COSC 4421.03/COSC5324.03 Introduction to Robotics

Using Sound Localization in Robot's navigation.

Project Report

February 6, 2001

By: **Andriy Pavlovych**
Student ID: **203468436**
cs account: **cs983373**

Abstract

Estimating the direction of arrival of sound in space is typically performed by a time-delay processing on a set of signals from an array of omnidirectional microphones (which requires specialized multichannel A/D hardware, and careful arrangement of the microphones into an array) or a set of two microphones with a controlled pose.

This work is motivated by the desire to use instead only one-channel processing on an input of only one microphone.

In this report, we describe the signal processing and control algorithm of a device with one directional microphone and one rotational degree of freedom.

Experimental results with real data are reported with both specially generated and real-life sounds in an untreated, normally reverberant indoor environment.

Table of content

1. Introduction

Sensors play an important part in robot's behaviour since they can provide it with various information about the environment. Most existing sensors that directly gather data about surrounding objects use electromagnetic radiation (radars, laser rangefinders – both are *active*; vision systems – usually *passive*), mechanical contact (contact sensors), or ultrasound (sonars). There are no known systems that would use information about the position of *existing* sound sources to judge about the environment (just like video camera "sees" the objects using only ambient light).

> Our system will use one directional microphone on top of one of the SuperScout robots to perform some simple task such as going towards the sound.

Some of the possible applications of such type of the sensor include:

• locating the direction from where the voice command came;

• locating the source of noise that requires attention;

• locating objects (that emit some sound) in case when other methods of sensing fail (e.g. malfunction, inappropriate lighting).

2. Sound Localization Problem

The problem of sound localization is closely related to acoustics.

> It is well known that humans derive the location of sound based on 3 major parameters:
>
> • interaural time difference, ITD (the ear closer to the source senses sound first – difference in phase)
>
> • interaural level difference, ILD (the ear closer to the source senses sound as being stronger in volume)
>
> • interaural spectral difference, ISD (the ear closer to the source senses sound as being "brighter" i.e. having stronger high-frequency components with lower being almost the same).
>
> One minor fact, often overlooked, is the shape of the outer ear. Although it's very individual and hard to model it provides humans with some ability to localize the sound in the third dimension (height).
>
> While not a sensing technique itself, humans ability to turn head (with ears on it) *helps* to locate sound by trying the orientations in which the precision is maximized.
>
> It should be noted that none of these methods is reliable on its own: ITD does not work well when sound wave period is shorter than the minimal measurable difference; ILD does not work well on low frequencies and on long distances; ISD will not with some simple spectrum signal (e.g. pure tones).

Based on these methods, several approaches are available (taken from [1]):

• Using several omnidirectional microphones (minimum 3 for 2D and 4 for 3D). This needs multichannel A-D equipment which is, while available now [3], is relatively expensive, and a specifically written software which increases the development costs. Similar to ITD. Current project of Bill Kapralos from York University.

• Using 2 microphones, simulate human ears directly using all three methods above. It is usually complex (for obvious reasons).

• Using active omnidirectional microphone pair. Done in [1]. Again, I think it will be unreliable at higher frequencies since it uses method similar to ITD. Reasonably easy.

• Using active directional microphone or active directional-fixed omnidirectional microphone pair. These are reported in [1] as being to sensitive to signal intensity inconsistencies. This is *in part* true. I suspect such inconsistency appeared because while the microphone is *supposed* to be directional it *is* directional enough only within certain range of frequencies and at frequencies 300 Hz and below it acts more like omnidirectional.

> My idea to use one directional microphone (active because it rotates with the robot) is similar to *both* of the last methods above. As a sound source I use a pair of sounds: 350 and 6000 Hz. As a result, one physical microphone acts like two logical. Another significant technical advantage of using one microphone is the fact that most sound cards have only one mono microphone input (the pin corresponding to R in the jack is used as +5V power) and using stereo line input would require two microphone amplifiers.

3. Implementation of the Project

The preliminary tests conducted at home before the actual work on this project began convinced me that my assumptions above are true (these tests could have potentially saved me from spending energy on a project which was not feasible). As a source of signal I used a PC sound output connected to my stereo system. As a source of ambient noise a TV operating, at average volume, was used. The microphone output was then captured by the microphone input of the sound card (using full-duplex mode of the sound controller). After isolating source signal frequencies (using 5-th order IIR filters implemented in the sound editing program, GoldWave) it was possible to observe high directionality at higher frequency and very low directionality at lower frequency. Distances tested: $0.5 - 3^1$ m; angles at which speaker faces the microphone: ±60°, angles at which the microphone faces the speaker: ±180°

The project itself consists of 3 subproblems to be addressed in order to complete it:

• capturing the sound from the microphone into the computer. Most computer system support this. New version of Java from Sun includes built-in classes to capture sound into the program.

• processing the sound: filtering, normalization, intensity analysis.

• making the decision based on the results from step 2 and giving the commands to the robot

It was decided to do all the programming in Java. Despite its "weaknesses" and "low performance" the biggest useful advantage was (and is) its portability. Different parts of the project were developed, tested and refined under different operating systems: MS Windows 98, Windows NT 4.0, Solaris OS. No modifications of the code were ever required due to platform differences.

•• For capturing and processing of the sound, a separate class, called `audioSensor`, was created. In addition to the methods that actually do what is required for this stage, some other methods were added to this class. These methods allow to monitor the behaviour of the class through providing the facilities to load a sound file into the program (instead of capturing it from the microphone), and to save it to a disk to be analysed by an outside sound processing software. These extra methods are not

---

required in the end product, and can be easily removed from the class which was done when preparing this document in order to save space. Complete code exists on-line.

• To control the robot according to the data from the new sensor, the `run()` method of the class `SuperScout` was modified. In fact, all Java classes from the course assignment were directly re-used in this project.

