DETECTING EMOTION IN THE VOICE

A Model and Software Implementation

© Harold Rodriguez
1231 SW 3rd Ave • Apt. 121
Phone 954.464.0839 • Email drharold@ufl.edu

Table of Contents

Chapter 1

A Model

How to use 3-Dimensional space to model emotions inherent in fluctuations of the voice.

There has been a lot of work done on voice recognition in the past few years, largely due to increased computing power. It is not uncommon today to have, after voice calibration, 95% accuracy in word recognition.

But word recognition and **emotion detection** are not the same thing. Whether one says, "Tell him to get down here," or screams, "*Tell him to get down here*!" into the microphone, the speech engine will gently scribe the former (assuming the microphone still works).

COMMONLY DETECTABLE EMOTIONS

- ☞ Anger
- ☞ Sadness
- ☞ Happiness
- ☞ Neutrality

Ultimately, one would like the unification of both recognitions, where, "Ah," and "Ah!" are distinct. While the word recognition camp seems to have made great strides in past decades, the emotion recognition camp is struggling to keep up.

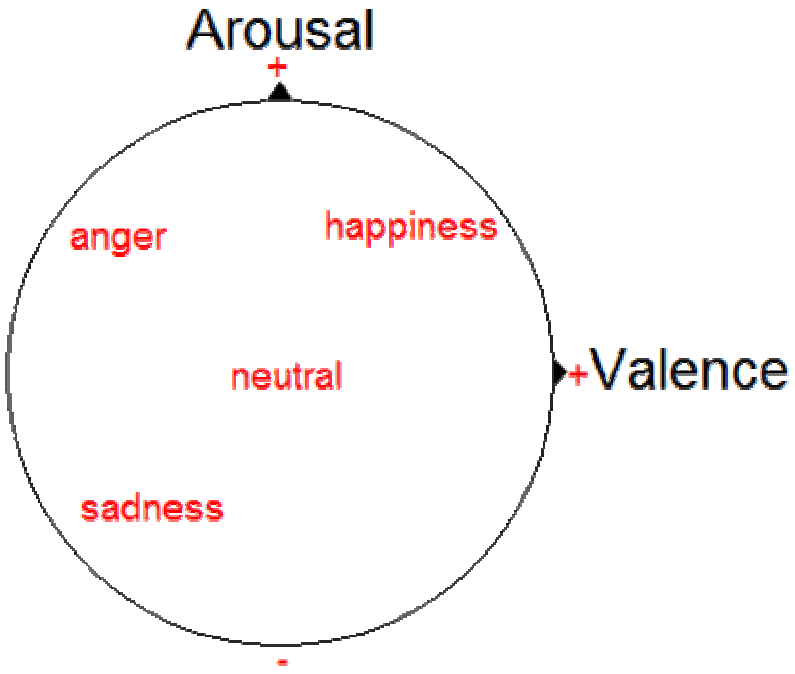
Current Detection Methods

Today's greatest need for emotion detection software comes from the telecommunications industry. For example, the manager of a large call center could run software on the telephone lines to analyze customer calls. This would alleviate his having to go through every call, and he would instead focus on those that were considered "angry". He might even consider re-evaluating an employee with a high number of "first happy, then angry" customers.

Current detection methods are only good at detecting four basic emotions: Anger, Sadness, Happiness, and Neutrality. Even in such a limited scope, the successful detection rate is near 75%, hardly close to the 95% for word recognition.

Symbolic Mapping

These methods usually rely on a relation between **arousal** and **valence** in the voice. Arousal can be measured by "highness" or "lowness", while valence can be measured in "positive" or "negative". For instance, 'anger' would have high arousal and negative valence.



Valence vs. Arousal
A summary of the four commonly detectable emotions.

These parameters have some drawbacks. For one, they don't translate into anything computational. One can not determine "valence" from a waveform. You may attempt to use, however, **pitch** and **volume** to roughly correspond to the parameters.

3-Dimensional Mapping

This research ambitiously proposes including several low-percentage emotions (those that are detected with $< \approx 40\%$ accuracy) with the use of a vector in $\mathbb{R}^3$ to analyze a voice.

The Dimensions

The voice will be modeled after the calculatable quantities: **pitch**, **volume**, and **idle percentage**. Both pitch and volume will be determined through waveform analysis, and idle percentage will be defined as the ratio: *duration of the inaudible segments* divided by the *duration of the audible segments* in the recording buffer.

Using these three parameters, we can attempt to detect Anger, Sadness, Happiness, Neutrality, Laughing, Boredom, and Defensiveness[1].

New Emotional State Definitions

Pitch, *Volume*, and *Idle Percentage* can be used to determine emotion using the following relations:

- **Anger** - Volume, pitch, & idle percentage are large.
- **Sadness** - All signs abysmal.
- **Happiness** – Elevated pitch and volume; less idle percentage than anger.
- **Neutrality** - Voice at *natural frequency*, centered *volume center*.
- **Laughing** – Idle percentage is large, as is volume, at the *natural frequency*.
- **Boredom** – Pitch is below *natural frequency*; volume and idle percentage are low.
- **Silence** - Pitch is less than *ambient frequency* + 7%.

Variable definitions:

ambient frequency **- Microphone no-sound threshold**
natural frequency **- Average frequency at which the person speaks**
volume center **– Average volume at which the person speaks**

The following page shows a segment diagram to express the *emotion-to-variables* relation in the way software would interpret.

