Acoustic Beamforming using a TDS3230 DSK: Final Report

Steven Bell
Student Member, IEEE

Nathan West
Student Member, IEEE

Electrical and Computer Engineering
Oklahoma Christian University

CONTENTS

I Introduction 1

II Theory 1
- II-A Delay and Sum Method 1
- II-B Microphone Array Design 1
  - II-B1 Microphone spacing 1

III Simulation 2
- III-A Source localization 2
- III-B Spatial Filtering 3

IV System Requirements 3

V Design 4
- V-A Hardware 4
- V-B DSP Software 4
- V-C Interface Software 5
  - V-C1 DSK-PC Interface 5
  - V-C2 Interface GUI 5

VI Test Methods 6
- VI-A Source localization 6
- VI-B Spatial filtering 6

VII Results 6

VIII Discussion 6
- VIII-A Sources of difficulty 6

IX Conclusion 7

References 7

Appendix A: MATLAB code for beam-sweep source localization 8

Appendix B: MATLAB code for spatial filtering 11

Appendix C: C code for main processing loop 14

Appendix D: C code for summing delays 16

Appendix E: C code for calculating delays 17

Appendix F: Java Code - Main 17

Appendix G: Java Code for Beamformer Communication 18

Appendix H: Java Code for Beam Display 21

***Abstract*—Acoustic beamforming is the use of a microphone array to determine the location of an audio source or to filter audio based on its direction of arrival. For this project, we simulated and implemented a real-time acoustic beamformer using MATLAB for simulations and the TDS3230 DSK for the real-time implementaion. Although the final system does not meet all of our initial goals, it does successfully demonstrate beamforming concepts.**

I. Introduction

Most broadly, beamforming is the use of an array of antennas - or in the case of audio, microphones - to perform signal processing based on the spatial characteristics of a signal. We will discuss two primary forms of beamforming in this document: source localization and spatial filtering. Source localization attempts to determine the location in space a signal originated from, while spatial filtering creates an electronically-steerable narrow-beam antenna, which has gain in one direction and strong attenuation in others. Spatial filtering systems and corresponding transmission techniques can replace physically moving antennas, such as those used for radar.

Acoustic source localization is familiar to all of us: we have two ears, and by using them together, our brain can tell where sounds come from. Similarly, our brains are able to focus on one particular sound and tune out the rest, even when the surrounding noise is much louder than the sound we are trying to hear.

II. Theory

A. Delay and Sum Method

If we have an array of microphones and sufficient signal-processing capability, we can measure the time delay from the time the sound strikes the first microphone until it strikes the second. If we assume waves originate far enough away that we can treat the edge of its propagation as a plane, then the delays are simple to model with trigonometry as shown in Figure 1.

Suppose we have a linear array of microphones, m_1 through m_n, each spaced D_{mic} meters apart. Then D_{delay}, the extra distance the sound has to travel for each successive microphone, is given by

$$D_{delay} = D_{mic} \cdot \cos(\theta) \tag{1}$$

At sea level, the speed of sound is $340.29\,\frac{\text{m}}{\text{s}}$, which means that the time delay is

$$T_{delay} = \frac{D_{mic}}{340.29\,\frac{\text{m}}{\text{s}}} \cdot \cos(\theta) \tag{2}$$

By reversing this delay for each microphone and summing the inputs, we can recover the original signal. If a signal comes from a different direction, the delays will be different, and as a result, the individual signals will not line up and will tend to cancel each other when added. This essentially creates a spatial filter, which we can point in any direction by changing the delays.

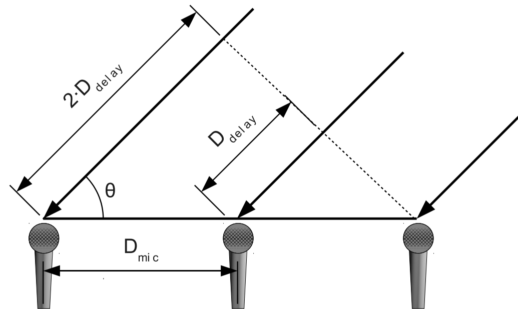

Figure 1. Planar wavefront approaching a linear microphone array.

To determine the direction a signal came from, we can sweep our beam around the room, and record the total power of the signal recieved for each beam. The source came from the direction with the highest-power signal.

B. Microphone Array Design

Microphone arrays can be nearly any shape: linear, circular, rectangular, or even spherical. A one-dimensional array allows beamforming in one dimension; additional array dimensions allow for 2-dimensional beamforming. Given the limited number of microphones and amount of time we have, a linear array is the best choice.

1) Microphone spacing: The spacing of the microphones is driven by the intended operating frequency range.

For spatial filtering, a narrower beam width is an advantage, because signals which are not directly from the intended direction are attenuated. A narrow beam width is analogous to a narrow transistion band for a tranditional filter. Lower frequencies will correlate better with delayed versions of themselves than high frequencies, so the lower the frequency, the broader the beam. Conversely, a longer array will result in a greater delay between the end microphones, and will thus reduce the beam width.

At the same time, the spacing between microphones determines the highest operating frequency. If the wavelength of the incoming signal is less than the spacing between the microphones, then spatial aliasing occurs. An example is shown in Figure 2.

The spacing between microphones causes a maximum time delay which, together with the sampling frequency, limits the number of unique beams that can be made.

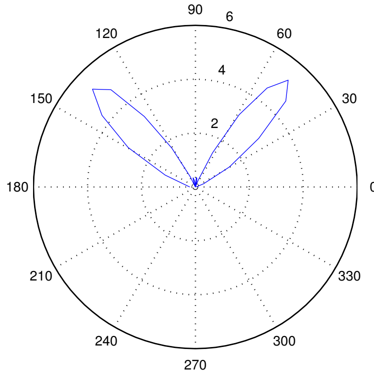

Figure 2. Spatial aliasing. The source is 1 kHz, located at 135 degrees. An image appears at approximately 50 degrees, which means that sound coming from that direction is indistinguishable from the desired source.

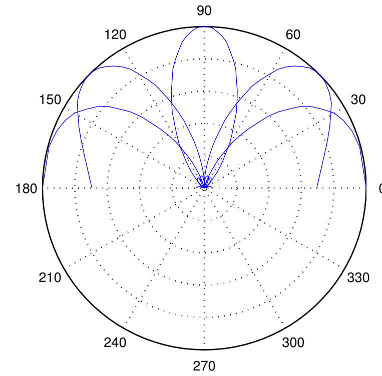

Figure 3. Source localization simulation using a 400 Hz source and a 4-microphone array. The source was located at 0, 45, 90, 135, and 180 degrees.

