Reduction of Near-Field Grating Lobes in Sparse Acoustic Phased Arrays

Dylan Rudolph
Embry-Riddle Aeronautical University - Daytona Beach

Follow this and additional works at: https://commons.erau.edu/edt

Part of the Electrical and Computer Engineering Commons

Scholarly Commons Citation
https://commons.erau.edu/edt/125

This Thesis - Open Access is brought to you for free and open access by Scholarly Commons. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of Scholarly Commons. For more information, please contact commons@erau.edu.
Reduction of Near-Field Grating Lobes in Sparse Acoustic Phased Arrays

Author: Dylan Rudolph

Completed: April 2013

Submitted in partial fulfillment of a Master of Science in Electrical and Computer Engineering
Reduction of Near-Field Grating Lobes in Sparse Acoustic Phased Arrays

by: Dylan Rudolph

This thesis was prepared under the direction of the candidate’s thesis committee chairman, Dr. William Barott, of the Electrical, Computer, Software, and Systems Engineering Department, and has been approved by the members of his thesis committee. It was submitted to the Electrical, Computer, Software, and Systems Engineering Department and was accepted in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering.

THESIS COMMITTEE:

Dr. William Barott
Committee Chairman

Dr. Richard Stansbury
Committee Member

Dr. Timothy Wilson
Committee Member & Department Chair
Electrical, Computer, Software, and Systems Engineering

Associate Vice President For Academics
Abstract

Reduction of Near-Field Grating Lobes
in Sparse Acoustic Phased Arrays

Dylan Rudolph

Acoustic phased arrays with inter-element spacings of greater than one half-wavelength will produce grating lobes that decrease the usefulness of the array. With many array configurations, the near-field and far-field character of these lobes is significantly different — an optimization of the array configuration to reduce grating lobes in the far-field can have little bearing on the performance of the array in the near-field.

The focus of this thesis is the reduction of grating lobes in the near-field by iterative optimization. Genetic algorithms are employed to choose inter-element spacings which are able to reduce the magnitude of grating lobes at select distances in the near-field. The genetic algorithm produced configurations that are theoretically able to suppress grating lobes in the near-field.

In order to verify the efficacy of these configurations, a hardware test platform was constructed. The platform permits largely automated evaluation of the near-field character of arbitrary array configurations. Using the test platform, several of the arrangements were constructed and evaluated. The results of these evaluations confirm that it is possible to reduce the grating lobes of sparse phased arrays in the near-field.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>Abbreviations</td>
<td>vi</td>
</tr>
<tr>
<td>Symbols</td>
<td>vii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Scope of Works</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Research Questions</td>
<td>2</td>
</tr>
<tr>
<td><strong>2 Background and Theory</strong></td>
<td>3</td>
</tr>
<tr>
<td>2.1 Phased Arrays</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Acoustic Elements</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Genetic Algorithms</td>
<td>11</td>
</tr>
<tr>
<td>2.4 Previous Research</td>
<td>12</td>
</tr>
<tr>
<td><strong>3 Hardware and Software</strong></td>
<td>14</td>
</tr>
<tr>
<td>3.1 Hardware</td>
<td>15</td>
</tr>
<tr>
<td>3.1.1 Electronics</td>
<td>15</td>
</tr>
<tr>
<td>3.1.2 Acoustics and Mechanical</td>
<td>27</td>
</tr>
<tr>
<td>3.2 Software</td>
<td>29</td>
</tr>
<tr>
<td>3.2.1 FPGA Firmware</td>
<td>29</td>
</tr>
<tr>
<td>3.2.2 Hardware Management</td>
<td>30</td>
</tr>
<tr>
<td>3.2.3 Simulation and Phased Array Related</td>
<td>31</td>
</tr>
<tr>
<td>3.2.4 Optimization</td>
<td>32</td>
</tr>
<tr>
<td><strong>4 Methodology</strong></td>
<td>33</td>
</tr>
<tr>
<td>4.1 Genetic Algorithm Specifications</td>
<td>33</td>
</tr>
<tr>
<td>4.2 Testing Methodology and Procedure</td>
<td>37</td>
</tr>
<tr>
<td><strong>5 Results and Analysis</strong></td>
<td>40</td>
</tr>
<tr>
<td>5.1 Theoretical Optimized Results</td>
<td>40</td>
</tr>
<tr>
<td>5.1.1 Simulated 2.25 Wavelength Minimum Spacing</td>
<td>41</td>
</tr>
</tbody>
</table>
5.1.2 Simulated 1.75 Wavelength Minimum Spacing ............................................. 42
5.1.3 Simulated 1.25 Wavelength Minimum Spacing ............................................. 43
5.1.4 Simulated 0.75 Wavelength Minimum Spacing ............................................. 44
5.2 Measured Optimized Results ........................................................................... 44
5.3 Imaging Sonar Application Results ................................................................. 47
5.4 Possible Sources of Error ................................................................................. 50

6 Conclusions ........................................................................................................ 58
6.1 Discussion of Results ....................................................................................... 58
6.2 Future Work ..................................................................................................... 59
6.3 Closing Thoughts ............................................................................................. 59

A Approximate Hardware Costs .......................................................................... 60

B Low Level Software .......................................................................................... 61
B.1 File: comm.py ................................................................................................ 61
B.2 File: lowlevelfunctions.py ........................................................................... 68

C Simulation & Other Software ........................................................................... 91
C.1 File: fastsim.py ............................................................................................... 91
C.2 File: highlevelfunctions.py .......................................................................... 96

D Optimization & Analysis Software ................................................................ 104
D.1 File: genetic.py ............................................................................................. 104
D.2 File: analysis.py ........................................................................................... 107

Bibliography .......................................................................................................... 110
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Wavefront Propagation Visualization</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Comparison of Several Phased Array Element Counts</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Pressure Magnitude Visualization</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>Comparison of Logarithmic and Linear Phased Arrays</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>Phased Array Focusing Visualization</td>
<td>9</td>
</tr>
<tr>
<td>2.6</td>
<td>Directional Acoustic Array Response</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>System Diagram</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Data Converter Block Diagrams</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>ADC Evaluation PCB</td>
<td>20</td>
</tr>
<tr>
<td>3.4</td>
<td>DAC Evaluation PCB</td>
<td>21</td>
</tr>
<tr>
<td>3.5</td>
<td>Finalized ADC and DAC PCB</td>
<td>22</td>
</tr>
<tr>
<td>3.6</td>
<td>Finalized VHDCI-Interface Board</td>
<td>23</td>
</tr>
<tr>
<td>3.7</td>
<td>Image of Electronics Assembly</td>
<td>24</td>
</tr>
<tr>
<td>3.8</td>
<td>Image of Electronics Assembly</td>
<td>25</td>
</tr>
<tr>
<td>3.9</td>
<td>Image of Electronics Assembly</td>
<td>26</td>
</tr>
<tr>
<td>3.10</td>
<td>Transducer Array</td>
<td>27</td>
</tr>
<tr>
<td>3.11</td>
<td>Pan-Tilt Assembly</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Measured Array Response</td>
<td>39</td>
</tr>
<tr>
<td>5.1</td>
<td>Simulated 2.25$\lambda$ Minimum-Spacing Optimized Configurations</td>
<td>41</td>
</tr>
<tr>
<td>5.2</td>
<td>Simulated 1.75$\lambda$ Minimum-Spacing Optimized Configurations</td>
<td>42</td>
</tr>
<tr>
<td>5.3</td>
<td>Simulated 1.25$\lambda$ Minimum-Spacing Optimized Configurations</td>
<td>43</td>
</tr>
<tr>
<td>5.4</td>
<td>Simulated 0.75$\lambda$ Minimum-Spacing Optimized Configurations</td>
<td>44</td>
</tr>
<tr>
<td>5.5</td>
<td>Measured 1.75$\lambda$ Minimum-Spacing Results</td>
<td>45</td>
</tr>
<tr>
<td>5.6</td>
<td>Imaging Sonar Setup</td>
<td>48</td>
</tr>
<tr>
<td>5.7</td>
<td>Imaging Sonar Periodic Configuration Evaluation</td>
<td>49</td>
</tr>
<tr>
<td>5.8</td>
<td>Imaging Sonar Aperiodic Configuration Evaluation</td>
<td>49</td>
</tr>
<tr>
<td>5.9</td>
<td>Simulated Element Positioning Error</td>
<td>51</td>
</tr>
<tr>
<td>5.10</td>
<td>Simulated Off-Broadside Phase Variation Error</td>
<td>53</td>
</tr>
<tr>
<td>5.11</td>
<td>Simulated Noise Error</td>
<td>55</td>
</tr>
<tr>
<td>5.12</td>
<td>Simulated Combination of all Errors</td>
<td>56</td>
</tr>
</tbody>
</table>
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Analog to Digital Converter</td>
</tr>
<tr>
<td>AWGN</td>
<td>Additive White Gaussian Noise</td>
</tr>
<tr>
<td>AFE</td>
<td>Analog Front End</td>
</tr>
<tr>
<td>BGA</td>
<td>Ball Grid Array</td>
</tr>
<tr>
<td>DAC</td>
<td>Digital to Analog Converter</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>GA</td>
<td>Genetic Algorithm</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>PCB</td>
<td>Printed Circuit Board</td>
</tr>
<tr>
<td>RMS</td>
<td>Root Mean Square</td>
</tr>
<tr>
<td>RSLN</td>
<td>Relative Side Lobe Level</td>
</tr>
<tr>
<td>SNR</td>
<td>Signal to Noise Ratio</td>
</tr>
<tr>
<td>SQNR</td>
<td>Signal to Quantization Noise Ratio</td>
</tr>
<tr>
<td>SPS</td>
<td>Samples Per Second</td>
</tr>
<tr>
<td>VHDCI</td>
<td>Very High Density Cable Interconnect</td>
</tr>
</tbody>
</table>
# Symbols

<table>
<thead>
<tr>
<th>Description</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>Meters</td>
</tr>
<tr>
<td>$c$</td>
<td>Meters per Second</td>
</tr>
<tr>
<td>$f$</td>
<td>Hertz</td>
</tr>
<tr>
<td>$t$</td>
<td>Seconds</td>
</tr>
<tr>
<td>$P$</td>
<td>Pascals</td>
</tr>
<tr>
<td>$\theta$</td>
<td>Radians / Degrees</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Unitless: $\alpha \geq 0$</td>
</tr>
<tr>
<td>$A_n$</td>
<td>Amplitude per Distance</td>
</tr>
<tr>
<td>$D_{xyn}$</td>
<td>Meters</td>
</tr>
<tr>
<td>$R_{xyn}$</td>
<td>Unitless: $0 \leq R_{xyn} \leq 1$</td>
</tr>
<tr>
<td>$d_n$</td>
<td>Meters</td>
</tr>
<tr>
<td>$\Delta X_i$</td>
<td>Meters</td>
</tr>
<tr>
<td>$\Delta \tau_n$</td>
<td>Seconds</td>
</tr>
<tr>
<td>$\max{x}$</td>
<td></td>
</tr>
<tr>
<td>$\forall$</td>
<td>“for all”</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Scope of Works

The focus of this thesis is the reduction of grating lobe magnitude in the near-field of sparse phased arrays by means of optimal element placement. To the author’s knowledge, this is a novel task; see Section 2.4 for additional details. Most of the analysis is generalized as to apply to both electromagnetic and acoustic phased arrays, but acoustic phased arrays are the primary interest.

Consideration is not given to other means of reducing the magnitude of grating lobes, such as unequal sizing, weighting, or shape of the elements. However, the techniques described here may be able to augment other methods of optimization.

1.2 Motivation

It is not always possible to arrange phased arrays such that the spacing between their elements is less than one half of their operating wavelength. This may be due to manufacturing constraints, or the need for a large aperture. Under this circumstance, effectiveness-robbing grating lobes will be produced.

In the case of an imaging sonar — the primary application focus of this thesis — grating lobes will cause ghosting and directional ambiguity. This is due to the inability of a beam-forming
system to effectively distinguish acoustic signals originating in the direction the grating lobes from those originating in the direction of the main lobe. See Section 5.3 for further discussion.

Reduction of grating lobes can be accomplished by cleverly choosing an aperiodic arrangement of the elements. Most methods of choosing an aperiodic spacing of the elements attempt to reduce grating lobes on the far-field — at distances much greater than the size of the array. For many applications, this is not a problem. However, for some phased array imaging applications, it may be necessary to operate in the near-field.

If it can be shown that it is possible to optimize sparse arrays to work reasonably well in the near-field, device manufacturing constraints can be relaxed, and devices with larger apertures can be produced. Larger apertures could permit greater transmit power and resolution.

Among the possible application fields are: medical ultrasound, mechanical test and inspection, architectural surveying, millimeter-wave antenna applications, and maritime/terrestrial acoustic imaging.

1.3 Research Questions

This thesis hopes to answer the following:

1. Is it possible to arrange the elements of a sparse phased array such that the array can be operated effectively in the near-field?

2. If the array is optimized to work well in the near-field, to what extent does it lose performance in the far-field?

3. How do the physical parameters of the array affect its performance in the near-field?

Additionally, it would be useful to find the cutoff points of physical array parameters at which the array no longer provides a certain level of performance. In order to do this, several representative test cases must be constructed.
Chapter 2

Background and Theory

2.1 Phased Arrays

Phased arrays, acoustic or electromagnetic, can be used to produce a desired beam directivity pattern [1]. This pattern can be produced in both transmitting and receiving arrays. The exact character of the directivity of the array is dependent on a number of its physical parameters. Phased arrays can be designed to produce a high radiated power in certain directions, while reducing the radiated power in other directions. They can also be designed to be particularly receptive to signals in one direction, while suppressing signals in another direction.

Though phased arrays are not limited to being arranged along a line — they could have any physically-realizable configuration — the analysis here will focus on line array configurations. A line array is sufficient for altering the directivity along one angular axis. A planar or three-dimensional array is required to control the directivity along both angular axes.

The principle on which acoustic phased arrays operate is the constructive and destructive interference caused by the superposition of propagating pressure waves. Similarly, the principle on which electromagnetic phased arrays operate is the superposition of propagating electric and magnetic fields.

The beam pattern of a phased array can be steered in different angular directions by introducing a phase delay in the elements. Figure 2.1 demonstrates this steering — a cascaded
delay in the elements causes the propagation of the combined wavefront to be directed. If Figure 2.1 depicted the pattern of an acoustic array, the different colors would indicate different pressure magnitudes. If it depicted the pattern of an electromagnetic phased array, the different colors would indicate the magnetic or electric field induced by the antennas.

The required time delay $\Delta \tau_n$ for element $n$ to produce a given steering angle $\theta$ is:

$$\Delta \tau_n = -\frac{d_n \sin \theta}{c}$$  \hspace{1cm} (2.1)

Where $c$ is the wave propagation speed in the medium, and $d_n$ is the distance between the element and some reference point along the array axis. In Figure 2.1, the reference point is declared to be the first element, so $d_2$ is the distance between element 2 and element 1.

---

1. All color-graded plots are colored such that blue indicates greater values, and red indicates lesser values.
2. This notation is common in papers concerned with underwater acoustics.
In two-dimensional Cartesian coordinates, the pressure $P(x, y, t)$ resulting from a single point source at a given time $t$ and point $(x, y)$ is:

$$P(x, y, t) = \frac{A}{D_{xy}} \cos \left( 2\pi f \left( t - \Delta \tau - \frac{D_{xy}}{c} \right) \right)$$

(2.2)

Where $A$ is an amplitude scaling factor for the point source, $f$ is the operating frequency of the source, $\Delta \tau$ is an induced steering delay, $c$ is the wave propagation speed, and $D_{xy}$ is the distance from the location of the source to the point $(x, y)$. Note that the amplitude scales inversely proportional to distance — the so-called one over $r$ law. An evaluation of Equation 2.2 is shown in part A of Figure 2.2.

![Figure 2.2](image)

**Figure 2.2:** Visualization of pressure fields as described in Equation 2.2 at one point in time, for several different numbers of elements. The color axis is scaled logarithmically to better illustrate the propagating wavefronts. Higher pressure magnitudes are shown in blue, lower pressure magnitudes are shown in red. All elements have the same $A$, $\Delta \tau$, and $f$.

Because the pressure as the result of many point sources will be the combined effect of each of the point sources, we can conclude that the pressure at any point $(x, y, t)$ for $N$ unique
point sources operating in a linear medium will then be:

\[ P(x, y, t) = \sum_{n=0}^{N} \left[ \frac{A_n}{D_{xyn}} \cos \left( 2\pi f \left( t - \Delta \tau_n - \frac{D_{xyn}}{c} \right) \right) \right] \]  

(2.3)

Where the amplitude scaling, steering delay, and location can be different for each element. Figure 2.2 is a visualization of the complex interactions that permit phased arrays to function. Because Equation 2.3 has a time-varying component, it does not lend itself well to the evaluation of the radiation pattern of an array. In order to resolve this, we can evaluate the pressure field over all time and find the maximum value. That is:

\[ P(x, y) = \max \{ P(x, y, t) \} \forall t \]  

(2.4)

Figure 2.3 is an example of this type of evaluation. Because the pressure at a given point will be periodic, it is only necessary to evaluate over one period.

Figure 2.3: Visualization of the pressure magnitude field and grating lobes of a twelve-element line array for two different inter-element spacings. Blue areas indicate greater pressure magnitude.

A sparse phased array is defined as one where the elements are spaced more than one half-wavelength (\( \lambda \)) apart. Sparse arrays suffer from grating lobes — side lobes with magnitude
equal to the main lobe, but different angular location. The angular location of the side lobes is dependent on the spacing between the elements. Wider element spacing causes side-lobes that are angularly closer to the main lobe. Figure 2.3 demonstrates this.

More generally, grating lobes will appear at angle \( \theta_l \) according to:

\[
\cos(\theta_l) - \cos(\theta_0) = \frac{n \lambda}{d}
\]

(2.5)

Where \( \theta_0 \) is the steered angle of the main lobe, and \( n \) is the set of integers. If we wanted the grating lobes to be no less than 90° off-axis, we would need an inter-element spacing of no more than \( \lambda \).

A more convenient method of plotting the character of side lobes is to plot the angle off-broadside along the horizontal axis, and the reduction from the maximum lobe along the vertical axis, for a given distance \( r \). That is:

\[
RPV(\theta) = 20 \log_{10} \left( P(r, \theta) \right) - \max \left( 20 \log_{10} \left( P(r, \theta) \right) \forall \theta \right)
\]

(2.6)

Where \( RPV(\theta) \) is the Reduction from Peak Value, given in decibels, and \( P(r, \theta) \) is the circular coordinate-system equivalent of \( P(x, y) \) as defined in Equation 2.4. This representation is shown in Figure 2.4.

The goal of an optimized phased array is to reduce the Relative Side Lobe Levels (RSLL). Figure 2.4 shows the RSLL for one angle. Under all conditions in which the main lobe is the lobe of greatest amplitude, \( RSLL(\theta) = RPV(\theta) \) for all \( \theta \) outside the main lobe. For \( \theta \) in the main lobe, the RSLL is undefined.

