"
1766.    android:layout_centerVertical="true"
1767.    android:text="@string/button_start_graph_time"
1768.  />

1770.  <Button
1771.    android:id="@+id/start_graph_freq"
1772.    android:layout_width="wrap_content"
1773.    android:layout_height="wrap_content"
1774.    android:layout_below="@id/start_graph_time"
1775.    android:layout_centerHorizontal="true"
1776.    android:layout_centerVertical="true"
1777.    android:text="@string/button_start_graph_freq"

1778.  />

1779. </RelativeLayout>
```

Listing 1) ***activity_sound_analysis.xml*** **layout file from the** ***WAT_AN_APP\res\layout folder***

```
package com.wat_an_app;
// NEW CODE – INSERT AFTER Article 3, Listing 4 Lines 501– 512
513. import android.content.Intent;
514. import android.os.Environment;
515. import android.util.Log;
516. import android.view.Menu;
517. import android.view.MenuInflater;
518. import android.view.MenuItem;
519. import android.view.View;
520. import android.widget.Button;
521. import android.widget.ProgressBar;
522. import java.io.IOException;

public class SoundAnalysis extends ActionBarActivity{
 //NEW CODE – INSERT AFTER Article 3, Listing 4 Lines 551– 556

557. final CounterClass timer = new CounterClass(5000,250);
558. private static final double REFSPL = 0.00002;  // Hearing reference level
559. private Button button_graph_time; private Button button_graph_freq;
560. private MediaPlayer mPlayer=null;

600. @Override protected void onCreate(Bundle savedInstanceState) {
  //NEW CODE – INSERT AFTER Article 3, Listing 4 Lines 601– 603
  //Delete Article 3 Listing 4 Line 604
605.  button_graph_time = (Button) findViewById(R.id.start_graph_time);
606.  button_graph_freq = (Button) findViewById(R.id.start_graph_freq);
607. }
 // Same methods as in Article 3
 //protected void onStart() //Article 3 Listing 5 Lines 750 to 759
 //protected void onPause() //Article 3 Listing 5 Lines 800 to 804

 //public class CounterClass // Article 3 Listing 6 Lines 850 to 877
 //protected void onPreExecute() //Article 3 Listing 7 Lines 910 to 919
 //protected void onCancelled() //Article 3 Listing 10 Lines 1300 to 1305
 //protected boolean detectImpulse() // Article 3 Listing 10 Lines 1350 to 1356
 //protected boolean detectTGBN( )  // Article 3 Listing 10 Lines 1400 to 1406

 //NEW AND MODIFIED METHODS
 //public boolean onCreateOptionsMenu(Menu menu) //Listing 3A Lines
       610 to 615
 //public void startPlaying() //Listing 3A Lines 650 to 657
 //public boolean onOptionsItemSelected(MenuItem item) //Listing 3A
       Lines 700 to 708
//public void onTick_Article4(long millisUntilFinished) // Listing 3C Lines 860
       to 865
 //protected Integer doInBackground() //Listings 4A and 4B Lines 910 to 1199
 //protected void onProgressUpdate(Integer ... data) //Listing 5 Lines 1200 to 1220
 //protected void onPostExecute(Integer data) //Listing 5 Lines 1250 to 1259
 //protected int doubleFFT(double[][] samples, int numRecords, int sampleSize)
      // Listing 6A Lines 1450 to 1468
 //public static int nearestPow2Length(int length) //Listing 6B Lines 1500 to 1505
 //public void DisplayGraph(View v)
      // Listing 6C Lines 1600 to 1603
```

Listing 2) The prologue of the *SoundAnalysis.java* file (*WAT_AN_APP\src* folder) sets up the User Interface. The *OnCreate()* method enables the UI composed amongst other of two buttons (Lines 605 and 606). The other methods in this class are detailed in the different Listings

```
610.  public boolean onCreateOptionsMenu(Menu menu) {
611.   // Inflate the menu items for use in the action bar
612.   MenuInflater inflater = getMenuInflater();
613.   inflater.inflate(R.menu.menu_sound_analysis,
      menu);
614.   return super.onCreateOptionsMenu(menu);
615.  }

650.  public void startPlaying() {
651.   mPlayer = new MediaPlayer();
652.   try {
653.    mPlayer.setDataSource(Environment.
        getExternalStorageDirectory()
            .getAbsolutePath()+
              "/MySounds/Chirp_50_1000Hz.wav");
654.    mPlayer.prepare();
655.    mPlayer.start();
656.   } catch (IOException e) {}
657.  }