The maximum number of beams, max_{beams}, is shown mathematically in Equation 3.

$$max_{beams} = 2 \cdot F_s \cdot time_{spacing} \quad (3)$$

The variable $time_{spacing}$ is the maximum amount of time it takes for sound to travel from one microphone to an adjacent microphone, as the case when the source is along the line created by the array.

III. Simulation

Source localization and spatial filtering both use essentially the same simulation code. First, we define a coordinate system where our microphones are placed, with the origin at the center of our linear array. We defined the distance between microphones to be 25 cm, making total array lengths of 75 cm for a four microphone array and 25 cm for a two microphone array.

A. Source localization

In the source localization code, a single source is located 10 meters away from the center of the array. A loop creates thirty beams which cover a 180-degree range, and the power for each beam is computed. Figure 3 shows the power sweeps for five different source angles. Note that in each case, the maximum power is exactly at the angle the sound originates from.

Using the beam-sweeping technique, we examined the relative effects of the number of microphones and the source frequency. Figure 4 shows a comparison of the beamwidth as a function of the number of microphones. Figure 5 shows a comparison of beam width versus frequency. Note that higher frequencies produce narrower beams, so long as spatial aliasing does not occur.

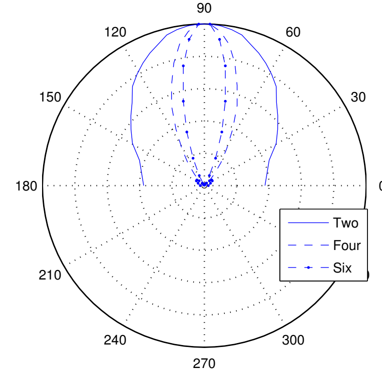

Figure 4. Comparison of beam width versus the number of microphones. The source was 400 Hz at 90 degrees.

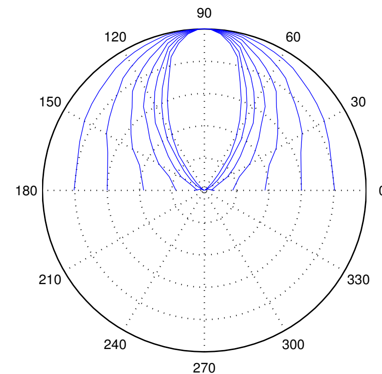

Figure 5. Comparison of beam width versus frequency using 2 microphones. The frequencies range from 200 Hz to 700 Hz in increments of 100 Hz. Higher frequencies produce narrower beams.

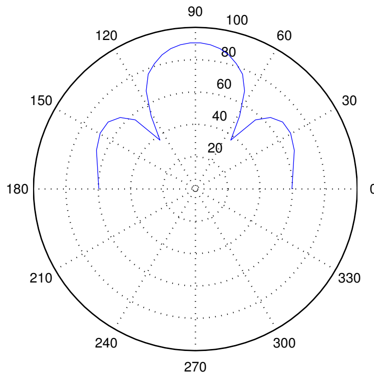

Figure 6. Decibel-scale plot of a 4-microphone array beampattern for a 600 Hz signal at 90 degrees. Note the relatively large sidelobes on both sides of the main beam.

Figure 6 shows a plot of the beam in decibels, which brings out a pair of relatively large sidelobes. These can be reduced by windowing the microphone inputs, in the same way that a filter can be windowed to produce lower sidelobes in exchange for a wider transition band. However, this windowing is unlikely to be useful for our small array.

The MATLAB source code is in Appendix A.

B. *Spatial Filtering*

For spatial filtering, one source is placed at 10 m normal to the array and 10 m tangentially from the center of the array. Using this location as a reference, we can place a secondary source equidistant from the center of the microphone array but some degree offset.

For this simulation set we used two offsets, 60 degrees and 90 degrees. The formed beam was looking 45 degrees away from the array (towards the source at 10, 10). Two microphones were used in simulations because of suggestions by Dr. Waldo and four were used because that is the maximum number of input channels on our hardware. In Figure 7 the simulation has a variable number of microphones with the sources separated by 60 and 90 degrees. This simulation was the influence for the specification calling for 3dB down on a 60 degree source separation. The SNR is improved drastically when the same sources are separated by 90 degrees still using a 2 microphone array, as seen in Figure 8.

Figures 9 and 10 show similar but improved results when the array is increased to four microphones.

The MATLAB source code is in Appendix B.

IV. System Requirements

- The system must have at least two microphones as its input.

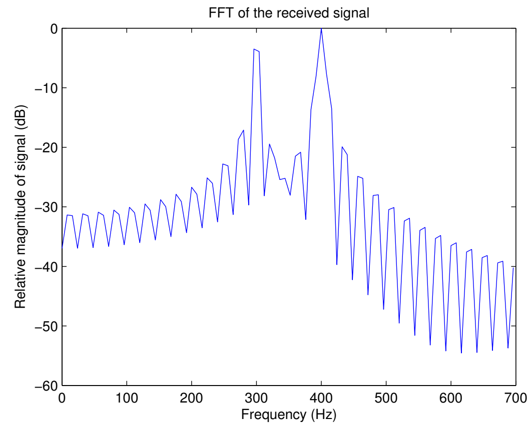

Figure 7. The noise source at 300Hz is attenuated by about 3dB using an array of two microphones with sources separated by 60 degrees.

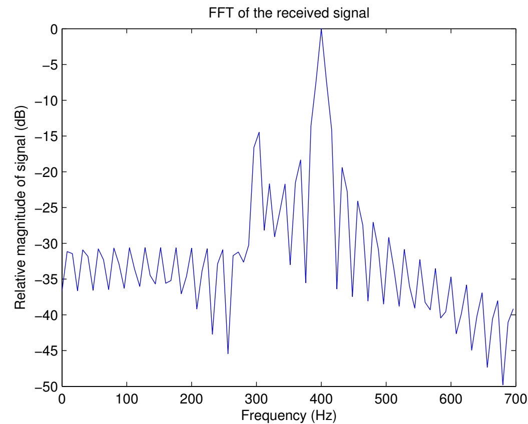

Figure 8. An array of two microphones with sources separated by 90 degrees will attenuate the noise source by about 13dB.