Because the RSLL is defined in terms of the main lobe amplitude, not the maximum value, if the main lobe is not the strongest lobe, the RSLL can have positive values, unlike \( RPV(\theta) \).

The Maximum RSLL is defined as: \( \max \{ RSLL(\theta) \forall \theta \} \).

For a sparse phased array, an aperiodic spacing of elements is required to reduce the Maximum RSLL. A logarithmic array configuration is one type of aperiodic configuration that is capable of reducing grating lobes in the far field; it has an inter-element spacing of 2:

\[
\Delta X_i = (1 + \alpha) \Delta X_{i-1}
\]

(2.7)
Where $\Delta X_i$ is the distance between element $i$ and the neighboring previous element, and $\alpha$ is a constant which determines the degree to which the elements are spaced out. The effect of a logarithmic array on the far-field grating lobes is shown in the blue line of Figure 2.4.

The boundary condition between near-field distances and far-field distances should be noted. For large arrays, this transition distance is the Fraunhofer distance, defined as:

$$d_f = \frac{2D^2}{\lambda}$$

(2.8)

Where $d_f$ is the Fraunhofer distance, $D$ is the maximum size of the array, and $\lambda$ is the wavelength of the radiator. For a sense of scale, an evaluation of Equation 2.8 for $D = 0.3$ meters and $\lambda = 0.0085$ meters, which corresponds to a 40 kHz transducer in air, yields:

$$d_f = \frac{2(0.3)^2}{0.0085} = 21.2 \text{ meters}$$

(2.9)
Chapter 2. Background and Theory

The phased arrays in Figure 2.4 are focused at infinity. That is: they produce a pattern which is designed to operate in the far field. This does not need to be the case. It is possible to apply delays to each of the elements such that they produce their maximum constructive interference at a single point in space. This is known as focusing, and is no different than the focusing of conventional visible-light optics.

![Figure 2.5: Comparison of the focusing performance of three different sixteen-element arrays in the near field. The yellow dots indicate the nominal focus points. The circles indicate the points of evaluation in the bottom plot. [Left, Black] is linear array with an inter-element spacing of $1.25\lambda$. [Middle, Blue] is a logarithmic array with an average spacing of $1.25\lambda$ and $\alpha = 0.05$. [Right, Green] is also a linear array with an inter-element spacing of $1.25\lambda$, now focused at a point off-axis.](image)

Figure 2.5 demonstrates the effect that the focusing of arrays can have on their response in the near field. There are several ways to determine the delays required at each element to focus to a certain point. One programmatically simple way to do so is to set time delays for each of the elements so that they will all arrive at the same time as the first element. This can be calculated based on the distances between the elements and the focusing point $(x, y)$. That is:

$$\Delta \tau_n = \frac{D_{xyn} - D_{xyl}}{c}, \quad n = 1, 2, \ldots N$$  \hspace{1cm} (2.10)
Where $\Delta \tau_n$ is the time delay at element $n$, $N$ is the number of elements, $D_{xyn}$ is the distance between the point $(x, y)$ and the element $n$, and $c$ is the propagation speed in the medium.

### 2.2 Acoustic Elements

Acoustic transducer elements, especially those with large apertures relative to their operating wavelengths, will have considerable directionality. The measured response of the transducer used in the array evaluations, the Kobitone 255-400SR12M-ROX[^1] is shown in the dashed black line of Figure 2.6. This response was obtained by the collection method described in Chapter 4, where a single element was used instead of an array.

![Response of acoustic phased array composed of elements with directionality.](image)

It might initially seem as if a highly directional response is beneficial, in that it reduces the magnitude of the grating lobes. However, for any application that requires an electronic steering of the beam, the directionality of the acoustic elements is of mixed benefit — if the beam is steered far enough to one side, the grating lobes and main lobe will become of equal magnitude. As seen in Figure 2.6, the grating lobes also steer along the envelope of the single-element response.

The model described in equation [2.3](#) needs to be modified for acoustic arrays to account for this directionality. The response $R_{xyn}$ of each of the elements will be multiplied by the

element patterns described in equation 2.3. This results in equation 2.11.

\[
P(x, y, t) = \sum_{n=0}^{N} \left[ A_n \frac{R_{xyn}}{D_{xyn}} \cos \left( 2\pi f \left( t - \Delta \tau_n - \frac{D_{xyn}}{c} \right) \right) \right]
\]  \hspace{1cm} (2.11)

Where \( R_{xyn} \) is dependent on the angular location of the point \((x, y)\) relative to element \(n\). Electromagnetic antenna elements can also have a directionality, which can cause an effect of the same form as Equation 2.11. Additionally, both electromagnetic and acoustic elements may have phase variation as a function of off-broadside angle [4]. This effect is not included in the model described by 2.11.

### 2.3 Genetic Algorithms

Genetic Algorithms (GA) are a biologically-inspired method of determining a viable solution to a problem, based on an iterative search. They mimic the evolutionary processes of mutation, crossover, and selection to find the most fit solutions to a complex problem. In broad terms, a GA does some combination of four things [5]:

- **Mutate**: Infrequently change attributes of population members at random.
- **Crossover**: Infrequently swap some attribute of a member of the population with another member of the population.
- **Prune**: On each iteration, choose the best members of the population.
- **Reproduce**: Based on the best members of the population, produce a new population.

The *population* is a set of possible solutions. The attributes of each solution are canonically called *chromosomes* or its *genotype*. The GA will usually start with a random initialization of chromosomes for each member of the population, followed by an iterative process composed of some combination of the above four functions, until some stop criterion is met.

The exact implementation of each of the GA functions is a substantial design decision. The details of the implementations used for this analysis are given in Chapter 4.

GAs can be used to provide a viable, if not near-optimal, solution to a complex problem [5]. In order to do this, in addition to properly creating the GA functions, a good way of
determining the fitness of a member of the population must be found; this is generally called the cost function. The cost function used for analysis is also discussed in Chapter 4.

2.4 Previous Research

The author was unable to find any published academic research concerning the optimization of inter-element spacing to reduce grating lobes in the near field for sparse acoustic arrays. However, many research efforts are quite related.

Both [6] and [7] describe phased array simulation models which are intended to be accurate in the near field. These provide a cross-reference for the model described in the previous sections of this chapter. The model in [7] is derived based on linear antenna arrays. While it is not directly applicable here, there are some key similarities. The model in [6] describes an electronically steered, linear acoustic phased array. Their model is the phasor-notation equivalent of Equation 2.11. The key difference between the model described in [6] and the model of this thesis is that in [6], two equations are derived — one for transmitter elements and one for receiver elements (although the resulting equations are virtually identical).

The far-field reduction of side lobes for electromagnetic phased arrays has been greatly researched. The author of [8] describes a deterministic method of reducing the sidelobe level in the far field for linear arrays. Another deterministic method for suppressing side lobes is suggested in [9], though the application focus is broadband planar electromagnetic phased arrays, so their research is less applicable.

Along with the deterministic methods, much research has been dedicated to iterative optimization of the locations of elements. The authors of [10] optimize the locations of dipole antenna elements by genetic algorithms. Because the elements of an acoustic array are physically large in surface area, unlike dipole antennas, the GA optimization scheme in [11] is closely related to the optimization scheme of this thesis.

Acoustic imaging is generally an active process (in that both transmitting and receiving elements are used), so research dedicated to the optimization of arrays combining transmit and receive elements is pertinent. Two examples of research into this manner of optimization are shown in [12] and [13].
Comparatively little research concerns acoustic imaging in air. However, the authors of [14] and [15] construct a device which makes use of a very similar type of transducer as is used in this thesis. They have built an active imaging sonar from a small number of sparse elements for the purpose of mobility aid for the visually impaired. Though the research goals of these documents are quite different than the research goals of this thesis, their method of construction and electronics are quite similar.

The authors of [16] show that it is possible to reduce the grating lobes of a sparse acoustic phased array for medical applications. They make use of focusing, directivity, and sparse randomized placement of elements to produce a desirable beam pattern.

It is worth mentioning, though not directly related to the goals of this thesis, that there are ways of reducing grating lobes without altering the placement of elements. One way to do this is to design element shapes that result in a desired beam directivity pattern. An example of this for electromagnetic antennas is shown in [17], an example for acoustic elements is shown in [18].
Chapter 3

Hardware and Software

This chapter describes the hardware and software that needed to be made to evaluate arbitrary phased array configurations. Many of the major design decisions for both the hardware and software, along with the reasons behind them, are presented.

It is worth noting that the design, implementation, and testing of the hardware represents roughly one third of the time spent in this thesis study. Similarly, roughly one third of the time was spent with the design and implementation of the software. So, the contents of this chapter describe a majority of the work done.

On the next page, Figure 3.1 shows a block diagram of all of the major system components. Within the Software/Computer superblock, the named subblocks indicate different software libraries and how they interact. The Application Software usually consists of trivial sequential function calls to these libraries, as they are abstracted to a very high level. Additionally, arrows interacting with the Software/Computer superblock show which libraries make use of each interface, based on the location that the arrows enter the block.

The Electronics superblock does not include things which are ancillary to system functionality, such as power supplies and the clock signal generator. These things, are, however addressed in the relevant sections of this chapter.
3.1 Hardware

A considerable amount of hardware was designed, built, and tested. The different hardware can be binned by purpose into two different categories: Electronics, and Acoustics/Mechanical. A list of approximate hardware costs is given in Appendix A.

3.1.1 Electronics

The electronics category includes all custom PCBs, purchased evaluation boards, power supplies, and mounting hardware. In order to fulfill the needs of the project, the following broad requirements were set:

1. At least sixteen channels of ADCs and sixteen channels of DACs are needed to permit reasonable array evaluation.
2. Each ADC channel must be able to sample at well above the Nyquist rate of the operating frequency of the transducers.

3. Each DAC channel must be able to update at a rate such that there will be a negligible loss of steering capability due to phase-delay quantization.

4. Each ADC channel must be amplified such that it can pick up the weakest anticipated signal with a sufficient number of least significant bits.

5. Each DAC channel must be amplified such that it can drive the transducers sufficiently under all operating conditions.

6. All of the data gathered from the ADCs must be communicated to an x86-based computer at the rate at which it is sampled. (This ensures that the data may be easily manipulated without the use of complication-creating buffers.)

7. The system must be reasonable portable. That is: it requires no large test equipment or power supplies, and can fit in a container that can be moved by one person.

8. The system must be of reasonable cost and complexity.

Some rough calculations needed to be made based on these requirements in order to begin making design decisions.

To satisfy the sample rate requirement, it is necessary to know the operating frequency of the transducers. Virtually all commercially available ultrasonic transducers which are impedance matched for air operate at either 25 kHz or 40 kHz. Because of this, a sample rate requirement on the order of 100-500 kSps (kilo-Samples per second) was set.

If all of the transmitting elements of a DAC-based phased array are synchronized to the same clock, the system must update the waveform of the transmitting elements quickly enough that the steering performance of the array is not unacceptably degraded due to quantization in phase delays. To satisfy the phase-quantization requirement, a requirement that no more than $3^\circ$ of quantization error would be permissible was set. Based on a number of small-steering-angle simulations, this results in a side-lobe error level of no more than 5% for array parameters similar to the array configurations under test. For a 40 kHz transducer, this

\footnote{The effect of phase-delay quantization on waveform error is well known and researched, see [19] or [20].}
results in a DAC refresh rate of:

\[
\text{DAC Refresh Rate} = \left( \frac{1}{40 \times 10^3 \ s^{-1}} \frac{3.0}{360.0} \right)^{-1} = 4.8 \times 10^6 \ \text{Hz} \quad (3.1)
\]

Additionally, the resolution of the ADCs and DACs needed to be picked. The driving factor for the ADC resolution was the dynamic range that would be needed to operate over a known distance range, assuming that \(1/r\) holds true and the amplifier gain is linear and fixed. For distance ranges far from the array, the signal needed to span a sufficient number of least significant bits (LSBs). For distance ranges close to the array, there could be no clipping of the signal as a result of it being over-amplified.

It was decided that the farthest measurement distance required for testing would be no farther than twenty times the closest measurement distance. Additionally, do ensure that ADC sampling quantization had a minimal effect on the measurements, it was decided that the power of the quantization noise should be on the order of 0.1\% to 0.01\% of the power of the signal, which corresponds to a Signal to Quantization Noise Ratio (SQNR) of 30 dB to 40 dB. An approximate relation between SQNR and the LSB count for a sinusoid is [21]:

\[
\text{SQNR} \approx 1.76 \ \text{dB} + 6.02 \log_2(\text{LSB Count}) \quad (3.2)
\]

Based on Equation 3.2, a minimum LSB count of 64 was decided. An LSB count of 64 is a round number which, with a corresponding SQNR of 37.9 dB, fulfills the SQNR criterion of 30 dB to 40 dB. Using this minimum LSB count and the operating distance ratio of twenty, the necessary bit count of the ADC could be calculated:

\[
\text{ADC Bit Count} = \log_2(64 \ \text{Counts} \times 20) = 10.32 \ \text{Bits} \quad (3.3)
\]

So, it was decided that a 12-bit ADC would be ideally suited, though a 10-bit ADC could provide marginally acceptable performance if necessary.

The required resolution for the DAC was determined to be dependent on the analog front-end (AFE) of the DAC. The DAC would likely need to only generate sinusoidal waveforms. Depending on the bandwidth of and slew rate of the amplifier, the resolution requirement might be relaxed because the AFE may not pass high-frequency components of the signal.
However, in order to begin making design decisions, it was decided that the DAC needed a resolution of at least 8 bits.

With the ADC sample rate, bit count, and channel count in mind, a rough estimation of data rate could be determined. This was calculated with a conservative additional protocol overhead factor of 1.2.

\[
\text{ADC Data Rate} = (16 \text{ Chan}) (12 \text{ Bits}) (4 \times 10^5 \text{ Sps}) (1.2) = 92.1 \text{ Mbps}
\] (3.4)

It was assumed that the device controlling the DACs would store the output waveforms locally, so no data rate calculation needed to be performed. The method of local storage could be pre-loaded sets of waveforms in firmware, on-demand calculation of the waveform points, or a system in which waveforms are loaded temporarily into memory.

Before making any additional design decisions, a system architecture must be chosen. It was decided that each transducer element should be connected to one amplifier, which should be connected to one ADC or DAC. This is the simplest configuration, and the most cost effective. A diagram of this type of system is shown in Figure 3.2.

![Figure 3.2: Data converter block diagrams. ADC-related components are shown in blue. DAC-related components are shown in red.](image)

Consideration needed to be given to amplifier selection. For the ADCs, the following requirement was set for the amplifier: The voltage across a receiver element must be amplified to
at least one volt peak-to-peak for a signal that was transmitted at five volts peak-to-peak and reflected off a wall located two meters away. One transmitter and one receiver element were connected to a function generator and oscilloscope to reproduce the requirement configuration. The voltage across the receiver element was measured to be 11 milli-volts peak to peak. This results in a required voltage gain of \[\frac{1}{0.011} = 90.9\]. Additionally, a fully differential amplifier would be of benefit to reduce the potential for common-mode noise.

It was decided that the DAC amplifier should have the bandwidth and slew rate to output a 15 volt peak-to-peak 40 kHz sine wave. 15 volts was chosen because that is the maximum suggested applied voltage for many of the 40 kHz transducers. Additionally, the DAC amplifier should be capable of sourcing 15 milliamps of current under all conditions. Again, this was based on the properties of the most common 40 kHz transducers.

Based on these requirements, the following design decisions were made:

- No off-the-shelf solutions exist which satisfy these requirements. A custom solution must be designed and constructed.
- FPGA evaluation boards are generally capable of performing the task of serializing the data and transferring it to and from a computer. Therefore, the design should be built around an existing FPGA evaluation board.
- Many existing FPGA boards are equipped with high-speed VHDCI connectors — an interconnect solution with sufficient bandwidth to interface with many ADCs or DACs simultaneously. Therefore, the custom boards should be designed to interface with the FPGAs over a VHDCI connector.
- In order to be able to stream the data to a computer in real time, the FPGA evaluation boards must have the capability for one or more of: High-Speed USB 2.0, USB 3.0, or 1000 BASE-T Ethernet.
- In order to keep the cost and complexity to a minimum, two-layer PCBs with traces no smaller than 8 mil are preferred. This eliminates many highly-integrated ultrasound-specific integrated circuits, which are often shipped only in high-density BGA packages.
- Separate PCB designs should be used for the ADC boards and DAC boards, this ensures that if a new revision is required of one of circuits, only that circuit need be redesigned.
• Two FPGA evaluation boards, each with their own VHDCI connector, should be used instead of a single board. By using two boards, one for the DACs, and one for the ADCs, the connection from the computer to each type of data converter is decoupled and largely unidirectional.

The most cost-effective FPGA evaluation boards which suit this purpose were the Nexys 3 variety from Digilent Inc. They are equipped with a VHDCI connector and High-Speed USB 2.0 controller, built around a Xilinx Spartan-6 FPGA.

![Four-channel ADC Evaluation PCB design](image)

**Figure 3.3:** Four-channel ADC Evaluation PCB design. Blue and red indicate copper layers. Green indicates plated through-holes. Black indicates silkscreen and component placement. The actual size of the board is roughly six inches wide.

The first boards to be designed were four-channel evaluation boards for the ICs that seemed to be likely candidates for use in the final design. These were designed in order to verify the schematics and pin out everything to tenth-inch spaced headers for quick evaluation. The designs of these evaluation boards are shown in Figures 3.3 and 3.4. The ADC boards were designed around the Texas Instruments ADC12010 high-speed analog to digital converter.

---

and the Texas Instruments LMH6550 differential high-speed operational amplifier. This combination permits a sample rate of 10 MSps at 12 bits of resolution, and a maximum voltage gain of roughly 1000 at 40 kHz, though there are many factors that determine the gain, such as the layout, reference voltage levels, and drive current.

The DAC boards were designed around the Texas Instruments DAC7821 high-speed digital to analog converter and the Texas Instruments OPA228-series operational amplifier. This pair permits 10 million DAC updates per second, an output of up to 20 volts peak-to-peak, and up to 25 milliamps of current, with sufficient slew rate and bandwidth for a 40 kHz sinusoid.

After the evaluation boards were assembled and tested, some changes to the schematics of each board were made to improve performance, and the final revision of the ADC and DAC boards were designed. It was decided that a number of stackable two-channel boards would

\[ http://www.ti.com/product/lmh6550 \]
\[ http://www.ti.com/product/dac7821 \]
\[ http://www.ti.com/product/opa228 \]
be preferable to a monolithic board for cost and complexity reasons. The finalized ADC and DAC boards are shown in Figure 3.5.

![Figure 3.5: Finalized stackable two-channel ADC PCB (left) and DAC PCB (right). Blue and red indicate copper layers. Green indicates plated through-holes. Black indicates silkscreen and component placement. The actual size of each board is roughly two and a half inches wide.](image)

The finalized boards perform considerably better than the evaluation boards in terms of noise performance and signal integrity. This is likely due to both the considerable amount of attention that was given to the ground planes and centralized grounding, and the impedance-control applied to some of the more important traces. On both of the finalized ADC and DAC boards, a number of surface-mount board-to-board connectors are placed around the edges so that each board can stack with any number of the same type of board. In practice, the stackability is limited to roughly twenty channels because of bandwidth limitations. The specifications of the finalized boards are:

- Sixteen ADC channels capable of sampling on all channels at software selectable rates between 100 kSps and 1.25 MSps. If fewer than sixteen channels are used, the rates can be up to 10 MSps.
• Sixteen DAC channels capable of updating each channel at least 3 million times per second, depending on the exact specifications of the waveform, due to firmware subtleties. Most waveforms permit 5 million updates per second or more.