700.  public boolean onOptionsItemSelected(MenuItem
      item) {
   // Handle presses on the action bar items
701.  switch (item.getItemId()) {
702.   case R.id.GenerateChirp:
703.     startPlaying();
704.   return true;
705.   default:
706.     return super.onOptionsItemSelected(item);
707.   }
708.  }
```

Listing 3A) Details of the methods dealing with the Action Bar, use to display an icon outputting a Chirp sound saved in the phone memory

```
2500. <item
2501.   android:id="@+id/GenerateChirp"
2502.   android:icon="@drawable/ic_action_play"
2503.   android:showAsAction="always"
2504.   android:title="@string/audio_play"
2505. />
```

Listing 3B) *menu_sound_analysis.xml* file from *WAT_AN_APP\res\menu* folder to setup the action bar in the activity

```
860.  public void onTick_Article4(long millisUntilFinished) {
861.   int capture_time_ms=getResources().
        getInteger(R.integer.capture_time) * 1000;
862.   if(millisUntilFinished> (capture_time_ms - 650) &&
        millisUntilFinished< (capture_time_ms - 400)){
863.    int playChirp =
         getResources().getInteger(R.integer.playChirp);
864.    if (playChirp==1) startPlaying();
865.    }
      }
```

Listing 3C) *onTick_Article4()* from the CounterClass class

```
      //COPY FROM Article 3, Listing 5 and Listing 6
      private class CaptureAudio extends AsyncTask<Void, Integer, Integer>{
        //NEW CODE – INSERT AFTER Article 3, Listings 7 and 8 Lines 910– 981
982.    detectBuffer = null;
        //TO UPDATE FROM Article 3
       //sampleBuffer = null; //Delete 983 to allow extended background task to work
984.   if (recorder != null) { recorder.release(); recorder = null; }
985.   if (!isCancelled()) publishProgress(-1, -1, -1, -1, 0, -1);
       //return 0;}  //Delete 986 and 987 to allow extended background task to work
990.    final int numRecords = getResources().getInteger(R.integer.num_records);
       // copy the buffer into a buffer of double
991.    double[][] samples =new double[numRecords][sampleBufferLength];
992.    double max = 0;
993.    for(int n = 0; n < sampleBufferLength; n++){
994.        samples[0][n] = (double) sampleBuffer[0][n]; // Identify maximum value
995.        if(max < Math.abs(samples[0][n])) {max = samples[0][n];}
996.     }
997.     for(int h = 0; h < sampleBufferLength; h++) {samples[0][h] /= max;}

      // Grab first record for analysis and display
1000.   double[] toStorage_time = new double[sampleBufferLength];
1001.    for (int n = 0; n < sampleBufferLength; n++) {
1002.       toStorage_time[n] = samples[0][n] / REFSPL;
1003.   }
1004.   if (isCancelled()) {return -1;}

