FPGA-BASED ADAPTIVE DIGITAL BEAMFORMING USING MACHINE LEARNING FOR MIMO SYSTEMS

by

John Gentile

A thesis submitted to Johns Hopkins University

in conformity with the requirements for the degree of

Master of Science

Baltimore, Maryland

May 2021

Abstract

In modern Multiple-Input and Multiple-Output (MIMO) systems, such as cellular and Wi-Fi technology, an array of antenna elements is used to spatially steer RF signals with the goal of changing the overall antenna gain pattern to achieve a higher Signal-to-interference-plus-noise ratio (SINR). Digital Beamforming (DBF) achieves this steering effect by applying weighted coefficients to antenna elements- similar to digital filtering- which adjust the phase and gain of the received, or transmitted, signals. Since real world MIMO systems are often used in dynamic environments, Adaptive Beamforming techniques have been used to overcome variable challenges to system SINR- such as dispersive channels or inter-device interference- by applying statistically-based algorithms to calculate weights adaptively.

However, large element count array systems, with their high degrees of freedom (DOF), can face many challenges in real application of these adaptive algorithms. These statistical matrix methods can be either computationally prohibitive, or utilize non-optimal simplifications, in order to provide adaptive weights in time for an application, especially given a certain system's computational capability; for instance, MIMO communication devices with strict size, weight and power (SWaP) constraints often have processing limitations due to use of low-power processors or Field-Programmable Gate Arrays (FPGAs).

Thus, this thesis research investigation will show novel progress in these adaptive MIMO challenges in a twofold approach. First, it will be shown that advances in Machine Learning (ML) and Deep Neural Networks (DNNs) can be directly applied to the computationally complex problem of calculating optimal adaptive beamforming weights via a custom Convolutional Neural Net (CNN). Secondly, the derived adaptive beamforming CNN will be shown to efficiently map to programmable logic FPGA resources which can update adaptive coefficients in real-time. This machine learning implementation is contrasted against the current state-of-the-art FPGA architecture for adaptive beamforming- which uses traditional, Recursive Least Squares (RLS) computation- and is shown to provide adaptive beamforming weights faster, and with fewer FPGA logic resources. The reduction in both processing latency and FPGA fabric utilization enables SWaP constrained MIMO processors to perform adaptive beamforming for higher channel count systems than currently possible with traditional computation methods.

Thesis Committee

Primary Readers

Jeff Houser(Advisor)
Johns Hopkins University Whiting School of Engineering
Engineering for Professionals

Ashutosh Dutta
Johns Hopkins University Whiting School of Engineering
Engineering for Professionals

Doug Wenstrand
Johns Hopkins University Whiting School of Engineering
Engineering for Professionals

Acknowledgments

Thanks to my friends and family for their continuous support throughout this whole research process.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern Multiple-Input-Multiple-Output (MIMO) systems, such as those used in cellular and Wi-Fi communication technologies, are often developed to optimally service multiple end users. To do so, the multiple antenna elements are usually coordinated in a fashion that allows the radiation gain pattern to be "steered" in space towards the direction of a specific user, so that user sees high signal strength, and other users see attenuation as to not interfere with that user's communication channel. This beamforming process can be visualized as a directional beam as in Figure 1.1.

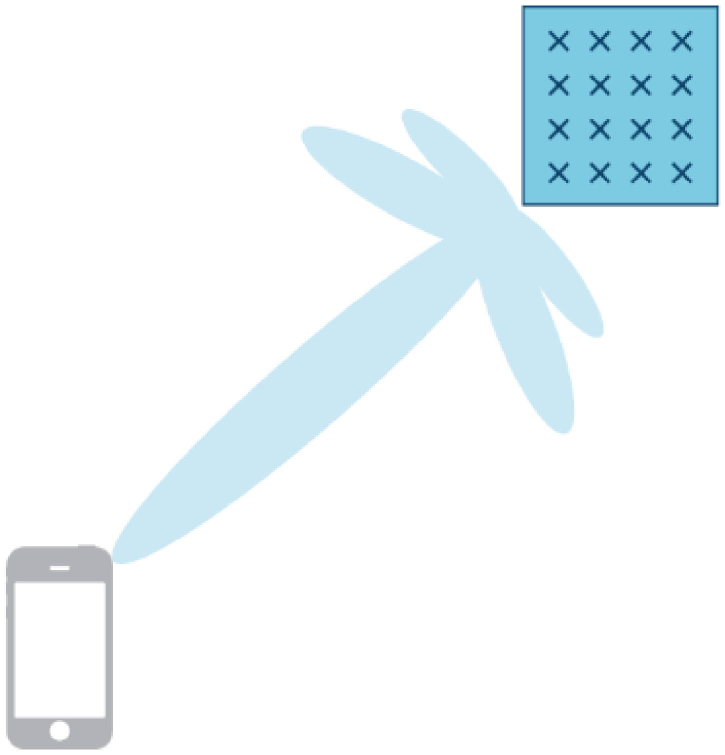


Figure 1.1: Beamforming a Signal from MIMO System to User

Source: Adapted from [1]

However, since real world MIMO systems are often used in dynamic environments, with constantly shifting sources of interference, Adaptive Beamforming techniques have been used to nullify interferers from disrupting the intended communication channel. Large element count array systems though, with their high degrees of freedom (DOF)- a metric of how many simultaneous interference sources can be nulled-, require significant processing in real calculation and application of these adaptive beamforming algorithms. For SWaP constrained, embedded implementations, these processing requirements may be too great for a given system to calculate the adaptive beamforming weights in time to be useful (e.g. to adapt to a rapidly changing spatial interference environment).

The problem worsens in massive MIMO systems where systems that must service multiple users, in the same or adjacent frequency band, compete for communication bandwidth and interfere with each other like in Figure 1.2 and as such, efficient adaptive beamforming algorithms will be necessary to practically support next generation massive MIMO systems [2].

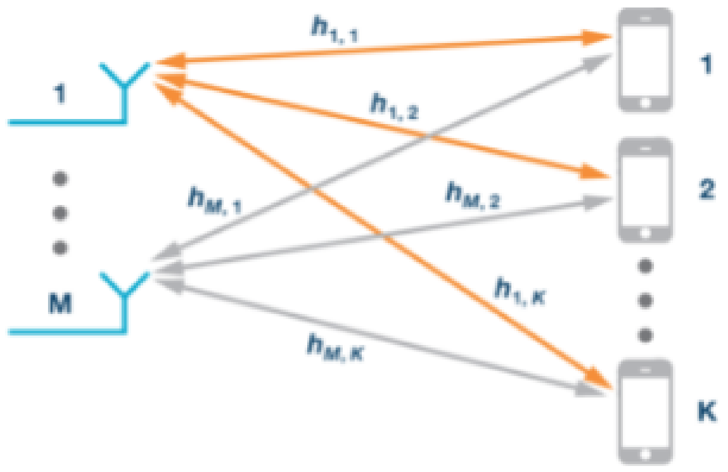


Figure 1.2: Multiple Simultaneous Users in MIMO System

Source: Adapted from [1]

Since processing complexity of adaptive beamforming grows exponentially with channel count, a more efficient adaptive beamforming algorithm than traditional methods could allow an edge device- such as a 5G radio tower- to calculate the adaptive beamforming weights directly at the edge. To support this end goal, this research will go over the background of traditional adaptive beamforming methodology and applicability to a low-power FPGA device, as commonly used in modern MIMO communication devices, like 5G radio heads [3], [4].

After a baseline implementation using the current state-of-the-art FPGA architecture is established, this research will show a novel method in applying recent advances in Deep Learning to the Adaptive Beamforming weight calculation problem set. To show a real-world, deployable implementation of the deep learning model, this work will also show an architecture, hosted in FPGA programmable-logic fabric, to compute adaptive weights in a more efficient manner than previous implementations.

References

[1] "Massive mimo and beamforming: The signal processing behind the 5g buzzwords," Analog Devices, Tech. Rep., 2017. [Online]. Available: https://www.analog.com/en/analog-dialogue/articles/massive-mimo-and-beamforming-the-signal-processing-behind-the-5g-buzzwords.html#.

[2] S. Rangan, T. S. Rappaport, and E. Erkip, "Millimeter-wave cellular wireless networks: Potentials and challenges," *Proceedings of the IEEE*, vol. 102, no. 3, pp. 366–385, 2014. DOI: 10.1109/JPROC.2014.2299397.

[3] "Verizon completes first successful end-to-end fully virtualized 5g data session in the world," 2020. [Online]. Available: https://www.verizon.com/about/news/verizon-fully-virtualized-5g-data-session.

[4] R. L. Maistre, "Xilinx unveils radio platform for second wave of 5g," 2020. [Online]. Available: https://www.telecomtv.com/content/5g/xilinx-unveils-radio-platform-for-second-wave-of-5g-40038/.

Chapter 2

Adaptive Beamforming Background

2.1 Introduction

In this section, the concept and application of beamforming, as well as the method of adaptive beamforming, is introduced. It's assumed the reader is familiar with the fundamentals of discrete signal processing techniques, such as Finite Impulse Response (FIR) filtering.

2.2 Beamforming and Array Basics

Filtering is a commonly used operation in signal processing; in the discrete sense, samples are passed through a set of filter coefficients, or "taps", to perform the convolution and achieve the desired response. Analogous to such temporal filtering, an array of sensors can be filtered *spatially* to produce a desired response across the elements.

Specifically in the context of Radio Frequency (RF) antenna arrays, this spatial filtering can be utilized to optimize the overall antenna pattern, in a process commonly known as *beamforming* [1]. The specific spatial optimization is often application dependent, however beamforming is generally seen as a method of beam steering, where gain is provided in a specific, desired direction- relative to the array's front-, with attenuation in other angles. Though the term "beamforming" sounds specific to transmitting applications- like radiating RF arrays-, beamforming, and consequently spatial filtering, can actually be performed on both the transmit and receive functions of any array, also known as *array reciprocity* [1], [2]. Beamforming is inclusive of non-RF arrays and applications as well, such as sound transducers used in SONAR arrays [1].

The fundamental operation of beamforming is derived from the properties of constructive and destructive interference of propagating waves in *phased array* systems. These systems are so named in that the individual array elements shift the phase of a received, or transmitted, signal to create a desired *far field* array pattern that culminates into a steered wavefront, as illustrated in Figure 2.1.

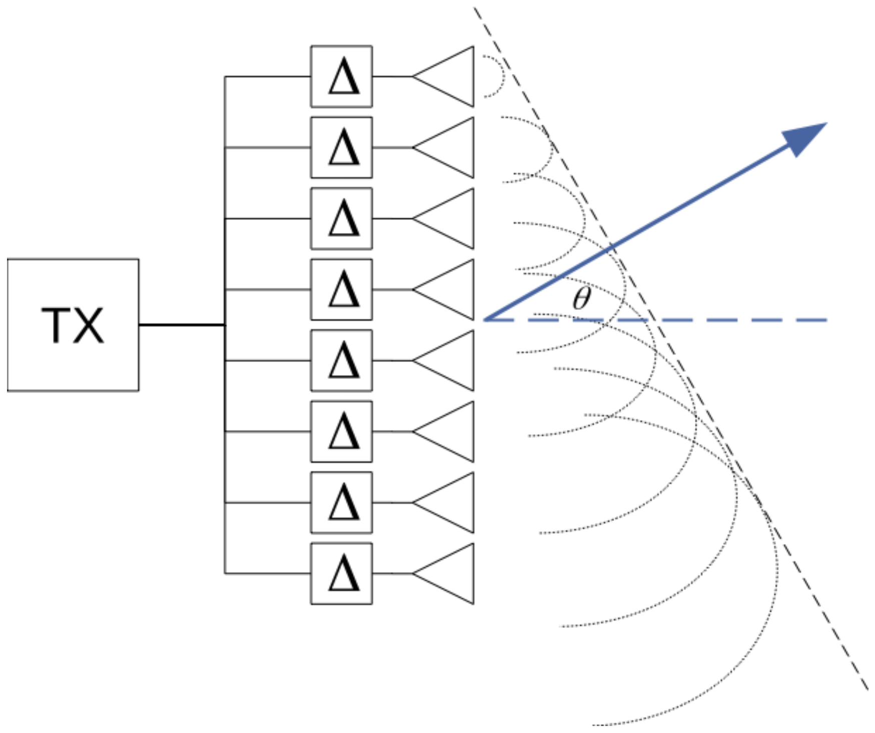


Figure 2.1: Phased Array Wavefront Steering

This phase shifting process can be acheived by digital or analog means. For instance in Figure 2.1, the Δ blocks could be analog phase shifter units that perform the beam steering in the RF/analog domain. In this case, each phase shifter unit is attached to an individual array antenna element, and is manifolded to a single receiver- such as an Analog to Digital Converter (ADC)- and/or a single transmitter- such as a Digital to Analog Converter (DAC). The benefits of such a system is simplicity in the digital and RF electronics, as there is only one ADC and/or DAC- and possibly one mixing/heterodyne system for the array-, however the system is much less flexible in that it can only steer in one direction at a time. For Single-Input Single-Output (SISO) systems, this architecture may suffice.