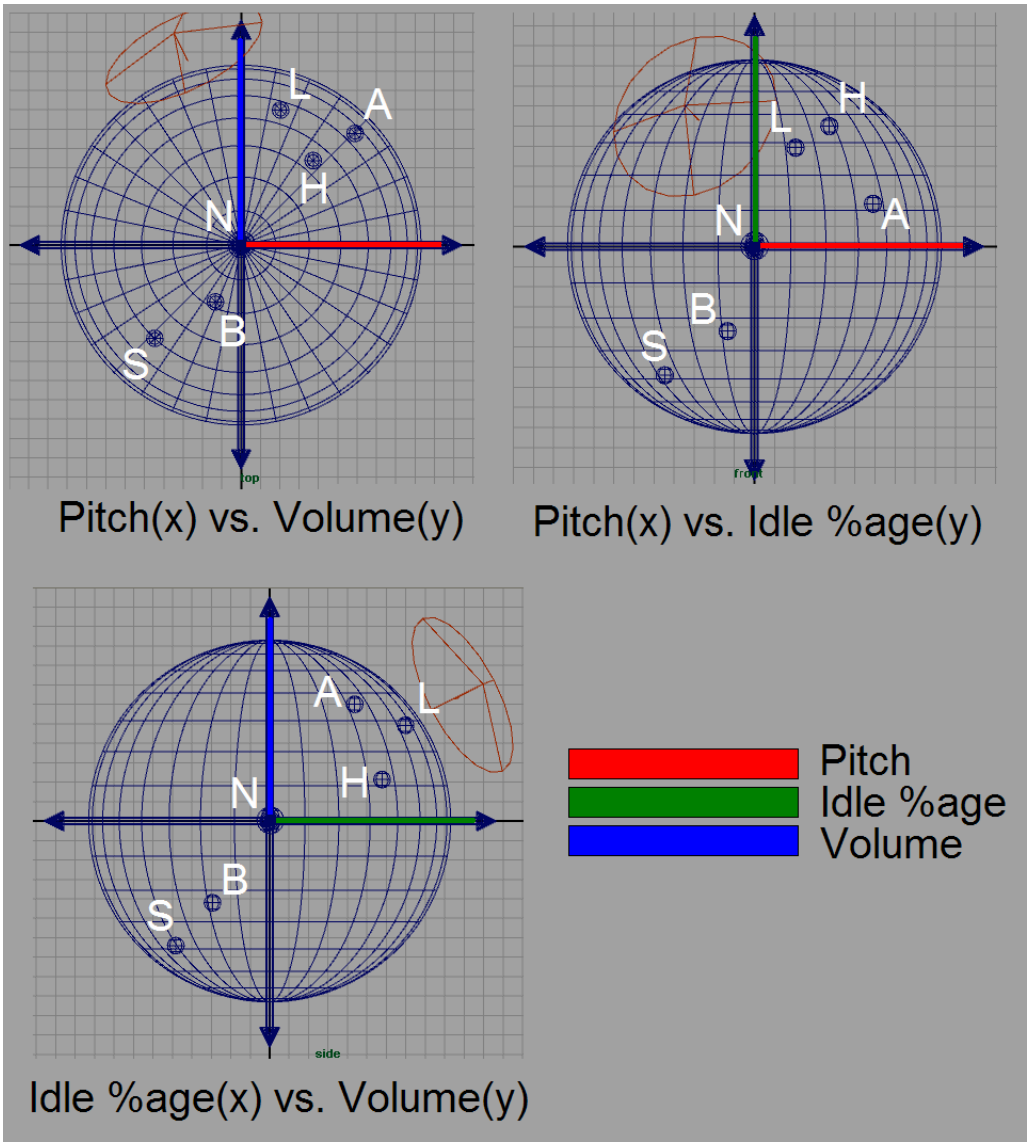
[1] Defensiveness was ultimately discarded, with definition: Voice is operating at *natural frequency* but with a large volume.

Voice Dimensions

Using Maya® 3-D modeling software, it is possible to visualize the voice interpreting scheme.

3-DIMENSIONAL EMOTIONS

- Anger
- Sadness
- Happiness
- Neutrality
- Laughing
- Boredom



Parameter Relations
A summary of the emotion configurations in R^3.

The letters correspond to the emotions' relative locations within a "sphere of absoluteness" (i.e. the arbitrary but finite limit for a parameter's value). This wireframe of the full model has superimposed colors corresponding to the emotion parameter the axis represents.

A fly-through movie of the real rendered model is available at: http://www.planetharold.com/download/files/voicedimensions.wmv.

Chapter 2

Software Implementation

How to manipulate waveforms using the FMOD Sound API.

Sound engineering is not child's play. Fortunately, the creators of the **FMOD Sound API** don't seem to think so.

The FMOD Sound API

The FMOD Sound API (hereby regarded as FMOD) is a highly successful architecture for working with audio. This enterprise-caliber sound library is highly respected worldwide, and it can be used in a variety of applications, including audio for console gaming systems (Gamecube, PS2/PS3, XBOX/XBOX360) as well as PC/MAC/Linux games.

FMOD comes with hundreds of functions for **manipulating and creating sounds**. FMOD can be used with Visual Basic or Delphi, but it is primarily intended for use in C++. FMOD comes with a dll, and import library, and C header.

NativeFMODEx

FMODEx is the latest version of FMOD, which boasts powerful new features.

Although FMOD provides no support for using its library in Java, computer scientist Jérôme Jouvie has ported the library using the Java Native Interface (JNI). What he dubs **NativeFMODEx** is what allows a programmer to access FMODEx functions while coding entirely in Java.

The code in this research is written entirely in Java, accessing the FMODEx Sound API through NativeFMODEx.

