Abstract

We have built an algorithm for sound localization. The main parts of the algorithm are in two C programs: “audio_record” and “localize”. The “audio_record” program repeatedly records 200 milliseconds of sound from the microphones and writes the data to an output port. The “localize” program repeatedly reads the data from an input port and processes the data. It first detects to see if there is a sound (as opposed to silence) in the audio signal. If there is no sound, it ignores that data and continues to process another 200 milliseconds of sound. If a sound is detected, it calculates the interaural time difference (ITD) between the left and right signals. It also calculates the horizontal angle between the microphones and the location of the original sound.

The hardware setup for our system is a two-channel microphone attached to a mannequin. The microphones are attached to the mannequin at the places where the two ears of a human would be. The microphone is connected to a preamplifier that is plugged into the “line in” audio input of the sound card. The sound card is connected to one of the sixteen nodes of our current hardware system. Since these nodes run the QNX operating system, we record the audio signal through the QNX audio interface.

Our algorithm has been tested on the mannequin setup. We recorded test samples of people speaking in front of the microphones, and test samples of people clapping. These sounds were recorded at different angles from the two-microphone system, but they were all at approximately the same horizontal plane with respect to the two microphones. Our system can calculate the horizontal angle to within 5 degrees of the correct value.

There are a number of improvements that can be made to the system. The localization works very well if the sound is approximately in the same horizontal plane of the two microphones. If not, there can be errors, which are expected. An easy way to compute the vertical angle with respect to the horizontal plane is to use the same algorithm with another set of microphones placed vertically from each other. Having noise in the background also creates errors. In order for the system to work well, it needs to be in an ideal setting where there is no other sound. In addition, the algorithm tends to work better for people’s voice than people clapping. Clapping may create echoes that can introduce noise other than the original sound.
**Description of Project**

Humans possess the ability to locate the source of a sound within approximately 5° in elevation and azimuth. The Head Related Transfer Function is a model that describes how this is done [1]. When a sound source is more than about 1 m away from the head, auditory cues such as the interaural time difference (ITD), the interaural intensity difference (IID), and the shape of the pinna and the head are used to localize a sound source. When the sound source is less than 1 m away from the head, the situation is different. We will focus only on the situation when the sound source is more than 1 m away from the head.

The purpose of this project is to build a system to implement auditory localization. Sound localization systems have been built and demonstrated by Bub, Hunke, and Waibel [2], and Silverman and Kirtman [3]. Their systems use arrays of multiple microphones. The redundant microphones provide additional information to further improve the accuracy of the localization process. Wasson [4] built a localization system with two microphones. His system is mostly based on the phase correlation between the two signals received by the microphones. The system to be implemented in this project will be similar to Wasson’s.

This document describes the work done on the project. We focus on the technical details of what we did and why we chose to do what we did. We also present the results that we have from testing our current system.

Eventually, we hope to use our sound localization algorithm on the robot in our lab. The idea is: when someone makes a sound in front of the robot, it will automatically turn towards that sound. This capability greatly improves a person’s willingness to interact and communicate with the robot.
**Pseudocode for “audio_record”**

Here is the pseudocode for the “audio_record” program. Each part of the program will be discussed in more detail in different sections of this document.

declare audio data structure, and buffer for storing data

define output port

set-up and open channel for capturing sound

while (1) {
  record 200 ms. of sound from microphones into buffer
  write data in buffer to output port
}
**Pseudocode for “localize”**

Here is the pseudocode for the “localize” program. Again, each part of the program will be discussed in more detail in different sections of this document.

declare data structure for left and right signals

define input and output ports

while (1) {
    read data (200 ms clips) from input port
    split into left and right signals, and convert into appropriate format
    detect for sound (as opposed to silence)
    if (sound detected) {
        reduce sampling rate to 11025, use cepstrum method to find delay (in number of samples) between left and right signals
        with sampling rate of 44100, repeat method, searching among a smaller range of possible delay values
        calculate ITD, and horizontal angle from microphones to location of sound
        write horizontal angle and other useful information to output port
        read 200 ms clip from input port to flush out old data
    }
}
Port System on QNX

The reason for using the port system is that we want our algorithm to run continuously. If we only have one process, the algorithm can either be recording audio or processing it. It cannot do both at the same time. Using the port system, we can create two processes. One of them (audio_record) will set-up the interface with the sound card and the audio driver, and continuously record the audio signal into an output port. The other (localize) will read the data from an input port, do the necessary calculations, and write the results to output ports.