However for Multiple-Input Multiple-Output (MIMO) or other systems that need more flexibility, these Δ phase shifting blocks could also be performed in the digital domain. In this case, each antenna element can be considered to be directly connected to an ADC and/or DAC and the associated phase shifts can be performed in digital logic- such as in a Field-Programmable Gate Array (FPGA) directly connected to each ADC/DAC- and then coherently combined to form the intended beam(s) [2]. The downside of a digitally beamformed system is increased complexity- and thereby often an increased cost- due to each channel requiring RF and sampling electronics that must be phase synchronous, however the upside is this system is much more flexible in how it can apply phase shifts, as well as it creates the opportunity for a system to create multiple spatial beams at one time [1].

For MIMO communication arrays, these properties of directional gain and attenuation can be exploited for servicing multiple users, such as in Spatial Multiplexing [3] where distinct users are assumed to be in different spatial locations or directions, so digital beamforming with multiple beams can be used to target each user independently at the same time.

In the case of a Uniform Linear Array (ULA)- which we will be using for the majority of this investigation- the standard digital beamforming architecture can be seen in Figure 2.2.

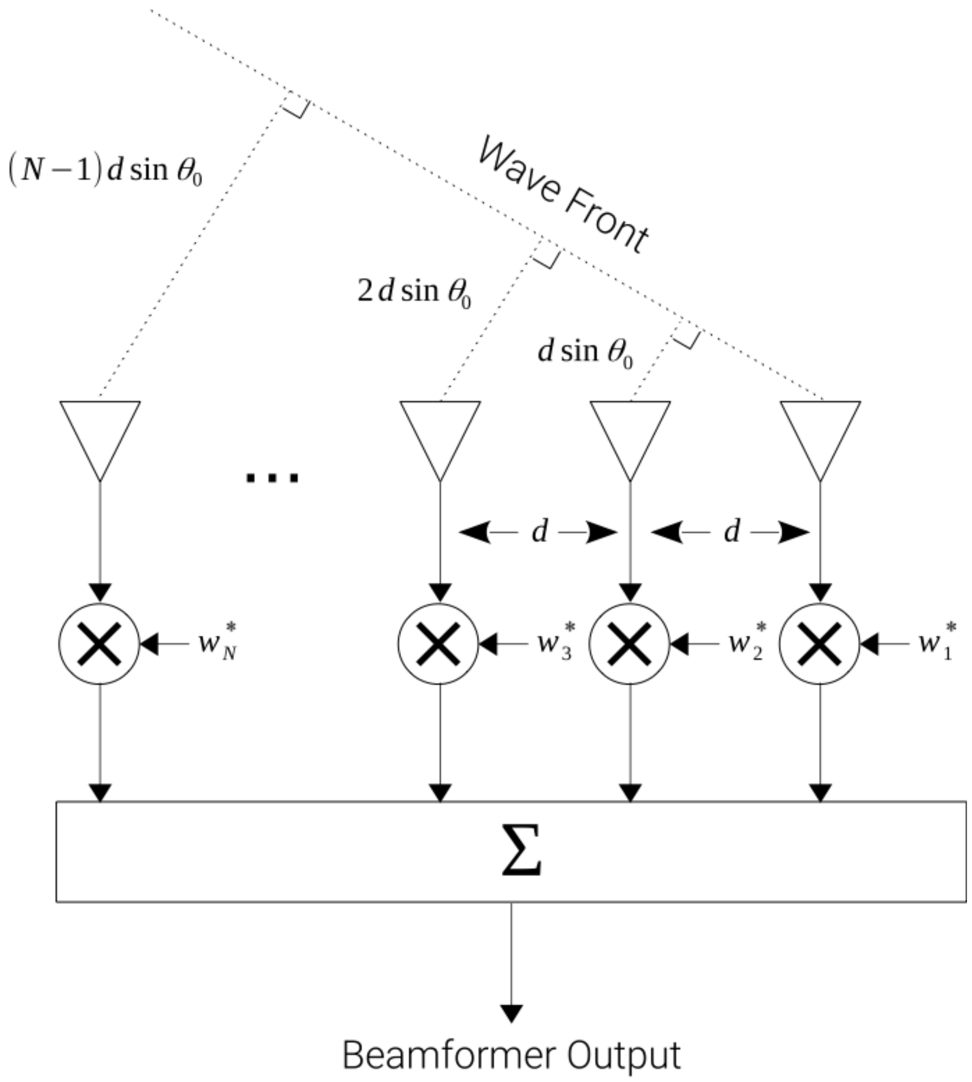


Figure 2.2: ULA Beamformer

A ULA is defined as an array with N elements equally spaced a distance d from each other along a linear axis [2], [4]. Each RF channel- related to an RF antenna element- is sampled synchronously such that the digital samples are aligned in time across all channels so coherent processing can be performed.

It can be seen that when dealing with a signal from the far field impinging on the array with angle, θ_0, the difference in propagation path length, L, between elements in a ULA is given by Equation 2.1 [2], [4].

$$L(n) = nd\sin(\theta_0), \quad 0 \leq n \leq N-1 \tag{2.1}$$

The reason we assume far field characteristics for the majority of this work to simplify the math and operations of phased arrays; for the case of a phased array receiver in the near field, an RF emitter is so close to the array that the incident angle of the received energy is different for every element due to the spherical wavefront of the source, as shown in Figure 2.3.

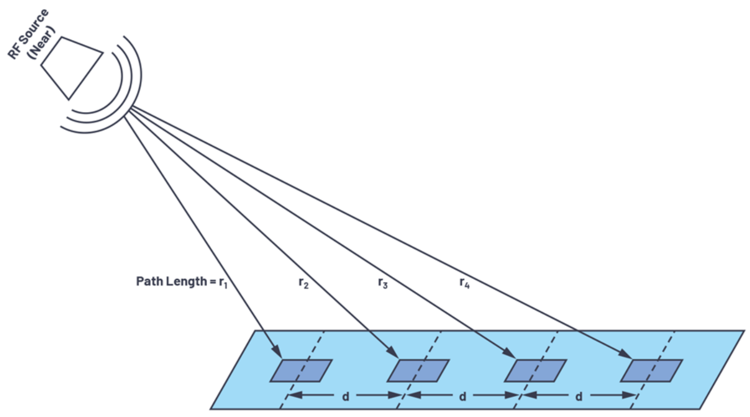


Figure 2.3: Near Field Response

Source: Adapted from [2]

However, in the far field, where the same emitter is farther away from the receiving array, the wavefronts become approximately linear, and each receive element sees an equivalent incidence angle, θ, of the arriving wave, as in 2.4.

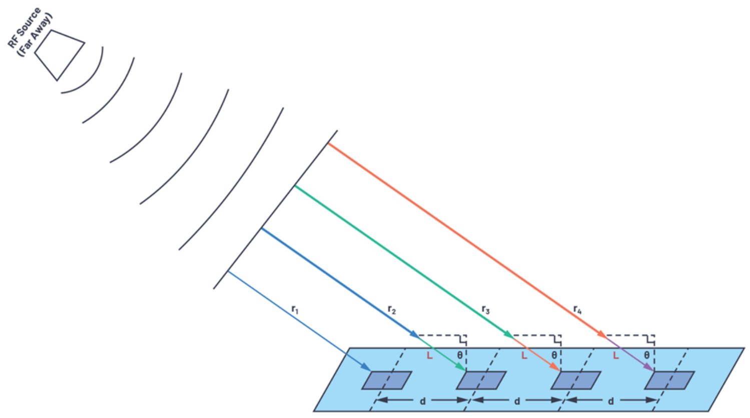


Figure 2.4: Far Field Response

Source: Adapted from [2]

The specific point at which a given system is operating in the far field is dependent on many factors of the array's antenna properties, however a general equation can be found from 2.2 based on an array's antenna diameter, D, and the wavelength of the operating carrier frequency, λ [2], [5].

$$FarField > \frac{2D^2}{\lambda} \tag{2.2}$$

2.2.1 Digital Beamforming Architecture

Once in the digital domain, there are several ways to perform phase shifting on the sampled baseband signals, mainly by way of time delay shifting or multiplication by a complex phasor [2], [4]. The decision to take a given approach is mainly dominated by RF system characteristics, though for narrowband signals that we will be using for this text, we will show that phase shifting is often more economical than time shifting.

True time delay shifting acts to match the time difference of a wavefront impinging on each different element such that when the time delayed channels are coherently summed together, the desired signal from that wavefront sees processing gain applied, while signals from other directions see attenuation [2]. Specifically, each antenna element will see a time shift of the same signal based on the specific path length L- defined from Equation 2.1- and the propagation speed of the signal over that distance- nominally assumed to be near the speed of light c, though varies on the medium and frequency- as in Equation 2.3.

$$t_{Delay}(n) = L(n)/c, \quad 0 \leq n \leq N-1 \tag{2.3}$$

However, achieving the exact time delay required for a given steering direction may not be practical. For instance, in digital logic, individual channels can be easily delayed using some number of registers in the datapath, however the time delay quanta is limited to the clock frequency of the logic, as $\Delta = 1/f_{Clk}$; so for a 200 MHz clock region in an FPGA, each signal can be delayed by an integer multiple of 5 nanosecond steps.

This level of delay precision may not be sufficient for some applications or frequencies, so instead of performing a true time delay on each channel's signal, often the time shift can be accurately approximated by an applied phase shift, especially for narrowband signals. When L is a fraction of the narrowband signal's wavelength, λ, the equivalent phase shift ϕ of the impinging signal at an incidence angle θ can be derived from Equation 2.4 [2], [6].

$$\phi = \frac{2\pi d}{\lambda} \sin\theta, \quad -\pi/2 \leq \theta \leq \pi/2 \tag{2.4}$$

The constraints on θ in 2.4 for ϕ to be valid are such that $d < \lambda/2$ so that there is no ambiguity between the value of the incidence angle and the desired phase shift; this *half-wavelength spacing* requirement of the array elements can be directly viewed as the *spatial analog of the Nyquist sampling theorem*, which similarly states a signal up to bandwidth B can be perfectly reconstructed given a sampling rate, f_s, that complies with $B < f_s/2$ [6].

Thus in narrowband systems that are designed for a specific carrier frequency, the array spacing often conforms to half the signal wavelength, which can lead to a further simplification of the elemental phase shift as in Equation 2.5.

$$\phi = \pi d \sin\theta, \quad d = \frac{\lambda}{2} \tag{2.5}$$

When an array uses digital phase shifting to perform beamforming on receive, the basic architecture can be seen in Figure 2.5 where each ADC channel is phase shifted by a complex weight value w_n. The digitized sample data from each channel, $x_n(k)$, is assumed to be complex- also commonly known as In-Phase and Quadrature data (I/Q) for digital RF systems- to retain phase information from each channel, where k is the time index for each sample.