       // reduce the size of our sample so the graph can load in a normal amount of time
1005.   int samplesPerPoint = getResources().
          getInteger(R.integer.samples_per_bin_time);
1006.   int width_time = toStorage_time.length / samplesPerPoint ;
1007.   int samplerate = getResources().getInteger(R.integer.sample_rate);
1008.   double maxYval_time = 0;
1009.   final double[] tempBuffer_time = new double[width_time];
1010.   for (int k = 0; k < tempBuffer_time.length; k++) {
1011.    for (int n = 0; n < samplesPerPoint; n++){
1012.     tempBuffer_time [k] += toStorage_time [k * samplesPerPoint + n];
1013.    }
1014.    tempBuffer_time [k] /= (double) samplesPerPoint;
1015.    if (maxYval_time < tempBuffer_time [k]){
1016.     maxYval_time = tempBuffer_time [k];}
1017.   }
       // scaling the x "time" values stored into xVals
1018.   final double[] xVals_time = new double[tempBuffer_time.length];
1019.   for (int k = 0; k < xVals_time.length; k++) { // xVales.length=512
1020.    xVals_time [k] = k * (1.0*samplesPerPoint) / (samplerate);
1021.   }
       //Adding properties to clicking on the "GRAPH IN TIME DOMAIN" button
1025.   button_graph_time.setOnClickListener(new View.OnClickListener() {
1026.    public void onClick(View arg0) {
1027.      String which_button_pressed = "1";
1028.      Bundle extras_time_values = new Bundle();
1029.      extras_time_values.putDoubleArray("key_x_time", xVals_time);
1030.      extras_time_values.putDoubleArray("key_y_time", tempBuffer_time);
1031.      extras_time_values.putString("button_pressed ", which_button_pressed);
1032.      Intent intent_graph_time = new
            Intent(SoundAnalysis.this,DisplayGraph.class);
1033.      intent_graph_time.putExtras(extras_time_values);
1034.      intent_graph_time.putExtras(extras_time_values);
1035.      intent_graph_time.putExtras(extras_time_values);
1036.      startActivity(intent_graph_time);
1037.    }
1038.}); // Continues in Listing 4B
```

Listing 4A) Saving recorded data into the time domain using the *doInBackground()* step from the *CaptureAudio* class

startPlaying() method, Lines 650 to 657, outputs the *Chirp_50_1000Hz.wav* file located in a folder "*MySounds*" we must add into the root directory of our phone's internal memory. We found that pushing the Chirp play-button by hand meant that we often lost the last part of the Chirp. Rather than increasing the recording and analysis time we modified the *onTick_Article4()* mentioned in Article 3 to automatically play the Chirp about 400 ms after recording had started, Listing 3C, Lines 862 to 865.

TIME DOMAIN ANALYSIS

Our Article 3 code allowed the capture of up to 6 sound records. For calculation time and other issues, we decided to analyse just one record by changing the variable "*num_records*" (Article 3 Listing 3 Line 2830) from "6" to "1" in the *ints.xml* file. As can be seen from Fig. 2A, one record provided frequency domain signals with a good signal-to-noise (SNR) ratio. You will have to synchronize the start of the Chirp output capture to better that 1 sample period (1/8000 s). This is not straight forward when you have Android tasks running in addition to our WAT-AN-APP.

In Article 3, we mentioned how the *SoundAnalysis* activity uses an asynchronous task, *CaptureAudio*, to record and analyze the sound, and to update the activity's UI with this class that makes use of four steps. The same first step, *onPreExecute*, performs any necessary setup. The second step, *doInBackground*, is invoked to perform any background computations that take a long time. *doInBackground* needs to be extended by adding the Listing 4A code to the TGBN detector *doInBackground* code from Article 3 Listing 8.

We need to do some housekeeping steps to allow the extended background activity to execute: after having inserted the code from Article 3 Listings 7 and 8, uncomment the line 956, as it will make sure that the sample buffer we are working with has a length that is a power of 2 for the FFT calculations. If you forget this step, the FFT calculations won't be processed, and an error message will appear on the screen. Remove the setting of *sampleBuffer* pointer to *null* in Line 983, remove the *return* statement in Line 986 and the curly bracket for Line 987.

Preparing data for graphing can be time consuming unless the mobile CPU has lots of horse-power. There are efficient Android graphing packages available if the data is stored within a SQL data-base. As employing those will take another series of articles, we have taken a straight forward, bull-at-gate, approach. You have to handle possible overflow issues when doing integer DSP calculations. So in Lines 991 to 997, we convert the audio record to doubles, allowing graphing of the recorded sound. Lines 1000 to 1003 allows grabbing the record for future analysis and display. The samples are stored into an array *toStorage_time* and divided by a

constant *REFSPL* which is the threshold of the human hearing, used here as the reference sound pressure level and equal to 0.00002 (Line 1002).

Displaying the sound signal generates another CATCH-22 scenario. If you display all the information – then you have 4 seconds off sound (32,000 point) displayed on a small screen. It takes for ever to zoom in. We decided to speed the time display and zooming by doing a rough form of down sampling, Lines 1005 to 1017. If you try this with the Chirp signal, then you will find that your display does not show constant sound amplitude. The strong amplitude variation shows that you are no longer satisfing the Nyquist sampling rate and you get "display aliasing"

The "x-axis" time values are generated through Lines 1018 to 1021. We make the button "*Graph in Time domain*" clickable by using *setOnClickListener(),* Line 1025. Line 1036 calls the graphing *DisplayGraph* activity we discuss later.

This call requires the graph's "x" and "y" values to be passed from the *SoundAnalysis* activity to the *DisplayGraph* one using a bundle, Line 1028. This bundle makes use of two strings, *"key_x_time"* and *"key_y_time",* Lines 1029 and 1030 to pass the arrays *"xVals_time"* and *"tempBuffer_time"*. We put the string containing the value of the button that has been pressed to start the DisplayGraph activity into a key *"button_pressed"*, Line 1031. We then declare an intent that will start the *DisplayGraph* activity, and pass our two arrays and one string via the bundle "*extras_time_values*", Lines 1033, 1034, as well as the button that has been pressed to call the graphing activity, Line 1035. The data in the time domain can now be graphed.

The quick guide section at the end of this article shows you a way to write array's values into an external .txt file stored in the phone internal memory for debugging analysis off the phone. If you are not interested in saving the values into a file, you can also display the content of the *tempBuffer* and *xVals* arrays in the LogCat, by using the "Log" API that sends log output in the LogCat window, as we show in Listing 4B Line 1124. This line of code displays the recorded data amplitude's values in the frequency domain.

CONCLUSION"

We are about half-way there to getting the full DSP code ready to assist out our neighbour's kid with his volunteer work. In the next article we will tackle getting the fast Fourier transform (FFT) code to handle spectral analysis. Then we head out into the world of Android graphics, and provide that promised picture of a ghost.

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 he is not singing with Calgary's Adult Recreational Choir *ARC's Up2Something,* his 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

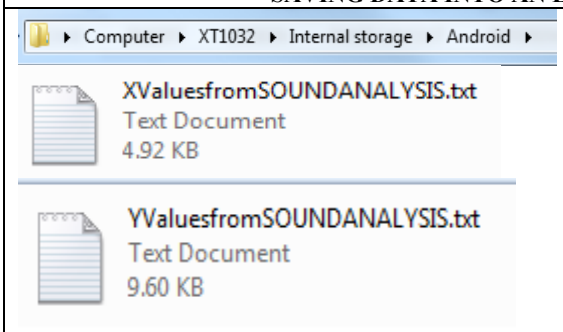
INSTALLING AN APK FILE ON A ANDROID DEVICE

Android application package file (APK) is the format of installable files on Android platform. In order to install an .apk on your phone, enable the "unknown sources" on Android (settings, security and check the box next to "unknown sources"). A dialogue box pops up asking you to confirm the action, knowing that "your phone and personal data are more vulnerable to attack by apps from unknown sources". Tap ok as this application is safe, but make sure to always know the source of the .apk files you install on your device (!). Connect your phone to the computer, and in the "USB Storage" folder, create a folder "My Applications". Get the application .apk from *CC website*, and copy it from your computer's download folder ("C:\Users\ajfgaspa\Downloads" by default) to the "My Applications" directory on your phone using your computer. From your phone, using the Google Play store, download and install an application as "file commander" (*play.google.com/store/apps/details?id=com.mobisystems.fileman&hl=en*) to take control over the files stored in your phone. Start file commander, tap on the "USB storage" folder, and then on "My Applications" folder: the application .apk file should be here. Tap it, a window pops up asking you to "complete action using". Select "Package installer", and confirm that you want to install this application. A few second later, the application has been installed. Click on "Open": the application starts!

DISPLAYING VALUES IN THE LOGCAT

Please note that if you want to display the values of a buffer on the LogCat, as for the tempBuffer containing our amplitudes "y" values, you have to use the line `// Log.d("ADebugTag", "Value of tempBuffer: " +Double.toString(tempBuffer[k]));` and import the Log library (*import android.util.Log;*)

SAVING DATA INTO AN EXTERNAL .TXT FILE SAVED ON THE PHONE MEMORY