The Code

In order to run the software, you must have the FMOD sound library and the NativeFMOD Java package. The following is the software implementation for detecting emotion in the human voice **in its entirety**. Elaborations follow.

```
/*=================================================================
Emotion Detection Software,   (c)Harold Rodriguez, 2006      v0.8
=================================================================*/
import java.nio.ByteBuffer;
import java.nio.FloatBuffer;
import java.io.*;
import java.awt.*;
import org.jouvieje.FmodEx.*;
import org.jouvieje.FmodEx.System;
import org.jouvieje.FmodEx.Defines.INIT_MODES;
import org.jouvieje.FmodEx.Enumerations.*;
import org.jouvieje.FmodEx.Exceptions.InitException;
import org.jouvieje.FmodEx.Misc.BufferUtils;
import org.jouvieje.FmodEx.Structures.FMOD_CREATESOUNDEXINFO;
import Tool.es;
import java.util.Calendar;
import static java.lang.System.*;
import java.io.*;
import static org.jouvieje.FmodEx.Defines.FMOD_INITFLAGS.*;
import static org.jouvieje.FmodEx.Defines.FMOD_MODE.*;
import static org.jouvieje.FmodEx.Enumerations.FMOD_CHANNELINDEX.*;
import static org.jouvieje.FmodEx.Enumerations.FMOD_DSP_FFT_WINDOW.*;
import static org.jouvieje.FmodEx.Enumerations.FMOD_SOUND_FORMAT.*;

public class EmoDetectionSoftware
{
   /*==================================*/ //Buffer declarations
   private final static int OUTPUTRATE = 48000;
   private final static int SPECTRUMSIZE = 8192;
   private final static float SPECTRUMRANGE = ((float)OUTPUTRATE / 2.0f);      //Nyquist frequency
   private final static float BINSIZE = (SPECTRUMRANGE / (float)SPECTRUMSIZE);
   public static BufferedReader filereader;                                                  //fI
   public static BufferedReader reader = new BufferedReader(new InputStreamReader(in));      //I
   public static BufferedWriter writer;                                                      //O
   private static char key;
   private static FloatBuffer spectrum = BufferUtils.newFloatBuffer(SPECTRUMSIZE);
   private static int recBufferSize = 50;   //Size of audio segment to analyze
   /*==================================*/

   public static void main(String[] args) throws IOException
   {
      /*==================================*/ //Start interface
      out.printf("================================================\n");
      out.printf("Emotion Detection Software,   (c)Harold Rodriguez, 2006\n");
      out.printf("================================================\n");
      out.printf("Press 'E' and ENTER at any time to quit.\n\n");
      out.printf("Waiting to calibrate surroundings. Please make your surroundings as quiet as
possible.\n");
      out.printf("    When finished, press ENTER to calibrate system.\n");
```

```
      String enter = "";
      try   {enter = reader.readLine();}
      catch(IOException ioe) {err.println("\n Error reading input.");exit(1);}
      if (enter.compareTo("e") == 0 || (enter.compareTo("E") == 0))
         {out.println("Quitting...");exit(0);}
      /*==================================*/ //NativeFmodEx Init
      try
      {Init.loadLibraries(INIT_MODES.INIT_FMOD_EX_MINIMUM);}
      catch(InitException e)
      {out.printf("NativeFmodEx error! %s\n", e.getMessage()); exit(1);}
      /*==================================*/ //Object declarations
      System system  = new System();
      Sound sound   = new Sound();
      Channel channel = new Channel();
      FMOD_RESULT result;
      /*==================================*/ //Create and initialize system
      result = FmodEx.System_Create(system);
      result = system.init(32, FMOD_INIT_NORMAL, null);
      /*==================================*/ //Channel preparation
      FMOD_CREATESOUNDEXINFO exinfo = FMOD_CREATESOUNDEXINFO.create();
      exinfo.setNumChannels(1);
      exinfo.setFormat(FMOD_SOUND_FORMAT_PCM32);
      exinfo.setDefaultFrequency(OUTPUTRATE);
      exinfo.setLength(exinfo.getDefaultFrequency() * BufferUtils.SIZEOF_SHORT *
exinfo.getNumChannels() * 5);
      result = system.createSound((String)null, FMOD_2D | FMOD_SOFTWARE | FMOD_LOOP_NORMAL |
FMOD_OPENUSER, exinfo, sound);
      exinfo.release();
      /*==================================*/ //Sound Engineering Variables
      long rightNow = Calendar.getInstance().getTimeInMillis(), lastNow = rightNow;    //Used for
time-splicing
      long startIdleClock = rightNow, startAnalysisClock = rightNow;
      float averageFreq = 90;            //Average vocal pitch
      float averageFreqProbe = 90;      //Prober for average pitch
      float dominanthz = 0;            //Instantaneous frequency
      float ambientFrequency = 60;      //Microphone no-sound threshold
      float volumeCenter = 0;          //Average volume at which the person speaks
      float idlePercentage = 0;         //Duration of silent moments/total moments
      int recBufferPointer = 0;         //Pointer to the current iteration (buffer location)
      int freqProbeCount = 0;          //Number of times frequency was probed as existing
      int idleProbeCount = 0;          //Number of times idleness was probed as existing
      int counter = 0;                //All-purpose counter
      Boolean isIdle = true;            //Determines whether the speaker is speaking or not
      int bin = 0;                    //Pointer to location in spectrum array
      float max = 0;                   //Previous value in spectrum array
      float volume = 0;                //Instantaneous volume
      /*==================================*/ //Emotion Analysis Variables
      String currentState = "Neutral";   //Current emotional state
      String analysis = "";             //Analysis to be output to file
      String lastAnalysis = "";         //Last analysis appended to file
      String lastLine;                //Last line of the file
      int isIdleCount = 0;             //Number of times user was idle
      int naturalFreqCount = 0;         //Number of times user was near natural frequency
      int happyCount = 0;               //Number of times user was happy
      int sadCount = 0;                //Number of times user was sad
      int angerCount = 0;               //Number of times user was angry
      int boredomCount = 0;            //Number of times user was bored
      int laughingCount = 0;            //Number of times user was laughing
      float naturalFrequency = 110;      //Average frequency at which the person normally speaks
      float previousAverageFreq = 100;   //Average frequency before latest update
```

```