Three pieces of information are being written by “localize” to output ports. “OutPortSound” contains a 0 or 1, indicating whether or not there is a sound (as opposed to silence) in the corresponding 200 ms data clip. “OutPortITD” is the delay in number of seconds between the left and right signals. “OutPortHorAngle” is the horizontal angle between the center of the two-microphone system and the location of the sound.

In C/C++, here is how we declare a port and associate it with a name:

```c
OutPutPortOf< int > OutPortRaw;
OutPortRaw.SetName("/audio_record/output/raw_int");
```

To write data to an output port, we do this:

```c
int s = 320;
OutPortRaw.Content() = s;
OutPortInt.Write();
```

And to read data from an input port:

```c
int r;
InPortRaw.Read();
r = InPortRaw.Content();
```
**Interface to Record Audio Data**

In order to record audio data in the QNX operating system, we need to use the QNX Sound Architecture drivers and library. There is QNX documentation [5] that explains this sound architecture in detail. We will briefly examine the parts that we need. Our system has these three main elements needed to produce/record sound: a sound card, device driver, and an Application Programming Interface. For the sound card, there is also a PCM device that is used for converting between the analog signal and digital sequences.

With these parts ready, we can record audio data in the following way. First, we must include the header file `/usr/include/sys/asoundlib.h` in the program and link the file `/lib/libasound.so` when compiling. This will allow the program to access the functions defined in the audio library. The most important functions that we need are:

- `snd_pcm_open_preferred()` – used to open the PCM device
- `snd_pcm_plugin_info()` – tells the device the format of the data
- `snd_pcm_plugin_prepare()` – prepares the opened channel to run
- `snd_pcm_close()` – close device when we are done

The user must also use the software mixer application to select the “line in” input as the channel to be read from. We can use this function to actually record data:

- `snd_pcm_plugin_read()` – read data from the audio driver and fills a specified buffer

The QNX documentation shows an example of using the `select()` function to record audio data in blocking mode. We wish to execute in blocking mode because doing so will allow the `read()` function to read data into the buffer fully (200 ms at a time) before writing it to the output port. However, we found that following the QNX example produces a delay in the overall response time, relative to not using `select()`. Hence we used this function to set the mode to blocking:

- `snd_pcm_nonblock_mode()`

and executed the `read()` function without calling `select()`. 
**Format of Audio Data**

There are a variety of data formats that the data can be recorded as. The QNX documentation [5] explains these in detail. We choose to record the data in stereo 8-bit mode, and at a sampling rate of 44100 per second. The data is being read as signed 8-bit numbers, and the data from the left and right signals are placed into the buffer in an interleaved fashion.

The “audio_record” program specifies a riff data type header structure. This can be written as the header of a wav file. We do not need to write the data as a file, but we still keep this structure to specify and clearly define the data parameters, and to make the format known to the function that records the actual data.

The data from the input port of “localize” needs to be converted into an appropriate format before we can process it. We perform these steps:

1) Separate data into left and right channels
2) Convert from signed 8-bit to unsigned 8-bit
3) Convert the range of the values to between –1 and 1

We convert the range to between –1 and 1 so that the data will be compatible with the usual audio data format in MATLAB. The algorithms that are used to process this data were originally written and tested in MATLAB. Keeping the data format compatible allows for a convenient comparison and maintenance of the code. Adding these extra steps to convert the format requires a minimal amount of time and will not affect the general efficiency of the algorithm.
Detection of Sound vs. Silence

The motivations for this step are to ignore the parts of the data that include no interesting sound and to allow the program to run more efficiently. If the recorded data has no interesting sound, we do not want to process it because trying to find the time delay between the left and right signals would be meaningless. In addition, we do not need to process this data in the usual way. If we can quickly check to see that a 200 ms clip of data does not contain any interesting sound, we can ignore the data immediately and not have to spend more time calculating any time delay value. It turns out that we can indeed do this in minimal time.

There exist many methods for differentiating between sound and silence in audio data. We use a popular approach that is simple and efficient. For each 200 ms clip, we compute the short-time energy function:

$$ Energy = \frac{1}{n} \sum_{t=1}^{n} [y_t]^2 $$

where \( n \) is the number of samples, and \( y_t \) is the intensity value at time \( t \). In general, the energy value of a sample with sound is greater than the energy value of a sample without sound. This can therefore be used to differentiate between the two. Figures 1 and 2 on the next page show an example of a waveform and the corresponding energy function.

We also calculate the short-time zero crossing rate (ZCR) of each 200 ms clip. The ZCR is the number of times the waveform crosses the intensity value 0. In general, the ZCR of a sample with sound is greater than the ZCR of a sample without sound. This property can also be used to distinguish between sound and silence. Figure 3 shows the corresponding ZCR values for the waveform in figure 1.