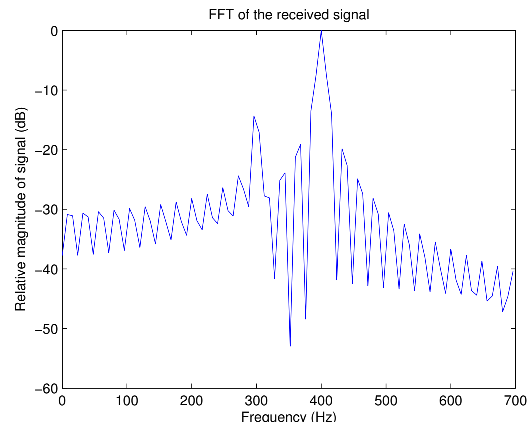

Figure 9. An array of four microphones with sources separated by 60 degrees results in the noise source attenuated by about 12dB.

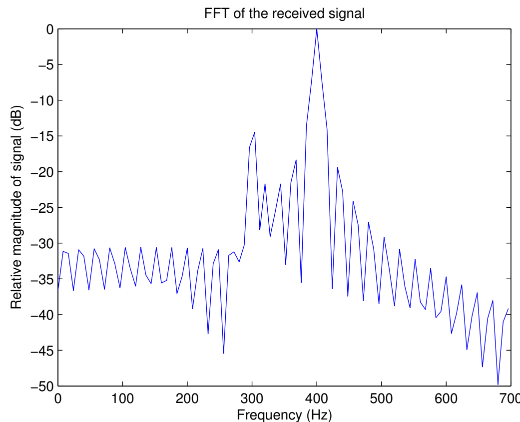

Figure 10. With an array of four microphones with sources separated by 90 degrees the noise signal is about 12dB down from the signal we attempted to isolate.

- The system must have a GUI running on a computer which communicates with the DSK board.
- In localization mode:
  - The system will be able to locate the direction of a 400 Hz tone between -90 and +90 degrees, with +/- 5 degrees error. This target tone will be significantly above the noise threshold in the room.
  - The GUI must display the direction of the incoming sound with a latency less than 1 second.
- In directional enhancement mode:
  - The GUI must allow the user to select a particular direction to listen from, between -90 and +90 degrees, and with a resolution of at least 10 degrees.
  - The system should play the selected directional output via one of the line out channels.
  - A 300 Hz noise source with the same power will be placed 60 degrees apart from the desired signal source. The directional output should give a signal-to-noise ratio of at least 3 dB.
- The system will operate properly when the sound sources are at least 4 meters away from the microphone array.

V. Design

The acoustic beam forming system consists of three high level subsystems: the microphone array, the DSK board, and a GUI running on a PC. A block diagram showing the interconnections of these subsystems is shown in Figure 11. The microphone array should contain two to four microphones placed in a straight line 25 cm apart. Theses microphones will be connected to the microphone inputs on the DSK_AUDIO4 daughtercard. The DSK will be programmed to process a 256 sample block at a time. The software delays each microphone's input by a different number of samples then adds each input to create the output. In source localization mode, the power of the output is calculated and sent to the PC GUI. In spatial filtering mode, the output is passed to one of the board's output channels.

Figure 11. Functional block diagram.

Figure 12. Flowchart for the main DSP software.

A. Hardware

In addition to the standard DSK board provided to the DSP class we will be using the Educational DSP DSK_AUDIO4 daughtercard, which has four input channels. Each of these channels will be connected to one of the generic mono microphones provided to the class by Dr. Waldo. The placement of the microphones introduces a physical variable to the system that is important to the operation of hardware: according to our specifications and simulations the microphones should be spaced 25 cm apart. Ideally, they will face the direction of incoming acoustic sources for maximum input power.

B. DSP Software

The foundation of our DSP software is the sample code provided by Educational DSP and modified by Dr. Waldo. This code initializes the DSP and the codec and calls a processing function that provides samples from all four input channels. Using this as a base we will implement our main function inside of the processing function provided. In this main file we will also include a file that defines our sum and delay functions which will exist in their own files.

The main software branch opens a RTDX channel with the PC GUI and keeps track of the current operating mode. The mode is a boolean value where a TRUE means spatial filtering mode and a FALSE means source localization mode. In the spatial aliasing mode we do a simple delay and sum of the input data. The delay is given by the GUI. In localization mode the current output is also the delay and sum of the input; however, the delay sweeps from a minimum possible delay to a maximum possible delay defined by the number of samples it takes to go from -90° to 90°. After every delay and sum operation the delay/power pairs will be sent to the GUI for display via the RTDX channel. The flow of this is shown in Figure 12.

Figure 13. Flowchart for the function that calculates the number of samples to delay each microphone's input by.

Figure 14. Flowchart for the DSK function that calculates the current output sample.

To determine how much to delay each microphone by we get the number of samples each microphone should be delayed by from the GUI and multiply by the order of that microphone in the array. The first microphone in the array is defined as microphone 0 and has a 0 delay. If the delay we get from the GUI is negative the order is reversed so that the earliest input always has a 0 delay. The calcDelay function will return a pointed to an array called delays. Figure 13 shows the structure and flow for calcDelay.

The sum function accepts the current four input samples and delay from the GUI and passes it to calcDelays. Figure 14 shows a program flow for this function. It will have a reverse index tracked buffer so that if `i=5` is the current input `i=6` is the oldest input and `i=4` is the second most recent input. This requires fewer rollover-error-checking comparisons relative to a forward-tracking buffer, which should give us an incremental speed improvement.

C. Interface Software

1) DSK-PC Interface: The PC will communicate with the DSK board using RTDX 2.0. For speed, the DSK will only perform integer operations; thus, the PC will perform the floating-point work of translating delays to angles and vice-versa. There will be two RTDX IO channels, one going each direction.

a) Power-vs-Delay to PC:

- Delay will be a 2's complement 8 bit signed number, which represents the relative delay between microphones. This will be calculated from the microphone spacing and the speed of sound.
- Power will be a 16-bit unsigned number, which holds the relative power of the signal for a particular beam.

In order for these two values to remain correlated, we will interleave both on one channel, and signal the start of each new delay-power pair using the value 0xFF. This will allow our system to recover from lost or delayed bytes.

This data will only be sent when the system is in source localization mode.

0	1	2	3	4	5	...
0xFF	Delay	Power (16 Bits)		0xFF	Delay	...