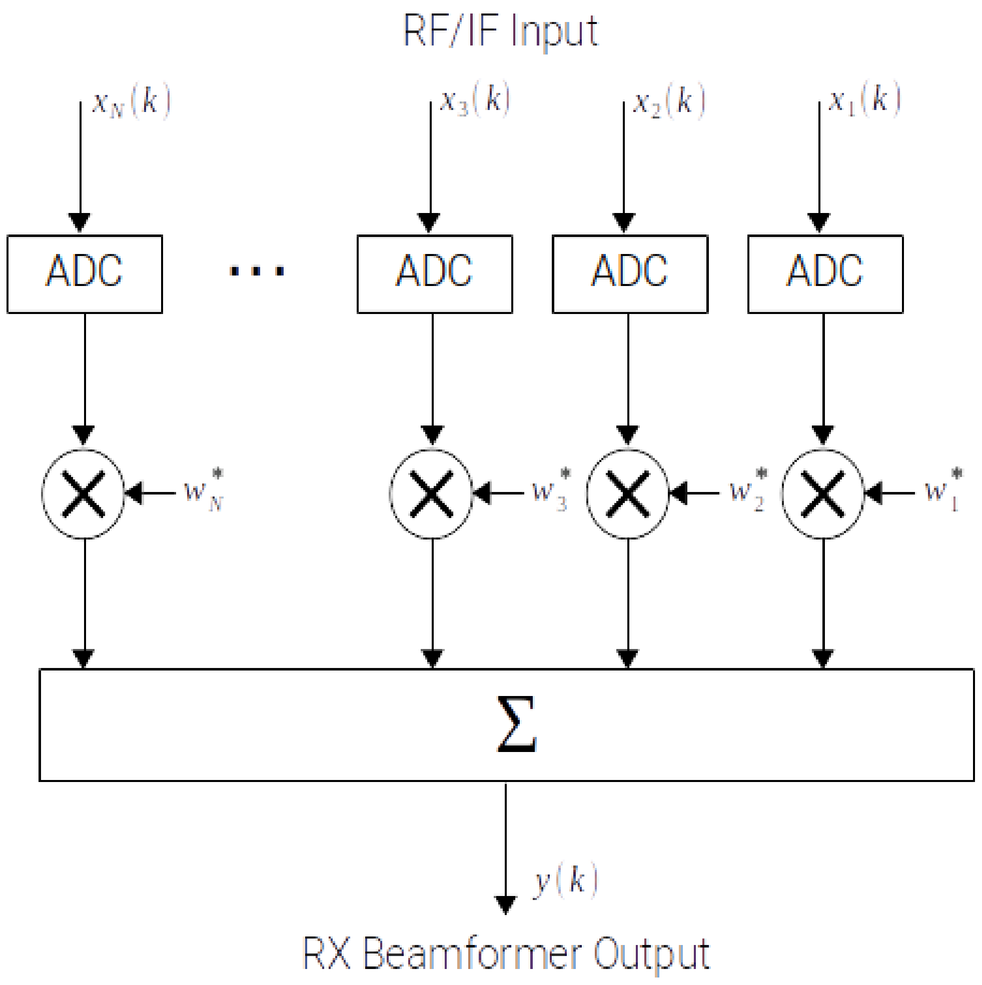


Figure 2.5: ULA Digital Beamforming on Receive

The beamformed output signal $y(k)$ is formed by summing the products of the complex conjugate of weights w_n and the input signals across each channel $x_n(k)$, as in Equation 2.6 [1], [6].

$$y(k) = \sum_{i=0}^{N-1} w_i^* x_i(k) \tag{2.6}$$

Equivalently in vector notation, this beamforming can be seen as the dot product of the N-by-1 complex weight column vector $\mathbf{w} = [w_1, w_2, \ldots, w_N]^T$

and the N-by-1 complex sample column vector $\mathbf{x}(k) = [x_1(k), x_2(k), \ldots, x_N(k)]^T$ at observation time k, as in Equation 2.7.

$$\mathbf{y}(k) = \mathbf{w}^H \mathbf{x}(k) \tag{2.7}$$

Note that the superscript $(\cdot)^H$ operator represents the Hermitian (complex conjugate) transpose applied to the weight vector, while $(\cdot)^T$ is the non-conjugate (normal) transpose operator [1].

Conversely, in the transmit case the opposite data flow occurs; we fan out- or DEMUX- a single transmit signal to each individual transmit channel, where again the complex weights are multiplied, and the resulting array's output has its beam steered in a given direction.

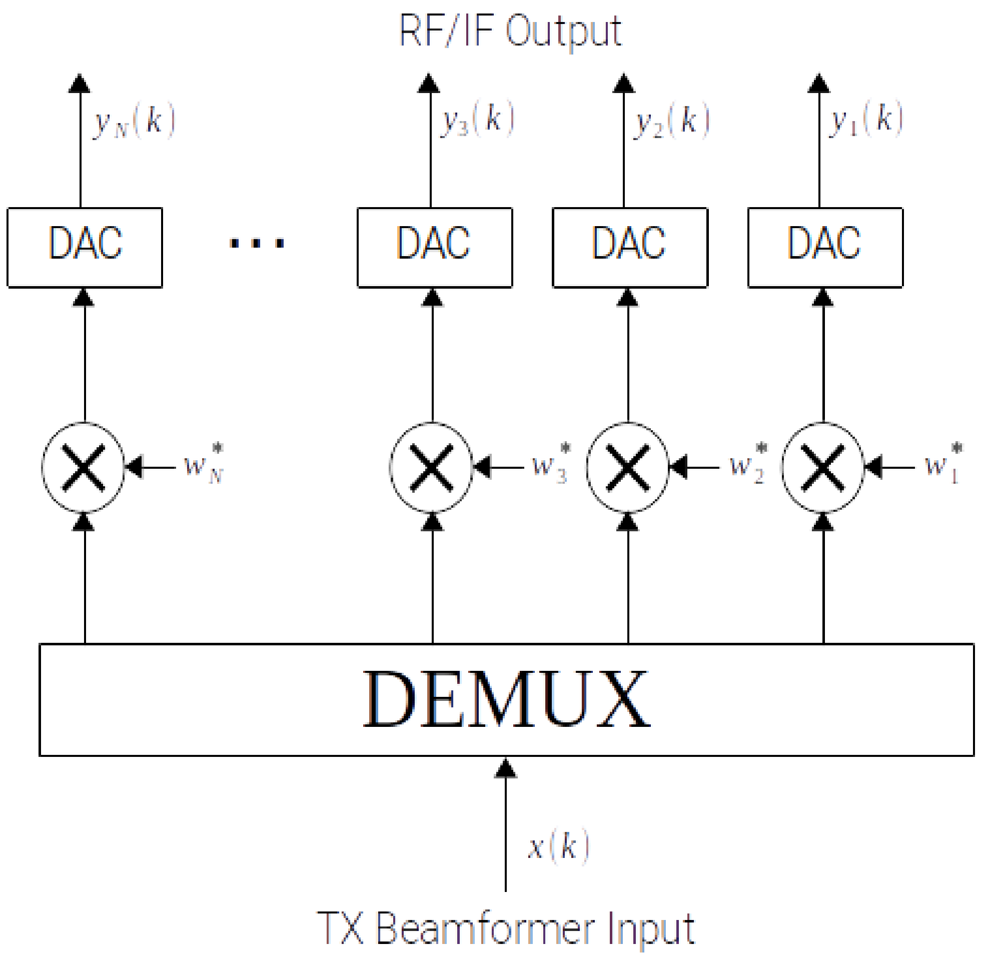


Figure 2.6: ULA Digital Beamforming on Transmit

2.2.2 Deterministic Beamforming

A standard beamformer, with quiescent beamforming weights w_n, can be called a *deterministic beamformer* [7]. The system is considered quiescent in the sense that the calculation of the beamforming weights need only depend on the intended steering direction of the array, with all other system properties static or not included as part of the calculation process.

For narrowband signals, the complex spatial response vector s_n is formed from the baseband envelope phasor at each ULA element [4]:

$$s_n = e^{j2\pi(n-1)\frac{d}{\lambda}\sin\theta_0} \quad 0 \leq n \leq N-1 \tag{2.8}$$

Equation 2.8 is a function of steering direction, or Angle of Arrival (AoA) θ_0, wavelength of the carrier frequency λ , and the elemental spacing of the n element array d [4], [6].

Again, analogous to temporal filtering, the ideal, quiescent beamforming weights can be seen to be the *matched filter* equivalent to the spatial response vector s_n to directly counteract the apparent phase shift across the array for an impinging wavefront, as in Equation 2.9. Note that the matched filter response is equivalent to the complex conjugate of the received response [6].

$$w_n = s_n^* \tag{2.9}$$

This quiescent response for $\theta = 0$ can be seen in the *radiation pattern plot* in Figure 2.7; a radiation pattern plot- also sometimes called an *azimuth cut* for a ULA- is used to show array gain versus angle of wave incidence, θ [6]. For example, a radiation plot of an ideal, omnidirectional, isotropic antenna would show constant gain across all angles. Here, we see that our main lobe expectedly shows up at an angle of 0, while all other angles show attenuation.

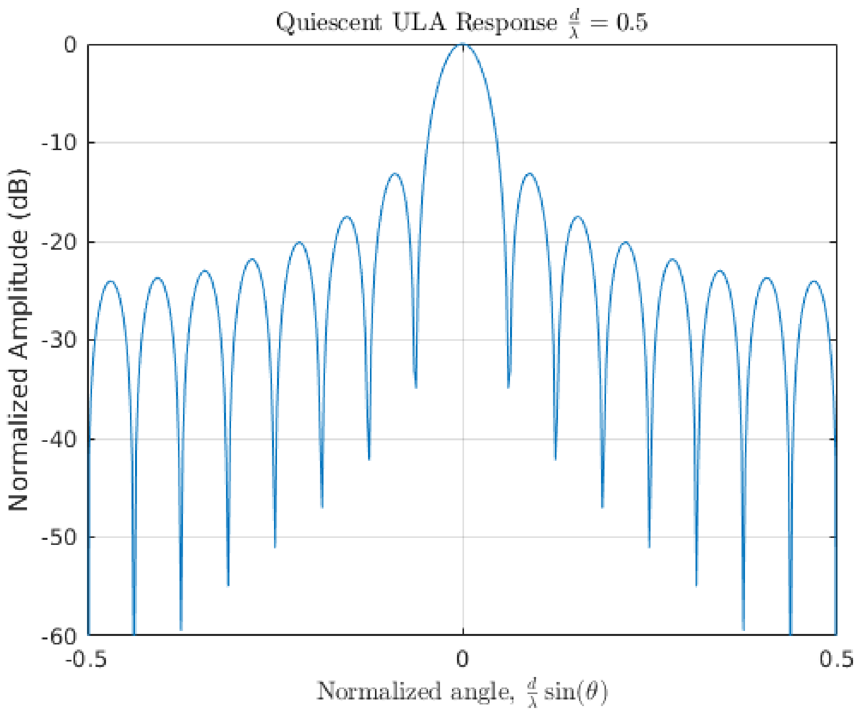


Figure 2.7: Normalized Radiation Pattern of ULA, $\theta = 0$, $N = 16$

Note also that it sometimes is useful to plot the x-axis of the radiation plot in *sine space*, where the angle is normalized by Equation 2.10 to show incidence angle based on array spacing and the incident signal's wavelength.

$$\theta_{Norm} = \frac{d}{\lambda}\sin(\theta) \tag{2.10}$$

This radiation plot is fundamentally showing the *array factor (AF)* of a system, which is defined in equation 2.11 as the total voltage response of an array as a sum of individual voltage responses, based on the difference between the steered angle ϕ and the incident angle θ [8].

$$AF(\theta) = \frac{1}{N}\sum_{n=1}^{N} e^{-j\left(2\pi(n-1)\frac{d}{\lambda}\sin(\theta_0-\phi_n)\right)} \tag{2.11}$$

When the incident angle θ is sweeped in 2.11, the resultant plot of Array Factor yields the radiation plot over incidence angles, as shown in Figure 2.7.

Note the similarity of the exponential/phasor term in the Array Factor equation 2.11 to the spatial response vector in Equation 2.8, which is again the complex conjugate of the quiescent beamforming weights; essentially, we can insert the beamforming weights directly into 2.11 to refactor the equation as in 2.12.

$$AF(\theta) = \frac{1}{N}\sum_{n=1}^{N} w_n e^{-j\left(2\pi(n-1)\frac{d}{\lambda}\sin(\phi_n)\right)} \tag{2.12}$$

This means we can take the same approach of sweeping receive incidence angles ϕ_n with Equation 2.12 with a set of beamforming weights to calculate the Array Factor, and create a resulting sine space plot.

For instance, to calculate the quiescent beamforming weights to steer the array to -10° with half-wavelength spacing, we simply plug $\theta_0 = -10^\circ$ into Equation 2.8 and then find the matched filter equivalent in Equation 2.9. Then we can plot the response over incident angles using Equation 2.12. The resultant sine space plot is shown in Figure 2.8 where we can see a gain peak at $0.5\sin(-10^\circ) \approx -0.09$ (the green vertical line plots exactly where we expect the gain peak in sine space).