• Each ADC channel is set for a voltage gain of 100.

• Each DAC channel can output at a peak to peak voltage of 15 volts, and up to 375 milliwatts.

• The system consumes between 10 and 25 watts depending on its state.

The acoustic elements can be connected to the boards by either SMA connector or 3.5 mm mono audio jack. To reduce cost, only the 3.5 mm audio jack was populated. In addition to the ADC and DAC boards, another board was needed to connect each stack of boards to the FPGAs over VHDCI. This board serves the functions of level conversion, multiplexing, pinout, and power-indication. The design for it is shown in Figure 3.6. This VHDCI-Interface board sits on top of a stack of ADC boards or a stack of DAC boards.

Figure 3.6: Finalized VHDCI-Interface, multiplexing, and pinout board. Blue and red indicate copper layers. Green indicates plated through-holes. Black indicates silkscreen and component placement. The board is roughly two and a half inches wide.
All of the boards were assembled and found to work acceptably as a system. The system is shown in Figures 3.7, 3.8, and 3.9. The ADC boards were found to run warmer than desirable (roughly 45° C) at higher clock rates, so an additional power-filtering and voltage regulation board was assembled and added to the system. This board dissipates 8 watts of power that would otherwise be dissipated by the ADC stack. This board can be seen connected to the black heatsink in Figures 3.7 and 3.9. Additionally, a small clock-cleaning and level-converting board was made for the ADCs. It can be seen wrapped in heat-shrink atop one of the FPGA evaluation boards in Figure 3.8.

Figure 3.7: Closeup of the back of the electronics assembly showing the power-filtering board and ADC clock distribution solution in more detail.

Because of the very particular requirements in clocking of the ADC boards, the clock distribution solution was given special consideration. It was not high frequency that was the driving factor, but rather a requirement for a near perfectly 50% duty cycle, as requested in the data sheet. It was decided that the clock line could not be passed through the boards as the data and power lines were, but rather needed to be connected and distributed externally, avoiding the impedance changes that result from these board-to-board connectors. The cables with gold-plated SMA connectors on top of the electronics assembly are used to
distribute the clock from board to board by daisy-chaining. The source of the clock signal is the FPGA which controls the ADCs. Additionally, the clock signal is further preserved and amplified by a clock cleaning and distributing IC on each board.
Figure 3.9: The full electronics assembly: Two Digilent Nexys 3 FPGA evaluation boards, eight two-channel ADC boards, eight two-channel DAC boards, two VHDCI-Interface boards, and a power-filtering board. The ADC boards can be identified by their blue soldermask. The DAC boards can be identified by their red soldermask. The VHDCI-Interface boards can be identified by black soldermask.
3.1.2 Acoustics and Mechanical

In addition to the electronics, it was necessary to create an easy way of producing and testing arbitrary array configurations. A slide system for the acoustic elements was produced. This can be seen in Figure 3.10. A maximum spacing between the farthest two elements of roughly fifteen inches can produced using this system.

![Figure 3.10: An array of sixteen receiving elements arrange in an evenly spaced line, mounted on the adjustable rail system atop the pan-tilt assembly.](image)

In order to avoid both the hassle and inaccuracies of manually steering the beam array, a pan-tilt system was made. It can be seen in Figure 3.11. The majority of the parts were purchased as part of a set designed to fit together, so little design work was involved. The source of the mechanics was Servocity.com. The controller for the pan-tilt is a repurposed custom board which can interface directly with a computer USB port. The power supply is a simple linear power supply which uses the body of the pan-tilt as a heatsink. Both the controller and the power supply can be seen in the right image of Figure 3.11. The pan-tilt mechanism is capable of an angular resolution of roughly 0.4 degrees, though the equipment to measure this precisely was not available.
Figure 3.11: Two views of the pan-tilt assembly used for evaluating the phased arrays. The quick-release mechanism and upper feedback potentiometer can be seen in the left view. The power supply, controller board, and lower feedback potentiometer can be seen on the right view.
3.2 Software

In order to properly support the hardware, a considerable amount of software needed to be developed. Additionally, much software unrelated to hardware needed to be made for simulation and optimization. The software can be broadly divided into four categories: FPGA Firmware, Hardware Management and Support, Simulation and Phased Array Related, and Optimization.

A large portion of the written software is included as an appendix. Inline comments are included only sparingly to keep the appendices terse, so a fair understanding of the language and libraries used is somewhat requisite. The bulk of the final software is written in Python to support quick development and readability, but the necessarily fast computations call C-compiled subroutines.

Many of the functions from each module are enumerated in the following subsections, for two reasons: to give an idea of the structure of the code, and to provide an index to the code in the appendices.

3.2.1 FPGA Firmware

The two FPGA boards serve different purposes. The one connected to the ADC board stack is responsible for:

- Generating a clock signal and control-flow signals for the ADCs.
- Acquiring the data from the ADCs.
- Formatting the parallel data such that it can be transferred to a computer.
- Responding to instructions from the computer.
- Sending the data to the computer.
- Holding a small data buffer to permit a slightly lax timing requirement on the part of the computer to account for the use of a non-real-time operating system.
- Notifying the computer of any issues, such as a data buffer overflow.

The FPGA board connected to the DAC stack is responsible for:
• Generating control-flow signals for the DACs.
• Responding to computer commands.
• Temporarily storing waveforms that are defined by the computer.
• Giving these waveforms to the individual DACs.
• Notifying the computer of any issues, such as improper waveform format or command buffer overflow.

Because the FPGA evaluation boards were based on Xilinx products, the use of Xilinx development tools was requisite. The Xilinx ISE Webpack was used as the primary development environment. All of the top-level modules are written in VHDL. A large portion of the code comes from Chris McClelland’s FPGALink library[^7] which was used as the basis for USB communication with the FPGA boards. Unlike the following sections, because VHDL code is quite verbose and little can be gleaned from quickly looking at it, this code is not as an appendix.

### 3.2.2 Hardware Management

In order to communicate with the FGPAs from the computer, a large amount of host-side software needed to be written. This is included in Appendix B containing the files named `comm.py` and `lowlevelfunctions.py`.

The best way to describe the level of abstraction of these modules is to list some of their functions. `comm.py` contains the following functions, which have been rewritten here in a more readable way:

- `connect(device_id)`
- `disconnect(device_id)`
- `awaitDevice(device_id, timeout)`
- `writeChannel(device_id, timeout, channel, bytes)`
- `some_bytes = readChannel(device_id, timeout, channel, n_bytes)`
- `loadUsbControllerFirmware(device_id)`
- `flashUsbControllerFirmware(device_id)`

The file `lowlevelfunctions.py` is used for configuration of the FPGAs, it contains the following functions, again re-written in a more readable way:

- `guidedConnectionSetup()`
- `setAdcChannelCount(n_channels)`
- `setDacChannelCount(n_channels)`
- `setAdcSampleRate(speed)`
- `setDacSampleRate(speed)`
- `setAdcSampleBlockAcquireSize(n_samples)`
- `waveform_array = makeWaveform(type, amplitude, cycle_count)`
- `uploadWaveform(waveform_array)`
- `setDacPhaseDelays(delays)`
- `queueDacTransmitWaveform()`
- `requestAdcSampleBlock()`
- `raw_bytes = obtainAdcSampleBlock()`
- `samples = parseAdcSampleBlock(raw_bytes)`

### 3.2.3 Simulation and Phased Array Related

Proper simulation and phased array related calculations are integral to the project. The first revision of the simulation software was written for use with either MATLAB or Octave. This implementation worked acceptably for visualizing pressure fields, but was not fast enough to be acceptably run several million times, as is needed for the genetic algorithm. To fulfill the need for a faster implementation, it was re-written in Python and C. In addition to the simulation, various calculations needed to be done to make use of the data obtained from the data converters.

The simulation is contained the `fastsim.py` file, which can be found in Appendix C. There are two types of simulation, one which calculates pressure magnitudes over a two-dimensional plane, and one which only calculates pressure magnitudes along an arc at some distance from the array. The inputs to the simulation, neglecting setup parameters, are the locations of the array elements, and the phase delay for each element.
Some functions that build upon those in `lowlevelfunctions.py` are contained in `highlevelfunctions.py`. These are rewritten for readability below:

- configureAdc(speed, sample_block_size, channels)
- configureDac(speed, output_block_size, channels)
- defineDacWaveform(type, waveform_parameters)
- samples = transmitWaveformAndSampleReturn()
- pressure_field = processDataForImagingSonar(samples)
- setupImagingSonarPlot()
- updateImagingSonarPlot(pressure_field)
- panTiltConnect()
- panTiltDisconnect()
- setPanTiltAngle(pan, tilt)
- saveSampleFile(name, samples)
- samples = loadSampleFile(name)

The whole of `highlevelfunctions.py` is included in Appendix C.

### 3.2.4 Optimization

The final category of software is related to the optimization and analysis of array configurations. Two files are used for this. The first, `genetic.py`, is a library of genetic-algorithm-related functions. It has function calls to this effect:

- setup(population_size, member_size, costs, variances)
- cost = findCost(population_member)
- new_population = prune(population)
- new_population = cross(population)
- new_population = mutate(population)

The second, `analysis.py`, contains miscellaneous functions mostly related to the visualization of the data produced by the other files. Both of these files are included in Appendix D.
Chapter 4

Methodology

4.1 Genetic Algorithm Specifications

Many attributes of the genetic algorithm must be specified in order for it to be reproducible. The most important among them are specified here. At the top level, the GA does the following:

```
1 Population ← Generate Random Population{};
2 while Best Individual Fitness Has Improved Within Last 25 Iterations do
3   Most Fit Individuals ← Prune{Population};
4   New Population ← Reproduce{Most Fit Individuals};
5   Mutated Population ← Mutate{New Population};
6   Population ← Cross{Mutated Population};
7   Print the parameters of the best member thus far;
8 end
```

To define any of the above functions, the attribute(s) which make up the chromosomes must be decided. We are interested in finding an optimal linear arrangement of transducers which reduce the side-lobe levels in the near-field. The chromosomes of the genetic algorithm must wholly characterize the array. So, because this is a linear arrangement, the chromosomes were chosen to be the set of inter-element distances. With this choice, for an array of 16 transducers, there will be 15 chromosomes.
The first function, the initialization of a random population, is simple:

\[
\text{input} : \text{Population Size, Chromosome Count, Variance, Chromosome Lower Limit} \\
\text{1 for } p \leftarrow 0 \text{ to Population Size do} \\
\text{2 for } c \leftarrow 0 \text{ to Chromosome Count do} \\
\text{3 Member}(c) \leftarrow |N(0, \text{Variance})| + \text{Chromosome Lower Limit}; \\
\text{4 end} \\
\text{5 Population}(p) \leftarrow \text{Member}; \\
\text{6 end} \\
\text{output: Population()}
\]

Where \(N(\mu, \sigma^2)\) is a single normally distributed random variable with mean \(\mu\) and variance \(\sigma^2\). This results in a population with chromosomes values that are all greater than the so called \textit{Chromosome Lower Limit}. A lower bound on the value of the chromosomes is set so that minimum-spacing requirements can be enforced, to optimize for different types of array configurations.

Pruning is done as follows:

\[
\text{input} : \text{Population()}, \text{Survival Rate} \\
\text{1 foreach Member() in Population() do} \\
\text{2 Determine Fitness\{Member()\};} \\
\text{3 end} \\
\text{4 } K \leftarrow (\text{Survival Rate})(\text{Population Size}); \\
\text{5 Survivors() \leftarrow The K most fit Members of Population();} \\
\text{output: Survivors()}
\]

Where it is necessary to define a cost function in order to determine the fitness of a member. The two components which were decided to be relevant are: the Maximum RSLL in the entire field of interest, and the total size of the array. It is important to constrain the size of the array so that it can be physically realized. The cost function (which is to be minimized) is then:

\[
\text{Cost} = K_{RSLL}(RSLL_{Max}) + K_{Size}(\text{Array Size}) \quad (4.1)
\]
Where $K_{RSLL}$ and $K_{Size}$ are tuned to produce an ideal result. The Array Size is the distance between the centers of the two farthest elements. Additionally, $RSLL_{Max}$ is found by the following:

```plaintext
1 foreach Distance to Evaluate do
2     Find the Maximum RSLL at this distance;
3     if This Measured Maximum RSLL is the Greatest Thus-Far then
4         Declare it to be $RSLL_{Max}$;
5     end
6 end
```

$RSLL_{Max}$ can be considered the global maximum RSLL for all evaluation distances. After pruning, reproduction must be done. The implementation of reproduction here is trivial: repeat the Survivors() until a new population of the appropriate size is made. Because there is little to this function, it is bundled with the `mutate()` function in the software call. Mutation is less trivial. It is done by:

```plaintext
input : Population(), Variance, Chromosome Lower Limit
1 foreach Member() in Population() do
2     foreach Chromosome in Member() do
3         if Should be Mutated then
4             Chromosome ← Chromosome + $N$(0, Variance);
5             if Chromosome < Chromosome Lower Limit then
6                 Chromosome ← Chromosome Lower Limit;
7         end
8     end
9 end
output: Population()
```

Where `Should be Mutated` is triggered randomly with a chosen probability. Crossover is defined in a similar fashion to mutation; instead of a random mutation happening with a chosen frequency, two population members will exchange one of their chromosomes with a
chosen frequency. The precise implementation of crossover makes use of language-specific subtleties, so the code should be referenced for a full understanding.

For the plots generated in the results section, these are the numerical parameters of the genetic algorithm used in each set of optimizations. The *Chromosome Lower Limit* is the same as the Minimum Element Spacing in these optimization sets.

<table>
<thead>
<tr>
<th>Quantity</th>
<th>First Set</th>
<th>Second Set</th>
<th>Third Set</th>
<th>Fourth Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum Element Spacing</td>
<td>2.25λ</td>
<td>1.75λ</td>
<td>1.25λ</td>
<td>0.75λ</td>
</tr>
<tr>
<td>Transducer Elements</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Population Size</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>Crossover Rate</td>
<td>5%</td>
<td>5%</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>Mutation Rate</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
</tr>
<tr>
<td>Survival Rate</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
</tr>
<tr>
<td>$K_{\text{Size}}$</td>
<td>5.0/2.25</td>
<td>5.0/1.75</td>
<td>5.0/1.25</td>
<td>5.0/0.75</td>
</tr>
<tr>
<td>$K_{\text{RSLL}}$</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Variance</td>
<td>0.016</td>
<td>0.012</td>
<td>0.008</td>
<td>0.006</td>
</tr>
</tbody>
</table>

It should be noted that $\lambda$ in air for a 40 kHz transducer is:

$$\lambda = \frac{c}{f} = \frac{340}{40000} = 0.0085 \text{ Meters}$$

So the variances defined above are actually quite large. Additionally, the $K_{\text{Size}}$ defined above are relatively small compared to the $K_{\text{RSLL}}$, because the size of the arrays is on the order of 0.1 to 0.4 meters, but the values of $RSLL_{\text{Max}}$ are on the order of 5 dB to 15 dB.
4.2 Testing Methodology and Procedure

Several array configurations needed to be evaluated as a function of off-broadside angle and distance. Evaluating them as a function of off-broadside angle allows the determination of the Maximum RSLL. Evaluating them as a function of distance allows the determination of how the Maximum RSLLs vary in the near field.

In order to do this, each element must first be calibrated. Primarily as a result of manufacturing tolerances, they might each have a different phase and amplitude response to the same input signal. Properties of the analog front-end of the electronics will also affect phase and amplitude. The calibration was done according to:

1. Arrange all receivers in a periodic linear configuration;
2. Place one transmitter at a known distance \( d \) broadside of the receivers;
3. for Five Separate Runs, to Average Results do
   4. Transmit a long pulse from the transmitter;
   5. Wait \( d/c \) for the pulse to get to the receivers;
   6. Sample for a short time on all the receivers;
   7. foreach Receiver’s Samples do
      8. Find the phase difference between this receiver and the first one;
      9. Find the amplitude of the samples for this receiver;
   end
   10. Determine the average of all these phase differences;
   11. Determine the average of all of these amplitudes;
   12. foreach Receiver do
      13. Phase Calibration Value ← Measured Phase − Average Phase;
      14. Amplitude Calibration Value ← Measured Amplitude − Average Amplitude;
   end
17. end
18. Average the determined calibration values of all five runs;

Calibration was performed any time a receiver was connected to a different input channel to account for differences in the analog front-ends.
Chapter 4. Methodology

The actual test procedure for determining the maximum RSLL of an array as a function of
distance is given below:

1 Arrange the receivers in the configuration under test using a set of calipers;
2 foreach Measurement Distance $d$ do
3 \hspace{1em} Place one transmitter broadside of the receivers at distance $d$;
4 \hspace{1em} for Number of runs to average do
5 \hspace{2em} Find the main-lobe amplitude at this distance;
6 \hspace{2em} Find the maximum side-lobe value at this distance;
7 \hspace{2em} Find the maximum RSLL based on the difference between them;
8 \hspace{1em} end
9 \hspace{1em} Average the maximum RSLL for all runs;
10 end

Where the determination of the maximum main lobe amplitude and side-lobe amplitudes is
done by the following automated process:

1 Set the tilt angle of the pan-tilt to point the receivers to the transmitter;
2 foreach Pan Angle to Measure do
3 \hspace{1em} Set the pan angle of the pan-tilt to this angle;
4 \hspace{1em} for Five Runs to Average Results do
5 \hspace{2em} Transmit a long pulse from the transmitter;
6 \hspace{2em} Wait $d/c$ for the pulse to get to the receivers;
7 \hspace{2em} Sample for a short time on all the receivers;
8 \hspace{2em} Find the response at this pan angle by delay-and-sum beamforming;
9 \hspace{1em} end
10 \hspace{1em} Average the determined response of the five runs;
11 end
12 Return the averaged measured response for each pan angle;

The step on line 8 requires explanation. The delay-and-sum beamforming was done accord-
ing to the model described in Chapter 2, where the beam was always electronically steered
Chapter 4. Methodology

broadside, and focused to the measurement distance. This means that the “averaged measured response for each pan angle” will be of the same form as the bottom plots of Figures 2.5 and 2.6. An example of a measurement of this form is shown in Figure 4.1.

![Figure 4.1: Comparison of measured (Top, Blue) versus theoretical (Bottom, Black) array response for a 16 element acoustic receiver array. The elements were spaced periodically at 1.85\(\lambda\), and electronically focused to 3.5 meters. The transmitter element is also located at 3.5 meters.](image)

It was subjectively determined that the measured results of the array were of sufficient quality that no statistical estimation of the true peak values and true RSLL were needed — the values could simply be read from the plots as-is. Some of the reasons for differences in the simulated and measured plots in Figure 4.1 are discussed in the next chapter.