float averageIdlePercent = 0;      //Average idle percentage over analysis period
float averageVolume = 0;         //Average volume over analysis period
/*==================================*/ //Start recording
result = system.recordStart(sound, true);
try   {Thread.sleep(recBufferSize);}  //Recording buffer time
catch(InterruptedException e){}
result = system.playSound(FMOD_CHANNEL_REUSE, sound, false, channel);
result = channel.setVolume(0);
/*==================================*/ //Determine ambient frequency
do
{
   bin = 0; //Pointer to location in spectrum array
   max = 0;
   result = channel.getSpectrum(spectrum, SPECTRUMSIZE, 0, FMOD_DSP_FFT_WINDOW_BLACKMAN);
   for(int i = 0; i < SPECTRUMSIZE; i++)
   {
      if(spectrum.get(i) > 0.01f && spectrum.get(i) > max)
      {
         max = spectrum.get(i);
         bin = i;
      }
   }
   dominanthz = (float)bin * BINSIZE;   //Instantaneous room frequency
   rightNow = Calendar.getInstance().getTimeInMillis();
   if (dominanthz != 0.0) //Exclude startup detection instances
   {
      averageFreq += dominanthz; counter++; //averageFreq will be divided by counter
   }
   system.update();
   try{Thread.sleep(1);}catch(InterruptedException e){} //Effective sampling rate
}
while(rightNow - lastNow < 3000);
if (counter != 0) {ambientFrequency = averageFreq/counter;} else {ambientFrequency = 0;}
out.printf("=====================================\n");
out.printf("The room has been analyzed to have ambient frequency of:%3.0f",ambientFrequency);
out.printf(" Hz.\n\n");
/*==================================*/ //Determine natural frequency and volume of speaker
out.printf("Please read the following text aloud in your 'regular', everyday voice.\n");
out.printf("When ready to speak the text, press ENTER.\n");
try   {enter = reader.readLine();}
catch(IOException ioe) {err.println("\n Error reading input.");exit(1);}
if (enter.compareTo("e") == 0 || (enter.compareTo("E") == 0))
   {out.println("Quitting...");exit(0);}
try{Thread.sleep(1000);}catch(InterruptedException e){}
/*==================================*/ //Background check for Exit requests
Thread keyHit = new Thread()
{
   public void run()
   {
      boolean end = false;
      while(!end)
      {
         key = es.readChar();
         if(key == 'e' || key == 'E')
            end = true;
      }
   }
};
keyHit.start(); //Begin exit-checker thread
keyHit.setPriority(keyHit.MIN_PRIORITY); //Don't want to bog our main research down
```

```
      /*==================================*/
      out.printf("======================================\n");
      out.printf("'This text I am reading will be used to calibrate the system. I should
remain\n");
      out.printf("calm and speak into the microphone using my natural voice.\n\n");
      try{Thread.sleep(6000);}catch(InterruptedException e){}
      counter = 0;averageFreq = 0; lastNow = rightNow;
      do
      {
         /*==================================*/ //Natural frequency portion
         bin = 0; //pointer to location in array spectrum
         max = 0;
         result = channel.getSpectrum(spectrum, SPECTRUMSIZE, 0, FMOD_DSP_FFT_WINDOW_BLACKMAN);
         for(int i = 0; i < SPECTRUMSIZE; i++)
         {
            if(spectrum.get(i) > 0.01f && spectrum.get(i) > max)
            {
               max = spectrum.get(i);
               bin = i;
            }
         }
         dominanthz = (float)bin * BINSIZE;   //Instantaneous voice frequency
         rightNow = Calendar.getInstance().getTimeInMillis();
         if (dominanthz > ambientFrequency*1.07) //Don't factor in ambient frequency samples
         {
            averageFreq += dominanthz; //Natural frequency = averageFreq/counter
            counter++;
         }
         /*==================================*/ //Volume center portion
         volume = 0;
         result = channel.getWaveData(spectrum, recBufferSize, 0);
         for(int i = 0; i < SPECTRUMSIZE; i++)
         {if(spectrum.get(i) > 0.01f && spectrum.get(i) > volume){
               volume = spectrum.get(i);
         }}
         volumeCenter += volume; //volumeCenter will be divided by counter
         /*==================================*/ //Update
         rightNow = Calendar.getInstance().getTimeInMillis();
         system.update();
         try{Thread.sleep(1);}catch(InterruptedException e){} //Effective sampling rate
      }
      while(rightNow - lastNow < 10000);
      if (counter != 0){naturalFrequency = averageFreq/counter;volumeCenter/=(float)counter; }
//Definitions
      else {out.printf("You must speak to calibrate. Quitting...");exit(0);}
      out.printf("\nThank you. Your voice has been analyzed.\n");
      try{Thread.sleep(6000);}catch(InterruptedException e){}
      /*==================================*/ //Main functionality
      try{writer = new BufferedWriter(new FileWriter("Emotion Analysis.txt"));} //Output to file
      catch (IOException e){}
      try   {writer.write("Analysis of Emotional States");writer.newLine();
      writer.write("==================================");writer.newLine();writer.flush();}
      catch (IOException e) {}
      do
      {
         /*==================================*/ //Detect instantaneous frequency
         bin = 0; //Pointer to location in array spectrum
         max = 0;
         result = channel.getSpectrum(spectrum, SPECTRUMSIZE, 0, FMOD_DSP_FFT_WINDOW_BLACKMAN);
         for(int i = 0; i < SPECTRUMSIZE; i++)
```

```
        {if(spectrum.get(i) > 0.01f && spectrum.get(i) > max){
              max = spectrum.get(i);
              bin = i;
        }}
        if (dominanthz*3 < (float)bin * BINSIZE)
        {
           dominanthz += 10;
        }
        else if ((float)bin * BINSIZE < dominanthz/3)
        {
           dominanthz -= 10;
        }
        else
        {dominanthz = (float)bin * BINSIZE;}   //Instantaneous voice frequency
        /*==================================*/ //Detect volume
        volume = 0;
        result = channel.getWaveData(spectrum, recBufferSize*50, 0);
        for(int i = 0; i < recBufferSize*50; i++)
        {if(spectrum.get(i) > 0.05f && spectrum.get(i) > volume){
           volume = (spectrum.get(i));
        }}
        /*==================================*/ //Calculate average frequency
        previousAverageFreq = averageFreq;
        rightNow = Calendar.getInstance().getTimeInMillis();
        if (rightNow - lastNow > recBufferSize)
        {
           lastNow = rightNow;
           if (freqProbeCount==1 || freqProbeCount==0) {averageFreqProbe/=freqProbeCount;} else