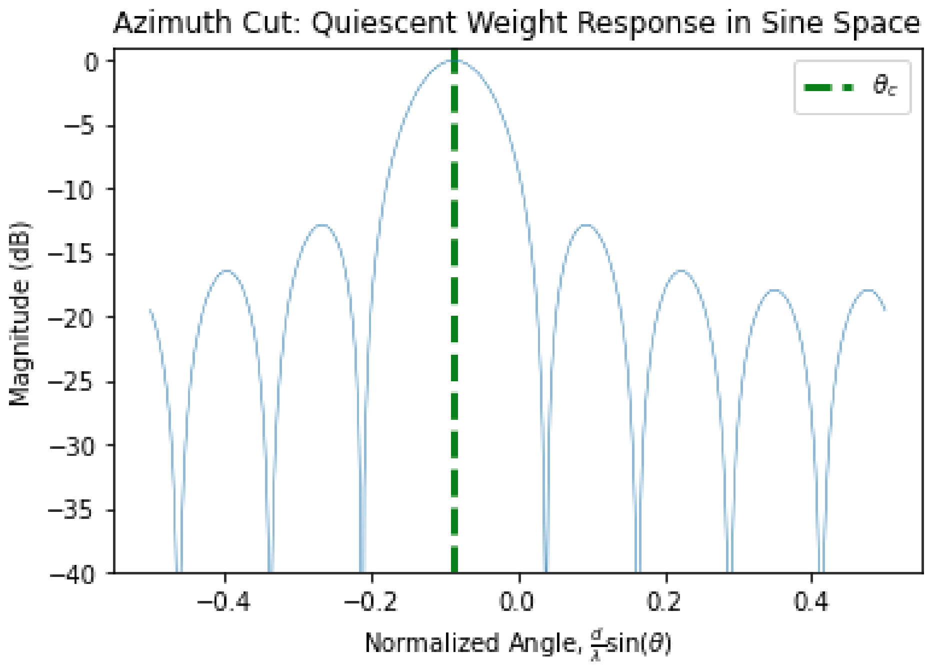


Figure 2.8: Sine Space Plot of Quiescent Weights

2.2.3 Beamforming Array Effects

In the previous section, we introduced the spatial sampling theorem for phased arrays where we implied a $d < \lambda/2$ elemental spacing restriction on the ULA. If however the $d < \lambda/2$ spacing requirement is violated, the array pattern of the beamformed system will experience *grating lobes*, which is the spatial equivalent of signal aliasing in the temporal domain by an undersampled system, where $B > f_s/2$ [6], [9]. These grating lobes appear as duplicate areas of gain in the array's radiation pattern plot. Such a radiation plot for an array showing grating lobes can be seen in Figure 2.9 for a 32-channel ULA steered to beamform at an incidence angle of 30 degrees.

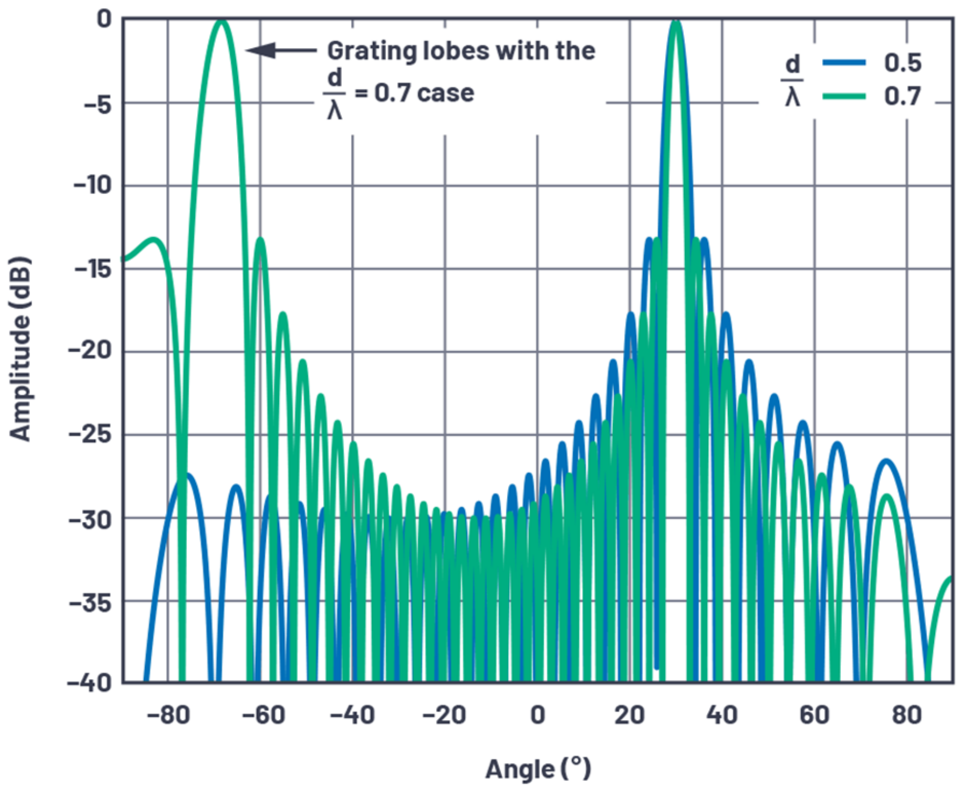


Figure 2.9: Normalized Radiation Pattern of ULA with different element spacing, $N = 32$

Source: Adapted from [9]

In Figure 2.9, it can be seen that when $d/\lambda = 0.5$, or half-wavelength, we see the expected gain peak at 30 degrees and attenuation in all other directions. However, when wavelength spacing is greater than half-wavelength, $d/\lambda = 0.7$, we unexpectedly see another gain peak at roughly -70 degrees. This could cause issues for a beamforming system when it is expecting to only receive signals from a specific steering direction, as signals impinging from -70 degrees would be at the same gain level as the intended direction.

Mentioned previously, we are assuming narrowband systems in this text for simplification of math and experimentation. However the issue that can arise when performing narrowband beamforming- such as multiplication of a complex phasor to apply the phase shift- when wideband signals are present- or even signals of different wavelengths then designed- is that of *beam squint* [2], [8]. This is shown in Figure 2.10 where a ULA designed for a carrier frequency of 3 GHz is receiving a 3.3 GHz signal at different incidence angles.

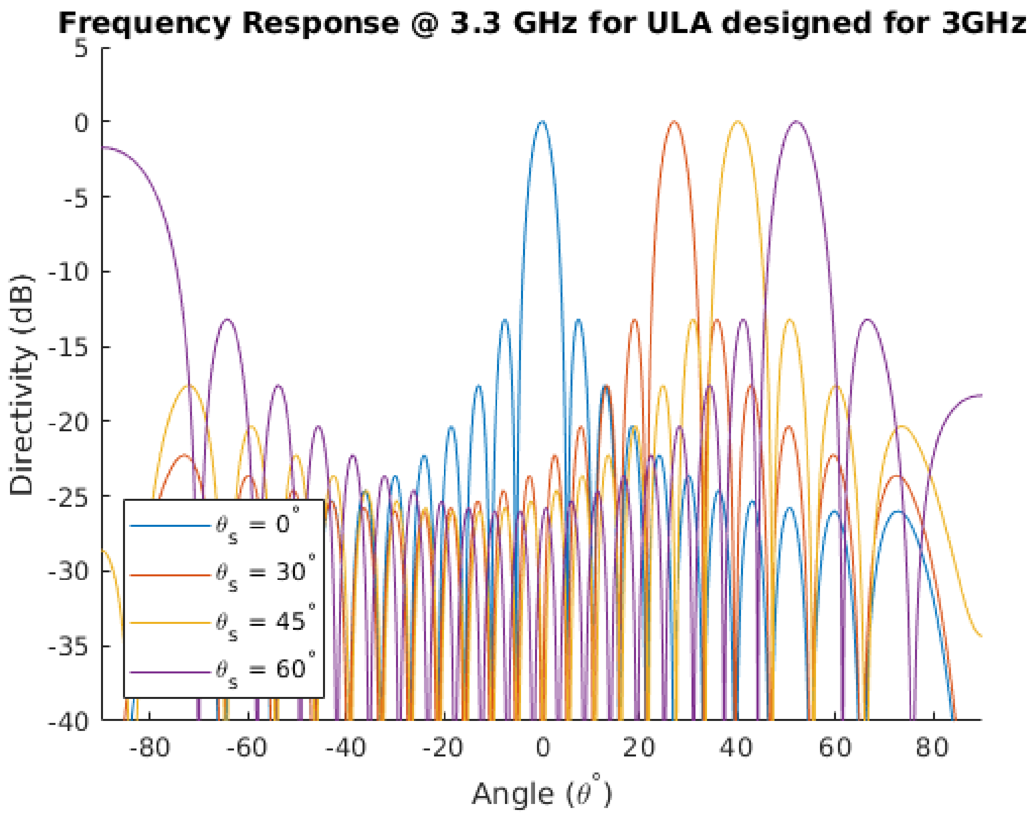


Figure 2.10: Beam Squint

Note that as the incidence angle moves past boresight (where $\theta = 0$), the main lobe broadens, and we even see some aliasing into other signal directions at larger angles.

Another simplification we are making in the discussion of these phased array systems is we assume that individual array elements are isotropic, however as shown in Figure 2.11, real antenna elements have real directivity and angular response of their own.

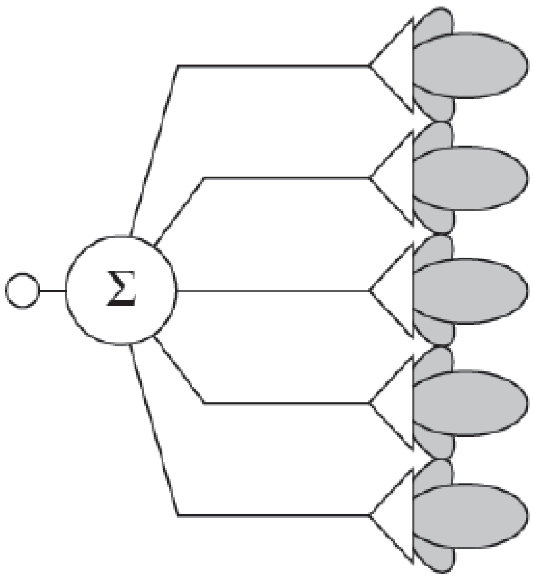


Figure 2.11: Individual Array Element Directivity

Source: Adapted from [8]

Thus, a more accurate description of the actual antenna array directivity, $E(\theta)$, is a linear combination of the Array Factor, $AF(\theta)$, and the individual element directivity $E_e(\theta)$ as shown in Equation 2.13 [8].

$$E(\theta) = E_e(\theta)AF(\theta) \tag{2.13}$$

The resulting combination can be shown in Figure 2.12 where the elemental pattern starts to show attenuation at large angles off boresight, therefore the overall array pattern also shows some attenuation at large angles.

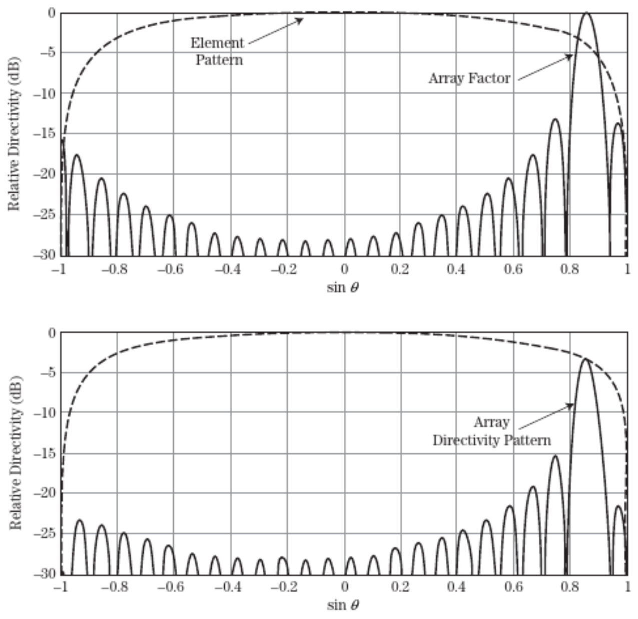


Figure 2.12: Total Antenna Array Directivity

Source: Adapted from [8]

Thus in real systems, we cannot always assume that every steering angle sees the exact same gain response.