Some numerical specifics of the testing methodology are specified below:

- The maximum RSLL was evaluated at each distance 3 times and then averaged.
- The measurement distances are 0.5 to 3.5 meters in half meter increments.
- The pan angles are -50 to 50 degrees in 0.4 degree increments.
- Five different array configurations were evaluated.

The results of these evaluations are given in the following chapter.
Chapter 5

Results and Analysis

5.1 Theoretical Optimized Results

Four different optimizations were carried out for each of four separate minimum spacing requirements, totaling sixteen independent GA-optimized configurations. The four minimum spacing requirements are $2.25\lambda$, $1.75\lambda$, $1.25\lambda$, and $0.75\lambda$. The four optimization distances include two optimizations for a single distance, 1.0 meter and 3.0 meters, and two optimizations for a range of distances, 0.5 to 2.0 meters and 1.0 to 4.0 meters.

The results are given in the form of Maximum RSLL versus distance for each configuration. In addition to the optimized configurations, each plot also includes the performance of a periodic configuration with an inter-element spacing equal to the minimum spacing requirement. This is to serve as a benchmark on which to judge the effectiveness of the GA optimization.

Analysis of, and commentary on, the results is included following each plot in the following four sections. It is worth noting again that the genetic algorithm did not optimize purely for Maximum RSLL, as array size is included in the cost function. The size of each array configuration is not reflected in this analysis. It is also worth noting that each evaluated configuration is only one of many configurations output by the GA. An attempt was made by the author to choose configurations that were representative of the set.
5.1.1 Simulated 2.25 Wavelength Minimum Spacing

As can be seen from Figure 5.1, it is somewhat possible to suppress grating lobes of acoustic arrays in the near-field with a minimum inter-element spacing of 2.25\(\lambda\). All configurations permit an improvement of at least 3 dB over the periodic configuration at their optimized distances. However, a Maximum RSLL of -7 dB to -9 dB falls short of the rule of thumb of -10 dB required for effective suppression. There are several notable features:

- As expected, each of the configurations is able to outperform the others at their respective optimization distances.
- The one meter optimized configuration narrowly outperforms the one-to-four meter optimized configuration, at the expense of performance in the far-field.
- The periodic configuration has better performance at the half-meter distance. With some investigation, the author concluded this is due to a differential in the focusing distance of the main lobe and the grating lobes. This effect can be seen in Figure 2.5.
- Unsurprisingly, the configurations which are optimized for distances farther from the array perform better approaching the far-field.
### 5.1.2 Simulated 1.75 Wavelength Minimum Spacing

[Figure 5.2: Comparison of the near-field performance of four different simulated GA-optimized 16 element phased array configurations constrained to a minimum inter-element spacing of 1.75\(\lambda\). A periodic arrangement with an inter-element spacing of 1.75\(\lambda\) is also shown. The dots indicate the distances of optimization for the genetic algorithm.]

As can be seen from Figure [5.2](#), it is somewhat possible to suppress grating lobes in the near-field with a minimum inter-element spacing of 1.75\(\lambda\). All configurations permit an improvement of at least 3 dB over the periodic configuration at their optimized distances. Coupled with the response of the acoustic elements, 9 dB to 11 dB of side lobe reduction is possible. Again, some things are worthy of note:

- All of the noted trends of the 2.25\(\lambda\) minimum spacing configurations can also be observed in the 1.75\(\lambda\) minimum spacing configurations.

- The Maximum RSLL of the 1.75\(\lambda\) configurations have been more-or-less shifted vertically compared to the 2.25\(\lambda\) configurations. This is due to the grating lobes being spaced farther from the main lobe, coupled with the reduced response of the acoustic elements at farther off-broadside angles.

The measured results of the 1.75\(\lambda\) configurations are given in Section 5.2. The array patterns for a select number of evaluation distances are also given in Section 5.4.
5.1.3 Simulated 1.25 Wavelength Minimum Spacing

As can be seen from Figure 5.3, suppression of grating lobes in the near-field with a minimum inter-element spacing of 1.25\( \lambda \) is very possible. Again, some things are worth noting:

- All of the noted trends of the 2.25\( \lambda \) minimum spacing configurations and the 1.75\( \lambda \) minimum spacing configurations can also be observed here.
- Though the absolute performance of each of the optimized 1.25\( \lambda \) minimum spacing configurations is better than the 1.75\( \lambda \) and 2.25\( \lambda \) configurations, there is less improvement over the periodic configuration.
5.1.4 Simulated 0.75 Wavelength Minimum Spacing

![Graph showing comparison of near-field performance of four different simulated GA-optimized 16 element phased array configurations constrained to a minimum inter-element spacing of 0.75\(\lambda\). A periodic arrangement with an inter-element spacing of 0.75\(\lambda\) is also shown. The dots indicate the distances of optimization for the genetic algorithm.]

Because the 0.75\(\lambda\) minimum spacing configurations will not have grating lobes within the \(\pm 90^\circ\) field of view of interest, this case is quite different than the three prior. This difference permits the GA to thoroughly suppress side lobes, with marked improvement over the periodic configuration.

5.2 Measured Optimized Results

Figure 5.5 shows the measured values of Maximum RSLL as a function of distance for the configurations shown in Figure 5.2. The inclusion of the periodic configuration serves as a baseline for evaluating the effectiveness of the measurement techniques.

The average difference between measured and theoretical values for the baseline configuration is 0.45 dB. The error increases with the farther distances. This is due, at least in part, to the ADCs getting closer to their noise floor. Further discussion of this error is given in Section 5.4.
Figure 5.5: Measured Maximum RSLL versus distance for the optimized configurations shown in Figure 5.2. The circles indicate the points of optimization. The diamonds indicate measured values.
The error in the measured results of the optimized configurations is much greater than the periodic configuration, but the trends of each configuration are generally maintained.

Section 5.4 provides a discussion of a number of the reasons for the discrepancies between the measured and theoretical values in Figure 5.5. Section 5.4 also includes some of the array patterns for select measurement points.
5.3 Imaging Sonar Application Results

To evaluate an optimized configuration in practice, the array was configured as an active imaging sonar. One transmitter was placed just above the receivers, facing the same direction. In brief, the plots shown in Figures 5.7 and 5.8 were produced by:

```
1 Configure the array and test-reflection objects;
2 Transmit a short pulse from the transmitter;
3 Sample on all receivers for 2d/c seconds;
4 foreach Angle and Distance to Plot do
   foreach Receiver’s Samples do
      Δτs ← steering delay for current transducer, by Equation 2.1
      Δτf ← focusing delay for current transducer, by Equation 2.10
      Offset Samples ← Estimate of samples at time t − Δτs − Δτf from the
                       original samples at time t by cubic interpolation;
   end
5 Plot Value (r, θ) ← Sum of the Offset Samples at sample time closest to 2r/c;
6 Multiply off-broadside values by 1/R^2_{xy} to account for transducer response;
7 end
```

Where d is the maximum desired measure distance. We sample for 2d/c because the transmitted pulse needs to reflect off an object before returning to the receivers. The test setup for the imaging sonar evaluation is shown in Figure 5.6. Two 1.5 inch square stainless-steel poles of 20 inch length are used as the acoustic reflectors. Though the perspective of the image does not show it, the plane of the acoustic elements intersects the poles 8 inches from their top. The exact locations of the poles with respect to the array is shown in the figure. They are rotated such that a flat face is directed at the transmitter. By running the imaging sonar in the absence of the poles, but leaving the mounting tripods, it was determined that the tripods were positioned and sized such that they would not affect the results.

Two array configurations were analyzed: A periodic configuration with an inter-element spacing of 1.75λ, and the 1.0 to 4.0 meter optimized configuration shown in Figure 5.5. The results of the imaging sonar evaluation are shown in Figures 5.7 and 5.8. For the 1.0 meter
distance pole in the periodic configuration, the first grating lobe is $-0.2$ dB down from the main lobe. This is in line with the expected $0.0$ dB. For the 1.25 meter distance pole, the first grating lobe is also $-0.2$ dB down from the main lobe. Again, this is to be expected, as we have normalized for the response of the transducers.

The choice of color grading does not convey the performance well, but the aperiodic configuration performs better than the periodic configuration. The Maximum RSLL of the 1.0 meter distance pole is now $-3.0$ dB. The results for the 1.25 meter distance pole are slightly more complicated. The first grating lobe is $-3.5$ dB down from the main lobe, but a second grating lobe which is only $-2.8$ dB down from the main lobe has appeared. This is due to the wider spacing of the elements.

The wider spacing also improves angular resolution performance. As the figures show, there is improvement in the ability of the array to resolve the size of the poles. For the periodic configuration, the poles do not show up a singular objects to the extent that they do in the aperiodic configuration.
Figure 5.7: Results of imaging sonar evaluation for the test setup shown in Figure 5.6 for a periodic array configuration with an inter-element spacing of 1.75\(\lambda\). Stronger reflections are indicated in blue. Weaker reflections and noise are indicated in red. The arrows indicate the true reflections. All other reflections are artifacts of the array.

Figure 5.8: Results of imaging sonar evaluation for the test setup shown in Figure 5.6 for an optimized aperiodic array configuration. The array configuration used was the 1.0 to 4.0 meter optimized configuration shown in Figure 5.2. Stronger reflections are indicated in blue. Weaker reflections and noise are indicated in red. The arrows indicate the true reflections. All other indicated reflections are artifacts.
5.4 Possible Sources of Error

There are many possible sources of error that would cause the differences between the theoretical and measured results in Figure 5.5. Among them are:

1. The alignment of the optimized configurations is more prone to errors than the periodic configurations, as it is not visually obvious that there is an alignment error.
2. Misalignment of the periodic configuration may actually improve performance, but this is unlikely to be the case for the optimized configurations.
3. The elements are more widely spaced in the optimized configurations, resulting in a more narrow focus point and more narrow main lobe because of the larger effective aperture. This requires greater care of positioning than the periodic configuration.
4. The system model does not account for the directionality of the transmitter element, it is assumed isotropic. As the configuration becomes larger, this assumption holds less true, resulting in more error.
5. The system model does not account for variation of phase of the elements with off-broadside angle. The configurations proposed by the GA, therefore, also do not account for this.
6. The AFEs of the system will introduce some noise to the signal, and digitizing the signal will introduce some quantization noise.

It is possible to simulate the effect of several of these sources of error. Three different types of error — alignment, phase-variation with angle, and system noise — will be examined by simulation. Each error simulation is accompanied by a corresponding measured response with the same parameters. The measured responses are taken from the data used to produce Figure 5.5.

Alignment error is the result of both human error in positioning, and manufacturer tolerances in the placement of the piezoelectric element inside the transducer case. It is approximated here by a normally-distributed random addition of distance to the nominal locations of each element. That is: \( \text{Location}_n = \mathcal{N}(\text{Location}_n, \text{Variance}) \) for each element \( n \). Figure 5.9 shows a simulation which does this — it contains an overlapping of many different configurations with misalignment.
Figure 5.9: Comparison of the measured response [Top, Red] and ideal response [Top, Gray] of a transducer array to the simulated responses of arrays with added positioning error. The measured results come from the 3.0 meter optimized, 1.75\(\lambda\) minimum spacing configuration, measured at 2.5 meters. These measured results are also shown as a single point in Figure 5.5. The simulated results each show 32 overlapped array responses with positioning error variances of 0.5 mm\(^2\) [Middle, Black] and 2.0 mm\(^2\) [Bottom, Black].
The middle plot of Figure 5.9 has an alignment variance of 0.5 mm$^2$. An inspection of the transducer elements shows that, because of manufacturing tolerances, the centers of some of the piezoelectric elements in the transducers are misaligned from the center of their cases by more than 0.5 mm. Therefore, the middle plot should be considered a good alignment. The bottom plot shows an alignment variance of 2.0 mm$^2$ — a poor alignment. An alignment error variance of 2.0 mm$^2$ produces an effect on the Maximum RSLL which is similar to the measured results for several of the 32 simulation runs.

Another possible source of error is a variation in phase of the transducer elements with off-broadside angle. Because of the extremely small positioning tolerances that would be required to measure this variation, it was excluded from the model. However, we can simulate its effect.

In the absence of any knowledge of the phase trend of the transducers, it is assumed that the phase at off-broadside angles is a discrete normal random walk of the form:

$$\Phi_\theta = \Phi_{\theta-1} + N(0, \text{Variance})$$  \hspace{1cm} (5.1)

Where $\Phi_\theta$ is the phase at off-broadside angle $\theta$ and $\Phi_{\theta-1}$ is the phase at the previous discrete angle. It is also assumed that all of the transducers in a given array have the same phase-variation profile — that is, the simulation does not account for manufacturing variance, only a non-zero off-broadside phase trend.

Figure 5.10 is the phase-variation equivalent of Figure 5.9. Because $\Phi_\theta$ is a random walk, the increment between $\Phi_\theta$ and $\Phi_{\theta-1}$ is important. In each of the bottom two plots of Figure 5.10, the increment is 0.1 degrees. The middle plot shows a phase variance of 0.2 degrees squared. The bottom plot shows a phase variance of 0.4 degrees squared.

Unaccounted-for phase variations with off-broadside angle will have a bigger effect in distances closer to the array, so a close-range measured value was used in Figure 5.10. The character of the errors induced by phase-variation is different than those produced by positioning errors. One thing to note concerning this type of error is that even for the low-variance case, high-magnitude side lobes just outside the main lobe are produced. This effect was seen in many of the measured responses, such as the top plot of Figure 4.1.
Figure 5.10: Comparison of the measured response [Top, Blue] and ideal response [Top, Gray] of a transducer array to the simulated responses of arrays with off-broadside phase profiles. The measured results come from the 1.0 meter optimized, 1.75λ minimum spacing configuration, measured at 0.5 meters. These measured results are also shown as a single point in Figure 5.5. The simulated results each show 32 overlapped array responses with phase variances of 0.2 [Middle, Black] and 0.4 [Bottom, Black] degrees squared.
Another type of error that can be readily simulated is the addition of quantization and system noise. These types of error will be more prevalent towards the far end of the distance ranges, where the received signal is of lower magnitude compared to the noise. These types of error can be simulated exactly, as the quantization and system noise are both measurable parameters.

The ADCs measure a signal of roughly 70 LSBs peak-to-peak at 3.5 meters in the test configuration, so the error simulation quantizes the simulated sinusoids into 70 bins. The ADCs and amplifiers were also measured to have a noise floor variance of 4.0 LSBs with the receivers connected (and less than one LSB with the receivers disconnected), so Additive White Gaussian Noise (AWGN) was added to the sinusoids with that variance. The noise-adding process was done before any phased-array processing, and can be expressed as:

<table>
<thead>
<tr>
<th>input</th>
<th>Sinusoid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 V ← 4.0;</td>
<td></td>
</tr>
<tr>
<td>2 Q ← 70;</td>
<td></td>
</tr>
<tr>
<td>3 A ← Peak-to-peak value of the Sinusoid;</td>
<td></td>
</tr>
<tr>
<td>4 Noisy Sinusoid ← Sinusoid + ( N(0, AV/Q) );</td>
<td></td>
</tr>
<tr>
<td>5 Noisy Quantized Sinusoid = Round{Noisy Sinusoid, Q}</td>
<td></td>
</tr>
<tr>
<td>output: Noisy Quantized Sinusoid</td>
<td></td>
</tr>
</tbody>
</table>

Where Round\{Wave, Q\} rounds each element of Wave to fit in one of Q linearly-spaced bins.

The results of adding this type of noise to the system is shown in Figure 5.11. The noise affects the apparent level of the side-lobes more than it does the apparent level of the main lobe. This is because of the logarithmic scaling of the vertical axis — were these results displayed on a linear axis, the noise would appear to be of equal magnitude everywhere.

It is apparent that this type of error is different than either the positioning error shown in Figure 5.9 or the phase-variation error shown in Figure 5.10. This type of error results in a response that appears to have been modified by the addition of AWGN — which is to be expected, as the response at each off-broadside angle is simply the linear combination of signals which have been modified by the addition of AWGN.
Chapter 5. Results

Figure 5.11: Comparison of the measured response [Top, Green] and ideal response [Top, Gray] of a transducer array to the simulated responses of arrays with added noise. The measured results come from the 1.0 to 4.0 meter optimized, 1.75\(\lambda\) minimum spacing configuration, measured at 3.5 meters. These measured results are also shown as a single point in Figure 5.5. The simulated results [Bottom, Black] contain 32 runs which include noise error added to the sinusoids.

It would be unjustifiable to not combine these three types of error to see if it were possible to produce results similar to the measured responses, so this simulated error-combination will be done for the sample array configuration shown in Figure 4.1. The periodic configuration was chosen for this analysis because it has an ideal response that is easily understood, and any deviation from the ideal response is visually apparent. There has been a key difference between the simulated and measured plots of the array responses thus-far; the measured responses are shown with 0.4\(^\circ\) off-broadside angle increments (pan-tilt limitation), but the simulated responses are shown with 0.1\(^\circ\) increments. So, Figure 5.12 was simulated with 0.4\(^\circ\) increments.

Figure 5.12 is simulated with alignment errors, off-broadside phase errors, and noise errors. It shows both 8 and 256 overlapped simulated configurations. The alignment error variance
Figure 5.12: Comparison of the measured response [Top, Blue] and ideal response [Top, Gray] of a transducer array to the simulated responses of arrays with all errors added. The measured results are for a 1.85λ spacing periodic configuration measured at 3.5 meters. These measured results are also shown in Figure 4.1. The simulated results show an overlay of 8 [Middle, Black] and 256 [Bottom, Black], runs with an alignment variance of 0.5 mm², phase variance of 0.1 degrees squared, and added noise.
used in the figure is 0.5 mm$^2$, and the phase variance used in the figure is 0.1 degrees squared. The noise error parameters are the same as in Figure 5.11.

Many of the aberrations in the measured response of Figure 5.12 also appear in the simulated errors. One notable aberration which is reflected in both the simulated error and the measured results is the high-magnitude side lobes just outside the main lobe. Another notable aberration which is reflected in both the simulated error and the measured results is the high level of the lobes between the main lobe and the grating lobes. Based on these simulated errors, we can conclude that the mismatch between the simulated and measured results can be largely accounted-for with the three simulated sources of error.
Chapter 6

Conclusions

6.1 Discussion of Results

The possibility of arranging the elements of a sparse acoustic phased array such that it can be operated effectively in the near-field has been confirmed. It has been shown that the grating lobes can be reduced by a considerable amount over the distances in which the array has been optimized.

The performance cost in the far-field for an array that is optimized in the near-field is dependent on how close the optimization distances are to the array. For configurations which are optimized to perform well at distances very close to the array, there can be a considerable far-field performance cost compared to configurations which are optimized to perform at distances farther from the array.