3.1 Capturing the Sound

A package `javax.sound.sampled.*` was used. Many functions were adapted from Java Sound demo contained in the distribution of the Java 2 SDK, Standard Edition 1.3. The task of capturing the sound is implemented in function `public void readSnd()` of the class `audioSensor`. Complete code is in the Appendix. Here are just highlights with brief explanation.

The tasks consists of four stages:

• open a line

```
// define the required attributes for our line,
DataLine.Info info = new DataLine.Info(TargetDataLine.class,format);

// get and open the target data line for capture.
    line = (TargetDataLine) AudioSystem.getLine(info);
    line.open(format, line.getBufferSize());
```

• read the required amount (LENGTH) of sound data into a byte array

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
int frameSizeInBytes = format.getFrameSize();
int bufferLengthInFrames = line.getBufferSize() / 8;
int bufferLengthInBytes = bufferLengthInFrames * frameSizeInBytes;
byte[] data = new byte[bufferLengthInBytes];

int numBytesRead;
int totalBytes = 0;

line.start();
while (totalBytes <= LENGTH) {
    if((numBytesRead = line.read(data, 0, bufferLengthInBytes))
        == -1) {break;}
    out.write(data, 0, numBytesRead);
    totalBytes+= numBytesRead;
}
```

• close the line

```
// we reached the end of the stream.  stop and close the line.
line.stop();
line.close();
line = null;

// stop and close the output stream
    out.flush();
    out.close();
```

• save read data into an array of bytes for later processing

```
audioBytes = out.toByteArray();
```

3.2 Processing the Sound

The objective of the processing stage is to be able to compute average powers of the two components of the signal from the microphone on top of the robot: 350 Hz and 6000 Hz. According to the preliminary tests and the idea in Section 2, based on the relation between these two components it will be possible to deduce the orientation of the directional microphone relative to the source of sound and, therefore, to solve our problem of locating the sound source

3.2.1 Conversion to integers

Audio output stream has allowed us to save captured sound as an array of bytes. Before the sound can be processed it must be converted to an array of (signed) integers. In our class it is done through a method `public int[] byteToInt(byte[] BYTE)`. It takes into account the fact that different sound file formats when viewed as a byte stream may have their bytes swapped (low order - high order). Another issue: when *processing* the sound, it is usually a good idea to use higher resolution (e.g. to avoid errors due to overflowing); in our case 32 bits are used instead of 16 – 32 is a standard size in Java for an int type.

3.2.2 Filtering

There are three basic types of digital filters. Each of them has its own advantages and disadvantages. In this project at this stage any type of the filter would be adequate. However, one must look at possible future applications of the code and try to choose the type that will be right for most of the applications.

• **FFT, Fast Furrier Transform.** Conceptually simple. But have many disadvantages: tend to be slow, require significant amount of processing power, have poor performance in audio applications – distort the phase information almost completely (at window sizes large enough to be useful). They are useful when it is needed to observe a complete spectrum of the signal.

• **FIR, Finite Impulse Response.** Tend to be the slowest. Easy to implement. Deliver the best quality. Do not shift the phase. Considered the best in digital audio processing. FIR is a special case of IIR – with no feedback.

Finite impulse response filters get their name from the following property: an impulse signal (e.g. the sample sequence {1,0,0,0,...}) fed into a FIR filter results in an output which decays to zero after a finite number of iterations.

The equation below describes how the output y of a finite impulse response filter is calculated from the input x. This equation simply says that the nth output is a weighted average of the most recent P inputs. If x is real-valued, then all the other variables can be real-valued as well without loss of generality. Otherwise, they should all be complex-valued.

$$y_n = \sum_{i=0}^{P-1} c_i x_{n-i}$$

When the subscript *n-i* is negative, we consider that input to be zero. The *c* array holds *P* arbitrary constants. These values "tune" the FIR filter to provide the desired frequency response.

• **IIR, Infinite Impulse Response.** The most widely used. The fastest and reasonably good. Introduce slight phase shifts to frequencies close to those that are altered. Can be obtained from FIR filter by allowing the portion of already filtered signal to be used when filtering current signal. IIR filters are the closest in their behaviour to analogue LC, RC filters. Unfortunately, they require computation in complex numbers (which tends to slow things down), and just like analogue filters, or any circuits with feedback, can suffer from stability problems if improperly designed (exactly what happened during the developing of the code).

$$ y_n = \sum_{i=0}^{P-1} c_i x_{n-i} + \sum_{j=1}^{Q} d_j y_{n-j} $$

The *d* array holds weighting coefficients for feeding back the previous *Q* outputs into the current output value. Note that an FIR filter is really just an IIR filter with *Q*=0. An additional complication with IIR filters is that, unlike FIR filters, they require complex-valued weighting coefficients even when you want only the real parts of the outputs. Both the real and imaginary components of outputs are fed back into the filter, and both affect the real components of future outputs

Some facts in the description of FIR and IIR filters are taken from [4].

Based on the properties of the filters and on the experience of using them, FIR type was chosen to be implemented in this project. If the time-delay estimation principle is ever added to this work it will be very useful to not have phase distortions which is achieved by this type of filters.

It is very hard to compute the coefficients for any filter based on required properties, therefore only filtering algorithm was implemented, and the array of the coefficients was computed by another program, [5], and plugged into the code.

We need two filters: one – to filter out the lower frequency, and one – higher. There is a separate code for each. Although the difference is only in an array of coefficients, making the codes separate allows to optimize each filter for its own domain.

Finally, the fragment of the code of one of the filters, 6000Hz:

```
/////////////Filter BP 6000 Hz//////////////
   public int[] filter6000 (int[] x){
/////////////////////////////////////////////
/*
Parks-McClellan FIR Filter Design
Filter type: Band pass
Passband: 0.133786848072562358276643990929705 - 0.138321995464852607709750566893424
Order: 119
Passband ripple: 1.0 dB
```

```
Transition band: 0.02
Stopband attenuation: 70.0 dB
*/
final int K= 119;
double[] a= new double[K+1];

a[0] =    1.92576721716378E-4      ;
a[1] =    4.816437665091255E-4     ;
.....
a[118] = 4.816437665091255E-4   ;
a[119] = 1.92576721716378E-4    ;

    int noSamples= x.length;
    int y[]= new int[noSamples];

// output will start with K zeros
    for (int i=0; i<K; i++){
        y[i]=0;
    }

    for (int i=K; i<noSamples; i++){
        double currentValue=0.0;

        for (int j= 0; j<=K; j++)
                currentValue+= x[i-j]*a[j];
        y[i]= (int)currentValue;
    }
    return y;
}
```

3.2.3 Measuring the strength of the filtered components.

To measure the average power of the filtered components, a simple method `avgPow()` was written which returns the RMS value of the input samples. A straightforward code.

```
//////////////////////////////////////////
    public double avgPow(int[] audioData){
//////////////////////////////////////////
        long sum=0;
        for(int i=0; i<audioData.length; i++){
            sum+=audioData[i]*audioData[i];
        }
        return Math.sqrt(sum/(double)audioData.length);
    }
```

3.3 Generating Commands for the Robot

After the sound is processed, commanding the robot is easy. As it was mentioned before, a procedure run() of the class SuperScout was modified; it is the method that runs from the beginning to the end of the motion of the robot.

The algorithm of control is this:

"1. Rotate the robot 360º in 45º increments. Find the position where the ratio of the 6000 Hz component to the 350 Hz is the highest and rotate the robot to it. Move forward by a fixed step.

2. From now on, repeat: adjust the robot towards the source of the sound using a Hill-climbing algorithm and move forward by a fixed step.

   Stop moving when the intensity of the sound becomes greater than a pre-defined threshold."

It should be noted that the robot this project was tested on had an array of sonars which could measure the distance to the obstacles. They could obviously be integrated with the new sensor in the project. But the goal here was to prove that the new sensor is powerful enough to work on its own. Several tests were made with sonars turned on (but ignored) – it was observed that they don't affect the performance of the new sensor. It is also reasonable to assume that the sonars' sensitivity will not be affected by the presence of the sounds present during testing since these sensors rely on different sources of sound: strong, short pulse for one and a continuous low-power – for another.

A question might appear, why don't we just use two (or three) appropriate microphone poses and then calculate the orientation based on the directionality pattern of the microphone (which can be measured via a calibration procedure, once; a picture below showing it was taken from [1])?
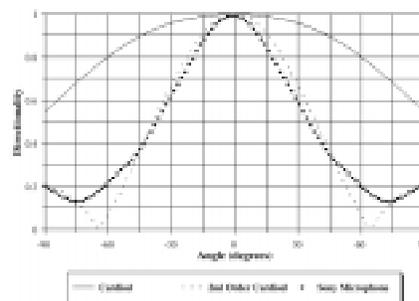
One major problem with this approach is that sounds with different frequencies travel differently in the environment. Higher freqencies are absorbed much faster. (One has no difficulty hearing a subwoofer in a passing car 300 meters (1000 ft.) away, yet it is difficult to hear the radio even 10 m (33 ft.) away in another room). Therefore, we would have at least to take the distance to the source into account, and we have no facilities for this in our setup.

Another problem, also related to distance, is a decrease in Signal to Noise Ratio with the increase of the distance which can dramatically reduce the accuracy of this alternative approach. Increase of the distance 10 times decreases the S/N ratio 100 times or by 20 dB (with the assumption that the sound is *not* absorbed by the environment). Most current 16 bit sound controllers have SNR close to 70 dB for line input and less than that for microphone input – not much room here. Environment noise is a factor here, too.

   The original approach implemented, although not elegant, but has proved to be reliable.

4. Experimental Results

Equipment Setup

A directional microphone was mounted on top of the mobile robot. An audio cable, about 6 m (20 ft.) long, was connecting it to the Sun workstation's microphone input. The program was running on that station. Commands to the robot were sent through a wireless Ethernet link. As a source of a sound, a personal CD player connected to a small computer speaker was used; the player and the speaker were put on the movable platform to allow for easy relocation within the room. Room was a typical lab room; no special preparations were made (except for clearing the floor from junk).

Results

Before starting the robot, the OS's environment variables were set:

```
setenv JMFHOME /cs/local/packages/JMF2.1
setenv CLASSPATH $JMFHOME/lib/jmf.jar:$JMFHOME/lib/sound.jar
setenv LD_LIBRARY_PATH $JMFHOME/lib
```

There were two sets of tests: one – with a recorded test signal consisting of two sine signals, 350 and 6000 Hz equally mixed to –3 dB DigitalFullScale, another with random selections of music.

A microphone input's sesitivity was adjusted so the robot stops within 0.3 m from the sound source

After that, the robot was behaving as expected.

The only problems noted were when the *spectral content* of the music was changing between taking the measurements from the microphone – sudden pauses, special effects; changes of the volume had no noticeable effect. The least reliable results were obtained with the kinds of music with very low high-frequency content (e.g. "Beethoven - Sym 5 Piano Concerto"). Here are some spectral diagrams corresponding to different signals with short comments (horizontal scale is 11025 Hz):
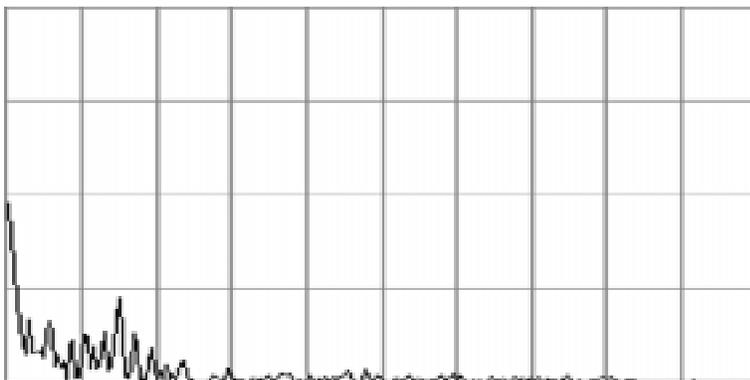
Ambient noise in the room. Most noise is concentrated below 100 Hz
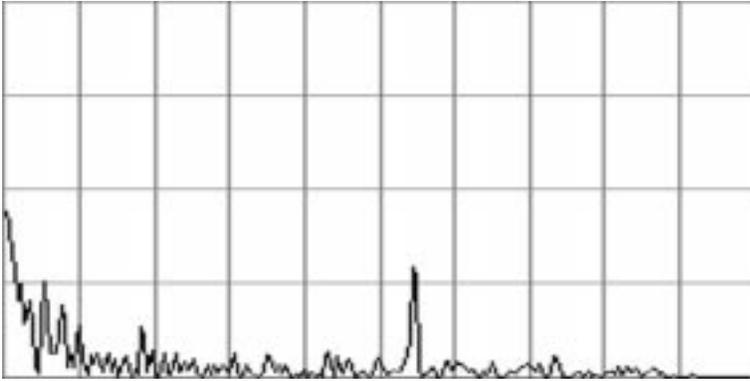
Specially generated test signal. 6000 Hz component is clearly visible.
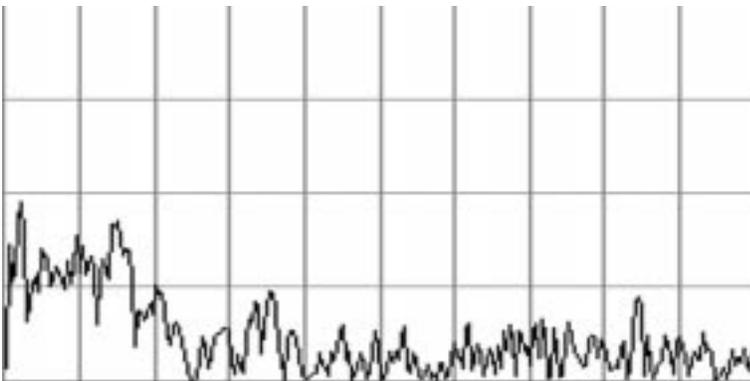
Popular music fragment. Rich in high frequencies. Works fairly well in this project
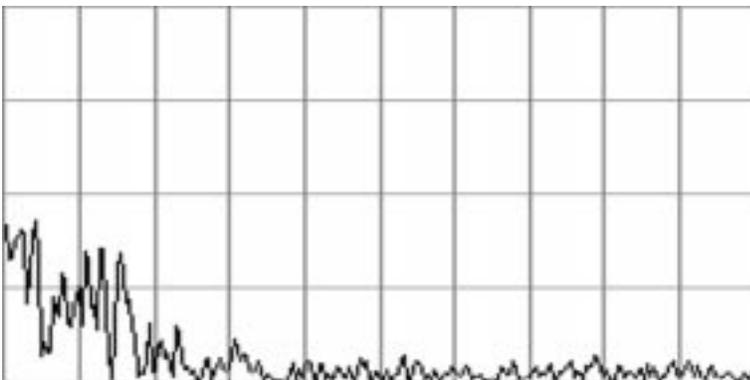
Classical music fragment. Very little of higher frequencies. Doesn't work well.

5. Discussion

I have presented the technique of estimating the direction of arrival of the sound in two dimensional space using only a directional microphone and a widely available equipment. This approach is similar to interaural spectral difference, ISD, a method used by humans to judge about the location of the sound. It was shown that the technique can be used together with other techniques which can make sensing in a robot more robust.

A natural extension of this work is using the implemented ideas to model humans' (or animals') hearing – definitely a complex task, but being perfected through many years of evolution, it is considered the most reliable sensor in the real world.

Appendix 1. Electret condenser directional microphone

Manufacturer : Sony
Model :   ECM-PB1C
Weight :   Approx.115 g
Dimensions :   174 x 160 x 70 mm (width/height/depth)
Cost :   Approx. US$80
Description :   The microphone used in this project consisted of an omnidirectional microphone supported at the focal point of a small parabolic reflector. This product is sold commercially as an adapter for personal video camera recorders adding directional sensitivity and gain to the audio recording.

Appendix 2. Source code of the software developed for this project.

Online version is at: http://www.ariel.cs.yorku.ca/~cs983373/4421/99f.zip

Note: the code for audiosensor.java is edited. The functions that are not used in this project and were used only at the developing stage are removed for simplicity and to save space.

```java
/**
 * audioSensor.java
 *
 * Description: A class that allows to capture the sound into a byte array and
 *              does basic signal processing  (filtering, measuring)
 * @author    Andriy Pavlovych, cs983373@ariel.cs.yorku.ca
 * @version    0.1
 */

import java.util.Enumeration;
import java.util.Random;
import java.io.*;
import javax.sound.sampled.*;
import java.awt.font.*;
import java.text.*;

public class audioSensor {

    final AudioFormat format= new AudioFormat(AudioFormat.Encoding.PCM_SIGNED,
                                              44100, 16, 1, 2, 44100, true);
    byte [] audioBytes = null;

    public audioSensor() { }

///////////////////////////////////////////////
    public void readSnd() {
///////////////////////////////////////////////
            final int LENGTH=22050;
            TargetDataLine line;
            duration = 0;
            audioInputStream = null;

            // define the required attributes for our line,
            // and make sure a compatible line is supported.