Finally on the topic of real directivity, another fundamental array effect that should be considered in phased array systems is that the main lobe width of the Array Factor is inversely related to the number of antenna elements. In the standard, non-windowed (rectangular) spatial response, the null-to-null width of a ULA can be found by 2.14 [4].

$$\theta_{MB} = 2\sin^{-1}\left[\frac{\lambda}{Nd} - \sin(\theta_0)\right] \tag{2.14}$$

This can also be seen in Figure 2.13 where higher channel count systems see much narrower main lobe beamwidths.

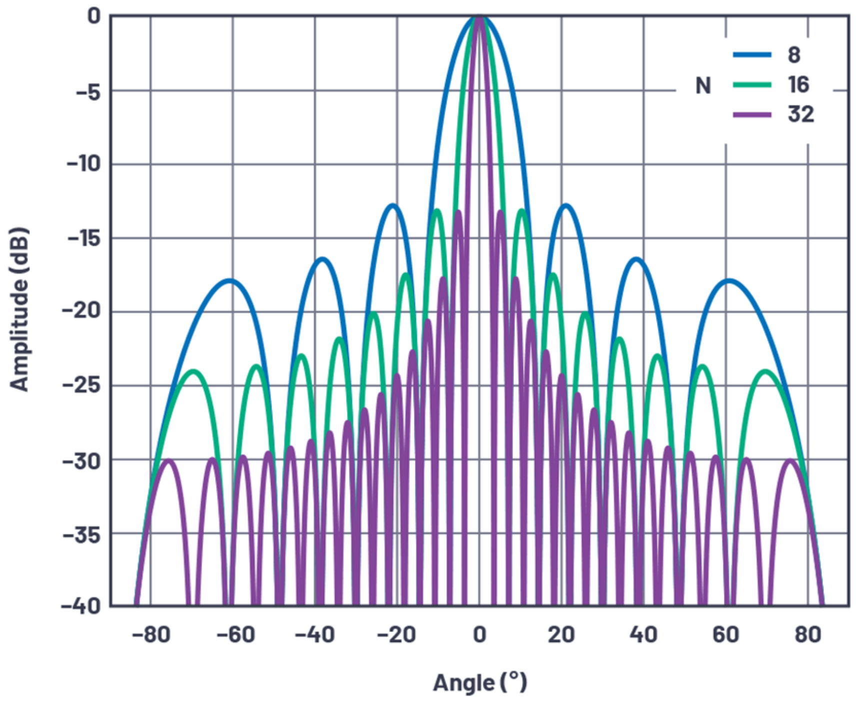


Figure 2.13: Array Radiation Plot over Different Element Counts, $d = \lambda/2$

Source: Adapted from [2]

It would seem obvious that designing a system with higher channel counts yields a better, more directional system, however increasing RF channel count increases system cost and complexity, especially if given a fixed power or space budget.

2.3 Adaptive Beamforming

In some systems just applying quiescent weights to steer the array may not be enough, as signals from non-intended directions can still make their way into the desired signal frequency band, especially if there are multiple users within this same band.

For instance, in the case of Figure 2.14 a desired signal at 300 MHz is impinging the array at $\theta_d = 5°$ and an interference source at 270 MHz is impinging the array at $\theta_{Inf} = 30°$. The sampling frequency is arbitrarily set to $f_s = 1GHz$.

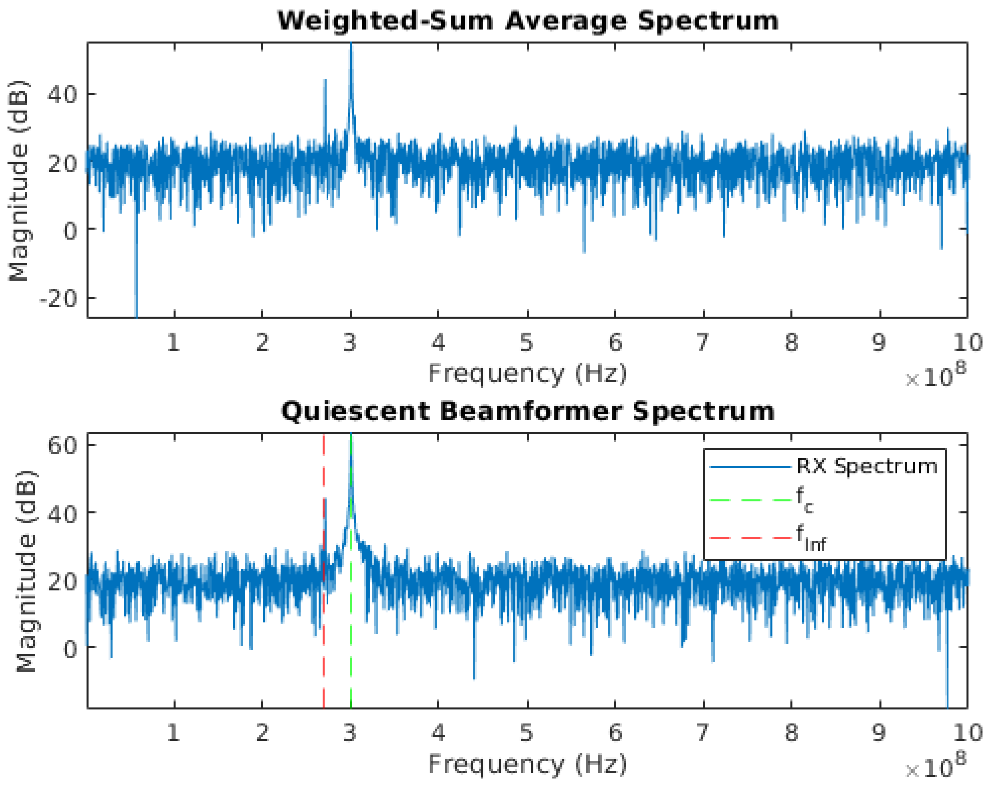


Figure 2.14: Quiescent Interference, $d = \lambda/2$

Note that in both the non-beamforming, weighted-sum averaging case (which is equivalent to beamforming weights equal to unity, or an array factor pointed at boresight) and in the quiescent beamforming case where we are applying digital beamforming with weights designed to steer the array to $\theta_d =$

5°, we still see the interference source in the resultant frequency spectrum. The cause of this is the fact that, even in steered/quiescent-beamformed systems, the side lobes of the Array Factor response do not attenuate interference sources enough, so they still show up in our output spectrum. This can be seen by referring back to a sine space plot of the Array Factor, such as in Figure 2.8, where the nearest side lobes are only $-13dB$ down from the main lobe- typical of a rectangular, non-windowed response in the frequency domain, which shows here as a familiar $sinc()$ function due to $sinc(f) = \frac{\sin(f)}{f} = \mathcal{F}\left[rect(t)\right]$ [2].

One basic approach to null, off-angle, undesired sources could be to apply *windowing* to the individual antenna elements (either digitally or applied via analog methods). As an example, a Hamming window can be used to create scalar values to multiply across our quiescent weights vector. Hamming weights can be derived from Equation 2.15; the Hamming window provides lower, equiripple side lobes, but at the expense of an increased main lobe width, as shown in Figure 2.15.

$$w_{hamming}(n) = 0.54 - 0.46\cos(\frac{2\pi n}{M-1}) \quad 0 \leq n \leq M-1 \tag{2.15}$$

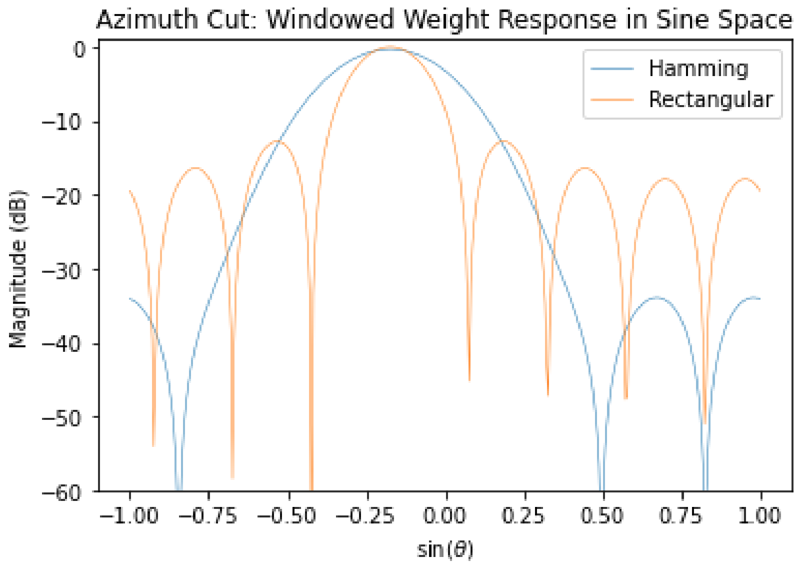


Figure 2.15: Non-windowed vs Hamming Window Sine Space Response, $N = 8$

However, windowing on its own may still not be ideal as the increased main lobe width means interference sources near the spatial direction of the desired source are at nearly the same gain as the desired direction. As well, even though side lobes have larger attenuation (less gain) in windowed responses, a very powerful interference source may still show up in the output spectrum, or band of interest, if that particular emitter does not fall within a natural null (e.g. the interference source falls within a side lobe).

It should also be noted that the difference in frequency between the desired and interference signals in the above examples are mainly for easier demonstration of the negative affects of certain digital beamforming setups where we view one output spectrum post-beamforming; the two signals may actually be at the exact same frequency, for instance two communications users occupying the same channel, in which case both signals may directly interfere with each other, possibly causing degradation or even loss of signal. Again, the only assumed difference between signals in these scenarios is that both interference and desired signals emit from different spatial locations, which provides the impetus for processing such as Adaptive Beamforming to seperate the desired signal from interference and noise.

2.3.1 MVDR Adaptive Beamforming

The basic function of Adaptive Beamforming is to calculate beamforming weights which- when applied in the same beamforming architecture covered previously- provides gain to signals incident from a desired direction, while dynamically *nulling* signals from other spatial locations. This nulling effect is achieved in spatial directions with interference sources by driving a spatial null- an area with large attenuation- in those locations, which is also why Adaptive Beamforming has been referred to as *null steering* [1].

Though there are many different algorithms and implementations for performing adaptive beamforming, the *Minimum-variance distortionless response (MVDR)* algorithm is a classical, *data-dependent* method for adaptive beamforming. MVDR is advantageous due to its fast convergence speed and ability to deal with many, complicated interference sources [1], [6]. The MVDR method may also be referred in literature as *Capon's method* from (Capon, 1969) [6].

The MVDR method works on a batch of sample data across each spatial channel at a given time, here called a *snapshot* of K samples. As such, an N-by-K sample data matrix $\mathbf{X}$ is given, as in Equation 2.16, where $\mathbf{X}$ is built of

N rows (each row corresponding to a distinct array element) of K samples of data.

$$\mathbf{X}_{N,K} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,k} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,k} \end{pmatrix} \tag{2.16}$$

From this input sample matrix $\mathbf{X}$, the estimated sample covariance matrix $\mathbf{M}$ can be formed [1], [6]. Note that in some texts this covariance matrix is denoted as $\mathbf{\Phi}$ or $\mathbf{R}$, however we wish to not conflict with similarly-named variables in this text, such as in QR matrix decomposition in the next section. We will assume for practical purposes that we are dealing with *overdetermined systems* in which the number of samples per channel in a snapshot, K, is greater than the number of channels, N [6]. Thus, in this case $\mathbf{M}$ is an N-by-N matrix formed by the expectation $\mathbb{E}(\cdot)$ of the matrix product of the N-by-K sample matrix $\mathbf{X}$ and its K-by-N Hermitian transpose $\mathbf{X}^H$ such as in 2.17 [6].

$$\mathbf{M} = \mathbb{E}[\mathbf{X}^H\mathbf{X}] \tag{2.17}$$

Given zero-mean input samples- as is assumed for most RF systems where the sampled input is a set of time varying signals with little to no direct current (DC) bias voltage- the covariance matrix $\mathbf{M}$ is equivalent to the autocorrelation matrix. Moreover, the mutually uncorrelated sources in $\mathbf{X}$ mean each sample is equiprobable, thus the expectation $\mathbb{E}(\cdot)$ of the matrix product is essentially the *time-average correlation matrix* as seen in Equation 2.18 [6], [10].

$$\mathbf{M} = \frac{1}{K}\sum_{n=1}^{K} \mathbf{x}^H(n)\mathbf{x}(n) \tag{2.18}$$