For each sample, we multiply the energy value with the ZCR. If this value is below a fixed threshold value, the audio sample is classified as silence. If the value is above the threshold, it is classified as sound. The threshold value needs to be recalibrated on different systems and conditions. For the system that we tested, we used a value of 0.0003. The user will need to re-calibrate this value depending on the system set-up. Figure 4 shows the corresponding values of energy multiplied by ZCR (the intensity values in the figure are on a different scale).
Figure 1. Waveform of signal

Figure 2. Short-time energy function
Figure 3. Short-time zero crossing rate (ZCR)

Figure 4. Energy X ZCR
**Microphone set-up**

The following diagram shows the geometric setup of the microphones:

![Diagram of microphone setup](image)

The horizontal angle is calculated from the center of the two-microphone system to the location of the sound. The angle is 0 degrees if the sound is directly in front of the mannikin. The angle is negative if it is on the left side and positive on the right side.

The microphones are placed 0.16 meters apart. This value is hard-coded in “localize”. The user must change the value in “localize” if the microphones are placed at a different distance than 0.16 meters. Furthermore, the speed is sound is assumed to be 343 meters per second, and this number is also coded into “localize”. If the temperature of the environment changes, this speed will need to be changed.

We tried some simple experiments to test our methods. Some of the test cases shown below are labeled with names such as “testing1” or “claps3”. “testing” means that the word “testing” was spoken and recorded, and “claps” means that the sound of a person clapping was recorded. The labels 1 to 5 specify the approximate location of where the sound is from, as shown above.
Waveform cross-correlation method

To calculate the horizontal angle between the microphones and the location of the sound, we need to first compute the time delay between the left and right signal. We first tried a popular method that computes the cross-correlation function of the waveform. This approach worked quite well. However, we then tried another method using the cepstrum function and our experiments showed that this worked better. Hence we implemented the latter approach in our algorithm. Here, we describe the cross-correlation method, and in the next section, we describe the cepstrum method.

Given the left and right signals $y_1[k]$ and $y_2[k]$, the delay between the two is the location $k_{12}$ of the maximum of the cross-correlation function:

$$\phi_{12}[k] = \sum_{m=-\infty}^{\infty} y_1[k + m] \ast y_2[m]$$

The index $m$ is the length of each of $y_1$ and $y_2$ that we are correlating. This length should be as large as possible, if the data is available. The index $k$ represents the offset between $y_1$ and $y_2$ for each correlation calculation. For a specific $k$, the higher the cross-correlation value, the better the match between $y_1$ and $y_2$. So the value of $k$ giving the maximum of the cross-correlation function gives us the delay in number of samples between $y_1$ and $y_2$.

Figure 5 on the next page shows an example of a waveform recorded on the left channel of the microphone. The one for the right channel is similar. Figure 6 shows the result of applying the cross-correlation function, where the correlation length is 3000 and the offset index is 30. We computed the correlation function 61 times, with the value of $k$ ranging from –30 to +30. The value of $k$ that maximizes the correlation function is +21. This is the delay in the number of samples between the left and right signals. The following table shows the results of some other test cases. We computed the delay values using the waveform method.

<table>
<thead>
<tr>
<th>test samples</th>
<th>“correct” delay from geometry</th>
<th>computed delay from waveform method</th>
</tr>
</thead>
<tbody>
<tr>
<td>testing1</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>testing2</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>testing3</td>
<td>0</td>
<td>-3</td>
</tr>
<tr>
<td>testing4</td>
<td>-11</td>
<td>-7</td>
</tr>
<tr>
<td>testing5</td>
<td>-18</td>
<td>-20</td>
</tr>
<tr>
<td>claps1</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>claps2</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>claps3</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>claps4</td>
<td>-11</td>
<td>-10</td>
</tr>
<tr>
<td>claps5</td>
<td>-18</td>
<td>-19</td>
</tr>
</tbody>
</table>
Figure 5. Waveform from left channel

Figure 6. Cross-correlation function with offset $k = -30$ to 30
Cepstrum method

We implemented a method using the cepstrum function to calculate the delay between the left and right signals. The cepstrum function is used in signal processing for detecting echoes in audio signals. For more information, refer to Oppenheim and Schafer’s book [6].