            DataLine.Info info = new DataLine.Info(TargetDataLine.class,
                format);

            if (!AudioSystem.isLineSupported(info)) {
                System.out.println("Line matching " + info + " not supported.");
                return;
            }

            // get and open the target data line for capture.

            try {
                line = (TargetDataLine) AudioSystem.getLine(info);
                line.open(format, line.getBufferSize());
```

```java
            } catch (LineUnavailableException ex) {
                System.out.println("Unable to open the line - CAPT: " + ex);
                return;
            } catch (SecurityException ex) {
                System.out.println(ex.toString());
                return;
            } catch (Exception ex) {
                System.out.println(ex.toString());
                return;
            }

            // play back the captured audio data
            ByteArrayOutputStream out = new ByteArrayOutputStream();
            int frameSizeInBytes = format.getFrameSize();
            int bufferLengthInFrames = line.getBufferSize() / 8;
            int bufferLengthInBytes = bufferLengthInFrames * frameSizeInBytes;
            byte[] data = new byte[bufferLengthInBytes];
            int numBytesRead;

            line.start();

            int totalBytes = 0;
            while (totalBytes <= LENGTH) {
                if((numBytesRead = line.read(data, 0, bufferLengthInBytes)) == -
1) {
                    break;
                }
                out.write(data, 0, numBytesRead);
                totalBytes+= numBytesRead;
            }

            // we reached the end of the stream.  stop and close the line.
            line.stop();
            line.close();
            line = null;

            // stop and close the output stream
            try {
                out.flush();
                out.close();
            } catch (IOException ex) {
                ex.printStackTrace();
            }

            // load bytes into the audio input stream for playback

            audioBytes = out.toByteArray();
        }

//////////////////////////////////////////////////
    public byte[] intToByte(int[] INT) {
//////////////////////////////////////////////////
            byte[] BYTE=null;
            int lengthInSamples = INT.length;
             if (format.getSampleSizeInBits() == 16) {
                 int lengthInBytes = lengthInSamples * 2 ;
```

```java
                BYTE = new byte[lengthInBytes];
                if (format.isBigEndian()) {
                    for (int i = 0; i < lengthInSamples; i++) {
                        /* First byte is MSB (high order) */
                        BYTE[2*i]= (byte)(INT[i]>>8);
                        /* Second byte is LSB (low order) */
                        BYTE[2*i+1]= (byte)(INT[i] & 255);
                    }
                } else {
                    for (int i = 0; i < lengthInSamples; i++) {
                        /* First byte is LSB (low order) */
                        BYTE[2*i]= (byte)(INT[i] & 255);
                        /* Second byte is MSB (high order) */
                        BYTE[2*i+1]= (byte)(INT[i]>>8);
                    }
                }
            } else if (format.getSampleSizeInBits() == 8) {
                BYTE = new byte[lengthInSamples];
                if (format.getEncoding().toString().startsWith("PCM_SIGN")) {
                    for (int i = 0; i < lengthInSamples; i++) {
                        BYTE[i] = (byte)INT[i];
                    }
                } else {
                    for (int i = 0; i < lengthInSamples; i++) {
                        BYTE[i] = (byte)(INT[i]+128);
                    }
                }
            }
        return BYTE;
    }


//////////////////////////////////////////////////
    public int[] byteToInt(byte[] BYTE) {
//////////////////////////////////////////////////
            int[] INT=null;
            if (format.getSampleSizeInBits() == 16) {
                int nlengthInSamples = BYTE.length / 2;
                INT = new int[nlengthInSamples];
                if (format.isBigEndian()) {
                    for (int i = 0; i < nlengthInSamples; i++) {
                        /* First byte is MSB (high order) */
                        int MSB = (int) BYTE[2*i];
                        /* Second byte is LSB (low order) */
                        int LSB = (int) BYTE[2*i+1];
                        INT[i] = MSB << 8 | (255 & LSB);
                    }
                } else {
                    for (int i = 0; i < nlengthInSamples; i++) {
                        /* First byte is LSB (low order) */
                        int LSB = (int) BYTE[2*i];
                        /* Second byte is MSB (high order) */
                        int MSB = (int) BYTE[2*i+1];
                        INT[i] = MSB << 8 | (255 & LSB);
                    }
                }
```

```
                    } else if (format.getSampleSizeInBits() == 8) {
                        int nlengthInSamples = BYTE.length;
                        INT = new int[nlengthInSamples];
                        if (format.getEncoding().toString().startsWith("PCM_SIGN")) {
                            for (int i = 0; i < BYTE.length; i++) {
                                INT[i] = BYTE[i];
                            }
                        } else {
                            for (int i = 0; i < BYTE.length; i++) {
                                INT[i] = BYTE[i] - 128;
                            }
                        }
                    }
                }
            return INT;
        }


//////////////Filter BP 300 Hz//////////////
   public int[] filter300 (int[] x){
////////////////////////////////////////////
/*
Parks-McClellan FIR Filter Design