It should also be noted that, although not perfectly known by the system during runtime, the covariance matrix is essentially made up of the desired signal of interest (SOI) covariance matrix $\mathbf{M}_d$ and the interference plus noise covariance matrix $\mathbf{M}_{i+n}$ [10].

$$\mathbf{M} = \mathbf{M}_d + \mathbf{M}_{i+n} \tag{2.19}$$

As alluded to in the previous section, the figure of merit for an adaptive beamforming algorithm, is how well it can increase the signal power of an SOI from a desired steering direction while attenuating sources of noise and interference from other directions; mathematically we can represent this value by the *Signal-to-Interference-Plus-Noise ratio (SINR)* of the system. SINR is basically calculated by dividing the signal power of the SOI P by the total noise N and interference powers I [6]:

$$SINR = \frac{P}{I+N} \tag{2.20}$$

SINR is also a valuable metric for both MIMO, and non-MIMO, communication systems. The linear (non-dB) SINR value can be used to estimate the theoretical upper bounds of a communication channel's capacity (in C bits/second), given a B Hz channel bandwidth, as in the *Shannon-Hartley* *theorem* of [6]:

$$C = B \log_2 (1 + SINR) \tag{2.21}$$

Given the N-by-N covariance matrices for SOI and noise-plus-interference defined above, and some N-by-1 beamforming weight vector $\mathbf{w}$, we can actually directly calculate the expected SINR given Equation 2.22 [10].

$$SINR = \frac{\mathbf{w}^H \mathbf{M}_d \mathbf{w}}{\mathbf{w}^H \mathbf{M}_{i+n} \mathbf{w}} \tag{2.22}$$

Also necessary for the data-dependent adaptive beamforming MVDR method, $\mathbf{s}(\theta)$ is to be given as the N-by-1 steering vector, which is equivalent to 2.23 given a desired array steering angle θ [6].

$$s_n(\theta) = [1, e^{-j\theta}, \dots, e^{-j(N-1)\theta}]^T \tag{2.23}$$

Note that the steering vector is equivalent to the complex conjugate of the spatial response vector defined in Equation 2.8 in the previous section. This makes sense given we also covered that the ideal quiescent response in the deterministic beamforming case is essentially the matched filter (complex conjugate) of the spatial response, as previously shown in Equation 2.9.

Thus, the optimum beamformer maximizes the SINR output of the system given the constraint $\mathbf{w}^H \mathbf{s}(\theta) = 1$ through the following minimization equation in 2.24 [6], [10]:

$$\hat{\mathbf{w}} = \min_{w} \mathbf{w}^H \mathbf{M} \mathbf{w} \tag{2.24}$$

From here we can introduce the mathematical definition of the MVDR solution for the adaptive beamforming N-by-1 weight vector $\hat{\mathbf{w}}$ in Equation 2.25 [6].

$$\hat{\mathbf{w}} = \frac{\mathbf{M}^{-1}\mathbf{s}(\theta)}{\mathbf{s}^H(\theta)\mathbf{M}^{-1}\mathbf{s}(\theta)} \tag{2.25}$$

An example of the MVDR beamformer from [6] can be shown in Figure 2.16 where the correlation/covariance matrix is built from the snapshot of K samples of data from each channel, and passed to an MVDR processor which calculates the adaptive beamforming weights which are applied across each channel, then coherently summed to form the output beam $y(n)$. Note, in [6], the channel count is denoted by M instead of N.

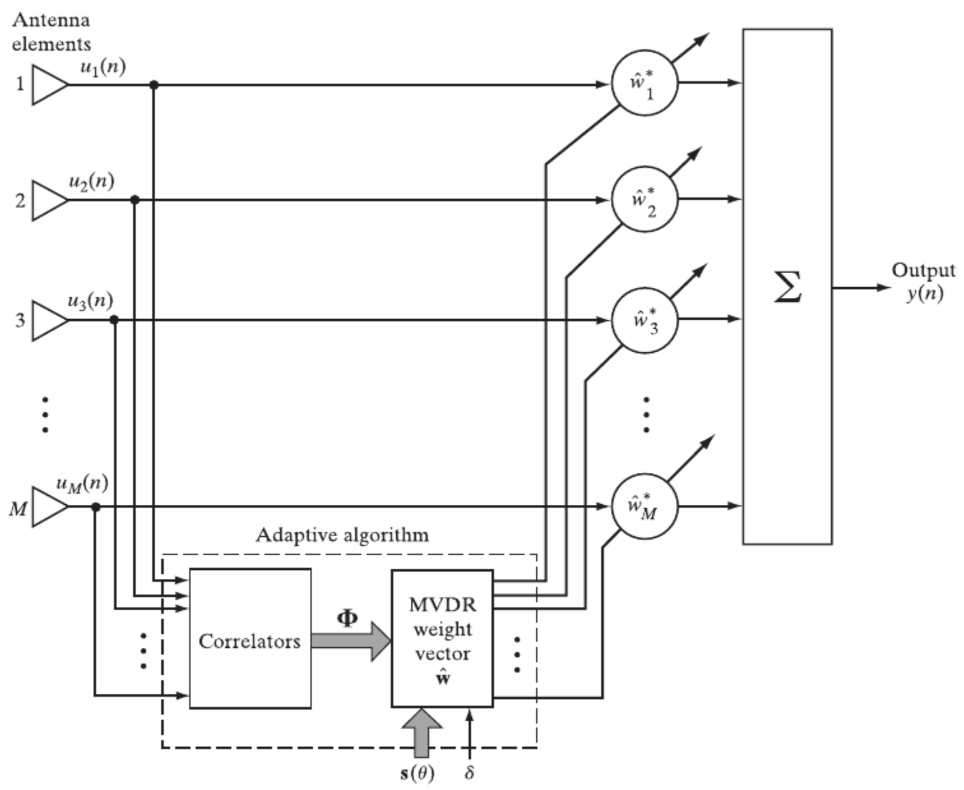


Figure 2.16: Block Diagram of a MVDR Beamformer for a ULA

Source: Adapted from [6]

Using the same example from the deterministic beamforming section, we can apply the MVDR-calculated beamforming weights and compare the spectrum to the quiescent beamforming spectrum from Figure 2.14. To repeat the scenario, a desired signal at 300 MHz is impinging the array at $\theta_d = 5°$ and an interference source at 270 MHz is impinging the array at $\theta_{Inf} = 30°$.

The MVDR narrowband beamformer output is thus given by 2.26.

$$y(k) = \hat{\mathbf{w}}^H \mathbf{x}(k) \tag{2.26}$$

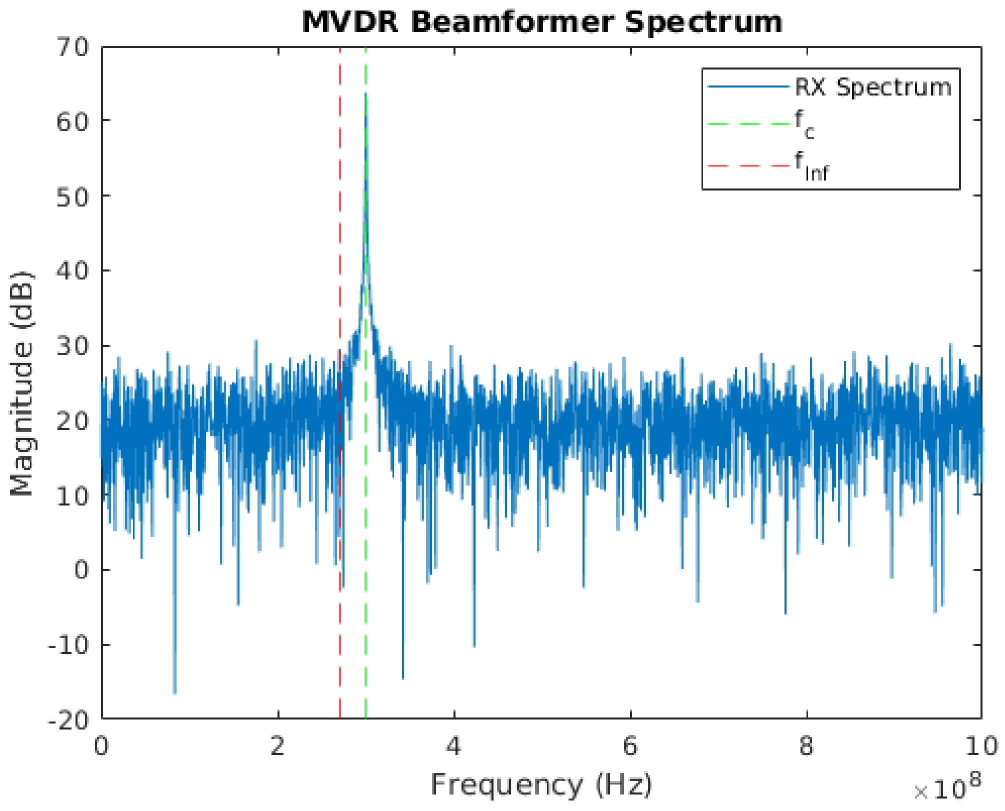


Figure 2.17: Output Spectrum from the MVDR Beamformer for a ULA

As seen in Figure 2.17, the desired SOI has a visibly high SINR, due to the applied beamforming gain, and the interference signal is no longer present in the output spectrum at all.

Further demonstration that the interference source has in-fact been nulled can be seen by examining the MVDR output weights in sine space. This can be seen in Figure 2.18 where the interference angle sees a deep null, and as such, the interference signal is attenuated.

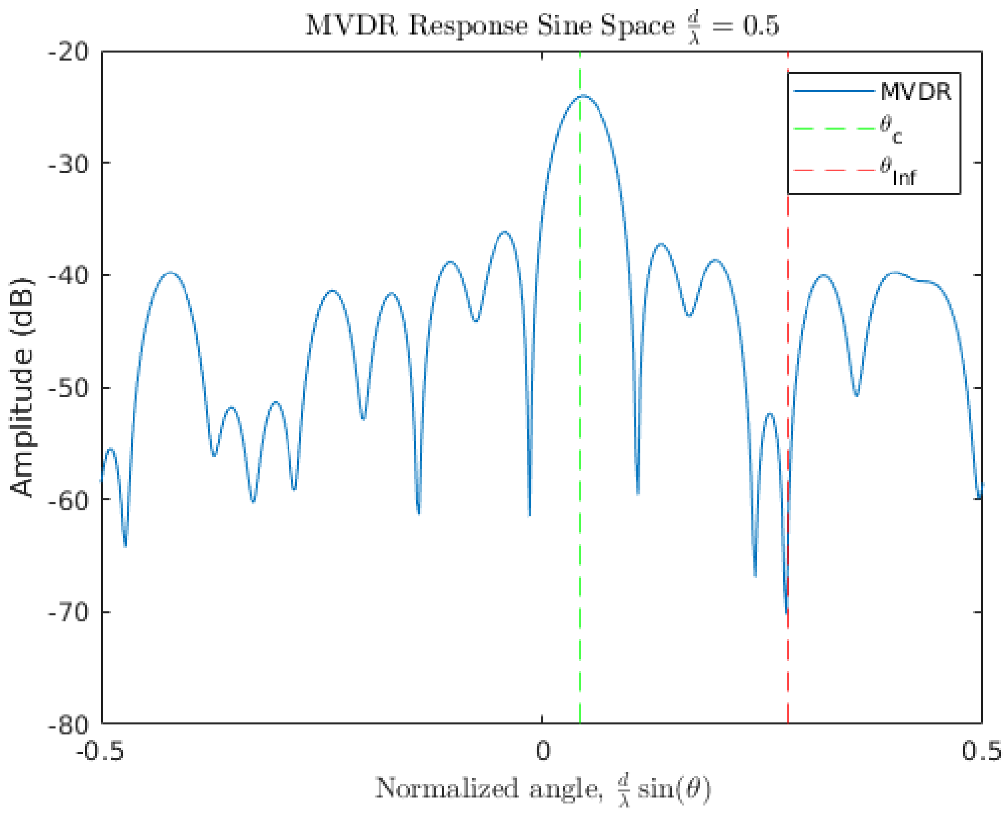


Figure 2.18: Radiation Plot of the MVDR Beamforming Weights

Based on the spatial response of a given number of antenna elements, the *degrees of freedom (DOF)* of an N element array is fundamentally driven by the number of independent nulls that can be produced by the MVDR algorithm, as defined in Equation 2.27.

$$DOF = N - 2 \tag{2.27}$$

In the context of interference mitigation, this means that *up to* $N - 2$ interference sources can be cancelled out using MVDR [6]. To show this in practice, an 8-element ULA can be shown to have 6 interference sources, at varying narrowband frequencies and varying incident angles, which completely muddies the output spectrum in the quiescent beamforming case as seen in Figure 2.19.