Given the left and right signals $y_1[k]$ and $y_2[k]$, the delay between the two is the location $k_{12}$ of the maximum of this function:

$$\theta_{12}[k] = \operatorname{mvalue}(\text{cepstrum}(y_1[k + m]y_2[m]))$$

$y_1[k+m]y_2[m]$ is the concatenation of the two sequences of samples. We take the cepstrum of this sequence. From the resulting output vector, we take only the middle value in the sequence. The index $m$ is the length of each of $y_1$ and $y_2$ that we are concatenating. This length should be as large as possible, if the data is available. The index $k$ represents the offset between $y_1$ and $y_2$ for each function calculation. For a specific $k$, the higher the $\theta$ function value, the better the match between $y_1$ and $y_2$. So the value of $k$ giving the maximum of the $\theta$ function gives us the delay in number of samples between $y_1$ and $y_2$.

Figure 7 on the next page shows an example of a waveform recorded on the left channel of the microphone. The one for the right channel is similar. Figure 8 shows the result of applying the $\theta$ function, where the correlation length is 5000 and the offset index is 30. We computed the correlation function 61 times, with the value of $k$ ranging from $-30$ to $+30$. The value of $k$ that maximizes the $\theta$ function is $-11$. This is the delay in the number of samples between the left and right signals. The following table shows the results of some other test cases. We computed the delay values using the cepstrum method.

<table>
<thead>
<tr>
<th>test samples</th>
<th>“correct” delay from geometry</th>
<th>computed delay from cepstrum method</th>
</tr>
</thead>
<tbody>
<tr>
<td>testing1</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>testing2</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>testing3</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>testing4</td>
<td>-11</td>
<td>-11</td>
</tr>
<tr>
<td>testing5</td>
<td>-18</td>
<td>-19</td>
</tr>
<tr>
<td>claps1</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>claps2</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>claps3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>claps4</td>
<td>-11</td>
<td>-11</td>
</tr>
<tr>
<td>claps5</td>
<td>-18</td>
<td>-20</td>
</tr>
</tbody>
</table>
Figure 7. Waveform from left channel

Figure 8. $\theta$ function with offset $k = -30$ to 30
Implementing cepstrum function and FFT in C

In order to implement the cepstrum method as shown above, we need to implement the cepstrum function and FFT in C. We choose to write our code in C to make it compatible with the other code that controls the robot in our lab. In addition, the C code will be more efficient than MATLAB. We therefore want to have simple and efficient implementations of the cepstrum and FFT in C.

For the FFT, we used code written by Don Cross. We are using his code because it is simple, efficient, and easy to interface with. The “README” files and actual code contain more documentation. The most important function that we use is:

```c
fft_double(unsigned int, int, double *, double *, double *, double *);
```

The second parameter can be 0 or 1, for the FFT or the inverse FFT respectively. Both of these are necessary for the cepstrum.

For the cepstrum, we implemented our own function:

```c
cceps_double(unsigned int, double *, double *);
```

The code is written to implement the cceps() function in MATLAB. The algorithms that are used to process the audio data were originally written and tested in MATLAB. Keeping our cepstrum function compatible with that of MATLAB’s allows for a convenient comparison and maintenance of the code.
Executive cepstrum method twice

In “localize”, we choose to execute the cepstrum method twice:

reduce sampling rate to 11025, use cepstrum method to find delay (in
number of samples) between left and right signals

with sampling rate of 44100, repeat method, searching among a smaller
range of possible delay values

The motivation for doing so is that it gives an accurate delay value in a short amount of
time. We could run the method once with a sampling rate of 11025. It would run in a
short time, but the computed delay value would not be as accurate in comparison. If we
run the method once with a sampling rate of 44100, the value would be much more
accurate. However, it would take too long to calculate the value. In fact, it would take
more than 3 seconds just to run this part of the algorithm. Since we want the eventual
response time to be less than 1 second, this is not acceptable either. So we choose to run
the cepstrum method twice, the first time at 11025 Hz and then at 44100 Hz.

When we run the cepstrum method with a sampling rate of 11025, we calculate
the value $\theta_{13}[k]$ 17 times with k from –8 to 8. The outcome will allow us to know the
approximate range of where the optimal k value lies. Given this approximate range, we
then run the cepstrum method again with a sampling rate of 44100. We calculate the
value $\theta_{13}[k]$ only 7 times, with the range of k centered at the corresponding value that
was found from the first run. So the first run is designed to find an approximate range of
where the optimal k lies, and the second run focuses on this approximate range to find the
optimal k value.

The overall method gives a reasonably accurate delay value. The total time that it
takes is about 0.5 seconds.
**ITD and horizontal angle calculation**

Once we have the time delay in terms of the number of samples, the time delay in terms of seconds is simply:

\[
\text{time delay in seconds} = \frac{k_{12}}{T}
\]

where \(k_{12}\) is the best value of \(k\) calculated previously, and \(T\) is the sampling rate (44100 samples per second in our case). This time delay is known as the interaural time difference (ITD).