Filter type: Low pass
Passband: 0 - 0.0015
Order: 119
Passband ripple: 1.0 dB
Transition band: 0.02
Stopband attenuation: 70.0 dB
*/
final int K= 119;
double[] a= new double[K+1];

a[0] =    3.8629782245587914E-4    ;
a[1] =    2.0925414545260878E-4    ;
a[2] =    2.6478066895547255E-4    ;
a[3] =    3.286575382600167E-4     ;
a[4] =    4.0172984848725027E-4    ;
a[5] =    4.8454901825600204E-4    ;
a[6] =    5.78299016287318E-4      ;
a[7] =    6.83255663357234E-4      ;
a[8] =    8.001355794639199E-4     ;
a[9] =    9.293026927515482E-4     ;
a[10] =   0.0010724351806418108    ;
a[11] =   0.0012285556566238927    ;
a[12] =   0.0013990867591264547    ;
a[13] =   0.0015845655687491585    ;
a[14] =   0.0017847895115405707    ;
a[15] =   0.002000714752473591     ;
a[16] =   0.0022322179517671853    ;
a[17] =   0.0024798074901276486    ;
a[18] =   0.002743248274113184     ;
a[19] =   0.003022738166926772     ;
a[20] =   0.0033181996968626826    ;
a[21] =   0.0036293831556151613    ;
a[22] =   0.003955909689013857     ;
```

```
a[23]  =    0.004297762313852435      ;
a[24]  =    0.004654004836156562      ;
a[25]  =    0.005024421428369434      ;
a[26]  =    0.005408037579321557      ;
a[27]  =    0.005804242479426323      ;
a[28]  =    0.006211921611664645      ;
a[29]  =    0.006630150554864636      ;
a[30]  =    0.007057777909653664      ;
a[31]  =    0.007493599486893597      ;
a[32]  =    0.007936205550931809      ;
a[33]  =    0.00838440610214323       ;
a[34]  =    0.008836516213337248      ;
a[35]  =    0.009291132053827764      ;
a[36]  =    0.009746554732354373      ;
a[37]  =    0.010201142457775671      ;
a[38]  =    0.010653144548075066      ;
a[39]  =    0.011100763206870685      ;
a[40]  =    0.011542267450974707      ;
a[41]  =    0.011975804217193065      ;
a[42]  =    0.012399556762954185      ;
a[43]  =    0.012811789433098063      ;
a[44]  =    0.013210673291895495      ;
a[45]  =    0.013594469430059717      ;
a[46]  =    0.013961495877361545      ;
a[47]  =    0.014310045579649598      ;
a[48]  =    0.014638581430104245      ;
a[49]  =    0.014945492655923764      ;
a[50]  =    0.015229429653249676      ;
a[51]  =    0.015488992979749854      ;
a[52]  =    0.015722955872094754      ;
a[53]  =    0.015930222392067817      ;
a[54]  =    0.016109783210010446      ;
a[55]  =    0.01626076313172103       ;
a[56]  =    0.016382455543789677      ;
a[57]  =    0.01647423277119074       ;
a[58]  =    0.0165356944354582        ;
a[59]  =    0.01656647928188185       ;
a[60]  =    0.01656647928188185       ;
a[61]  =    0.0165356944354582        ;
a[62]  =    0.01647423277119074       ;
a[63]  =    0.016382455543789677      ;
a[64]  =    0.01626076313172103       ;
a[65]  =    0.016109783210010446      ;
a[66]  =    0.015930222392067817      ;
a[67]  =    0.015722955872094754      ;
a[68]  =    0.015488992979749854      ;
a[69]  =    0.015229429653249676      ;
a[70]  =    0.014945492655923764      ;
a[71]  =    0.014638581430104245      ;
a[72]  =    0.014310045579649598      ;
a[73]  =    0.013961495877361545      ;
a[74]  =    0.013594469430059717      ;
a[75]  =    0.013210673291895495      ;
a[76]  =    0.012811789433098063      ;
a[77]  =    0.012399556762954185      ;
a[78]  =    0.011975804217193065      ;
```

```
a[79]  =   0.011542267450974707    ;
a[80]  =   0.011100763206870685    ;
a[81]  =   0.010653144548075066    ;
a[82]  =   0.010201142457775671    ;
a[83]  =   0.009746554732354373    ;
a[84]  =   0.009291132053827764    ;
a[85]  =   0.008836516213337248    ;
a[86]  =   0.00838440610214323     ;
a[87]  =   0.007936205550931809    ;
a[88]  =   0.007493599486893597    ;
a[89]  =   0.007057777909653664    ;
a[90]  =   0.0066301505548646365   ;
a[91]  =   0.006211921611664645    ;
a[92]  =   0.005804242479426323 5   ;
a[93]  =   0.005408037579321557    ;
a[94]  =   0.005024421428369434    ;
a[95]  =   0.004654004836156562    ;
a[96]  =   0.004297762313852435    ;
a[97]  =   0.003955909689013857    ;
a[98]  =   0.0036293831556151613   ;
a[99]  =   0.0033181996968626826   ;
a[100] =   0.003022738166926772    ;
a[101] =   0.002743248274113184    ;
a[102] =   0.0024798074901276486   ;
a[103] =   0.0022322179517671853   ;
a[104] =   0.002000714752473591    ;
a[105] =   0.0017847895115405707   ;
a[106] =   0.0015845655687491585   ;
a[107] =   0.0013990867591264547   ;
a[108] =   0.0012285556566238927   ;
a[109] =   0.0010724351806418108   ;
a[110] =   9.293026927515482E-4    ;
a[111] =   8.001355794639199E-4    ;
a[112] =   6.83255663357234E-4     ;
a[113] =   5.78299016287318E-4     ;
a[114] =   4.8454901825600204E-4   ;
a[115] =   4.0172984848725027E-4   ;
a[116] =   3.286575382600167E-4    ;
a[117] =   2.6478066895547255E-4   ;
a[118] =   2.0925414545260878E-4   ;
a[119] =   3.8629782245587914E-4   ;

    int noSamples= x.length;
    int y[]= new int[noSamples];

    for (int i=0; i<K; i++){
        y[i]=0;
    }

    for (int i=K; i<noSamples; i++){
        double currentValue=0.0;

        for (int j= 0; j<=K; j++)
                currentValue+= x[i-j]*a[j];
        y[i]= (int)currentValue;
    }
```