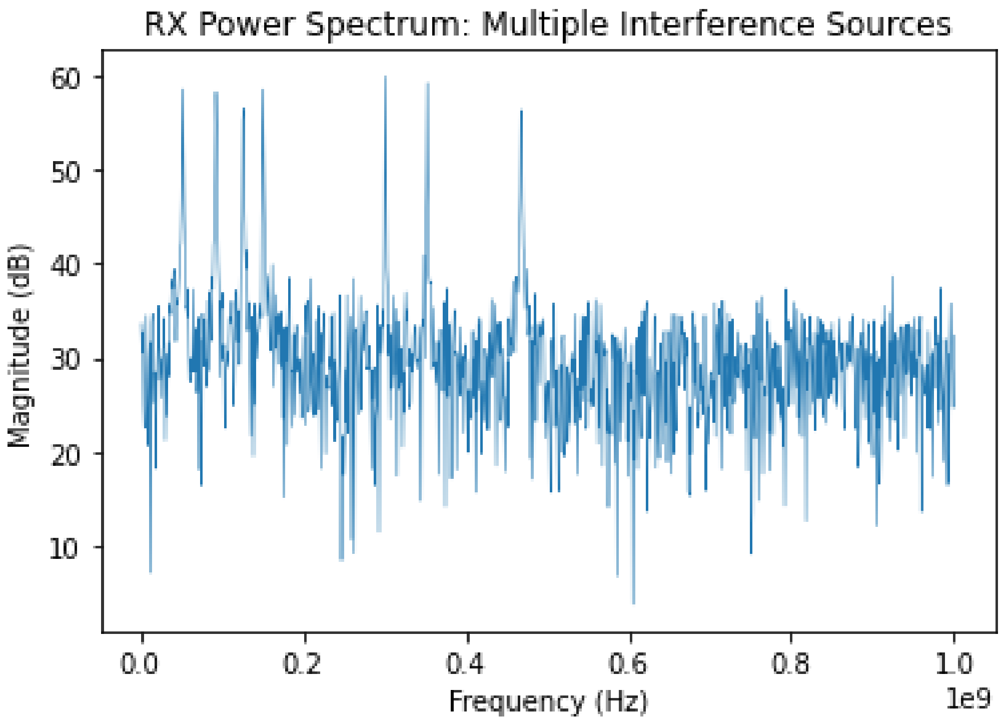


Figure 2.19: Output Spectrum with Multiple Interference Sources, $N = 8$

Post-MVDR adaptive beamforming, the output spectrum again in Figure 2.20 is cleaned up with only the intended signal from the desired direction present.

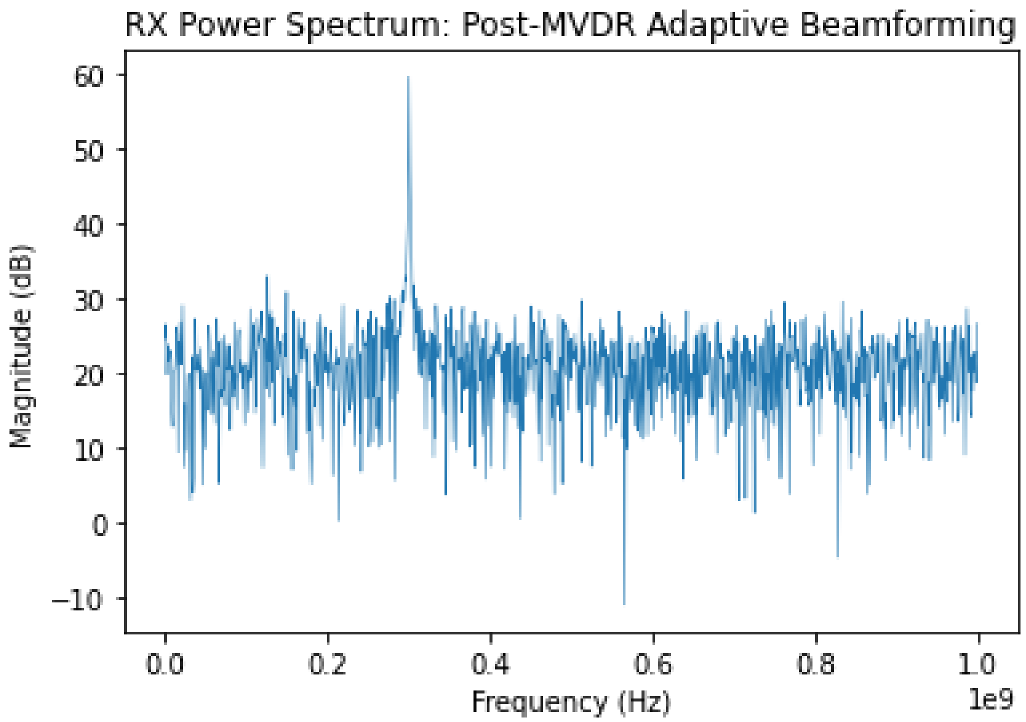


Figure 2.20: MVDR Output Spectrum with Multiple Interference Sources, $N = 8$

Again in sine space, we can see the different spatial interference directions adaptively nulled in Figure 2.21.

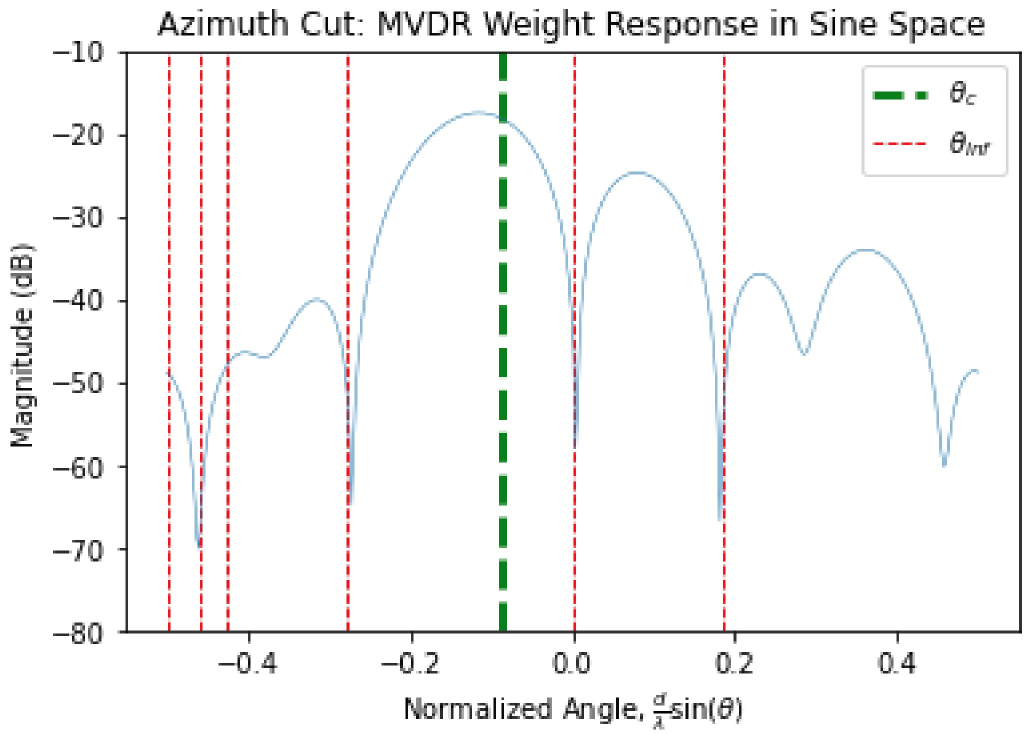


Figure 2.21: MVDR with Multiple Interference Sources in Sine Space, $N = 8$

Note that in some cases, MVDR- or really any adaptive beamforming algorithm- can not perfectly null all interference sources. Usually this is due to where interferers fall spatially relative to each other and the desired look direction, which can be seen from the sine space plot in Figure 2.22 where a ULA has $N = 8$ channels and a desired signal at impinging at the array at $\theta_d = 5°$, but the interference source is spatially very close, impinging the array at $\theta_{Inf} = 8°$.

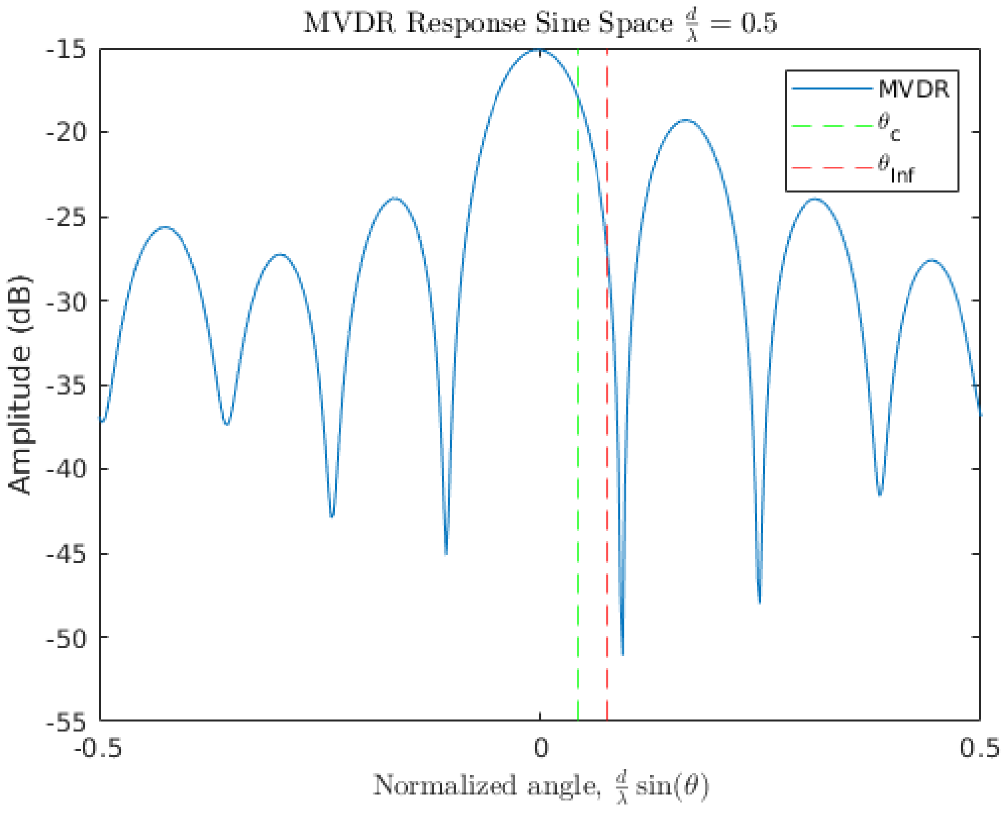


Figure 2.22: MVDR with Interference Source Too Close in Sine Space, $N = 8$

Notice here that the algorithm is trying to force the interference direction into a null while maintaining gain in the desired steering direction, however the main lobe beamwidth of the 8-element ULA is too wide to perform both. The takeaway of this effect is that more antenna elements not only gives a system more numerous nulls to place with adaptive beamforming processes like MVDR, but also tighter lobes- as shown in Figure 2.13 from the previous section- which can more easily null interference directions with close spacing relative to each other and/or the desired look direction.

References

[1] B. D. V. Veen and K. M. Buckley, “Beamforming: A versatile approach to spatial filtering,” *IEEE ASSP Magazine*, 1998.

[2] “Phased array antenna patterns- part 1: Linear array beam characteristics and array factor,” Analog Devices, Tech. Rep., 2020. [Online]. Available: https://www.analog.com/en/analog-dialogue/articles/phased-array-antenna-patterns-part1.html#.

[3] S. Rangan, T. S. Rappaport, and E. Erkip, “Millimeter-wave cellular wireless networks: Potentials and challenges,” *Proceedings of the IEEE*, vol. 102, no. 3, pp. 366–385, 2014. DOI: 10.1109/JPROC.2014.2299397.

[4] J. Guerci, *Space-Time Adaptive Processing for Radar*, 2nd ed. Artech House, 2015.

[5] R. Mailloux, *Phased Array Antenna Handbook*, 2nd ed. Artech House, 2005.