Figure 15. Byte transmission order for DSK→ PC communication

Table I
BYTE SIGNALS FOR PC→DSK COMMUNICATION

Use	Code	Range
Microphone time spacing		0-100
Number of microphones		2-4
Mode		1,2
Beam selection		-100 to +100

0	1	2	3	4	5	...
0xFF	Code	Value	0xFF	Code	Value	...

Figure 16. Byte transmission order for PC → DSK communication

We found that we did not need any synchronization bytes, and we trimmed the communictation down to a single byte for the power paired with a single byte for delay. This was intended to speed up RTDX communication, but was not successful.

b) Settings to DSK Board: The GUI will control several settings, summarized in Table I Physical variables such as microphone spacing and number of microphones need to be easily changed by software in real time, and the GUI provides an interface for users to the DSK to change the operation as required.

The byte stream will follow the same format as the DSK to PC stream. Data will be sent in 3 byte packets as shown in Figure 16. Packets will only be sent when there is new data, so we expect the number of packets per second to be very low.

We did not implement this due to time constraints.

2) Interface GUI: The interface software will be written in Java. An overview of the main classes is shown in Figure 17.

The Beamformer class handles the low-level translation and communication with the DSK board. It converts from sample delays to angles and vice-versa, converts booleans to code values, and floating-point values to integers for high-speed communication.

The BeamDisplay is a GUI class which displays the beam power levels and sound direction on a polar plot, similar to the simulation plots. It will use the Java AWT/Swing toolkit and associated drawing classes to implement a custom drawable canvas. Clicking on the display will select the direction to use when in spatial filtering mode.

The Settings panel will contain a series of standard Swing components to select the operating mode, number of microphones, microphone spacing, the number of beams per sweep. It will have a pointer to the Beamformer object, so it can configure it directly.

The MainWindow class creates the main GUI window, the display and control panels, and the connection to the DSK board. Once running, it will regularly poll the beamformer class for new data, and update the

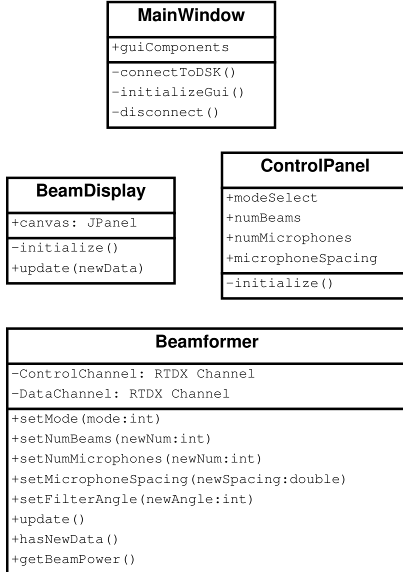

Figure 17. Class Diagram for the GUI

BeamDisplay accordingly.

VI. Test Methods

A. Source localization

To test performance of the source localization method we set up two microphones 25cm apart. Using a meter stick to keep a contsant radius, an amplified speaker was swept radially across the plane level with the microphones. A digital oscilloscope was used to measure the RMS voltage of the processed output. We also observed the beam displayed on the GUI.

B. Spatial filtering

Because of time constraints and unexpected difficulties with other parts of the project, we did not explicitly test spatial filtering.

To test the spatial filtering mode, we would have set up two sources at for different frequencies equidistant from the array with a 90 degree arc length seperation. The DSK would process the data with a beam formed towards one of the sources and feed the output was to a digital oscilloscope or MATLAB/Simulink GUI set to view the FFT. Using this method, we could compare the relative strengths of the signals.

VII. Results

Our system is able to do beamforming with 2 microphones, and display the result on a PC GUI. By observation, the beamwidth is about what we expected for a 2-microphone array.

We were unable to get any kind of beamforming to work with four microphones. After getting the 2-microphone system working, we switched to 4 by changing a `#define` in our code and connecting two additional microphones to the DSK expansion board. When we ran the GUI, we were unable to observe any kind of localization.

Each adjacent pair of microphones worked in a 2-microphone beamformer, so we believe that it was not a simple case of one microphone not working properly or a result of mixing up the order of the microphones. We performed a test where we merely averaged the inputs from the four microphones and sent that value to the output, which is equivalent to forming a beam perpendicular to the array. Using one of the speakers as an audio source and measuring the RMS voltage of the output on the oscilloscope, we manually checked the directionality of the array. We were unable to measure any significant difference between the sound source being directly orthogonal to the array and at a small angle to it. This indicates that at least part of the problem is not due to our software. There may have been additional problems with our software for four microphones, but because we were unable to get this simple test working, we did not test our software further.

VIII. Discussion

A. Sources of difficulty

We identified several factors which contributed to our system's failure to meet its requirements.

First, there were several acoustic problems. We found that the supplied microphones are very directional - they are specifically labeled "unidirectional". Because the beamforming system assumes omnidirectional input, unidirectional microphones will cause oblique signals to be attenuated. This is fine when the desired beam is perpendicular to the array, but is suddenly a problem when the goal is to amplify oblique signals over orthogonal ones.

The audio input level from the microphones seemed to be rather low, producing averaged output values in the single-millivolt range. We are not sure if there is a way to configure additional gain on the DSK board, or if using different microphones would help. Due to the low input level, we had to play the signal very loudly on the speakers and keep them within one or two meters of the array.

A 1-meter source distance clearly violates the farfield plane-wave assumption - our initial specification assumed that 4 meters was farfield. This may have caused some problems, but re-running our MATLAB simulation using a source only 1m away produced a beam virtually indistinguishable from a 10-meter beam.

We also experienced very strong echoes in the room, because of its cement construction and solid wood lab benches. Working in an environment with reduced echoes would probably have yielded better results. This was evident during late nights in the lab when people working in the ASAP room and janitorial staff would peek into the room to find the very loud noise echoing throughout the HSH.

On the software side, we experienced difficulties getting the bandwidth we desired over RTDX. I had also seen poor RTDX performance before, when sending data from a PC to the DSK during the cosine generation lab. There are several possible causes of this: One is that we only send a few bytes at a time, but do it dozens of times per second. Recommendations we found online suggested that filling the RTDX queue and sending large packets of data is the fastest way to send data.