Array performance is strongly affected by slight errors in positioning. The small amount of positioning error that is caused by manufacturing tolerances in the placement of the piezoelectric element is enough to affect array performance. Additionally, a variation in phase with off-broadside angle has the potential to affect array performance if it has not been compensated-for in some way.
6.2 Future Work

The following were not included in the scope of this thesis, and would be appropriate steps to take in the future:

1. Expand the analysis to include transmitter arrays.
2. Use several different brands and models of transducer elements, ideally also of varied sizes and shapes.
3. Update the system model to include the effects of phase variation with off-broadside angle, and measure this variation for a real transducer.
4. Include effective aperture and angular resolution in the analysis.
5. Include steering performance (off-broadside electronic steering) of the array in the analysis.
6. Characterize array performance in more practical imaging sonar applications, and perform more signal analysis on the imaging sonar results to identify objects and edges.
7. Use the genetic algorithm to optimize for many different minimum spacing requirements, so that a plot of Maximum RSLL versus minimum spacing can be produced, with lines of constant optimization distance. This would provide more insight into the points at which a certain performance criterion can no longer be met.

6.3 Closing Thoughts

The research questions of this thesis have been answered. Unfortunately, because of limited time, the scope of the thesis has been constrained. The hardware which was constructed permits much of the future work to be done, but each item of the future work is a considerable expansion of scope.

That said, the primary goal of this thesis study — determine and verify the possibility of suppressing near-field grating lobes in sparse acoustic phased arrays — has been completed and explored to a satisfactory extent.
Appendix A

Approximate Hardware Costs

The costs below include the approximate price of items that did not need to be purchased. The prices are meant to reflect the money that would be required to build a new system.

<table>
<thead>
<tr>
<th>Item</th>
<th>Quantity</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nexys 3 Evaluation Boards</td>
<td>2</td>
<td>$240</td>
</tr>
<tr>
<td>Populated Dual-Channel ADC Boards</td>
<td>8</td>
<td>$750</td>
</tr>
<tr>
<td>Populated Dual-Channel DAC Boards</td>
<td>8</td>
<td>$450</td>
</tr>
<tr>
<td>Populated VHDCI-Pinout Boards</td>
<td>2</td>
<td>$80</td>
</tr>
<tr>
<td>Transducers</td>
<td>16-32</td>
<td>$80-$160</td>
</tr>
<tr>
<td>Coaxial Cables and Audio Jacks</td>
<td>16-32</td>
<td>$90-$170</td>
</tr>
<tr>
<td>USB Cables and VHDCI Cables</td>
<td>2</td>
<td>$60</td>
</tr>
<tr>
<td>Transducer Rail System</td>
<td>1</td>
<td>$120</td>
</tr>
<tr>
<td>Pan-Tilt Head</td>
<td>1</td>
<td>$300</td>
</tr>
<tr>
<td>Misc Mounting and Assembly Hardware</td>
<td>-</td>
<td>$100</td>
</tr>
<tr>
<td>Power Supplies</td>
<td>-</td>
<td>$40</td>
</tr>
</tbody>
</table>

Total $2310-$2470
Appendix B

Low Level Software

B.1 File: comm.py

The file comm.py is a wrapper of C-compiled functions for interfacing with the USB controllers on the FPGA evaluation boards.

```python
from ctypes import *

import time

import array

import sys

import os

os.environ['PATH'] += os.pathsep + os.path.dirname(os.path.abspath('comm.py'))

# Define types
```

class FLContext (Structure):
    pass

class FLErrException (Exception):
    pass

FLCHandle = POINTER (FLContext)
FLStatus = c_uint
FL_SUCCESS = 0
uint32 = c_uint
uint16 = c_ushort
uint8 = c_ubyte
ErrorString = c_char_p

# Get DLL
if (sys.platform == "linux2"):
    os.environ["PATH"] += (os.pathsep +
        os.path.dirname(os.path.abspath('comm.py')) + "lib/")
    cdll.LoadLibrary(os.path.dirname(os.path.abspath('comm.py'))
        + "/lib/libfpgalink.so")
    fpgalink = CDLL(os.path.dirname(os.path.abspath('comm.py'))
        + "/lib/libfpgalink.so")

elif (sys.platform == "darwin"):
    cdll.LoadLibrary("libfpgalink.dylib")
    fpgalink = CDLL("libfpgalink.dylib")

elif (sys.platform == "win32"):
    os.environ["PATH"] += (os.pathsep +
        os.path.dirname(os.path.abspath('comm.py'))
        + "\\lib\\")
    windll.LoadLibrary(os.path.dirname(os.path.abspath('comm.py'))
        + "\\lib\\libfpgalink.dll")
    fpgalink = WinDLL(os.path.dirname(os.path.abspath('comm.py'))
        + "\\lib\\libfpgalink.dll")

# Miscellaneous Functions
fpgalink.flInitialise.argtypes = []
fpgalink.flInitialise.restype = None
fpgalink.flFreeError.argtypes = [c_char_p]
fpgalink.flFreeError.restype = None
fpgalink.flSleep.argtypes = [uint32]
fpgalink.flSleep.restype = None
fpgalink.flLoadFile.argtypes = [c_char_p, POINTER(uint32)]
fpgalink.flLoadFile.restype = POINTER(uint8)
Appendix B. Low-Level Software

```python
67 fpgalink.flFreeFile.argtypes = [POINTER(uint8)]
68 fpgalink.flFreeFile.restype = None
69 fpgalink.flScanChain.argtypes = [FLHandle, POINTER(uint32), POINTER(uint32),
70    uint32, POINTER(ErrorString)]
71 fpgalink.flScanChain.restype = FLStatus
72 fpgalink.flPortAccess.argtypes = [FLHandle, uint16, uint16, POINTER(uint16),
73    POINTER(ErrorString)]
74 fpgalink.flPortAccess.restype = FLStatus
75
76 # Connection Lifecycle
77 fpgalink.flOpen.argtypes = [c_char_p, POINTER(FLHandle), POINTER(ErrorString)]
78 fpgalink.flOpen.restype = FLStatus
79 fpgalink.flClose.argtypes = [FLHandle]
80 fpgalink.flClose.restype = None
81
82 # Device Capabilities and Status
83 fpgalink.flIsDeviceAvailable.argtypes = [c_char_p, POINTER(uint8),
84    POINTER(ErrorString)]
85 fpgalink.flIsDeviceAvailable.restype = FLStatus
86 fpgalink.flIsNeroCapable.argtypes = [FLHandle]
87 fpgalink.flIsNeroCapable.restype = uint8
88 fpgalink.flIsCommCapable.argtypes = [FLHandle]
89 fpgalink.flIsCommCapable.restype = uint8
90
91 # CommFPGA Operations
92 fpgalink.flIsFPGARunning.argtypes = [FLHandle, POINTER(uint8),
93    POINTER(ErrorString)]
94 fpgalink.flIsFPGARunning.restype = FLStatus
95 fpgalink.flWriteChannel.argtypes = [FLHandle, uint32, uint8, uint32,
96    POINTER(uint8), POINTER(ErrorString)]
97 fpgalink.flWriteChannel.restype = FLStatus
98 fpgalink.flReadChannel.argtypes = [FLHandle, uint32, uint8, uint32,
99    POINTER(uint8), POINTER(ErrorString)]
100 fpgalink.flReadChannel.restype = FLStatus
101
102 # NeroJTAG Operations
103 fpgalink.flPlayXSVF.argtypes = [FLHandle, c_char_p, POINTER(ErrorString)]
104 fpgalink.flPlayXSVF.restype = FLStatus
105
106
107 # Open a connection to the FPGALink device
108 def connect(vp):
109    handle = FLHandle()
110    error = ErrorString()
111    status = fpgalink.flOpen(vp.encode('ascii'), byref(handle), byref(error))
```
if (status != FL_SUCCESS):
    s = str(error.value)
    fpgalink.f1FreeError(error)
    raise FLException(s)
return handle

# Close the FPGALink connection
def disconnect(handle):
    fpgalink.f1Close(handle)

# Await renumeration - return true if found before timeout
def AwaitDevice(vp, timeout):
    error = ErrorString()
    isAvailable = uint8()
    fpgalink.f1Sleep(1000)
    while (True):
        fpgalink.f1Sleep(100)
        status = fpgalink.f1IsDeviceAvailable(vp.encode('ascii'),
                                             byref(isAvailable), byref(error))
        if (status != FL_SUCCESS):
            s = str(error.value)
            fpgalink.f1FreeError(error)
            raise FLException(s)
        timeout = timeout - 1
        if (isAvailable):
            return True
        if (timeout == 0):
            return False

# Query NeroJTAG capability
def IsNeroCapable(handle):
    if (fpgalink.f1IsNeroCapable(handle)):
        return True
    else:
        return False

# Query CommFPGA capability
def IsCommCapable(handle):
    if (fpgalink.f1IsCommCapable(handle)):
        return True
    else:
return False

# Scan the JTAG chain

def ScanChain(handle):
    #error = ErrorString()
    ChainType = (uint32 * 16)  # Guess there are fewer than 16 devices
    chain = ChainType()
    length = uint32(0)
    #status = fpgalink.flScanChain(handle, byref(length), chain,
    # 16, byref(error))
    if (length > 16):
        # We know exactly how many devices there are, so try again
        ChainType = (uint32 * length.value)
        chain = ChainType()
        # status = fpgalink.flScanChain(handle, None, chain, length, byref(error))
    return chain

# Access the I/O ports on the micro

def PortAccess(handle, portWrite, ddr):
    error = ErrorString()
    portRead = uint16()
    status = fpgalink.flPortAccess(handle, portWrite, ddr, byref(portRead),
    byref(error))
    if (status != FL_SUCCESS):
        s = str(error.value)
        fpgalink.flFreeError(error)
        raise FLException(s)
    return portRead.value

# Return true if the FPGA is actually running

def IsFPGARunning(handle):
    error = ErrorString()
    isRunning = uint8()
    status = fpgalink.flIsFPGARunning(handle, byref(isRunning), byref(error))
    if (status != FL_SUCCESS):
        s = str(error.value)
        fpgalink.flFreeError(error)
        raise FLException(s)
    if (isRunning):
        return True
    else:
return False

# Write one or more bytes to the specified channel
def WriteChannel(handle, timeout, chan, values):
    error = ErrorString()
    if (isinstance(values, bytearray)):
        # Write the contents of the byte array:
        numValues = len(values)
        BufType = uint8 * numValues
        buf = BufType.from_buffer(values)
        status = fpgalink.flWriteChannel(handle, timeout, chan, numValues,
                                           buf, byref(error))
    elif (isinstance(values, int)):
        # Write a single integer
        if (values > 0xFF):
            raise FLException("Supplied value won’t fit in a byte!")
        status = fpgalink.flWriteChannel(handle, timeout, chan, 1,
                                          (uint8 * 1)(values), byref(error))
    else:
        # Write the contents of a file
        fileLen = uint32()
        fileData = fpgalink.flLoadFile(values.encode('ascii'), byref(fileLen))
        if (fileData == None):
            raise FLException("Cannot load file!")
        status = fpgalink.flWriteChannel(handle, timeout, chan, fileLen,
                                          fileData, byref(error))
        fpgalink.flFreeFile(fileData)
    if (status != FL_SUCCESS):
        s = str(error.value)
        fpgalink.flFreeError(error)
        raise FLException(s)

# Read one or more values from the specified channel
def ReadChannel(handle, timeout, chan, count=1):
    error = ErrorString()
    if (count == 1):
        # Read a single byte
        buf = uint8()
        status = fpgalink.flReadChannel(handle, timeout, chan, 1,
                                         byref(buf), byref(error))
        returnValue = buf.value
    else:
# Read multiple bytes
byteArray = bytearray(count)

BufType = uint8 * count
buf = BufType.from_buffer(byteArray)
status = fpgalink.flReadChannel(handle, timeout, chan, count,
    buf, byref(error))

returnValue = byteArray

if (status != FL_SUCCESS):
    s = str(error.value)
    fpgalink.flFreeError(error)
    raise FLException(s)

return returnValue

# Append channel write to init buffer

def AppendWriteChannelCommand(handle, chan, values):
    error = ErrorString()
    if (isinstance(values, (list, tuple, array.array))):
        numValues = len(values)
        nV = numValues
        status = fpgalink.flAppendWriteChannelCommand(handle, chan, numValues,
            (uint8 * nV)(*values),
            byref(error))
    elif (isinstance(values, int)):
        status = fpgalink.flAppendWriteChannelCommand(handle, chan, 1,
            (uint8 * 1)(values),
            byref(error))
    else:
        fileLen = uint32()
        fileData = fpgalink.flLoadFile(values, byref(fileLen))
        if (fileData == None):
            raise FLException("Cannot load file!")
        status = fpgalink.flAppendWriteChannelCommand(handle, chan, fileLen,
            fileData, byref(error))
        fpgalink.flFreeFile(fileData)
    if (status != FL_SUCCESS):
        s = str(error.value)
        fpgalink.flFreeError(error)
        raise FLException(s)

def LoadStandardFirmware(curVidPid, newVidPid, jtagPort):
    error = ErrorString()
    status = fpgalink.flLoadStandardFirmware(curVidPid.encode('ascii'),
newVidPid.encode('ascii'),
  jtagPort.encode('ascii'),
byref(error))

if (status != FL_SUCCESS):
  s = str(error.value)
  fpgalink.flFreeError(error)
  print "Error"
  raise FLException(s)
return status == FL_SUCCESS

# Flash standard firmware into the FX2's EEPROM

def FlashStandardFirmware(handle, newVidPid, jtagPort, eepromSize,
xsvfFile=None):
  error = ErrorString()
  status = fpgalink.flFlashStandardFirmware(handle, newVidPid.encode('ascii'),
                                           jtagPort.encode('ascii'),
                                           eepromSize,
                                           xsvfFile.encode('ascii'),
                                           byref(error))
  if (status != FL_SUCCESS):
    s = str(error.value)
    fpgalink.flFreeError(error)
    raise FLException(s)

B.2 File: lowlevelfunctions.py

The file lowlevelfunctions.py contains a class which handles, as its name suggests, a number of functions which are abstracted only slightly from hardware interface.
#### Common Setup Parameters

NEXYS_3_DEFAULT_VIDPID = "1443:0007"

ADC_FPGA_VIDPID = "1443:0009"

DAC_FPGA_VIDPID = "1443:0008"

JTAG_PORT = "D0234"

FPGA_CLOCK = 100000000 # Hz

ADC_BIT_COUNT = 12

DAC_BIT_COUNT = 12

SHORT_TIMEOUT = 20 # Timeouts for operations, in ms.

MED_TIMEOUT = 200

LONG_TIMEOUT = 10000

TINY_SLEEP = 0.0001 # Seconds

SMALL_SLEEP = 0.001 # Seconds

---

#### ADC FPGA Setup Parameters

ADC_RETURN_BUFFER_SIZE = 32768 # Bytes

ADC_CALIB_OFFSETS = [-2.6, -7.1, 4.0, -7.18, 9.6, -10.1, -1.0, 0.6, 11.3, -6.5, -5.6, -14.4, -0.8, 5.8, 1.85, -4.5, 0.0, 0.0, 0.0]

ADC_READ_FIFO_SIZE_ADDR_1 = 0x01 # Location of 8 MSBs of read fifo size

ADC_READ_FIFO_SIZE_ADDR_2 = 0x02 # Location of 8 LSBs of read fifo size

ADC_WRITE_FIFO_SIZE_ADDR_1 = 0x3 # Location of 8 LSBs of write fifo size

ADC_PACKET_HEADER_BYTE = 0xFF

ADC_PACKET_FOOTER_BYTE = 0x00

ADC_CHAN_COUNT_NYBBLE = 0x1 # Choose the number of channels.

ADC_SAMPLE_PERIOD_NYBBLE = 0x2 # fs.

ADC_DASCH_NYBBLE = 0x3 # Delay after the sample clock goes high

ADC_BLOCK_SIZE_NYBBLE = 0x4 # Choose Sample Block Size.

ADC_PROPOGATION_DELAY_NYBBLE = 0x5 # Choose expected propagation time.

ADC_READ_TIME_NYBBLE = 0x6 # How long it takes to evaluate sample.

ADC_PRW_NYBBLE = 0x7 # Time to keep oe low after read.

ADC_NULL_TIME_NYBBLE = 0x8 # Time to keep oe high between reads.

ADC_SAMPLE_NOW_BYTE = 0xFF # Instruction to start sampling.
# Nybble lookup dictionary

```python
ADC_CHAN_COUNT_OPT = {2: 0x1, 4: 0x2, 6: 0x3, 8: 0x4,
                      10: 0x5, 12: 0x6, 14: 0x7, 16: 0x8, 18: 0x9, 20: 0xA}
```

```python
ADC_SAMPLE_PERIOD_OPT = {50: 0x0, 100: 0x1, 120: 0x2, 140: 0x3,
                         168: 0x4, 192: 0x5, 220: 0x6, 256: 0x7,
                         300: 0x8, 360: 0x9, 420: 0xA, 480: 0xB,
                         540: 0xC, 600: 0xD, 720: 0xE, 840: 0xF}
```

```python
ADC_BLOCK_SIZE_OPT = {2400: 0x0, 2700: 0x1, 3000: 0x2, 3400: 0x3,
                      3900: 0x4, 4400: 0x5, 5000: 0x6, 6000: 0x7,
                      7200: 0x8, 8500: 0x9, 10000: 0xA, 12000: 0xB,
                      15000: 0xC, 20000: 0xD, 30000: 0xE, 50000: 0xF}
```

```python
ADC_DASCH_OPT = {0: 0x0, 1: 0x1, 2: 0x2, 3: 0x3,
                 4: 0x4, 5: 0x5, 6: 0x6, 7: 0x7,
                 8: 0x8, 9: 0x9, 10: 0xA, 11: 0xB,
                 12: 0xC, 13: 0xD, 14: 0xE, 15: 0xF}
```

```python
ADC_PROPORGATION_OPT = {0: 0x0, 1: 0x1, 2: 0x2, 3: 0x3,
                       4: 0x4, 5: 0x5, 6: 0x6, 7: 0x7,
                       8: 0x8, 9: 0x9, 10: 0xA, 11: 0xB,
                       12: 0xC, 13: 0xD, 14: 0xE, 15: 0xF}
```

```python
ADC_READ_TIME_OPT = {0: 0x0, 1: 0x1, 2: 0x2, 3: 0x3,
                     4: 0x4, 5: 0x5, 6: 0x6, 7: 0x7,
                     8: 0x8, 9: 0x9, 10: 0xA, 11: 0xB,
                     12: 0xC, 13: 0xD, 14: 0xE, 15: 0xF}
```

```python
ADC_POST_READ_WAIT_OPT = {0: 0x0, 1: 0x1, 2: 0x2, 3: 0x3,
                         4: 0x4, 5: 0x5, 6: 0x6, 7: 0x7,
                         8: 0x8, 9: 0x9, 10: 0xA, 11: 0xB,
                         12: 0xC, 13: 0xD, 14: 0xE, 15: 0xF}
```

```python
ADC_NULL_TIME_OPT = {0: 0x0, 1: 0x1, 2: 0x2, 3: 0x3,
                     4: 0x4, 5: 0x5, 6: 0x6, 7: 0x7,
                     8: 0x8, 9: 0x9, 10: 0xA, 11: 0xB,
                     12: 0xC, 13: 0xD, 14: 0xE, 15: 0xF}
```

# SETTINGS FOR "SLOW" OPERATION

The settings are of the form: {chan_count: sample_period}

```python
ADC_SAMPLE_RATES_SLOW = {2: 600, 4: 600, 6: 600, 8: 600,
```
ADC_SLOW_DASCH = {2: 10, 4: 10, 6: 15, 8: 10, 10: 10, 12: 10, 14: 10, 16: 10, 18: 10, 20: 10}

ADC_SLOW_PROPOGATION_DELAY = {2: 8, 4: 8, 6: 15, 8: 8, 10: 8, 12: 8, 14: 8, 16: 8, 18: 8, 20: 8}

ADC_SLOW_NULL_TIME = {2: 10, 4: 10, 6: 10, 8: 15, 10: 10, 12: 10, 14: 10, 16: 10, 18: 10, 20: 10}

# SETTINGS FOR "MEDIUM" OPERATION

# The of the form: {chan_count: sample_period}


ADC_MED_NULL_TIME = {2: 1, 4: 1, 6: 1, 8: 1, 10: 1, 12: 1, 14: 1, 16: 1, 18: 1, 20: 1}

# SETTINGS FOR FAST" OPERATION

# The of the form: {chan_count: sample_period}

ADC_SAMPLE_RATES_FAST = {2: 50, 4: 50, 6: 100, 8: 100, 10: 100, 12: 100, 14: 140, 16: 140, 18: 168, 20: 168}

ADC_FAST_DASCH = {2: 6, 4: 6, 6: 6, 8: 6, 10: 6, 12: 6, 14: 6, 16: 6, 18: 6, 20: 6}

ADC_FAST_PROPOGATION_DELAY = {2: 6, 4: 6, 6: 6, 8: 6, 10: 6, 12: 6, 14: 6, 16: 6, 18: 6, 20: 6}

ADC_FAST_NULL_TIME = {2: 0, 4: 0, 6: 0, 8: 0, 10: 0, 12: 0, 14: 0, 16: 0, 18: 0, 20: 0}
# SETTINGS COMMON TO ALL SPEEDS

ADC_SLOW_READ_TIME = 1
ADC_MED_READ_TIME = 1
ADC_FAST_READ_TIME = 1

ADC_SLOW_PRW_TIME = 0
ADC_MED_PRW_TIME = 0
ADC_FAST_PRW_TIME = 0

DAC_CPWE_NYBBLE = 0x1  # Choose cycles-per-wave-element.
DAC_CHAN_COUNT_NYBBLE = 0x2  # Choose the number of channels.
DAC_BLOCK_SIZE_NYBBLE = 0x3  # Choose Sample Block Size.
DAC_PROPOGATION_DELAY_NYBBLE = 0x4  # Choose expected propagation time.
DAC_WRITE_TIME_NYBBLE = 0x5  # How long it takes to evaluate sample.
DAC_NULL_TIME_NYBBLE = 0x6  # Time to keep oe high between reads.
DAC_OUTPUT_NOW_BYTE = 0xFF  # Instruction to start sampling.