{averageFreqProbe /= (freqProbeCount-1);}
           if (averageFreqProbe > ambientFrequency*.90f)
              averageFreq = averageFreqProbe;
           averageFreqProbe = 0; freqProbeCount=0;
        }
        else
        {
           if ((averageFreqProbe/freqProbeCount)*1.1f < dominanthz && dominanthz <
naturalFrequency+500)
           {
              averageFreqProbe += dominanthz; //freqProbeCount--;
           }
        }
        freqProbeCount++;
        /*==================================*/ //Calculate idle percentage
        if (dominanthz < ambientFrequency*1.07) //The user is idle
        {isIdle = true;}
        else                            //The user is not idle
        {isIdle = false;}

        if (rightNow - startIdleClock < 1000) //Analyze for a second
        {
           if (isIdle)
           {idleProbeCount++;}
        }
        else
        {
           idlePercentage = (float)idleProbeCount/142f;
           idleProbeCount=0;
           startIdleClock = rightNow;
        }
        /*==================================*/ //Analysis Execution
```

```
         /*==================================*/
         if (rightNow - startAnalysisClock < 16000)   //Analyze for 16 seconds
         {}
         else
         {
            startAnalysisClock = rightNow;
            isIdleCount=0;
            isIdleCount = 0;
            naturalFrequency = 90;
            averageIdlePercent = 0;
         }
         /*==================================*/ //Silence
         if (isIdle)
         {
            isIdleCount++;
            if (isIdleCount >= 500)
            {
               currentState = "Silent";
               if (isIdleCount >= 900)
               {
                  analysis = "Silence";
                  isIdleCount = 0;
               }
            }
         }
         else
         {
            isIdleCount = 0;
         }
         /*==================================*/ //Neutral - Voice at natural frequency, centered
volumeCenter
         if ((averageFreq > naturalFrequency*.55f || isIdle) && (averageFreq < naturalFrequency*2f)
&& volume < volumeCenter+.2f)
         {
            if (!isIdle) {naturalFreqCount++;currentState = "Neutral";}
         }
         else if (averageFreq>naturalFrequency+600f) {naturalFreqCount++;} //Deviant S's
         else
         {
            if (naturalFreqCount >= 800f)   //Good case for being neutral
            {
               naturalFreqCount -= 1f;
            }
            else if (previousAverageFreq*3f - averageFreq < -10f)   //Be leniant towards sudden
outliers
            {
               naturalFreqCount -= 5f;
            }
            else
            {
               if (volume > volumeCenter)      //Punish any real outbursts
                  {naturalFreqCount -= 6f;}
               naturalFreqCount--;
            }
         }
         if (naturalFreqCount < 0) naturalFreqCount = 0;
         if (naturalFreqCount >= 800f)
         {
            analysis = "Neutral";
            naturalFreqCount = 0;
```

```
         }
         /*==================================*/ //Happy - Higher pitch and volume, less Idle %age
than angry
         if (volume<volumeCenter+.4f)
         {
            if ((averageFreq > naturalFrequency+5f || isIdle) && (averageFreq <
naturalFrequency*2.2f) && volume > volumeCenter-.3f)
            {
               if (!isIdle && idlePercentage < 45)
               {
                  happyCount++;
                  if (angerCount<150 && lastAnalysis.compareTo("Angry") !=0 &&
lastAnalysis.compareTo("Very angry") !=0
                        && lastAnalysis.compareTo("Sad") !=0) //Don't have a history of
anger/sadness
                  {
                     if (volume-volumeCenter<.3f)
                     currentState = "Happy";
                  }
               }
            }
            else if (averageFreq>naturalFrequency+600f) {happyCount++;} //Deviant S's
            else
            {
               if (happyCount >= 250f)   //Good case for being happy
               {
                  happyCount -= 1f;
               }
               else if (previousAverageFreq*3f - averageFreq < -10f)   //Be leniant towards sudden
outliers
               {
                  happyCount -= 3f;
               }
               else
               {
                  if ((!isIdle) && (volume < volumeCenter-.2f || averageFreq < naturalFrequency-
20))      //Punish any downers
                     {
                        if (volume < volumeCenter-.2f) {happyCount -= 1f;}
                        else
                        {
                           if (averageFreq>ambientFrequency+5)
                           happyCount -= 4f;
                        }
                     }
               }
            }
            if (happyCount < 0) happyCount = 0;
            if (happyCount >= 900f)
            {
               analysis = "Happy";
               happyCount = 0;
            }
            if (currentState.compareTo("Happy") != 0) {happyCount--;} else {happyCount++;}
         }
         /*==================================*/ //Laugh - Idle%age is large as is volume
         if (volume-volumeCenter>.2f && averageFreq < 2.5f*naturalFrequency && idlePercentage>80)
            {laughingCount += 5f; currentState="Laughing";}
         else
         {
```

```
            if (laughingCount >= 250f)   //Good case for being bored
            {
               laughingCount -= 1f;
            }
            else
            {
               if ((!isIdle) && (volume < volumeCenter+.1f))   //Punish any downers
                  {
                     boredomCount -= 1f;
                  }
            }
         }
         if (boredomCount < 0) boredomCount = 0;
         if (boredomCount >= 500f)
         {
            analysis = "Laughing"; boredomCount = 0;
         }
         /*==================================*/ //Angry - Volume, pitch, and idle %age are large
         if (volume-volumeCenter>.2f && averageFreq > 2.5f*naturalFrequency && idlePercentage>60)
            {angerCount += 5f; currentState="Angry";}
         if (volume-volumeCenter>.5f)
            {angerCount += 7f;currentState="Very angry";}
         if ((averageFreq > naturalFrequency+50f || isIdle) && (averageFreq <
naturalFrequency*2.5f) && volume > volumeCenter)
         {
            if (!isIdle)
            {
            if (volume-volumeCenter>.1f)
               {angerCount++;}
            else if (volume-volumeCenter>.2f)
               {angerCount += 2f;}
            else if (volume-volumeCenter>.3f)