Another likely cause is our DSK boards. We read a mailing list correspondence between someone who is familiar with this family of DSKs and a hapless hobbyist having similar issues. The "expert" claimed that there is obfuscation hardware on the DSK boards introduced by Spectrum Digital that slows down RTDX communication. It was mentioned that a possible workaround is using the JTAG connector on the DSK board, but the hardware required for this is not readily available (and the JTAG port is blocked by the DSP_AUDIO_4). We looked at high-speed RTDX, but it is not supported on this platform.

IX. Conclusion

Although we did not meet our initial specifications, our project was still somewhat successful. We were able to implement a working beamforming system, and most of the problems we encountered were limitations of our equiment which were not under our control.

References

[1] M. Brandstein and D. Ward, eds., *Microphone Arrays: Signal Processing Techniques and Applications*. Berlin, Germany: Springer, 2001.

[2] J. Benesty, J. Chen, and Y. Huang, *Microphone Array Signal Processing*. Berlin, Germany: Springer, 2008.

[3] J. Chen, L. Yip, J. Elson, H. Wang, D. Maniezzo, R. Hudson, K. Yao, and D. Estrin, "Coherent Acoustic Array Processing and Localization on Wireless Sensor Networks," *Proceedings of the IEEE*, vol. 91, no. 8, pp. 1154–1162, 2003.

[4] J. C. Chen, K. Yao, and R. E. Hudson, "Acoustic Source Localization and Beamforming: Theory and Practice," *EURASIP Journal on Applied Signal Processing*, pp. 359–370, 2003.

[5] T. Haynes, "A primer on digital beamforming," *Spectrum Signal Processing*, 1998.

[6] W. Herbordt and W. Kellermann, "Adaptive Beamforming for Audio Signal Acquisition," *Adaptive Signal Processing: Applications to Real-World Problems*, pp. 155–196, 2003.

[7] R. A. Kennedy, T. D. Abhayapala, and D. B. Ward, "Broadband nearfield beamforming using a radial beampattern transformation," *IEEE Transactions on Signal Processing*, vol. 46, no. 8, pp. 2147–2156, 1998.

[8] R. J. Lustberg, "Acoustic Beamforming Using Microphone Arrays," Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1993.

[9] N. Mitianoudis and M. E. Davies, "Using Beamforming in the Audio Source Separation Problem," in *7th Int Symp on Signal Processing and its Applications*, no. 2, 2003.

APPENDIX A
MATLAB CODE FOR BEAM-SWEEP SOURCE LOCALIZATION

```
1  % General setup
2  clear;
3
4  % Speed of sound at sea level
5  speedOfSound = 340.29; % m/s
6
7  % Source setup
8  % Frequency of the audio source
9  sourceFreq = 700; % Hz
10
11 % Just so we know - since this will affect our mic placement
12 wavelength = speedOfSound / sourceFreq;
13
14 % xy location of audio source
15 sourceLocX = 0; % in meters
16 sourceLocY = 10;
17
18 % Microphone setup
19 numMicrophones = 4;
20
21 % Distance between microphones - determines the highest frequency we can
22 % sample without spatial aliasing
23 micSpacing = 0.25; % meters
24
25 % Minimum amount of time it takes for sound to travel from one mic to the next
26 timeSpacing = micSpacing/speedOfSound;
27
28 % Total width of the array - determines the lowest frequency we can
29 % accurately locate
30 arrayWidth = (numMicrophones - 1) * micSpacing;
31
32 % xy locations of the microphones
33 micLocX = linspace(-arrayWidth/2, arrayWidth/2, numMicrophones); % in meters
34 micLocY = zeros(1, numMicrophones);
35
36
37 % Sampling rate
38 Fs = 44100; % Hz
39
40
41 % Distance from the source to the mic
42 propDistance = hypot(sourceLocX - micLocX, sourceLocY - micLocY);
43
44 timeDelay = propDistance/speedOfSound;
45
46
47 % Create some of the signal
48 soundTime = 0:1/Fs:.125;
49
50 sourceSignal = sin(2 * pi * sourceFreq * soundTime);
51
```

```
52 % Delay it by the propagation delay for each mic
53 for ii = 1:numMicrophones
54   received(ii,:) = delay(sourceSignal, timeDelay(ii), Fs);
55 end
56
57 % Plot the signals received by each microphone
58 % plot(received');
59
60 % Now it's time for some beamforming!
61 % Create an array of delays
62 numBeams = 60;
63 beamDelays = linspace(-timeSpacing, timeSpacing, numBeams);
64
65 for ii = 1:numBeams
66   summedInput = zeros(1, length(sourceSignal));
67   for jj = 1:numMicrophones
68     if(beamDelays(ii) >= 0)
69       % Delay the microphones by increased amounts as we go left to right
70       summedInput = summedInput + ...
71                     delay(received(jj,:), beamDelays(ii) * (jj - 1), Fs);
72     else
73       % If the beam delay is negative, that means we want to increase the
74       % delay as we go right to left.
75       summedInput = summedInput + ...
76         delay(received(jj,:), -beamDelays(ii) * (numMicrophones - jj), Fs);
77     end
78   end
79   % Calculate the power of this summed beam
80   power(ii) = sum(summedInput .^ 2);
81 end
82
83 % directions = linspace(pi, 0, numBeams);
84 directions = acos(linspace(-1, 1, numBeams));
85
86 polar(directions, power/max(power), '-b');
```

```
function [outputSignal] = delay(inputSignal, time, Fs)
% DELAY - Simulates a delay in a discrete time signal
%
% USAGE:
%   outputSignal = delay(inputSignal, time, Fs)
%
% INPUT:
%   inputSignal - DT signal to operate on
%   time - Time delay to use
%   Fs - Sample rate

if(time < 0)
  error('Time delay must be positive');
end

outputSignal = [zeros(1, floor(time * Fs)), inputSignal];

% Crop the output signal to the length of the input signal
outputSignal = outputSignal(1:length(inputSignal));
```