DAC_COARSE_PHASE_CRUMBYL = 0b01  # Coarse phase adjustment bit pair.
DAC_FINE_PHASE_CRUMBYL = 0b10  # Fine phase adjustment bit pair.

DAC_WAVE_MSBS_CRUMBYL = 0b10
DAC_WAVE_LSBS_CRUMBYL = 0b01

DAC_CONTROL_ADDR = 0
DAC_WAVE_ADDR = 127

DAC_SKIP_RATE = 8

DAC_WRITE_FIFO_CHAN_ADDR = {0: 0x01, 1: 0x02, 2: 0x03, 3: 0x04,
4: 0x05, 5: 0x06, 6: 0x07, 7: 0x08,
8: 0x09, 9: 0x0A, 10: 0x0B, 11: 0x0C,
12: 0x0D, 13: 0x0E, 14: 0x0F, 15: 0x10,
16: 0x11, 17: 0x12, 18: 0x13, 19: 0x14}

DAC_CHAN_COUNT_OPT = {1: 0x1, 2: 0x2, 3: 0x3, 4: 0x4,
5: 0x5, 6: 0x6, 7: 0x7, 8: 0x8,
9: 0x9, 10: 0xA, 11: 0xB, 12: 0xC,
13: 0xD, 14: 0xE}

DAC_BLOCK_SIZE_OPT = {0: 0x0, 1: 0x1, 2: 0x2, 3: 0x3,}
Appendix B. Low-Level Software

DAC_PROPOGATION_OPT = {0: 0x0, 1: 0x1, 2: 0x2, 3: 0x3, 4: 0x4, 5: 0x5, 6: 0x6, 7: 0x7, 8: 0x8, 9: 0x9, 10: 0xA, 11: 0xB, 12: 0xC, 13: 0xD, 14: 0xE, 15: 0xF}

DAC_WRITE_TIME_OPT = {0: 0x0, 1: 0x1, 2: 0x2, 3: 0x3, 4: 0x4, 5: 0x5, 6: 0x6, 7: 0x7, 8: 0x8, 9: 0x9, 10: 0xA, 11: 0xB, 12: 0xC, 13: 0xD, 14: 0xE, 15: 0xF}

DAC_NULL_TIME_OPT = {0: 0x0, 1: 0x1, 2: 0x2, 3: 0x3, 4: 0x4, 5: 0x5, 6: 0x6, 7: 0x7, 8: 0x8, 9: 0x9, 10: 0xA, 11: 0xB, 12: 0xC, 13: 0xD, 14: 0xE, 15: 0xF}

DAC_SLOW_PROPOGATION_DELAY = 8
DAC_MED_PROPOGATION_DELAY = 4
DAC_FAST_PROPOGATION_DELAY = 1

DAC_SLOW_WRITE_TIME = 8
DAC_MED_WRITE_TIME = 4
DAC_FAST_WRITE_TIME = 1

DAC_SLOW_NULL_TIME = 8
DAC_MED_NULL_TIME = 4
DAC_FAST_NULL_TIME = 1

def __init__(self):
    comm.fpgalink.flInitialise() # Start the fpgalink library.

# ############## Handles and Awareness Parameters ##############

self.firmware_good_adc = False
self.firmware_good_dac = False

self.adc_configured = False
self.dac_configured = False

self.adc_fpga_handle = 'None'
self.dac_fpga_handle = 'None'
self.adc_chan_count = 'None'
self.adc_sample_rate = 'None'
self.adc_block_size = 'None'
self.adc_sample_period = 'None'

self.adc_dasch = 'None'
self.adc_prop_delay = 'None'
self.adc_read_time = 'None'
self.adc_prw_time = 'None'
self.adc_null_time = 'None'

self.adc_exact_sample_rate = 'None'
self.adc_buffer_full_time = 'None'
self.adc_bytes_per_block = 'None'
self.adc_bytes_per_sample = 'None'
self.adc_data_rate = 'None'

self.adc_n_blocks_requested = 0
self.adc_n_blocks_fulfilled = 0

self.dac_chan_count = 'None'
self.dac_waveform = 'None'
self.dac_amplitude = 'None'

self.dac_prop_delay = 'None'
self.dac_write_time = 'None'
self.dac_null_time = 'None'

self.dac_block_size = 'None'

def guidedSetup(self):

time_to_load_and_connect = 5

adc_loaded = comm.AwaitDevice(self.ADC_FPGA_VIDPID, 2)
dac_loaded = comm.AwaitDevice(self.DAC_FPGA_VIDPID, 2)
if adc_loaded and dac_loaded:
    PS.say("Both Nexys 3 Boards are Connected With Proper Firmware.")
    PS.instruct("Make sure the USB drives are connected.")
    PS.pause()
    PS.say("Jolly good then.")

elif adc_loaded:
    PS.say("ADC Nexys 3 Board is Connected and Loaded.")
    PS.instruct("Plug in the DAC Nexys 3 Board (RED) Now.")
    PS.pause()
    PS.say("Attempting to load firmware to DAC Nexys 3.")
    self.programFirmware("DAC")

    for i in range(time_to_load_and_connect):
        time.sleep(1)
        PS.say("...")

    dac_now_loaded = comm.AwaitDevice(self.DAC_FPGA_VIDPID, 2)

    if dac_now_loaded:
        PS.say("Success. Firmware Loading Complete.")
        PS.instruct("Now, plug in the USB drives.")
        PS.pause()
        PS.say("Okay, moving right along.")

    elif dac_loaded:
        PS.say("DAC Nexys 3 Board is Connected and Loaded.")
        PS.instruct("Plug in the ADC Nexys 3 Board (BLUE) Now.")
        PS.pause()
        PS.say("Attempting to load firmware to ADC Nexys 3.")

        self.programFirmware("ADC")

        for i in range(time_to_load_and_connect):
            time.sleep(1)
            PS.say("...")

        adc_now_loaded = comm.AwaitDevice(self.ADC_FPGA_VIDPID, 2)

        if adc_now_loaded:
            PS.say("Success. Firmware Loading Complete.")
PS. instruct("Now, plug in the usb drives.")
PS. say("Okay, moving right along.")

else:
    PS. say("No boards connected with proper firmware.")
    PS. instruct("Unplug any usb drives from the Nexys 3 boards.")
    PS. instruct("Plug in the DAC Nexys 3 Board (RED) Now.")
    PS. pause()
    PS. say("Attempting to load firmware to DAC Nexys 3.")
    self. programFirmware("DAC")
    for i in range(time_to_load_and_connect):
        time.sleep(1)
        PS. say("...")
    dac_now_loaded = comm.AwaitDevice(self.DAC_FPGA_VIDPID, 2)
    if dac_now_loaded:
        PS. say("Success. DAC FPGA Firmware Loading Complete.")
    else:
        PS. error("DAC FPGA firmware did not load.")
    PS. instruct("Plug in the ADC Nexys 3 (BLUE) Board Now.")
    PS. pause()
    PS. say("Attempting to load firmware to ADC Nexys 3.")
    self. programFirmware("ADC")
    for i in range(time_to_load_and_connect):
        time.sleep(1)
        PS. say("...")
    adc_now_loaded = comm.AwaitDevice(self.ADC_FPGA_VIDPID, 2)
    if adc_now_loaded:
        PS. say("Success. ADC FPGA Firmware Loading Complete.")
    else:
        PS. error("ADC FPGA firmware did not load.")
    PS. instruct("Now, plug in the usb drives.")
    PS. say("Okay, moving right along.")

def programFirmware(self, which):
    current_vidpid = self.NEXYS_3_DEFAULT_VIDPID
    jtag = self.JTAG_PORT
    if which == "ADC":
new_voidpid = self.ADC_FPGA_VIDPID
already_done = comm.AwaitDevice(new_voidpid, 5)
if not already_done:
    try:
        comm.LoadStandardFirmware(current_vidpid, new_voidpid, jtag)
        self.firmware_good_adc = True
    except:
        PS.error("ADC Firmware Load Failed.")
        self.firmware_good_adc = False
else:
    self.firmware_good_adc = True

elif which == "DAC":
    new_voidpid = self.DAC_FPGA_VIDPID
    already_done = comm.AwaitDevice(new_voidpid, 5)
    if not already_done:
        try:
            comm.LoadStandardFirmware(current_vidpid, new_voidpid, jtag)
            self.firmware_good_dac = True
        except:
            PS.error("DAC Firmware Load Failed.")
            self.firmware_good_dac = False
    else:
        self.firmware_good_dac = True
return already_done

def connectTo(self, which):
    if which == "ADC":
        self.adc_fpga_handle = comm.connect(self.ADC_FPGA_VIDPID)
    elif which == "DAC":
        self.dac_fpga_handle = comm.connect(self.DAC_FPGA_VIDPID)

def disconnectFrom(self, which):
    if which == "ADC":
        comm.disconnect(self.adc_fpga_handle)
    elif which == "DAC":
        comm.disconnect(self.dac_fpga_handle)

def getFromFpgaBuffer(self, which, acquire_size=1):
    if which == "ADC":
if self.adc_fpga_handle != 'None':
    the_data = comm.ReadChannel(self.adc_fpga_handle,
                               self.LONG_TIMEOUT, 0x00,
                               acquire_size)
else:
    PS.warn("ADC Not Connected, cannot collect data.")
    the_data = 'None'

elif which == "DAC":
    if self.dac_fpga_handle != 'None':
        the_data = comm.ReadChannel(self.dac_fpga_handle,
                                     self.LONG_TIMEOUT, 0x00,
                                     acquire_size)
    else:
        PS.warn("DAC Not Connected, cannot collect data.")
        the_data = 'None'
return the_data

def sendToFpga(self, which, the_channel, the_byte, timeout=0):
    if timeout == 0:
        timeout = self.SHORT_TIMOUT
    if which == "ADC":
        if self.adc_fpga_handle != 'None':
            comm.WriteChannel(self.adc_fpga_handle, timeout,
                               the_channel, the_byte)
        else:
            PS.warn("ADC Not Connected, cannot send data.")
    elif which == "DAC":
        if self.dac_fpga_handle != 'None':
            comm.WriteChannel(self.dac_fpga_handle, timeout,
                               the_channel, the_byte)
        else:
            PS.warn("DAC Not Connected, cannot send data.")

def loadOptions(self, which):
    if which == "ADC":
        if self.adc_fpga_handle != 'None':
            if (self.adc_sample_period != 'None'
                and self.adc_chan_count != 'None'
                and self.adc_block_size != 'None'
                and self.adc_prop_delay != 'None'
                and self.adc_null_time != 'None'
and self.adc_read_time != 'None'
    and self.adc_dasch != 'None'):

    # Set the number of channels.
    chan_byte = ((self.ADC_CHAN_COUNT_NYBBLE << 4) +
                 self.ADC_CHAN_COUNT_OPT[self.adc_chan_count])
    self.sendToFpga("ADC", 0x00, chan_byte)
    time.sleep(self.SMALL_SLEEP)

    # Set the Delay Between Data Sets.
    samp_byte = ((self.ADC_SAMPLE_PERIOD_NYBBLE << 4) +
                  self.ADC_SAMPLE_PERIOD_OPT[self.adc_sample_period])
    self.sendToFpga("ADC", 0x00, samp_byte)
    time.sleep(self.SMALL_SLEEP)

    # Set the Delay After Sample High.
    dasch_byte = ((self.ADC_DASCH_NYBBLE << 4) +
                  self.ADC_DASCH_OPT[self.adc_dasch])
    self.sendToFpga("ADC", 0x00, dasch_byte)
    time.sleep(self.SMALL_SLEEP)

    # Set the Block Size.
    bsiz_byte = ((self.ADC_BLOCK_SIZE_NYBBLE << 4) +
                 self.ADC_BLOCK_SIZE_OPT[self.adc_block_size])
    self.sendToFpga("ADC", 0x00, bsiz_byte)
    time.sleep(self.SMALL_SLEEP)

    # Set the propagation delay time.
    prop_byte = ((self.ADC_PROPOGATION_DELAY_NYBBLE << 4) +
                 self.ADC_PROPOGATION_OPT[self.adc_prop_delay])
    self.sendToFpga("ADC", 0x00, prop_byte)
    time.sleep(self.SMALL_SLEEP)

    # Set the read time.
    rtim_byte = ((self.ADC_READ_TIME_NYBBLE << 4) +
                  self.ADC_READ_TIME_OPT[self.adc_read_time])
    self.sendToFpga("ADC", 0x00, rtim_byte)
    time.sleep(self.SMALL_SLEEP)
# Set the post-read-wait time.
prwt_byte = ((self.ADC_PRW_NYBBLE << 4) +
            self.ADC_POST_READ_WAIT_OPT[self.adc_prw_time])

self.sendToFpga("ADC", 0x00, prwt_byte)
time.sleep(self.SMALL_SLEEP)

# Set the null time.
null_byte = ((self.ADC_NULL_TIME_NYBBLE << 4) +
              self.ADC_NULL_TIME_OPT[self.adc_null_time])

self.sendToFpga("ADC", 0x00, null_byte)
time.sleep(self.SMALL_SLEEP)

self.adc_configured = True

else:
    PS.error("Not all ADC options set, cannot load config.")
else:
    PS.error("ADC Not Connected, cannot load config.")

elif which == "DAC":
    if self.dac_fpga_handle != 'None':
        if (self.dac_chan_count != 'None' 
            and self.dac_block_size != 'None' 
            and self.dac_prop_delay != 'None' 
            and self.dac_write_time != 'None' 
            and self.dac_null_time != 'None'
        ):;

        # Set the number of channels.
        chan_byte = ((self.DAC_CHAN_COUNT_NYBBLE << 4) +
                     self.DAC_CHAN_COUNT_OPT[self.dac_chan_count])

        self.sendToFpga("DAC", self.DAC_CONTROL_ADDR, chan_byte)
time.sleep(self.SMALL_SLEEP)

        # Set the Block Size.
        bsiz_byte = ((self.DAC_BLOCK_SIZE_NYBBLE << 4) +
                     self.DAC_BLOCK_SIZE_OPT[self.dac_block_size])

        self.sendToFpga("DAC", self.DAC_CONTROL_ADDR, bsiz_byte)
time.sleep(self.SMALL_SLEEP)

        # Set the propogation delay time.
prop_byte = ((self.DAC_PROPOGATION_DELAY_NYBBLE << 4) +
    self.DAC_PROPOGATION_OPT[self.dac_prop_delay])

self.sendToFpga("DAC", self.DAC_CONTROL_ADDR, prop_byte)
time.sleep(self.SMALL_SLEEP)

# Set the write time.
wtim_byte = ((self.DAC_WRITE_TIME_NYBBLE << 4) +
    self.DAC_WRITE_TIME_OPT[self.dac_write_time])

self.sendToFpga("DAC", self.DAC_CONTROL_ADDR, wtim_byte)
time.sleep(self.SMALL_SLEEP)

# Set the null time.
null_byte = ((self.DAC_NULL_TIME_NYBBLE << 4) +
    self.DAC_NULL_TIME_OPT[self.dac_null_time])

self.sendToFpga("DAC", self.DAC_CONTROL_ADDR, null_byte)
time.sleep(self.SMALL_SLEEP)

self.dac_configured = True

else:
    PS.error("Not all DAC options set, cannot load config.")
else:
    PS.error("DAC Not Connected, cannot load config.")

def setChannelCount(self, which, n_channels):
    if which == "ADC":
        if n_channels in self.ADC_CHAN_COUNT_OPT:
            self.adc_chan_count = n_channels
        else:
            PS.warn("Invalid ADC Channel Count.")

    elif which == "DAC":
        if n_channels in self.DAC_CHAN_COUNT_OPT:
            self.dac_chan_count = n_channels
        else:
            PS.warn("Invalid DAC Channel Count.")

def setBlockSize(self, which, the_block_size, prints=True):
    if which == "ADC":
        if the_block_size in self.ADC_BLOCK_SIZE_OPT:
self.adc_block_size = the_block_size
self.adc_bytes_per_sample = self.adc_chan_count * 3 / 2 + 1
self.adc_bytes_per_block = self.adc_bytes_per_sample * \
    self.adc_block_size

if prints:
    PS.say("There are " +
            repr(self.adc_bytes_per_block / 1000) +
            " kB in each sample block.")
else:
    PS.warn("Invalid ADC Block Size.")

elif which == "DAC":
    if the_block_size in self.DAC_BLOCK_SIZE_OPT:
        self.dac_block_size = the_block_size
    else:
        PS.warn("Invalid DAC Block Size.")
else:
    PS.warn("'Which' must be ADC or DAC.")