               {angerCount += 3f;}
            else if (volume-volumeCenter>.4f)
               {angerCount += 5f;}
            }
            if (currentState.compareTo("Very angry")!=0) currentState = "Angry";
         }
         else if (averageFreq>naturalFrequency+600f) {angerCount++;} //Deviant S's
         else
         {
            if (angerCount >= 150f)   //Good case for being angry
            {
               angerCount -= 1f;
            }
            else if (previousAverageFreq*3f - averageFreq < -10f)   //Be leniant towards sudden
outliers
            {
               angerCount -= 3f;
            }
            else
            {
               if ((!isIdle) && (volume < volumeCenter-.2f || averageFreq < naturalFrequency-20))
//Punish any downers
                  {
                     if (volume < volumeCenter-.2f) {angerCount -= 4f;}
                     else
                     {
                        if (averageFreq>ambientFrequency+5)
                           angerCount -= 1f;
```

```
                }
             }
          }
       }
       if (angerCount < 0) angerCount = 0;
       if (angerCount >= 200f)
       {
          if (volume-volumeCenter>.5f)
             {angerCount += 7f;analysis="Angry";}
          else if (volume-volumeCenter>.6f)
             {angerCount += 8f;analysis="Angry";}
          else if (volume-volumeCenter>.7f)
             {angerCount += 9f;analysis="Very angry";}
          else if (volume-volumeCenter>.8f)
             {angerCount += 11f;analysis="Very angry";}
          else if (volume>volumeCenter+.9f)
             {angerCount += 15f;analysis="Very angry";}
          angerCount = 0;
       }
       if (currentState.compareTo("Anger") != 0) {} else {angerCount++;}

       /*==================================*/ //Bored - Pitch, Volume, and Idle %age are below
natural frequency
       if (volume<volumeCenter)
       {
          if (((averageFreq > naturalFrequency*.8) && averageFreq < naturalFrequency*1.1) ||
isIdle)
          {
             if (!isIdle)
             {
                if (idlePercentage > 95)
                {
                   boredomCount += 23f;;
                }
                else
                   boredomCount += 17f;;
                if (!isIdle) currentState = "Bored";
             }
          }
          else if (averageFreq>naturalFrequency+600f) {boredomCount++;} //Deviant S's
          else
          {
             if (boredomCount >= 250f)   //Good case for being bored
             {
                if (currentState.compareTo("Sad")!=0) boredomCount -= 1f;
             }
             else if (previousAverageFreq*3f - averageFreq < -10f)   //Be leniant towards sudden
outliers
             {
                if (currentState.compareTo("Sad")!=0) boredomCount -= 2f;
             }
             else
             {
                if ((!isIdle) && (volume > volumeCenter+.3f))   //Punish any outbursts
                   {
                      if (currentState.compareTo("Sad")!=0) boredomCount -= 1f;
                   }
             }
          }
          if (boredomCount < 0) boredomCount = 0;
```

```
            if (boredomCount >= 100f)
            {
               analysis = "Bored"; boredomCount = 0;
            }
         }
         /*==================================*/ //Sad - All signs abysmal
         if (volume<volumeCenter)
         {
            if (averageFreq < naturalFrequency-10f || isIdle)
            {
               if (!isIdle)
               {
                  sadCount++;
                  if (idlePercentage > 60)
                  {
                     sadCount++;
                  }
                  currentState = "Sad";
               }
            }
            else if (averageFreq>naturalFrequency+600f) {sadCount++;} //Deviant S's
            else
            {
               if (sadCount >= 200f)   //Good case for being sad
               {
                  sadCount -= 1f;
               }
               else if (previousAverageFreq*3f - averageFreq < -10f)   //Be leniant towards sudden
outliers
               {
                  sadCount -= 3f;
               }
               else
               {
                  if ((!isIdle) && (volume > volumeCenter || averageFreq > naturalFrequency+20))
//Punish any outbursts
                     {
                        if (volume < volumeCenter+.2f) {sadCount -= 1f;}
                        else
                        {
                           if (averageFreq>ambientFrequency+5)
                           sadCount -= 2f;
                        }
                     }
               }
            }
            if (sadCount < 0) sadCount = 0;
            if (sadCount >= 500f)
            {
               analysis = "Sad";
               sadCount = 0;
            }
         }
         /*==================================*/ //Write to file
         try {filereader = new BufferedReader(new FileReader("Emotion Analysis.txt"));} catch
(FileNotFoundException e1) {e1.printStackTrace();}
         while ((lastLine = filereader.readLine()) != null) {lastAnalysis = lastLine;}
         if (analysis !="" && analysis.compareTo(lastAnalysis) !=0 && !(analysis.compareTo("Angry")
==0 && lastAnalysis.compareTo("Happy")==0 && !(analysis.compareTo("Happy") ==0 &&
lastAnalysis.compareTo("Angry")==0)))
```

```
         {
            try   {writer.write(analysis);writer.newLine();writer.flush();} catch (IOException e)
{}
         }
         else
         {
            //The user is in a steady state of emotion
         }
         /*==================================*/
         /*==================================*/ //Universal  updates
         system.update();
         if (recBufferPointer<recBufferSize-1){recBufferPointer++;}else{recBufferPointer=0;}
//Reset buffer pointer
         try{Thread.sleep(1);}catch(InterruptedException e){} //Effective sampling rate
         if (!(dominanthz < 70 || volume < 0.01 || averageFreq < 70))
         {
            if (rightNow - lastNow < recBufferSize)
            {out.printf("Your status: %s",currentState);}
            //out.printf(" AVF: %3.0f, DOM: %3.0f, VOL: %3.0f, IDL: %3.0f,
\r",averageFreq,dominanthz,volume*100,idlePercentage*100);
         }
      }
      while(keyHit.isAlive()); //Quit if Thread dies (pressing E, Enter)
      /*==================================*/ //Shutdown
      writer.close();
      out.println("\nEmotion Analysis.txt has been written to disk.\n");
      result = sound.release();
      result = system.release();
      exit(0);
   }
}
```