```
String file_path4 = Environment.getExternalStorageDirectory()
  .getAbsolutePath() + "/Android/"; //Store file in the Android folder from the phone
     Internal memory
File file4 = new File(file_path4 + "/YValuesfromSOUNDANALYSIS.txt"); //file's name
  sendBroadcast(new Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE,
  Uri.fromFile(file4))); //force the file to be displayed when we look for it from the
computer
FileOutputStream fos4 = null;
try {
  fos4 = new FileOutputStream(file4);
  } catch (FileNotFoundException e1) {//exception: file not found
    e1.printStackTrace();
    }
OutputStreamWriter osw4 = new OutputStreamWriter(fos4);
try {
  String b4 = Arrays.toString(ReceivedyValues); //display the array "ReceivedYvalues" in
                a   text file as a string
  osw4.write(b4); osw4.flush();osw4.close();
  } catch (FileNotFoundException e) {
    e.printStackTrace();
    } catch (IOException e) {
    e.printStackTrace();
    }
```

FORCING AN ACTIVITY TO START IN THE LANDSCAPE SCREEN ORIENTATION

To force the graphing *DisplayGraph* activity to start in the landscape screen orientation, you have to modify the *AndroidManifest.xml* file. When we have created the activity, the lines containing the *android:name* and *android:label* have automatically been added to the manifest file. Add the lines *android:screenOrientation* and *android:configChanges* as shown on the sideline code to respectively display the activity in the landscape screen and hide the phone's keyboard.

```
<activity
  android:name=".DisplayGraph"
  android:screenOrientation="landscape"
  android:configChanges="orientation|keyboardHidden"
  android:label="@string/title_activity_display_graph">
</activity>
```

Table 1: Quick Help Guide