Appendix B
MATLAB code for spatial filtering

```
1
2
3  % General setup
4  clear;
5
6  % The frequencies we are interested in are:
7  % Right now this is optimized for sources between 100 and 500 Hz
8  % Change this to whatever range you are interested in.
9  fmin = 00;  % Hz
10 fmax = 700;% Hz
11
12 % Speed of sound at sea level
13 speedOfSound = 340.29; % m/s
14
15 % Source setup
16 % Frequency of the audio sources
17 source1Freq = 300; % Hz
18 source2Freq = 400; % Hz
19
20 % Just so we know - since this will affect our mic placement
21 wavelength1 = speedOfSound / source1Freq;
22 wavelength2 = speedOfSound / source2Freq;
23
24 degreesApart=60;
25 % xy location of audio source
26 source1LocX = sqrt(10^2 + 10^2)*cosd(45+degreesApart); % in meters
27 source1LocY = sqrt((10^2 + 10^2)-source1LocX^2);
28 source2LocX = 10; % Secondary source
29 source2LocY = 10;
30
31 % Microphone setup
32 numMicrophones = 2;
33
34 % Distance between microphones - determines the highest frequency we can
35 % sample without spatial aliasing
36 micSpacing = 0.25; % meters
37
38 % Minimum amount of time it takes for sound to travel from one mic to the next
39 timeSpacing = micSpacing/speedOfSound;
40
41 % Total width of the array - determines the lowest frequency we can
42 % accurately locate
43 arrayWidth = (numMicrophones - 1) * micSpacing;
44
45 % xy locations of the microphones
46 micLocX = linspace(-arrayWidth/2, arrayWidth/2, numMicrophones); % in meters
47 micLocY = zeros(1, numMicrophones);
48
49
50 % Sampling rate
51 Fs = 96000; % Hz
```

```
52
53
54 % Distance from the source to the mic
55 prop1Distance = hypot(source1LocX - micLocX, source1LocY - micLocY);
56 prop2Distance = hypot(source2LocX - micLocX, source2LocY - micLocY);
57
58 time1Delay = prop1Distance/speedOfSound;
59 time2Delay = prop2Distance/speedOfSound;
60
61 % Create some of the signal
62 soundTime = 0:1/Fs:.125;
63
64 source1Signal = sin(2 * pi * source1Freq * soundTime);
65 source2Signal = sin(2 * pi * source2Freq * soundTime);
66
67 % Delay it by the propagation delay for each mic
68 for ii = 1:numMicrophones
69   received(ii,:) = delay(source1Signal, time1Delay(ii), Fs)...
70       + delay(source2Signal, time2Delay(ii), Fs);
71 end
72
73
74 % Direct the beam towards a location of interest
75 angleWanted = 45; % Degrees (for simplicity)
76 angleToDelay = angleWanted * pi/180; % Convert to radian
77
78 % We want to take the fft of the signal only after everymicrophone is
79 % getting all of the data from all sources
80 deadAirTime = (max([time1Delay, time2Delay]));
81 deadAirSamples = deadAirTime * Fs;
82 endOfCapture = length(received(1,:));
83
84 % Start off with an empty matrix
85 formedBeam = zeros(1, max(length(received)));
86
87 % For each microphone add together the sound received
88 for jj = 1:numMicrophones
89     formedBeam = formedBeam + ...
90         delay(received(jj,:), + timeSpacing*sin(angleToDelay) * (jj-1), Fs);
91 end
92
93 % Get the PSD object using a modified covariance
94 beamPSD = psd(spectrum.mcov, formedBeam,'Fs',44100);
95 % Get the magnitude of the PSD
96 formedSpectrum = abs(fft(formedBeam));
97 % The fft sample # needs to be scaled to get frequency
98 fftScaleFactor = Fs/numel(formedSpectrum);
99
100 % The frequencies we are interested in are:
101 % Right now this is optimized for sources between 100 and 500 Hz
102 %fmin = 0;  % Hz
103 %fmax = 600;% Hz
104
105 % Plot the PSD of the received signal
```

```
106 figure(1);
107 % Get the frequencies and data out of the PSD object
108 beamPSDfreqs=beamPSD.Frequencies;
109 beamPSDdata = beamPSD.Data;
110 % Get the indeces that correspond to frequencies specified above
111 indexesToPlot=find(beamPSDfreqs>fmin-1):find(beamPSDfreqs>fmax,1);
112 % Actually plot it (in a log10 scale so we have dB)
113 plot(beamPSDfreqs(indexesToPlot),20*log10(beamPSDdata(indexesToPlot)));
114 title('PSD using a modified covariance of the received signal');
115 ylabel('Power/Frequency (dB)');
116 xlabel('Frequency (Hz)');
117 
118 % Plot the fft of our received signals
119 figure(2);
120 maxLimit=round(fmax/fftScaleFactor);
121 minLimit=round(fmin/fftScaleFactor);
122 if minLimit <= 0
123     minLimit = 1;
124 end
125 f=linspace(0,44100,44100/fftScaleFactor);
126 
127 % This gets all of the fft frequencies we want to look at
128 fOfInterest = f(minLimit:maxLimit);
129 % Grab the portion of the fft we want to look at
130 spectrumOfInterest = formedSpectrum(minLimit:maxLimit);
131 % Normalize this so that the max amplitude is at 0db.
132 spectrumOfInterest = spectrumOfInterest/max(formedSpectrum);
133 % Plot it
134 plot(fOfInterest,20*log10(spectrumOfInterest));
135 title('FFT of the received signal');
136 ylabel('Relative magnitude of signal (dB)');
137 xlabel('Frequency (Hz)');
```

APPENDIX C
C CODE FOR MAIN PROCESSING LOOP

```
1
2  void processing(){
3    Int16 twoMicSamples[2];
4
5    Int32 totalPower; // Total power for the data block, sent to the PC
6    Int16 tempPower; // Value used to hold the value to be squared and added
7    Int16 printCount; // Used to keep track of the number of loops since we last sent data
8    unsigned char powerToSend;
9
10   Int8 firstLocalizationLoop; // Whether this is our first loop in localization mode
11   Int8 delay; // Delay amount for localization mode
12
13         QUE_McBSP_Msg tx_msg,rx_msg;
14         int i = 0; // Used to iterate through samples
15
16   RTDX_enableOutput( &dataChan );
17
18         while(1){
19
20                 SEM_pend(&SEM_McBSP_RX,SYS_FOREVER);
21
22                 // Get the input data array and output array
23                 tx_msg = QUE_get(&QUE_McBSP_Free);
24                 rx_msg = QUE_get(&QUE_McBSP_RX);
25
26     // Spatial filter mode
27
28     // Localization mode
29     if(firstLocalizationLoop){
30       delay = DELAYMIN;
31       firstLocalizationLoop = FALSE;
32     }
33     else{
34       delay++;
35       if(delay > DELAYMAX){
36         delay = DELAYMIN;
37       }
38     }
39
40                 // Process the data here
41     /* MIC1R data[0]
42      * MIC1L data[1]
43      * MIC0R data[2]
44      * MIC0L data[3]
45      *
46      * OUT1 Right - data[0]
47      * OUT1 Left - data[1]
48      * OUT0 Right - data[2]
49      * OUT0 Left - data[3]
50      */
51
```

```
52      totalPower = 0;
53
54                  for(i=0; i<QUE_McBSP_LEN; i++){
55
56        // Put the array elements in order
57        twoMicSamples[0] = *(rx_msg->data + i*4 + 2);
58        twoMicSamples[1] = *(rx_msg->data + i*4 + 3);
59
60        tx_msg->data[i*4] = 0;
61        tx_msg->data[i*4 + 1] = 0;
62        tx_msg->data[i*4 + 2] = sum(twoMicSamples, delay);
63        tx_msg->data[i*4 + 3] = 0;
```