This is the end of the program.

Initialization

```
/*==================================*/ //NativeFmodEx Init

try

{Init.loadLibraries(INIT_MODES.INIT_FMOD_EX_MINIMUM);}

catch(InitException e)

{out.printf("NativeFmodEx error! %s\n", e.getMessage()); exit(1);}

/*==================================*/ //Object declarations

System system  = new System();

Sound sound   = new Sound();

Channel channel = new Channel();

FMOD_RESULT result;

/*==================================*/ //Create and initialize system

result = FmodEx.System_Create(system);

result = system.init(32, FMOD_INIT_NORMAL, null);
```


The first part of the program is the necessary NativeFMODEx initialization. Then we instantiate our System, Sound, and Channel objects (as opposed to declaring pointers in FMOD's C++ implementation). Finally, we initialize the master System object and call FMOD's Init function, which is usually defaulted to mix at 44100 Hz and use 32 software channels.

RESULT

☞ FMOD_RESULT result is the backbone to FMOD. All objects, on successful method execution, return FMOD_OK, which we store in result. Otherwise, an FMOD_ERR enumeration is returned.

Ambient Frequency

```
do
{
        bin = 0; //Pointer to location in spectrum array
        max = 0;
        result = channel.getSpectrum(spectrum, SPECTRUMSIZE, 0, FMOD_DSP_FFT_WINDOW_BLACKMAN);
        for(int i = 0; i < SPECTRUMSIZE; i++)
        {
                if(spectrum.get(i) > 0.01f && spectrum.get(i) > max)
                {
                        max = spectrum.get(i);
                        bin = i;
                }
        }
        dominanthz = (float)bin * BINSIZE;      //Instantaneous room frequency
        rightNow = Calendar.getInstance().getTimeInMillis();
        if (dominanthz != 0.0) //Exclude startup detection instances
        {
                averageFreq += dominanthz; counter++; //averageFreq will be divided by counter
        }
```

The ambient frequency is the microphone no-sound threshold. That is, the frequency of the user's silent room as picked up by the microphone. The room's actual acoustic frequency is not, in reality, being measured (the microphone's internal resistance is), but we may think of this frequency as the "room's frequency" (when we go to ignore it). This code takes a bunch of samples of room frequency and divides them by the number of samples.

Natural Frequency

```
do
{
        /*==================================*/ //Natural frequency portion
        bin = 0; //pointer to location in array spectrum
        max = 0;
        result = channel.getSpectrum(spectrum, SPECTRUMSIZE, 0, FMOD_DSP_FFT_WINDOW_BLACKMAN);
        for(int i = 0; i < SPECTRUMSIZE; i++)
        {
                if(spectrum.get(i) > 0.01f && spectrum.get(i) > max)
                {
                        max = spectrum.get(i);
                        bin = i;
                }
        }
        dominanthz = (float)bin * BINSIZE;      //Instantaneous voice frequency
        rightNow = Calendar.getInstance().getTimeInMillis();
        if (dominanthz > ambientFrequency*1.07) //Don't factor in ambient frequency samples
        {
                averageFreq += dominanthz; //Natural frequency = averageFreq/counter
                counter++;
        }
```

Natural frequency is the frequency the speaker normally uses when discoursing. We sample his voice and take the average.

Instantaneous Frequency and Volume

```
/*==================================*/ //Detect instantaneous frequency

bin = 0; //Pointer to location in array spectrum

max = 0;

result = channel.getSpectrum(spectrum, SPECTRUMSIZE, 0, FMOD_DSP_FFT_WINDOW_BLACKMAN);

for(int i = 0; i < SPECTRUMSIZE; i++)

{if(spectrum.get(i) > 0.01f && spectrum.get(i) > max){

                max = spectrum.get(i);

                bin = i;

}}

if (dominanthz*3 < (float)bin * BINSIZE)

{

        dominanthz += 10;

}

else if ((float)bin * BINSIZE < dominanthz/3)

{

        dominanthz -= 10;

}

else

{dominanthz = (float)bin * BINSIZE;}    //Instantaneous voice frequency
```

To obtain the frequency of the user's voice, we tap into the channel's current waveform, represented in a float buffer we called **spectrum**. The variable *bin* holds a 'pointer' into the buffer used for obtaining the value we want.

The second part of the code is for smoothing out irregularities in the waveform. Sometimes, large leaps may be detected erroneously, so a simple rectifying scheme is applied.

Idle Percentage

```
if (dominanthz < ambientFrequency*1.07) //The user is idle

        {isIdle = true;}

        else                            //The user is not idle

        {isIdle = false;}



        if (rightNow - startIdleClock < 1000) //Analyze for a second

        {

                if (isIdle)

                {idleProbeCount++;}

        }

        else

        {

                idlePercentage = (float)idleProbeCount/142f;

                idleProbeCount=0;

                startIdleClock = rightNow;

        }
```

The idle percentage is the ratio defined as: the amount of time a user is idle in a segment divided by the length of the segment. The size of the segment, in this case, is determined by Δt = <current time> - <time a while ago>.

Analysis Execution

```
if (some relations between averageFreq, naturalFrequency, volume, idle percentage, etc.)
        {
                if (!isIdle) {Count++;currentState = "TheState";}
        }
        else if (averageFreq>naturalFrequency+600f) {Count++;} //Deviant S's
        else
        {
                if (Count >= Xf)        //Good case for being TheState
                {
                        Count -= 1f;
                }
                else if (previousAverageFreq*3f - averageFreq < -10f)  //Be leniant towards sudden utliers
                {
                        Count -= 5f;
                }
                else
                {
                        if (volume > volumeCenter)              //Punish any real outbursts
                                {Count -= 6f;}
                        Count--;
                }
        }
        if (Count < 0) Count = 0;
        if (Count >= Yf)
        {analysis = "The State"; Count = 0;}
```

Most of the emotions have a structure similar to the pseudocode above. It starts out with a basic assumption relating to some of the basic vocal dimensions. If, for instance, we are attempting to detect sadness, we would want *volume* to be a little lower than *volumeCenter*. If true, and the user is not idle (to give positive reinforcement), the state will be recognized.

There is also a condition for Deviant S's. These are 's' sounds that resonate at the end of plurals, etc. They cause the pitch detection algorithm to skyrocket (as much as 5Khz). One way to eliminate this is to assume if the pitch is very high, everything is "normal".

Otherwise, punish the counters of the state using caution to various degrees. If it is highly likely that the state, now being threatened by evidence of other characteristics, is a state that will persist further, punish it only a little bit. For example, decrement the state's counter. We must also be aware of sudden outliers in the data, such as **trailing harmonic frequencies** or normal emphases. Usually, if the outlier is slightly less than 3 times the frequencies immediately before it, the state will still be penalized, but only a little bit.

Lastly, we declare our analysis so that the output buffer may save this to disk.

Chapter 3

Testing

Evaluating the limitations of real-life test trials and the effectiveness of the software.

Part of what makes testing the software so difficult are the limitations of the user's environment and the human voice's inadequacy in exactly replicating itself. The following discusses some of the issues involved with testing the software.

Microphone Issues

Early on, the concept of an *ambient frequency* variable was established to, basically, deal with the internal resistances of microphones. Due to the variability in quality of microphones, some would have a tendency to "hiss", while some would have a tendency to "hum", and others "whine".

Choosing a high-quality microphone is a good idea to remove excess pollution of the voice stream. However, the ambient frequency variable adapts to the environment by using the microphone's "no-sound" frequency as the basis for silence.