def binaryString(x, width):
    return ''.join(str((x >> i) & 1) for i in xrange(width - 1, -1, -1))

# #######################################################################
# ####################### ADC - SPECIFIC FUNCTIONS ########################
# #######################################################################

def setAdcSampleRate(self, how_fast="Slow", prints=True):
    if self.adc_fpga_handle != 'None':
        if self.adc_chan_count != 'None':
            if how_fast == "Slow":
                self.adc_sample_period = \
                    self.ADC_SAMPLE_RATES_SLOW[self.adc_chan_count]
                self.adc_dasch = \
                    self.ADC_SLOW_DASCH[self.adc_chan_count]
                self.adc_prop_delay = \
                    self.ADC_SLOW_PROPOGATION_DELAY[self.adc_chan_count]
                self.adc_read_time = \
                    self.ADC_SLOW_READ_TIME
self.adc_prw_time = \
    self.ADC_SLOW_PRW_TIME

self.adc_null_time = \
    self.ADC_SLOW_NULL_TIME[self.adc_chan_count]

elif how_fast == "Medium":
    self.adc_sample_period = \
        self.ADC_SAMPLE_RATES_MEDIUM[self.adc_chan_count]
    self.adc_dasch = \
        self.ADC_MED_DASCH[self.adc_chan_count]
    self.adc_prop_delay = \
        self.ADC_MED_PROPAGATION_DELAY[self.adc_chan_count]
    self.adc_read_time = \
        self.ADC_MED_READ_TIME
    self.adc_prw_time = \
        self.ADC_MED_PRW_TIME
    self.adc_null_time = \
        self.ADC_MED_NULL_TIME[self.adc_chan_count]

elif how_fast == "Fast":
    self.adc_sample_period = \
        self.ADC_SAMPLE_RATES_FAST[self.adc_chan_count]
    self.adc_dasch = \
        self.ADC_FAST_DASCH[self.adc_chan_count]
    self.adc_prop_delay = \
        self.ADC_FAST_PROPAGATION_DELAY[self.adc_chan_count]
    self.adc_read_time = \
        self.ADC_FAST_READ_TIME
    self.adc_prw_time = \
        self.ADC_FAST_PRW_TIME
    self.adc_null_time = \
        self.ADC_FAST_NULL_TIME[self.adc_chan_count]

    self.adc_exact_sample_rate = \
```python
self.adc_data_rate = \\
    (math.ceil(float(self.adc_chan_count) * 3 / 2) * self.adc_exact_sample_rate * 12)
else:
    PS.warn("You Must Declare channel counts before sample rate.")
else:
    PS.warn("ADC is not connected, cannot set sample rate.")

if prints:
    PS.say("ADC Sample Rate is " + repr(int(self.adc_exact_sample_rate / 1000)) + " kSps.")

return self.adc_exact_sample_rate

def getAdcFpgaReadBufferSize(self):
    if self.adc_fpga_handle != 'None':
        msB = comm.ReadChannel(self.adc_fpga_handle, self.MED_TIMEOUT,
                                self.ADC_READ_FIFO_SIZE_ADDR_1, 1)
        lsB = comm.ReadChannel(self.adc_fpga_handle, self.MED_TIMEOUT,
                                self.ADC_READ_FIFO_SIZE_ADDR_2, 1)
        the_size = (msB << 8) + lsB
    else:
        PS.warn("ADC Not Connected, cannot get buffer size.")
        the_size = 'None'

    return the_size

def getAdcSampleBlock(self):
    blocks_unfulfilled = \
        self.adc_n_blocks_requested - self.adc_n_blocks_fulfilled

    if self.adc_configured and blocks_unfulfilled > 0:
        new_block = self.getFromFpgaBuffer("ADC", self.adc_bytes_per_block)
        self.adc_n_blocks_fulfilled += 1
    else:
        PS.warn("ADC has not yet been configured, or there are" +
```
def requestAdcSampleBlock(self):
    if self.adc_configured:
        self.sendToFpga("ADC", 0x00, self.ADC_SAMPLE_NOW_BYTE)
        self.adc_n_blocks_requested += 1
    else:
        PS.warn("ADC has not yet been configured, cannot get sample block.")

def makeAdcSampleArray(self):
    if self.adc_configured:
        self.current_samples = np.zeros((self.adc_chan_count,
                                          self.adc_block_size))
    else:
        PS.warn("ADC has not yet been configured, cannot make zero array.")

def parseAdcSampleBlock(self, the_block):
    # For all the sample blocks.
    for i in range(len(the_block) / self.adc_bytes_per_sample):
        # For each element in one sample block.
        for j in range(self.adc_bytes_per_sample):
            if not ((j - 1) % 3): # If j is multiple of 3. [0,3,6]
                hors_index = (i * self.adc_bytes_per_sample) + j
                # [0, 2, 4], i
                self.current_samples[(j * 2 / 3), i] = (the_block[hors_index] << 4) +
                                                      ((the_block[hors_index + 1] & 0xF0) >> 4))

            elif not (j - 2) % 3: # Or [1, 4, 7]
                hors_index = (i * self.adc_bytes_per_sample) + j
                # [1, 3, 5], i
                self.current_samples[((j - 1) * 2 / 3 + 1), i] = (((the_block[hors_index] & 0x0F) << 8) +
                                                        the_block[hors_index + 1])
# Trim the values of the ADCs

```python
for i in range(self.adc_chan_count):
    self.current_samples[i, :] = np.add(self.current_samples[i, :],
                                         self.ADC_CALIB_OFFSETS[i])
```

```python
return self.current_samples
```

def getOneAdcSampleBlock(self, prints=True):
    self.requestAdcSampleBlock()
    start_time = float(time.time())
    new_block = self.getAdcSampleBlock()
    end_time = float(time.time())

    if prints:
        dt = float(end_time - start_time)
        rate = (float(self.adc_bytes_per_block) / dt / 1000000)
        PS.say("ADC data transfer rate over USB is \%0.3f MBps" % (rate))

    self.makeAdcSampleArray()
    the_good_samples = self.parseAdcSampleBlock(new_block)
    return the_good_samples

def obtainAdcSampleBlock(self, prints=True):
    start_time = float(time.time())
    new_block = self.getAdcSampleBlock()
    end_time = float(time.time())

    if prints:
        dt = float(end_time - start_time)
        rate = (float(self.adc_bytes_per_block) / dt / 1000000)
        PS.say("ADC data transfer rate over USB is \%0.3f MBps" % (rate))

    self.makeAdcSampleArray()
    the_good_samples = self.parseAdcSampleBlock(new_block)
    return the_good_samples

def calculateAdcTimeArray(self, offset=0):
    ...
```python
t = [(i / self.adc_exact_sample_rate + offset)
     for i in range(self.adc_block_size)]

    return t

# #######################################################################
# ####################### DAC-SPECIFIC FUNCTIONS ########################
# #######################################################################

def makeOneInvertedCosine(self, freq, amp, offset=0.0):
    the_length = self.FPGA_CLOCK / (freq * self.DAC_SKIP_RATE)
    the_array = [0 for i in range(int(the_length))]

    for i in range(int(the_length)):
        phase = float(i) / float(the_length) * 2 * 3.14159
        val = amp * 2 ** (self.DAC_BIT_COUNT - 1) * (-1 * math.cos(phase) + 1) + offset * 2.0 ** (self.DAC_BIT_COUNT)
        the_array[i] = int(val)

    return the_array

def makeLinearChirp(self, start_freq, end_freq, amp, wave_cycles):
    avg_freq = (start_freq + end_freq) / 2
    total_time = float(wave_cycles) * (1 / float(avg_freq))
    total_fpga_clock_cycles = int(total_time * self.FPGA_CLOCK)
    total_sample_size = int(total_fpga_clock_cycles / self.DAC_SKIP_RATE)

    the_array = [0 for i in range(total_sample_size)]

    k = (start_freq - end_freq) / (total_time)

    for i in range(total_sample_size):
        time = float(i) * self.DAC_SKIP_RATE / self.FPGA_CLOCK
        phase = 2 * 3.14159 * ((start_freq * time) + (k / 2 * time ** 2))
        val = amp * 2 ** (self.DAC_BIT_COUNT - 1) * (-1 * math.cos(phase) + 1)
        the_array[i] = int(val)

    return the_array
```

---

Appendix B. Low-Level Software

87
```python
def makeLinearCyclicChirp(self, start_freq, end_freq, amp, wave_cycles):
    avg_freq = (start_freq + end_freq) / 2
    total_time = float(wave_cycles) * (1 / float(avg_freq))
    total_fpga_clock_cycles = int(total_time * self.FPGA_CLOCK)
    rising_fpga_clock_cycles = int(total_time / 2 * self.FPGA_CLOCK)
    falling_fpga_clock_cycles = int(total_time / 2 * self.FPGA_CLOCK)
    total_sample_size = total_fpga_clock_cycles / self.DAC_SKIP_RATE
    rising_sample_size = rising_fpga_clock_cycles / self.DAC_SKIP_RATE
    falling_sample_size = falling_fpga_clock_cycles / self.DAC_SKIP_RATE
    the_array = [0 for i in range(total_sample_size)]
    k_up = 2 * (start_freq - end_freq) / (total_time)
    k_down = 2 * (start_freq - end_freq) / (total_time)
    for i in range(rising_sample_size):
        time = float(i) * self.DAC_SKIP_RATE / self.FPGA_CLOCK
        phase = 2 * 3.14159 * ((start_freq * time) +
                               (k_up / 2 * time ** 2))
        val = amp * 2 ** (self.DAC_BIT_COUNT - 1) *\
             (-1 * math.cos(phase) + 1)
        the_array[i] = int(val)
    for i in range(falling_sample_size):
        time = float(i) * self.DAC_SKIP_RATE / self.FPGA_CLOCK
        phase = 2 * 3.14159 * ((end_freq * time) +
                               (k_down / 2 * time ** 2))
        val = amp * 2 ** (self.DAC_BIT_COUNT - 1) *\
             (-1 * math.cos(phase) + 1)
        the_array[i + rising_sample_size] = int(val)
    return the_array

def uploadDacWaveform(self, the_wave_array):
```
for i in range(len(the_wave_array)):
    if the_wave_array[i] > 4095:
        PS.warn("Wave Array has Value Above 4095. It will be clipped.")
        the_wave_array[i] = 4095
    elif the_wave_array[i] < 0:
        PS.warn("Wave Array has value below 0, it will be clipped.")
        the_wave_array[i] = 0

msbs = (the_wave_array[i] >> 6) + (self.DAC_WAVE_MSBS_CRUMBYL << 6)

# Use 8 bit mode, neglect the last 4 digits.
lsbs = (the_wave_array[i] & 48) + (self.DAC_WAVE_LSBS_CRUMBYL << 6)

self.sendToFpga("DAC", self.DAC_WAVE_ADDR, lsbs, 30)
self.sendToFpga("DAC", self.DAC_WAVE_ADDR, msbs, 30)

self.sendToFpga("DAC", self.DAC_WAVE_ADDR, 0xC0)

def setDacRefreshSpeed(self, the_speed="Slow", prints=True):
    if the_speed == "Slow":
        self.dac_prop_delay = self.DAC_SLOW_PROPOGATION_DELAY
        self.dac_write_time = self.DAC_SLOW_WRITE_TIME
        self.dac_null_time = self.DAC_SLOW_NULL_TIME
    elif the_speed == "Medium":
        self.dac_prop_delay = self.DAC_MED_PROPOGATION_DELAY
        self.dac_write_time = self.DAC_MED_WRITE_TIME
        self.dac_null_time = self.DAC_MED_NULL_TIME
    elif the_speed == "Fast":
        self.dac_prop_delay = self.DAC_FAST_PROPOGATION_DELAY
        self.dac_write_time = self.DAC_FAST_WRITE_TIME
        self.dac_null_time = self.DAC_FAST_NULL_TIME
    else:
        PS.warn("Speed not available.")

    if prints:
        dac_refresh_rate = (self.FPGA_CLOCK / (self.dac_prop_delay + 
                                  self.dac_write_time + self.dac_null_time) 
                                  * (1.0 / self.dac_chan_count))
        PS.say("DAC refresh rate is %0.4f" % (dac_refresh_rate))

    return dac_refresh_rate

def setDacCoarsePhase(self, the_channel, the_coarse_phase):
if the_channel in self.DAC_WRITE_FIFO_CHAN_ADDR:
    if int(the_coarse_phase / 64) < (2 ** 6 - 1):
        coarse_byte = (self.DAC_COARSE_PHASE_CRUMBYL << 6) + \
            int(the_coarse_phase / 64)
        self.sendToFpga("DAC", the_channel + 1, coarse_byte)
    else:
        PS.warn("Invalid coarse phase value.")
else:
    PS.warn("Invalid Channel")

def setDacFinePhase(self, the_channel, the_fine_phase):
    if the_channel in self.DAC_WRITE_FIFO_CHAN_ADDR:
        if int(the_fine_phase) < (2 ** 6 - 1):
            fine_byte = (self.DAC_FINE_PHASE_CRUMBYL << 6) + \
                int(the_fine_phase)
            self.sendToFpga("DAC", the_channel + 1, fine_byte)
        else:
            PS.warn("Invalid fine phase value.")
    else:
        PS.warn("Invalid Channel")

def queueTransmitBlock(self):
    self.sendToFpga("DAC", self.DAC_CONTROL_ADDR, self.DAC_OUTPUT_NOW_BYTE)

if __name__ == "__main__":
    setup = LowLevelFunctions()
    setup.guidedSetup()
Appendix C

Simulation & Other Software

C.1 File: fastsim.py

The file fastsim.py contains two classes for different forms of phased-array simulation. The first class simulates over an entire 2d grid, the second class simulates only at one distance from the array.

```python
from scipy.interpolate import interp1d
import numpy as np
import time

class PhasedArrayGridSim():
    SIM_STEPS = 11 # Number of timesteps in simulation
    GMF = 0.95 # Fraction of grid sized used for evaluation
    NORM = 1.0 # Normalization factor for amplitudes, if y-axis values matter
    SOUND_SPEED = 340.0 # Speed of Sound
    GS = [800, 425] # Grid Size
    MP = 360 # Number of measurement points
    RD = 180.0 / np.pi # Radians to degrees

    def __init__(self):
        pass

    def setup(self, locations, delays = None, freq=40000, dist=3.0):
```
if not delays:
    self.delays = [0 for i in range(len(locations))]
else:
    self.delays = delays
self.dist = dist
self.ZOF = dist * 2. / (self.GS[0] * self.GMF)
self.xmid = self.ZOF * self.GS[0] / 2.
loc = np.array(locations)
self.locations = np.subtract(loc, np.average(loc))
self.elements = len(self.locations)
self.bounds = [[self.GS[0] * self.ZOF / -2., 0.],
self.freq = freq * 2 * np.pi
self.sim_freq = freq * self.SIM_STEPS / (2 * np.pi)
self.plot_angles = [(-90. + i * (180. / self.MP))
                    for i in range(self.MP)]
self.elementResponse()

def elementResponse(self):
    response = [0.05, 0.10, 0.18, 0.30, 0.52, 0.64, 0.75, 0.86, 0.96,
                1.0, 0.96, 0.86, 0.75, 0.64, 0.52, 0.30, 0.18, 0.10, 0.05]
    response = [1.0 for i in range(19)]
                      0., 10., 20., 30., 40., 50., 60., 70., 80., 90.]
    self.response = interp1d(response_angles, response, kind='cubic')

def preSimulation(self):
    state_size = (self.GS[1], self.GS[0], self.elements)
    self.pressures = np.zeros((self.GS[1], self.GS[0], self.SIM_STEPS))
    self.distances = np.zeros(state_size)
    self.angles = np.zeros(state_size)
    self.off_axis = np.zeros(state_size)
    self.rangex = np.arange(self.bounds[0][0], self.bounds[1][0], self.ZOF)
    self.rangey = np.arange(self.bounds[0][1], self.bounds[1][1], self.ZOF)
    zerooffx = np.tile(self.rangex, (self.GS[1], 1))
```python
zerooffy = np.tile(self.rangey.reshape(self.GS[1], 1), (1, self.GS[0]))

for i in range(self.elements):
    distx = np.subtract(zerooffx, self.locations[i])
    disty = np.subtract(zerooffy, 0.0)
    self.distances[:, :, i] = np.sqrt(np.add(np.power(distx, 2),
                                              np.power(disty, 2)))

    self.angles[:, :, i] = np.subtract(np.multiply(
                                             np.arctan2(disty, distx), self.RD), 90)

    self.angles[np.where(self.angles < -90.0)] += 180
    self.off_axis = self.response(np.abs(self.angles))

    self.inv_dist = np.reciprocal(np.add(self.distances, self.ZOF))
    self.combined_effect = np.multiply(self.inv_dist, self.off_axis)

def simulation(self):
    for t in range(self.SIM_STEPS):
        norm_amp = self.calculatePressureAmplitude(t)
        self.pressures[:, :, t] = np.sum(norm_amp, axis=2)

    self.findMaximumPressures(dim=2)

def calculatePressureAmplitude(self, the_time):
    dd = np.divide(self.distances, self.SOUND_SCOE)
    current_time = the_time / self.sim_freq
    pd = np.add(self.delays, current_time)
    phase = np.multiply(self.freq, np.subtract(pd, dd))
    amp = np.cos(phase)

    normalized_amp = np.multiply(self.combined_effect, amp)

    return normalized_amp

def findMaximumPressures(self, dim=2):
    self.max_pressures = np.multiply(np.log10(np.multiply(
                                           (np.amax(np.abs(self.pressures), dim)),
                                           self.NORM)), 20)

def evaluate(self):
```
pts = np.array(range(self.MP))
cosx = np.cos(np.divide(pts, (float(self.MP) / np.pi)))
oxoff = np.multiply(cosx, self.GS[0] * self.GMF / 2.)
x = np.add(xoff, self.GS[0] / 2.0).astype(int)
siny = np.sin(np.divide(pts, (float(self.MP) / np.pi)))
y = np.multiply(siny, self.GS[0] * self.GMF / 2.).astype(int)

self.measurements = [self.max_pressures[y[i]][x[i]]
    for i in range(self.MP)]
return (self.measurements, self.plot_angles,
    self.max_pressures, (self.rangex, self.rangey),
    (x, y))

class PhasedArrayQuickSim():