APPENDIX D
C CODE FOR SUMMING DELAYS

Listing 1. sum.h

```
#ifndef _SUM_H_BFORM_
#define _SUM_H_BFORM_
extern Int16 sum(Int16* newSamples, int delay);
#endif
```

Listing 2. sum.c

```

#include <std.h>

#include "definitions.h"
#include "calcDelay.h"

/* newSamples is an array with each of the four samples.
 * delayIncrement is the amount to delay each microphone by, in samples.
 * This function returns a "beamed" sample. */
Int16 sum(Int16* newSamples, int delayIncrement){
        static int currInput = 0; // Buffer index of current input
        int delays[NUMMICS]; // Amount to delay each microphone by
        int mic = 0; // Used as we iterate through the mics
        int rolloverIndex;
        Int16 output = 0;
        static Int16 sampleBuffer[NUMMICS][MAXSAMPLEDIFF];

        // Calculate samples to delay for each mic
  // TODO: Only do this once
        calcDelay(delayIncrement, delays);

  // We used to count backwards - was there a good reason?
        currInput++; // Move one space forward in the buffer

  // Don't run off the end of the array
  if(currInput >= MAXSAMPLEDIFF){
    currInput = 0;
  }

        // Store new samples into sampleBuffer
        for(mic=0; mic < NUMMICS; mic++){
                // Divide by the number of microphones so it doesn't overflow
    // when we add them
                sampleBuffer[mic][currInput] = newSamples[mic]/NUMMICS;
        }

        // For each mic add the delayed input to the current output
        for(mic=0; mic < NUMMICS; mic++){
                if(currInput - delays[mic] >= 0){// Index properly?
                        output += sampleBuffer[mic][currInput - delays[mic]];
                }
                else{
      // The delay index is below 0, so add the length of the array
      // to keep it in bounds
```

```
45                         rolloverIndex = MAXSAMPLEDIFF + (currInput - delays[mic]);
46                         output += sampleBuffer[mic][rolloverIndex];
47                 }
48         }
49
50         return output;
51
52 }
```

Appendix E
C code for calculating delays

Listing 3. calcDelay.h

```
1 #ifndef _CALC_DELAY_H_
2 #define _CALC_DELAY_H_
3 extern void calcDelay(int delayInSamples, int* delays);
4 #endif
```

Listing 4. calcDelay.c

```
1  /* calcDelay
2   * Accepts delays in samples as an integer
3   * and returns a pointer to an array of delays
4   *  for each microphone.
5   *
6   *  Date: 9 March 2010
7   */
8
9  #include "definitions.h"
10
11 void calcDelay(int delayInSamples, int* delays){
12         int mic = 0;
13         if(delayInSamples > 0){
14                 for(mic=0; mic < NUMMICS; mic++){
15                         delays[mic] = delayInSamples*mic;
16                 }
17         }
18         else{
19                 for(mic=0; mic < NUMMICS; mic++){
20                         delays[mic] = delayInSamples*(mic-(NUMMICS-1));
21                 }
22         }
23 }
```

Appendix F
Java Code - Main

Listing 5. MainWindow.java

```
1 /* MainWindow.java - Java class which constructs the main GUI window
2  * for the project, and sets up communication with the DSK.  It also
3  * contains the main() method.
4  *
5  * Author: Steven Bell and Nathan West
6  * Date: 9 March 2010
7  * $LastChangedBy$
8  * $LastChangedDate$
```

```
9   */
10
11 package beamformer;
12
13 import java.awt.*; // GUI Libraries
14
15 import javax.swing.*;
16
17 public class MainWindow
18 {
19   JFrame window;
20   DisplayPanel display;
21   ControlPanel controls;
22   static Beamformer beamer;
23
24   MainWindow()
25   {
26     beamer = new Beamformer();
27
28     window = new JFrame("Acoustic␣Beamforming␣GUI");
29     window.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
30     window.getContentPane().setLayout(new BoxLayout(window.getContentPane(), BoxLayout.LIN
31
32     display = new DisplayPanel();
33     display.setPreferredSize(new Dimension(500,500));
34     display.setAlignmentY(Component.TOP_ALIGNMENT);
35     window.add(display);
36
37     controls = new ControlPanel();
38     controls.setAlignmentY(Component.TOP_ALIGNMENT);
39     window.add(controls);
40
41     window.pack();
42     window.setVisible(true);
43
44     beamer.start();
45   }
46
47   public static void main(String[] args)
48   {
49     MainWindow w = new MainWindow();
50     while(true){
51       beamer.update();
52       w.display.updatePower(beamer.getPower());
53     }
54   }
55 }
```