General Testing Comments

- **Segmentation** – No particular work went into ensuring proper protection for all variables; get and set methods were unnecessary. The code eventually became very imperative, losing any advantage Java might provide. This was compounded by the manner in which no external methods were called; rather, all functionality was performed in loops as updates. This is the way most FMOD code examples illustrate 'proper form'.
- **Software Complexity** – In the first few iterations of the program, detecting emotions was easy (since there were only 3 or

4 emotions to choose from). After the implementation of a fifth emotion and forward, the chances for the software misinterpreting an emotion went up seemingly exponentially.

- **Market Competition** – After testing and evaluating "enterprise" emotion recognition programs, it is not clear that any one of them is better than this one.

- **Explicit Thinking** – When a human is thinking of something and wishes to announce his intention for expressing his idea verbally, he might mumble, "Uh…" But this noise confuses the software's pitch algorithm, resulting in unusually high (500+ Hz) frequencies for such a deep sound. Measures were taken to alleviate problems caused by *uhs*.

- **Hard Consonants** – The biggest issue with pitch detection resulted from words that have stressed hard consonants. Words that end in 's' were particularly notorious for bringing the average frequency up passed 4-5 KHz almost instantly.

Experiment Duplication

One way to solve the problem of random voice pollution and self-produced vocal inconsistencies when testing the software is to tap into the system's internal mixer. When testing the software, it is often difficult to determine whether the software code is incorrect, or if your, "Hello, testing, 1, 2, 3," was different from your last.

By using the **system mixer**, you are virtually guaranteed to get precisely the same output with which to formulate control groups. In this way, you can isolate code when performing tests instead of worrying about environmental dynamics.

However, it should be noted that just because the mixer outputs the same stream every time doesn't mean the input to FMOD's spectrum array is the same. The array's values are updated only every-so-often, and accessing it in rapid succession could lead to the same values being returned; depending on when you commence this succession, you might receive a different number of same-values.

Pitch, Volume, & Sample Accuracy

FMOD's sound recording interface allows for fairly accurate representations of pitch and volume. Once the programmer taps into the **spectrum** array, he can analyze and extract waveform data that corresponds to pitch and volume levels. While this method is faithful most of the time, there are some flaws.

1. *Harmonic frequencies* : Sometimes[1], the detected pitch is not accurately recorded, and FMOD reports a frequency that is *n* times above the desired frequency λ. That is, FMOD may report frequencies of $n\lambda$, where *n* can be as large as 3 or 4.

2. *Volume representation* : FMOD uses a float between 0 and 1 to specify the volume of a channel, where 0 is silence and 1 is full-volume. There is a drawback in how it determines full-volume, though, because it treats distorted material as full-volume. So, a user with a miscalibrated microphone could be speaking in a neutral voice but be at 1.0 volume for the duration of his speech.

3. *Leading/Trailing vibrations* : Due to the high sampling rate of the software, it is possible to have large leaps between consequent frequency detections. For example, a leading vibration occurs when the principal frequency of a sound (perhaps a word in speech) is preceded by a radically different micro-sound with a different frequency (such as a high-pitched "E" sound before saying, "You."). In speech, points 1 and 3 may collaborate to result in **leading/trailing *harmonic* vibrations**.

Corpora

Corpora included both downloaded material and in-house recordings.

To test the software, corpora were created for the various emotions the software attempts to detect. Other sound materials were downloaded from various websites on the internet using the sound search engine www.findsounds.com. For *concept testing*, speaking directly into the microphone was used. For *control testing*, the following corpora were used:

[1] For example, when using "MIDI piano-keyboard" programs, pressing the "A" key might lead FMOD to detect the frequency 1320 Hz, instead of 440 Hz.

OTHER CORPORA

- A pitch-declining, "Ewww!"
- A toddler's giggle/cooing
- Elmer Fudd

Anger

What do you mean fired? You can't fire me! I quit! Ever since the day I came in here you've been bothering me. Ugh.

Sadness

I don't know what's wrong with me. I just broke up with my girlfriend. What's the use? Everything about me is terrible.

Happiness

Oh, my God! Guess what? I got a raise at work! Now we'll be able to take that vacation to the Bahamas!

Neutrality

Hello, my name is John Smith, and I am speaking into this microphone in order to test the system.

Laughing

[laughter].

Note: The actual corpus lexicon is irrelevant. Its primary function is actor facilitation.

Boredom

Nah, we did that last week. I guess we could do it if you really wanted to, but... I really don't care to do that.

Defensiveness[1]

Hey, I never said that, so stop putting words in my mouth.

[1] Ultimately discarded.

Conclusion

At a certain point, as was proven, it becomes difficult to test the system because one's **'real' emotions suppress the ones trying to be tested**. For example, when testing the *Neutral* emotion, I found it frustrating when after a long period of constructive evidence for keeping the *Neutral* code, it would "fail" and say I wasn't being neutral. Frustrated, I couldn't figure out what was wrong until I realized: the further I got along in testing the *Neutral* code, and the more I saw that my outputs were telling me favorable things, I started to unconsciously get happy and change my voice, causing the code to "fail".

If for no other reason, this should be why the code has at least *a bit* of decorum.

Extensions and Applications

There exist many ways to extend this project, either to improve on it or to apply it.

One way to improve the program would be to allow user profiles to be saved to disk. The user would record his voice on a 'non-emotional' day, where it would be analyzed and saved as the "Neutral voice for <John> <Smith>." This way, he wouldn't have to *act naturally* as he calibrated the software upon initialization. There would still need to be some calibration, however, as the user might be further or closer to the microphone on different sessions (affecting the volume center, ambient frequency, etc.).

This software can be applied for use in the large, aforementioned[1] call centers, whose managers need to perform quick analyses of customer behavior responses. Because the manager or an employee does not need to take the time to actually listen to the conversation, they can use their spare time to listen to calls flagged by the software as "questionably *X*", heavily reducing their workload.

One could also use the statistics provided by the software to shape the face of a virtual character in real time. There would be no need for motion-capture reference points on an actor's face. The (voice) actor would simply speak his lines into the microphone while the modeling/animation program would use the stream data to articulate facial points on the dummy.

[1] Page 1.