With the ITD, we can then proceed to find the horizontal angle \(\alpha\):

We assume the microphones are mounted on a spherical shaped object with radius “\(a\)”. The sound is coming at an angle \(\alpha\) towards the microphones. From the diagram, we see that the sound arrives at the right side before the left, and it has to travel an extra distance of approximately \((a \alpha + a \sin \alpha)\) to reach the left side. We obtain the ITD by dividing this distance over the speed of sound, \(c\):

\[
ITD = \frac{a}{c} (\alpha + \sin \alpha)
\]

\[-\frac{\pi}{2} \leq \alpha \leq \frac{\pi}{2}\]

For our setup, “\(a\)” equals 0.08 meters and “\(c\)” is assumed to be about 343 meters per second.
We can actually use this formula to estimate the largest possible time delay in terms of the number of samples. The largest possible angle $\alpha$ is ($\pi/2$), so the maximum ITD is:

$$\text{max ITD} = \frac{0.08}{343} \cdot (\pi/2 + 1) \text{ seconds}$$
$$= 0.0005996026 \text{ seconds}$$

Multiplying this by the sampling rate gives us the delay in the number of samples:

$$\text{max delay} = 0.0005996026 \cdot 44100$$
$$= 26.44 \text{ number of samples}$$

At a sampling rate of 11025, the maximum delay would be about 6 to 7 samples. This justifies the calculation of $\theta_1: [k]$ 17 times with $k$ from $-8$ to $8$ in our first run of the cepstrum method. There is no need to search for values of $k$ below $-8$ or above $8$ because the maximum delay should not be in those ranges.
Results and known problems

The following table shows some of the results from testing our system. The angles in the column headings are the “correct” angles. “claps” means some clapping or sharp sounds were made, and “voice” represents a person speaking. For each case, the corresponding cell in the table shows the results of ten different trials. The angle values are in degrees.

<table>
<thead>
<tr>
<th></th>
<th>60</th>
<th>30</th>
<th>0</th>
<th>-30</th>
<th>-60</th>
</tr>
</thead>
<tbody>
<tr>
<td>claps</td>
<td>58 58</td>
<td>31 31</td>
<td>3 3</td>
<td>-28 -28</td>
<td>-58 6</td>
</tr>
<tr>
<td></td>
<td>58 58</td>
<td>61 31</td>
<td>3 3</td>
<td>-28 -28</td>
<td>-6  -54</td>
</tr>
<tr>
<td></td>
<td>54  8</td>
<td>31 31</td>
<td>-47 3</td>
<td>-28 3</td>
<td>-54 -25</td>
</tr>
<tr>
<td></td>
<td>6   58</td>
<td>8 31</td>
<td>-3 3</td>
<td>11 -25</td>
<td>-54 -54</td>
</tr>
<tr>
<td></td>
<td>58 58</td>
<td>-25 31</td>
<td>-6 3</td>
<td>-28 -25</td>
<td>-58 -54</td>
</tr>
</tbody>
</table>

| voice | 54  69| 41 28 | -3 6  | -25 -25| -58 -54|
|       | 58  73| 31 31 | 3 6  | -41 -28| -58 -69|
|       | 61  58| 31 34 | 3 -3  | -28 -25| -54 -54|
|       | 61  54| 34 31 | 0 3  | -25 -23| -50 -58|
|       | 58  50| 31 31 | 3 0  | -25 -28| -54 -58|

The localization algorithm works well under these conditions: the sound is made from about the same horizontal plane as the microphones, the sound is a human voice, and there is minimal background noise. Furthermore, if the sound is sustained for more than 1 second, the algorithm works almost perfectly.

We expect that the sound must be in approximately the same horizontal plane as the microphones. The system can be improved by adding a method for computing the vertical angle. This is difficult with just two microphones. With more than two microphones, we can have two microphones placed vertically away from each other, and the same algorithm will give the vertical angle. In addition, more microphones can lead to more accurate results.

The results in the table show that the system works very well for human voices, but not as well for clapping sounds or sharp noises. Clapping sounds may have created echoes that can affect the results. Human voices also tend to sustain for a longer time.

Background noise is a problem for the system. If the microphones are near a computer terminal, or in a room with many people, the algorithm tends to not work as well. This is a general problem in signal processing. Again, using more than two microphones could help. With more microphones, we can run the same algorithm more than once in parallel, and average the results.
References

www.qnx.com/developer/docs/momentum_nc_docs/neutrino/audio/about.html