```java
    return y;
}




///////////////Filter BP 6000 Hz///////////////
   public int[] filter6000 (int[] x){
/////////////////////////////////////////////
/*
Parks-McClellan FIR Filter Design

Filter type: Band pass
Passband: 0.13378684807256235827664399 0929705 -
0.13832199546485260770975056689 3424
Order: 119
Passband ripple: 1.0 dB
Transition band: 0.02
Stopband attenuation: 70.0 dB
*/
final int K= 119;
double[] a= new double[K+1];

a[0]  =    1.92576721716378E-4       ;
a[1]  =    4.816437665091255E-4      ;
a[2]  =    1.278514094546578E-4      ;
a[3]  =   -1.9568058401945885E-4     ;
a[4]  =   -6.848455079831354E-4      ;
a[5]  =   -7.293647925285123E-4      ;
a[6]  =   -1.794063642330517E-4      ;
a[7]  =    7.842737954803895E-4      ;
a[8]  =    0.0014646246437380738     ;
a[9]  =    0.0011709288402034256     ;
a[10] =   -2.0928937914061613E-4     ;
a[11] =   -0.0018892643759880518     ;
a[12] =   -0.0025692725972561777     ;
a[13] =   -0.0013757787070776011     ;
a[14] =    0.0012731078233119814     ;
a[15] =    0.0036445573047389217     ;
a[16] =    0.0037592597501957237     ;
a[17] =    9.452911393402906E-4      ;
a[18] =   -0.0032892762408816465     ;
a[19] =   -0.005946934660911436      ;
a[20] =   -0.004594874228653746      ;
a[21] =    5.782762839020691E-4      ;
a[22] =    0.006357956534349563      ;
a[23] =    0.008415792428016593      ;
a[24] =    0.004472574892789296      ;
a[25] =   -0.003551528625213956      ;
a[26] =   -0.010272904882156242      ;
a[27] =   -0.010393625804356677      ;
a[28] =   -0.0027732060122493745     ;
a[29] =    0.0080510554786143        ;
a[30] =    0.014456246432733515      ;
a[31] =    0.011056474906864455      ;
a[32] =   -9.3193467950603E-4        ;
a[33] =   -0.013733379675750388      ;
```

```
a[34] =   -0.018007287201484225   ;
a[35] =   -0.009623681218273904   ;
a[36] =    0.006670065600296276   ;
a[37] =    0.019798082339030583   ;
a[38] =    0.019878014501101486   ;
a[39] =    0.005616509944373052   ;
a[40] =   -0.013937213136008422   ;
a[41] =   -0.025098543510996097   ;
a[42] =   -0.019149213266181183   ;
a[43] =    9.197950605165496E-4   ;
a[44] =    0.021709265070431215   ;
a[45] =    0.02838923693716875    ;
a[46] =    0.015321904718742672   ;
a[47] =   -0.009313797425248887   ;
a[48] =   -0.028621503081541672   ;
a[49] =   -0.028653010623286245   ;
a[50] =   -0.008533310726157147   ;
a[51] =    0.018341489596146916   ;
a[52] =    0.033282614618204996   ;
a[53] =    0.02540883774904061    ;
a[54] =   -3.890115719992118E-4   ;
a[55] =   -0.026476322967869764   ;
a[56] =   -0.034634984978037194   ;
a[57] =   -0.018896322165246695   ;
a[58] =    0.01006981489219454    ;
a[59] =    0.03225247458646928    ;
a[60] =    0.03225247458646928    ;
a[61] =    0.01006981489219454    ;
a[62] =   -0.018896322165246695   ;
a[63] =   -0.034634984978037194   ;
a[64] =   -0.026476322967869764   ;
a[65] =   -3.890115719992118E-4   ;
a[66] =    0.02540883774904061    ;
a[67] =    0.033282614618204996   ;
a[68] =    0.018341489596146916   ;
a[69] =   -0.008533310726157147   ;
a[70] =   -0.028653010623286245   ;
a[71] =   -0.028621503081541672   ;
a[72] =   -0.009313797425248887   ;
a[73] =    0.015321904718742672   ;
a[74] =    0.02838923693716875    ;
a[75] =    0.021709265070431215   ;
a[76] =    9.197950605165496E-4   ;
a[77] =   -0.019149213266181183   ;
a[78] =   -0.025098543510996097   ;
a[79] =   -0.013937213136008422   ;
a[80] =    0.005616509944373052   ;
a[81] =    0.019878014501101486   ;
a[82] =    0.019798082339030583   ;
a[83] =    0.006670065600296276   ;
a[84] =   -0.009623681218273904   ;
a[85] =   -0.018007287201484225   ;
a[86] =   -0.013733379675750388   ;
a[87] =   -9.3193467950603E-4     ;
a[88] =    0.011056474906864455   ;
a[89] =    0.014456246432733515   ;
```

```
a[90]  =   0.0080510554786143       ;
a[91]  =  -0.0027732060122493745    ;
a[92]  =  -0.010393625804356677     ;
a[93]  =  -0.010272904882156242     ;
a[94]  =  -0.003551528625213956     ;
a[95]  =   0.004472574892789296     ;
a[96]  =   0.008415792428016593     ;
a[97]  =   0.006357956534349563     ;
a[98]  =   5.782762839020691E-4     ;
a[99]  =  -0.004594874228653746     ;
a[100] =  -0.005946934660911436     ;
a[101] =  -0.0032892762408816465    ;
a[102] =  9.452911393402906E-4      ;
a[103] =  0.0037592597501957237     ;
a[104] =  0.0036445573047389217     ;
a[105] =  0.0012731078233119814     ;
a[106] =  -0.0013757787070776011    ;
a[107] =  -0.0025692725972561777    ;
a[108] =  -0.0018892643759880518    ;
a[109] =  -2.0928937914061613E-4    ;
a[110] =  0.0011709288402034256     ;
a[111] =  0.0014646246437380738     ;
a[112] =  7.842737954803895E-4      ;
a[113] =  -1.794063642330517E-4     ;
a[114] =  -7.293647925285123E-4     ;
a[115] =  -6.848455079831354E-4     ;
a[116] =  -1.9568058401945885E-4    ;
a[117] =  1.278514094546578E-4      ;
a[118] =  4.816437665091255E-4      ;
a[119] =  1.92576721716378E-4       ;