Appendix G
Java Code for Beamformer Communication

Listing 6. Beamformer.java

```
1 /* Beamformer.java - Java class which interacts with the DSK board
2  * using RTDX.
3  *
```

```
4    * Author: Steven Bell and Nathan West
5    * Date: 11 April 2010
6    * $LastChangedBy$
7    * $LastChangedDate$
8    */
9
10  package beamformer;
11
12  //Import the DSS packages
13  import com.ti.ccstudio.scripting.environment.*;
14  import com.ti.debug.engine.scripting.*;
15
16  public class Beamformer {
17
18    DebugServer debugServer;
19    DebugSession debugSession;
20
21    RTDXInputStream inStream;
22
23    int[] mPowerValues;
24
25    public Beamformer() {
26      mPowerValues = new int[67]; // TODO: Change to numBeams
27
28      ScriptingEnvironment env = ScriptingEnvironment.instance();
29      debugServer = null;
30      debugSession = null;
31
32      try
33      {
34        // Get the Debug Server and start a Debug Session
35        debugServer = (DebugServer) env.getServer("DebugServer.1");
36        debugServer.setConfig("Z:/2010_Spring/dsp/codecomposer_workspace/dsp_project_trunk/ds
37        debugSession = debugServer.openSession(".*");
38
39        // Connect to the target
40        debugSession.target.connect();
41        System.out.println("Connected␣to␣target.");
42
43        // Load the program
44        debugSession.memory.loadProgram("Z:/2010_Spring/dsp/codecomposer_workspace/dsp_projec
45        System.out.println("Program␣loaded.");
46
47        // Get the RTDX server
48        RTDXServer commServer = (RTDXServer)env.getServer("RTDXServer.1");
49        System.out.println("RTDX␣server␣opened.");
50
51        RTDXSession commSession = commServer.openSession(debugSession);
52
53        // Set up the RTDX input channel
54        inStream = new RTDXInputStream(commSession, "dataChan");
55        inStream.enable();
56      }
57      catch (Exception e)
```

```
58        {
59          System.out.println(e.toString());
60        }
61      }
62
63      public void start()
64      {
65        // Start running the program on the DSK
66        try{
67          debugSession.target.restart();
68          System.out.println("Target␣restarted.");
69          debugSession.target.runAsynch();
70          System.out.println("Program␣running....");
71          Thread.currentThread().sleep(1000); // Wait a second for the program to run
72        }
73        catch (Exception e)
74        {
75          System.out.println(e.toString());
76        }
77      }
78
79      public void stop()
80      {
81        // Stop running the program on the DSK
82        try{
83          debugSession.target.halt();
84          System.out.println("Program␣halted.");
85        }
86        catch (Exception e)
87        {
88          System.out.println(e.toString());
89        }
90
91      }
92
93      public void setMode()
94      {
95
96      }
97
98      public void setNumBeams()
99      {
100
101     }
102
103     public void update(){
104       try{
105         // Read some bytes
106         byte[] power = new byte[1];
107         byte[] delay = new byte[1];
108
109         inStream.read(delay, 0, 1, 0); // Read one byte, wait indefinitely for it
110         inStream.read(power, 0, 1, 0); // Read one byte, wait indefinitely for it
111
```

```
112       int intPower = power[0] & 0xFF; // Convert to int value from unsigned byte
113 
114       mPowerValues[delay[0] + 33] = intPower; // Save them
115 
116       System.out.println("D:" + (int)delay[0] + "  P: " + intPower);
117     }
118     catch (Exception e)
119     {
120       System.out.println(e.toString());
121     }
122   } // END update()
123 
124   public int[] getPower()
125   {
126     return mPowerValues;
127   }
128 
129   /*
130   public int[] getDelays()
131   {
132     return int
133   }*/
134 
135 } // END class
```

Appendix H
Java Code for Beam Display

Listing 7. DisplayPanel.java

```
1  /* DisplayPanel.java - Java class which creates a canvas to draw the
2   * beamformer output on and handles the drawing.
3   *
4   * Author: Steven Bell and Nathan West
5   * Date: 9 March 2010
6   * $LastChangedBy$
7   * $LastChangedDate$
8   */
9 
10 package beamformer;
11 
12 import java.awt.*; // GUI Libraries
13 import javax.swing.*;
14 import java.lang.Math;
15 
16 public class DisplayPanel extends JPanel
17 {
18 
19   int mDelay[];// = {-3, -2, -1, 0, 1, 2, 3};
20   double mPower[];// = {0, .25, .5, 1, .5, .25, 0};
21   int mNumPoints = 67;
22 
23   double mSpacing = 33;
24 
25   DisplayPanel()
26   {
```

```
27      mDelay = new int[mNumPoints];
28      mPower = new double[mNumPoints];
29
30      for(int i = 0; i < mNumPoints; i++){
31        mDelay[i] = i - 33;
32        mPower[i] = .5;
33      }
34    }
35
36    void updatePower(int[] newPower)
37    {
38      for(int i = 0; i < mNumPoints; i++)
39      {
40        mPower[i] = (double)newPower[i] / 255;
41        if(mPower[i] > 1){
42          mPower[i] = 1;
43        }
44      }
45      this.repaint();
46    }
47
48    static final int FIGURE_PADDING = 10;
49
50    // Overrides paintComponent from JPanel
51    protected void paintComponent(Graphics g)
52    {
53      super.paintComponent(g);
54
55      Graphics2D g2d = (Graphics2D)g; // Cast to a Graphics2D so we can do more advanced pai
56      g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_O
57
58      // Determine the maximum radius we can use.  It will either be half
59      // of the width, or the full height (since we do a semicircular plot).
60      int maxRadius = this.getWidth()/2;
61      if(maxRadius > this.getHeight()){
62        maxRadius = this.getHeight();
63      }
64      maxRadius = maxRadius - FIGURE_PADDING;
65
66      // Pick our center point
67      int centerX = this.getWidth() / 2;
68      int centerY = this.getHeight()-(this.getHeight() - maxRadius) / 2;
69
70      // Calculate all of the points
71      int[] px = new int[mNumPoints];
72      int[] py = new int[mNumPoints];
73
74      for(int i = 0; i < mNumPoints; i++)
75      {
76        double angle = delayToAngle(mDelay[i]);
77        px[i] = centerX - (int)(maxRadius * mPower[i] * Math.sin(angle));
78        py[i] = centerY - (int)(maxRadius * mPower[i] * Math.cos(angle));
79      }
80
```

```
81      g2d.setPaint(Color.BLUE);
82      g2d.drawPolygon(px, py, mNumPoints);
83
84      // Draw the outline of the display, so we have some context
85      g2d.setPaint(Color.BLACK);
86      float[] dash = {5, 4};
87      g2d.setStroke(new BasicStroke((float).5, BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER,
88      g2d.drawLine(10, centerY, this.getWidth() - 10, centerY);
89      g2d.drawArc(10, (this.getHeight() - maxRadius) / 2, 2*maxRadius, 2*maxRadius, 0, 180);
90    }
91
92    // Takes a delay and converts it to the equivalent in radians
93    private double delayToAngle(int delay)
94    {
95      return(Math.PI/2 - Math.acos(delay/mSpacing));
96    }
97
98    // Takes an angle in radians (-pi/2 to +pi/2) and converts it to a delay
99    private int angleToDelay(double angle)
100   {
101     return(int)(mSpacing * Math.cos(angle + Math.PI/2));
102   }
103 }
```