    SIM_STEPS = 15 # Number of timesteps in simulation.
    NORM = 1.0 # Normalization factor for amplitudes, if y-axis values matter.
    SOUND_SPEED = 340.0 # Speed of Sound

    def __init__(self):
        pass

    def setup(self, locations, delays = None, freq = 40000, points = 500):
        if not delays:
            self.delays = [0 for i in range(len(locations))]
        else:
            self.delays = delays

        loc = np.array(locations)
        self.locations = np.subtract(loc, np.average(loc))
        self.elements = len(self.locations)

        self.freq = freq * 2 * np.pi
        self.sim_freq = freq * self.SIM_STEPS

        self.angles = np.array([(-90. + i * (180. / points))
            for i in range(points)])

        self.elementResponse()
Appendix C. Simulation & Other Software

```
    def elementResponse(self):
        response = [0.05, 0.10, 0.18, 0.30, 0.52, 0.64, 0.75, 0.86, 0.96, 1.0, 0.96, 0.86, 0.75, 0.64, 0.52, 0.30, 0.18, 0.10, 0.05]
        response = [1.0 for i in range(19)]
        self.response = interp1d(response_angles, response, kind='cubic')

    def evaluate(self, distance):
        quadangles = np.multiply(np.add(self.angles, 90.0), np.pi / 180.0)
        x = np.tile(np.multiply(np.cos(quadangles), distance),
                    (len(self.locations), 1)).T
        y = np.tile(np.multiply(np.sin(quadangles), distance),
                    (len(self.locations), 1)).T
        time = np.array([float(t) / self.sim_freq
                         for t in range(self.SIM_STEPS)])
        dx = np.subtract(x, np.tile(self.locations, (len(self.angles), 1)))
        dy = y
        pressures = np.zeros((len(self.angles), self.SIM_STEPS))
        distances = np.sqrt(np.add(np.power(dx, 2), np.power(dy, 2)))
        transangs = np.subtract(np.multiply(np.arctan2(dy, dx), 180.0 / np.pi), 90.)
        off_axis = self.response(np.abs(transangs))
        inv_dist = np.reciprocal(np.add(distances, 0.001))
        combined_effect = np.multiply(inv_dist, off_axis)
        distdelay = np.tile(np.divide(distances, self.SOUND_SPEED),
                             (len(time), 1, 1))
        transdelay = np.tile(np.array(self.delays), (len(time),
                                                  len(self.angles), 1))
        timedelay = np.tile(np.reshape(time, (len(time), 1, 1)),
                            (1, len(self.angles), len(self.delays)))
        totaldelay = np.add(distdelay, np.add(transdelay, timedelay))
        totalphase = np.multiply(totaldelay, self.freq)
```
totalamp = np.cos(totalphase)
normamp = np.multiply(combined_effect, totalamp)
measurements = np.multiply(np.log10(np.multiply(
    np.amax(np.abs(np.sum(normamp, axis=2)), axis=0), self.NORM)), 20)

return np.flipud(measurements), self.angles

C.2 File: highlevelfunctions.py

The file highlevelfunctions.py contains a class which manipulates data obtained from the phased arrays.

```python
import lowlevelfunctions
import datatools
import numpy as np
import serial
import time
import csv

LLF = lowlevelfunctions.LowLevelFunctions()
vis = datatools.DataVisualize()

class HighLevelFunctions():
    
    SOUND_SPEED = 340.0

    def __init__(self):
        
        self.dac_waveform_fpga = 'None'
        self.dac_waveform_calc = 'None'
        
        self.rx_locations = 'None'
        self.rx_distances = 'None'
        self.distance_array = 'None'
        
        self.dac_rate = 'None'
        self.adc_rate = 'None'
        self.adc_sample_times = 'None'

        self.adc_channels = 'None'
```
def connect(self):
    LLF.connectTo("ADC")
    LLF.connectTo("DAC")

def disconnect(self):
    LLF.disconnectFrom("ADC")
    LLF.disconnectFrom("DAC")

def configureDac(self, speed, blocks=1, channels=1, phases='None'):
    if phases != 'None':
        for i in range(len(phases)):
            coarse_phase = phases[i] - phases[i] % 64
            fine_phase = phases[i] % 64
            LLF.setDacCoarsePhase(i, coarse_phase)
            LLF.setDacFinePhase(i, fine_phase)
    LLF.setChannelCount("DAC", channels)
    self.dac_channels = channels
    self.dac_rate = LLF.setDacRefreshSpeed(speed)
    LLF.setBlockSize("DAC", blocks)
    self.dac_blocks = blocks
    LLF.loadOptions("DAC")

def configureAdc(self, speed, blocks=6400, channels=2, trim_time=0.000):
    LLF.setChannelCount("ADC", channels)
    rate = LLF.setAdcSampleRate(speed)
    LLF.setBlockSize("ADC", blocks)
    LLF.loadOptions("ADC")
    self.adc_sample_times = LLF.calculateAdcTimeArray(offset=trim_time)
    self.distance_array = self.timeToDistance(self.adc_sample_times)
    self.adc_channels = channels
    self.adc_blocks = blocks
    self.adc_rate = rate
def defineDacWaveform(self, the_type, params):

    if the_type == "InvertedCosine":
        freq = params["Frequency"]
        calc_freq = freq * LLF.DAC_SKIP_RATE * (self.adc_rate / self.dac_rate)

        amp = params["Amplitude"]

        self.dac_waveform_fpga = LLF.makeOneInvertedCosine(freq, amp)
        self.dac_waveform_calc = LLF.makeOneInvertedCosine(calc_freq, amp)

    elif the_type == "LinearChirp":
        start_freq = params["StartFreq"]
        calc_start_freq = start_freq * 8.0 * float(self.dac_rate / self.adc_rate)

        end_freq = params["EndFreq"]
        calc_end_freq = end_freq * 8.0 * float(self.dac_rate / self.adc_rate)

        amp = params["Amplitude"]
        cycle_count = params["CycleCount"]

        self.dac_waveform_fpga = LLF.makeLinearChirp(start_freq, end_freq, amp, cycle_count)
        self.dac_waveform_calc = LLF.makeLinearChirp(calc_start_freq, calc_end_freq, amp, cycle_count)

    elif the_type == "LinearCyclicChirp":
        start_freq = params["StartFreq"]
        calc_start_freq = start_freq * LLF.DAC_SKIP_RATE * (self.adc_rate / self.dac_rate)

        end_freq = params["EndFreq"]
        calc_end_freq = end_freq * LLF.DAC_SKIP_RATE * (self.adc_rate / self.dac_rate)

        amp = params["Amplitude"]
        cycle_count = params["CycleCount"]

        self.dac_waveform_fpga = LLF.makeLinearChirp(start_freq, end_freq, amp, cycle_count)
        self.dac_waveform_calc = LLF.makeLinearChirp(calc_start_freq, calc_end_freq, amp, cycle_count)
calc_end_freq, amp, cycle_count)

LLF.uploadDacWaveform(self.dac_waveform_fpga)

def defineRxTransducerLocations(self, locations):
    self.rx_locations = locations
    self.rx_distances = locations

def obtainSamples(self):
    samples = LLF.getOneAdcSampleBlock()
    return samples

def transmitWaveform(self):
    LLF.queueTransmitBlock()

def sampleAndTransmit(self, first="sample", time_between=0.0):
    if first == "sample":
        self.time_between_sample_and_transmit = time_between
        samples = self.obtainSamples()
        time.sleep(time_between)
        self.transmitWaveform()

    elif first == "transmit":
        self.time_between_sample_and_transmit = -1 * time_between
        self.transmitWaveform()
        time.sleep(time_between)
        samples = self.obtainSamples()

    return samples

def timeToDistance(self, times):
    distance_array =
        [times[i] * self.SOUND_SPEED / 2 for i in range(len(times))]
    return distance_array

def interpolate(self, the_array, new_times, method='linear'):
    interp_array = np.zeros(the_array.shape)
if method=='linear':
    for j in range(the_array.shape[0]):
        interp_array[j, :] = np.interp(new_times[j, :],
                                       self.adc_sample_times,
                                       the_array[j, :])
elif method=='cubic':
    for j in range(the_array.shape[0]):
        f = interp1d(self.adc_sample_times, the_array[j, :], kind='cubic',
                     bounds_error=False, fill_value=2048)
        interp_array[j, :] = f(new_times[j, :])
return interp_array

def trimValuesAndAddVertically(self, the_array, trim=2048):
    out_array = np.abs(np.average(np.subtract(the_array, trim), 0))
    return out_array

def correlate(self, the_array, waveform):
    out_array = np.correlate(the_array, waveform, 'same')
    return out_array

def averageSamplesToFit(self, the_array, new_size):
    out_array = np.log10(np.average(np.reshape(the_array, (new_size, -1)), 1))
    return out_array

def computeTimeArrays(self, angles, samples='Default', channels='Default'):
    if samples == 'Default':
        samples = self.adc_blocks
    if channels == 'Default':
        channels = self.adc_channels
    self.time_arrays = np.zeros([len(angles), channels, samples])
    for i in range(len(angles)):
        delays = np.multiply(self.rx_distances,
```python
np.sin(np.radians(angles[i])) / self.SOUNDC_SPEED

for j in range(channels):
    self.time_arrays[i, j, :] = np.add(self.adc_sample_times,
                                        delays[j])

def processForImSonPlot(self, new_samples, angles):
    plot_size = len(angles)

    plot_data = np.zeros((plot_size, plot_size))

    for i in range(len(angles)):
        plot_data[:, i] = 
            self.averageSamplesToFit(
                self.correlate(
                    self.trimValuesAndAddVertically(
                        self.interpolate(new_samples, self.time_arrays[i])
                    ), self.dac_waveform_calc
                ), plot_size
            )

    return plot_data

def initializeImagingSonarPlot(self, plot_angles):

    vis.imagingSonarPlotInitialize(len(plot_angles),
                                   (plot_angles[0], plot_angles[-1]),
                                   (self.distance_array[0],
                                    self.distance_array[-1]),
                                    colors="RdBu")

def updateImagingSonarPlot(self, new_data, colorbar, vmin, vmax):

    vis.imagingSonarPlotUpdate(new_data, colorbar, vmin, vmax)

def panTiltConnect(self, port="/dev/ttyUSB0"):
    self.pan_tilt_ser = serial.Serial(port, 115200);
    time.sleep(3)

def panTiltDisconnect(self):
    self.pan_tilt_ser.close()
    time.sleep(1)
```
Appendix C. Simulation & Other Software

```python
def setPanTiltAngle(self, pan_angle, tilt_angle):
    if tilt_angle < 100:
        tilt_string = "T0" + repr(int(tilt_angle*10))
    else:
        tilt_string = "T" + repr(int(tilt_angle*10))

    if pan_angle < 100:
        pan_string = "P0" + repr(int(pan_angle*10))
    else:
        pan_string = "P" + repr(int(pan_angle*10))

    self.pan_tilt_ser.write(pan_string)
    self.pan_tilt_ser.write(tilt_string)

def makeSampleFileName(self, elements, trans_type, sample_rate, pan, tilt):
    type_elements_string = trans_type + repr(elements)
    sample_rate_string = "R" + repr(sample_rate)
    pan_tilt_string = "P" + repr(pan) + "T" + repr(tilt)
    extention = ".data"
    full_name = (type_elements_string + sample_rate_string +
                  pan_tilt_string + extention)

    return full_name

def saveSampleFile(self, samples, elements, trans_type, sample_rate, pan,
                    tilt):
    file_name = makeFileName(elements, trans_type, sample_rate, pan, tilt)
    the_file = open(file_name, 'wb')
    writer = csv.writer(the_file, quoting=csv.QUOTE_ALL)
    writer.writerows(samples)

def loadSampleFile(self, file_name):
    file_name = makeFileName(elements, trans_type, sample_rate, pan, tilt)
    the_file = open(file_name, 'wb')

    return the_file

def saveAmplitudesFile(self, amplitudes, pans, tilts, name):
    file_name = name
    fh = open(file_name, 'wb')
```

for i in range(len(amplitudes)):
    panstr = "P" + repr(pans[i])
    tiltstr = "T" + repr(tilts[i])
    ampstr = "A" + repr(amplitudes[i]) + "\n"
    wholestr = panstr + tiltstr + ampstr
    fh.write(wholestr)

fh.close()

def openAmplitudesFile(self, name):
    fh = open(name)
    pans = []
    tilts = []
    amps = []

    for line in fh.readlines():
        pan = line[(line.find("P") + 1):line.find("T")]
        tilt = line[(line.find("T") + 1):line.find("A")]
        amp = line[(line.find("A") + 1):line.find(";")]
        pans.append(float(pan))
        tilts.append(float(tilt))
        amps.append(float(amp))

    return pans, tilts, amps
Appendix D

Optimization & Analysis Software

D.1 File: genetic.py

The file genetic.py contains the functions required for implementation of the genetic algorithms.

```python
import numpy as np
import fassim

class PAGA():
    def __init__(self):
        pass

    def setup(self, distances, angles=250, lobe_cost=-1., size_cost=2.0,
              survival_rate=0.2, pop_size=200, member_size=16,
              member_minimum=0.014, mutation_variance=0.01):
        self.eval_dist = distances
        self.angles = angles
        self.lobe_cost = lobe_cost
        self.size_cost = size_cost
        self.survival_rate = survival_rate
        self.pop_size = pop_size
        self.member_size = member_size
        self.min_size = member_minimum
        self.variance = mutation_variance
```
def prune(self):
    member_cost = [99999 for i in range(len(self.population))]
    for i in range(len(self.population)):
        member_cost[i] = self.findCost(self.population[i])
    remaining = self.survival_rate * len(self.population)
    survivor_index = np.array(member_cost).argsort()[:remaining]
    survivors = [self.population[item] for item in survivor_index]
    best_member = self.population[np.argmin(member_cost)]
    self.ax.clear()
    self.sim.setup(self.makeLocations(best_member), points=self.angles)
    for i in range(len(self.eval_dist)):
        result, a = self.sim.evaluate(self.eval_dist[i])
        self.ax.plot(a, np.subtract(result, np.amax(result)))
    self.ax.set_ylim([-50, 0])
    self.ax.grid(True)
    self.fig.canvas.draw()
    self.printToTerminal(member_cost[np.argmin(member_cost)], best_member)
    return survivors

def printToTerminal(self, cost, member):
    print "Best Cost: " + repr(cost)
    print "Locations:"
    print self.makeLocations(member)

def cross(self):
    #print "\nCrossing Chromosomes Now."
for i in range(int(self.pop_size / 20)):
    ma = int(np.random.rand() * self.pop_size)
    mb = int(np.random.rand() * self.pop_size)
    ca = int(np.random.rand() * self.member_size)
    cb = int(np.random.rand() * self.member_size)

    temp = self.population[ma][ca]
    self.population[ma][ca] = self.population[mb][cb]
    self.population[mb][cb] = temp

    def mutate(self, survivors):
        # print "\nMutating Population Now."
        new_population = np.repeat(survivors, int(1/self.survival_rate), axis=0)

        y = np.shape(new_population)[0]
        x = np.shape(new_population)[1]
        mutation = np.multiply(np.random.randn(y, x), self.variance)

        # Only mutate 20% of the chromosomes.
        mutation[np.where(np.random.rand(y, x) > 0.2)] = 0.0

        self.population = np.add(new_population, mutation)
        self.population[np.where(self.population <= self.min_size)] = self.min_size

        def findCost(self, member):
            ratios = [0. for i in range(len(self.eval_dist))]
            locations = self.makeLocations(member)
            self.sim.setup(locations, points=self.angles)

            for i in range(len(self.eval_dist)):
                result, a = self.sim.evaluate(self.eval_dist[i])
                ratios[i] = self.findLobeRatio(result)

            lobe_ratio_cost = np.amin(ratios) * self.lobe_cost
            width_cost = (locations[0] + locations[-1]) * self.size_cost
            cost = lobe_ratio_cost + width_cost

            return cost

        def findLobeRatio(self, values):
            center = self.angles / 2
main_lobe_value = values[center]
values[(center - self.ignore_width):(center + self.ignore_width)] = -99
side_lobe_value = values[np.argmax(values)]
ratio = main_lobe_value - side_lobe_value
return ratio

def makePopulation(self, members, member_size):
    self.population = np.add(np.abs(np.multiply(
        np.random.randn(members, member_size), self.variance)), self.min_size)
def makeLocations(self, member):
    locations = [0 for i in range(len(member))]
    for i in range(1, len(member)):
        locations[i] = locations[i-1] + member[i]
    return locations

D.2 File: analysis.py

The file analysis.py contains several high-level functions applicable to both measured and simulated phased arrays.

from pylab import imshow, colorbar
import matplotlib.pyplot as plt
import numpy as np
import fastsim
import genetic
import time

QS = fastsim.PhasedArrayQuickSim()
GS = fastsim.PhasedArrayGridSim()

def makeArrayConfig(configuration):
    if configuration['type'] == 'log':
        k = configuration['value']
        minimum = configuration['minimum']
count = configuration["count"]
spacings = [minimum for i in range(count)]
for i in range(1, count):
    spacings[i] = spacings[i-1] * (1 + k)

print np.average(spacings) * 40000./340.

elif configuration["type"] == "linear":
    minimum = configuration["minimum"]
count = configuration["count"]
spacings = [minimum for i in range(count)]

locations = makeLocations(spacings)
return locations

def makeLocations(spacings):
    locations = [0 for i in range(len(spacings))]
    for i in range(1, len(spacings)):
        locations[i] = locations[i-1] + spacings[i]
    return locations

def plotPressure(pressure, axes=False, transducers=False, eval_pts=False, extra_lines=False):
    plt.figure(facecolor='white', dpi=105)
    ax = plt.subplot(111)
    if eval_pts:
        ax.plot(eval_pts[0], eval_pts[1], '--', lw=3, alpha=0.25, mfc='blue')
    if extra_lines:
        width = np.shape(pressure)[1]
        for i in range(len(extra_lines)):
            new_eval_pts = np.multiply(eval_pts, extra_lines[i])
x_avg = int(np.average(new_eval_pts[0]))
            ax.plot(np.add(np.subtract(new_eval_pts[0], x_avg), width/2),
                    new_eval_pts[1], '--', lw=3, alpha=0.25)
xim = imshow(pressure, origin='lower', cmap='RdBu', interpolation='bicubic')
    if transducers:
        ax.plot(locx, locy, '0', ms=5, lw=2, alpha=0.7, mfc='orange')
61  if axes:
62      y = ax.get_yaxis()
63      ylen = len(y.get_major_ticklocs()) - 1
64      y_lab = ["%0.2f"%i for i in np.linspace(axes[1][0], axes[1][-1], ylen)]
65      y_lab.insert(0, ' ')
66      y_lab[1] = ' '
67      y.set_ticklabels(y_lab)
68      y.set_label_text("Position (Meters)")
69
70      x = ax.get_xaxis()
71      xlen = len(x.get_major_ticklocs()) - 1
72      x_lab = ["%0.2f"%i for i in np.linspace(axes[0][0], axes[0][-1], xlen)]
73      x_lab.insert(0, ' ')
74      x_lab[1] = ' '
75      x.set_ticklabels(x_lab)
76      x.set_label_text("Position (Meters)")
77
78      ax.grid(True)
79
80      # plt.colorbar(im)
81      plt.show()
82
83  def plotPolarMagnitude(plot_angles, measurements):
84      fig = plt.figure(facecolor='white', dpi=105)
85      for i in range(len(measurements)):
86          ax = fig.add_subplot(111)
87          ax.plot(plot_angles, measurements[i], alpha=0.5, lw=2)
88          ax.grid(True)
89          plt.xlabel("Off-Axis Angle (Degrees)")
90          plt.ylabel("Reduction From Main Lobe (dB)")
91          plt.show()
92
93  def normalize(the_array):
94      new_array = np.subtract(the_array, np.amax(the_array))
95      return new_array
Bibliography