    int noSamples= x.length;
    int y[]= new int[noSamples];

    for (int i=0; i<K; i++){
        y[i]=0;
    }

    for (int i=K; i<noSamples; i++){
        double currentValue=0.0;

        for (int j= 0; j<=K; j++)
                currentValue+= x[i-j]*a[j];
        y[i]= (int)currentValue;
    }
    return y;
}



/////////////////////////////////////
   public double avgAmpl(int[] audioData){
/////////////////////////////////////
     long sum=0;
     for(int i=0; i<audioData.length; i++){
        sum+=audioData[i];
```

```
        }
        return sum/(double)audioData.length;
    }


/////////////////////////////////////
    public double avgPow(int[] audioData){
/////////////////////////////////////
        long sum=0;
        for(int i=0; i<audioData.length; i++){
            sum+=audioData[i]*audioData[i];
        }
        return Math.sqrt(sum/(double)audioData.length);
    }


}
```

Method run() of the class SuperScout

```
  public void run() {

    double[] intensity = new double[8];
    audioSensor aSensor= new audioSensor();
    int [] audioData;
    double mxInt= 0;
    int mxIntI= 0;

    int rotSign=1;
    int tshd= 1000; //max avg power reaching which should stop (too close to source)
    int delay= 2000; //time in ms needed to execute a command "move" or "rotate"
    double iA;
    double i300;
    double i6000;
    double iRatio;
    double oldR;
    boolean peakPassed= false;

    System.out.println("Starting super scout");


    // Part 1. Initialization
    //1.1 sense the sound around the robot
    for(int n=1; n<=7; n++){
      aSensor.readSnd();
      audioData = aSensor.byteToInt(aSensor.audioBytes);
      intensity[n]= aSensor.avgPow(aSensor.filter6000(audioData))/
              aSensor.avgPow(aSensor.filter300(audioData));
      rotate(45);
      try {
          sleep(delay);
      } catch(InterruptedException e) {}
    }
```

```
//1.2 Find the orientation with the highest intensity
mxInt= intensity[0];
mxIntI= 0;
  for(int i=1; i<8; i++){
    if (intensity[i]>mxInt){
      mxInt= intensity[i];
          mxIntI=i;
    }
  }


//1.3 rotate the robot to it
rotate(((mxIntI+4)%8-4)*45);
  try {
      sleep(delay);
  } catch(InterruptedException e) {}



//1.4 move-wait
moveForward(0.10f);
  try {
      sleep(delay);
  } catch(InterruptedException e) {}


 // Part 2. Move until commanded to stop
for(;;){

 //2.1 read current data
  aSensor.readSnd();
  audioData = aSensor.byteToInt(aSensor.audioBytes);

  iA= aSensor.avgPow(audioData);
  i300= aSensor.avgPow(aSensor.filter300(audioData));
      i6000= aSensor.avgPow(aSensor.filter6000(audioData));
      iRatio= i6000/i300;

  System.out.println(aSensor.audioBytes.length);
  System.out.println(iA);
  System.out.println(iRatio);
  System.out.println();

  if(iA<tshd){

 //2.2 determine in which direction to rotate
 oldR= iRatio;

    rotate(rotSign*20);
    try {
        sleep(delay);
    } catch(InterruptedException e) {}

    aSensor.readSnd();
    audioData = aSensor.byteToInt(aSensor.audioBytes);

        i300= aSensor.avgPow(aSensor.filter300(audioData));
```

```
            i6000= aSensor.avgPow(aSensor.filter6000(audioData));
            iRatio= i6000/i300;
            if (iRatio< oldR){ rotSign= -rotSign;}


    //2.3 rotate until find the peak in intensity
     peakPassed= false;
     while(!peakPassed){
     oldR= iRatio;

        rotate(rotSign*20);
        try {
            sleep(delay);
        } catch(InterruptedException e) {}

        aSensor.readSnd();
        audioData = aSensor.byteToInt(aSensor.audioBytes);

     i300= aSensor.avgPow(aSensor.filter300(audioData));
     i6000= aSensor.avgPow(aSensor.filter6000(audioData));
     iRatio= i6000/i300;
            if (iRatio< oldR){ peakPassed= true; rotSign= -rotSign;}
      }

        rotate(rotSign*20);
      try {
          sleep(delay);
      } catch(InterruptedException e) {}

    //2.4 move-wait
     moveForward(0.20f);
       try {
           sleep(delay);
       } catch(InterruptedException e) {}


    }
    else try {
            sleep(delay);
          } catch(InterruptedException e) {}
    }
}
```

References

[1] Greg L. Reid, Evangelos Milios. Active Stereo Sound Localization. Technical Report CS-1999-09. York University, December 15, 1999.

[2] http://www.mtu-net.ru/pinetar/dm/ –describes various filters for sound processing (in Russian).

[3] http://ixbt.stack.net/multimedia.shtml – publishes news, reviews, testing of various new computer equipment, including multimedia equipment (in Russian).

[4] http://www.intersrv.com/~dcross/timefilt.html – describes various time domain filtering techniques for digital audio (in English).

[5] http://www.dsptutor.freeuk.com/remez/RemezFIRFilterDesign.html – designs FIR filters