[6] S. Haykin, *Adaptive Filter Theory*, 5th ed. pearson, 2014, ISBN: 978-0-132-67145-3.

[7] E. P. Tsakalaki, L. Ã. M. R. de TemiÃśo, T. Haapala, J. L. RomÃąn, and M. A. Arauzo, “Deterministic beamforming for enhanced vertical sectorization and array pattern compensation,” in *2012 6th European Conference on Antennas and Propagation (EUCAP)*, 2012, pp. 2789–2793. DOI: 10.1109/EuCAP.2012.6206403.

[8] M. A. Richards, J. A. Scheer, and W. A. Holm, *Principles of Modern Radar: Basic Principles*. IET, 2010, ISBN: 9781891121524.

[9] “Phased array antenna patterns- part 2: Grating lobes and beam squint,” Analog Devices, Tech. Rep., 2020. [Online]. Available: https://www.analog.com/en/analog-dialogue/articles/phased-array-antenna-patterns-part2.html.

[10] D. Li, Q. Yin, P. Mu, and W. Guo, “Robust mvdr beamforming using the doa matrix decomposition,” *IEEE-ISAS*, 2011.

Chapter 3

FPGA Implementation of Adaptive Beamforming

In this chapter, we survey the different methods for practical implementation of Adaptive Beamforming in an embedded Field-Programmable Gate Array (FPGA) processor. Thus, it is assumed the reader has some familiarity with FPGAs and digital logic implementations, as in Very Large Scale Integration (VLSI) circuits. The reasoning for choosing an FPGA as a processor for Adaptive Beamforming is not universal, however usually an FPGA is already used as a common interconnect between sensors (such as RF ADCs and DACs used in RF communication systems) and other computer systems (for instance an FPGA commonly acts as "glue logic" by transferring digital samples over some common protocol, such as Ethernet or PCIe, or even locally processing data within the FPGA or larger System on Chip). FPGAs also give flexibility and re-programmability to algorithms without having to be fixed functions as in the case of Application Specific Integrated Circuits (ASICs). Modern FPGAs are also popular in certain embedded, sensor processing devices due to low power consumption for certain algorithms compared to a fixed processor, like a CPU.

Since embedded is a relative term, we are mainly looking at approaches based on performance- such as processing latency to determine the adaptive beamforming weights- as well as on Size, Weight and Power (also known as SWaP). For "edge" devices, especially those designed to process RF communications in deployed locations and operational environments, we cannot ignore processing power and resources. Most embedded devices have strict constraints as well, such as a fixed power budget or size constraint to perform all processing within one physical processor board.

3.1 Comparison of Standard Architectures

Since there are many different architectures for performing Adaptive Beamforming in FPGA logic, we will first compare and contrast the standard methods and choose one as our optimal method which balances processing latency as well as logic resource utilization.

3.1.1 Systolic Arrays for QR Decomposition in Digital Logic

In the previous chapter, we showed that the MVDR method applied to adaptive digital beamforming yielded great results for nulling interference sources from a desired SOI, given a sample data covariance matrix $\mathbf{M}$ and a desired array steering vector $\mathbf{s}(\theta)$. However, the MVDR equation for finding optimal adaptive beamforming weights assumes some relatively complex math when calculating in an embedded system, namely the inversion of the covariance matrix, $\mathbf{M}^{-1}$.

For instance, directly performing the matrix inversion- also aptly called *Sample Matrix Inversion (SMI)* [1]- using Gaussian elimination has a high computational complexity of $\mathcal{O}(n^3)$ [2]. As well, fixed-point (integer) direct computations of SMI often have poor numerical robustness and stability [3], [4].

A common method for avoiding the pitfalls of direct matrix inversion is that of *QR Decomposition (QRD)*, so called because the operation decomposes some full rank $n \times p$ matrix $\mathbf{A}$ into an $n \times p$ orthogonal matrix $\mathbf{Q}$ and an upper-triangular $p \times p$ matrix $\mathbf{R}$ (where the lower triangle is all zeros) [1]–[3]:

$$\mathbf{A} = \mathbf{Q}\begin{bmatrix} \mathbf{R} \\ \hdashline 0 \end{bmatrix} \tag{3.1}$$

Using a rotation algorithm such as *Gram-Schmidt orthogonalization*, *Householder transformations* or *Givens rotations*, the pseudo-inverse of matrix $\mathbf{A}$ can be found by Equation 3.2 [2]–[5]:

$$\mathbf{A}^{-1} = (\mathbf{A}^H\mathbf{A})^{-1}\mathbf{A}^H = (\mathbf{R}^H\mathbf{R})^{-1}\mathbf{R}^H\mathbf{Q}^H \rightarrow \mathbf{A}^{-1} = \mathbf{R}^{-1}\mathbf{Q}^H \tag{3.2}$$

As well, since $\mathbf{Q}$ is a unitary matrix, we can fundamentally achieve the identify matrix $\mathbf{I}$ (a matrix with ones on the main diagonal and zeros elsewhere) from 3.3:

$$\mathbf{Q}^H\mathbf{Q} = \mathbf{I} \tag{3.3}$$

These properties of QRD allow us to perform the *Recursive Least Squares (RLS)* algorithm (combined known as QRD-RLS) to find the inverse of the matrix **A** in the context of solving the system of linear equations for **x** given the general form in Equation 3.4 by minimizing the least square error $|\mathbf{b} - \mathbf{A}\mathbf{x}|$ [1]–[3], [6]:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{3.4}$$

In the context of Adaptive Beamforming, we can setup a similar system of linear equations to optimally solve for adaptive weights using QRD-RLS, instead of direct matrix inversion as with MVDR, given the same optimization goals of maximizing SINR. For instance, we can surmise that, given a constant, ideal spatial signal column-vector, $\mathbf{s}(\theta)$- which is the same steering vector from the MVDR process-, a system with only one SOI present, which experiences zero noise or interference, would expect to observe each spatial channel (row) of the sample matrix **X** ($\mathbf{x_n}$) be directly related with the associated spatial element s_n when the phase relationship of θ is exact (albeit by a scalar relationship based on relative powers of each) [3], [6]. Said another way, **s** is optimum when the sample covariance matrix **M** was proportional to the Identity Matrix, such that it appeared that there was equal and independent noise from each array element [6]. Thus the system of linear equations in 3.5 can be solved for the ideal adaptive beamforming weight vector $\hat{\mathbf{w}}$ given only the covariance matrix **M** and the ideal steering vector **s** [3], [4], [6]:

$$\mathbf{M}\hat{\mathbf{w}} = \mathbf{s} \tag{3.5}$$

To solve for the adaptive beamforming weights, we can start by substituting the QR decomposition relationship for the covariance matrix from Equation 3.1 and 3.2, we can refactor 3.5 into Equation 3.6:

$$\mathbf{M}\hat{\mathbf{w}} = \mathbf{s} \xrightarrow{\text{QRD}} \mathbf{QR}\hat{\mathbf{w}} = \mathbf{s} \tag{3.6}$$

Given the identity matrix relationship in Equation 3.3, we can rearrange terms as in Equation 3.7 [4]:

$$\mathbf{Q}^H\mathbf{QR}\hat{\mathbf{w}} = \mathbf{Q}^H\mathbf{s} \rightarrow \mathbf{IR}\hat{\mathbf{w}} = \mathbf{Q}^H\mathbf{s} \rightarrow \mathbf{R}\hat{\mathbf{w}} = \mathbf{Q}^H\mathbf{s} \tag{3.7}$$

From here, we can find $\hat{\mathbf{w}}$ using back-substitution such as in Equation 3.8, where $c = \mathbf{Q}^H\mathbf{s}$ for ease of notation [4], [5], [7]:

$$\hat{w}_j = \frac{1}{r_{j,j}}\left[c_j - \sum_{k=j+1}^{N} r_{j,k}\hat{w}_k\right] \tag{3.8}$$

QRD-RLS not only solves the numerical stability issues SMI experiences [1], [2], [8], but moreover QRD allows for very efficient computation in digital (e.g. FPGA, ASIC, VLSI, etc.) logic. This is due to the common form of QRD-RLS in digital systems to be a *systolic array* which exploits the inherent parallelism of digital architectures when using rotation algorithms such as Givens Rotations, which allow distributed rotation cells as processing elements [1]–[3]. The common systolic array structure and *signal flow graph (SFG)* for QRD-RLS can be seen in Figure 3.1 which is common to QRD methods from [1]–[3], [5], [8], [9].

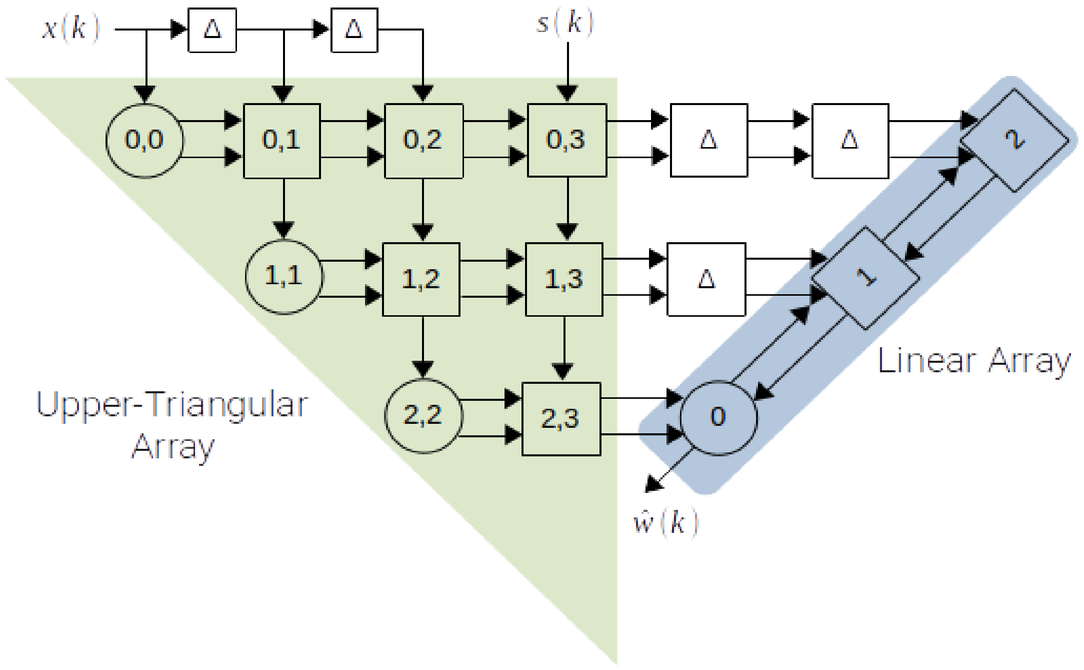


Figure 3.1: QRD Systolic Array with Linear Back-Substitution Section, $N = 3$

The triangular systolic array process consists of two main cell types: a *Boundary Cell (BC)* and an *Internal Cell (IC)*. These cells of the systolic array perform the Givens Rotations on each element of the input matrix to zero out unwanted elements to form the upper-triangular matrix [2], [3]. The elements stored within the upper-triangular systolic array directly correspond with the elements of the **R** matrix from QRD, indexed as $r_{i,j}$ as seen by the indices in each cell of Figure 3.1 [10].

Values move top-down and left-to-right in the SFG. Input samples $x(k)$ from the covariance matrix **M** can either be staggered via a tapped delay line- as shown- or using handshaking signals to each channel to ensure proper timing of data flow through the processing systolic array. The steering vector **s** is directly fed into the column to the right of the last input sample column.

The linear array performs the necessary back-substitution operation to form the final adaptive beamforming output weights $\hat{w}(k)$.

Due to the commonality in some of the rotation processing elements, some particularly resource-constrained implementations forego the parallel systolic array for a folded implementation which utilizes a single, common processing element to perform all operations. Some extra state control logic (or even a SW programmable co-processor) then iterates over the matrix space to give partial products to the Processing Element [7]. This approach, however, causes much greater processing latency, and so for this work was not pursued (essentially there is an area versus throughput trade with QRD implementations in digital logic).

The Boundary Cell (circular node in Figure 3.1) accomplishes the *vectoring* operation on complex input samples denoted x_{in}, which essentially transforms the sample from complex (e.g. I/Q) to a magnitude and phase [2], [5], [9]. The output of the BC are the rotation angles from the vectoring operation and are directly fed to an adjacent Internal Cell within the same systolic array row. The mathematical functions of the Boundary Cell can be seen below:

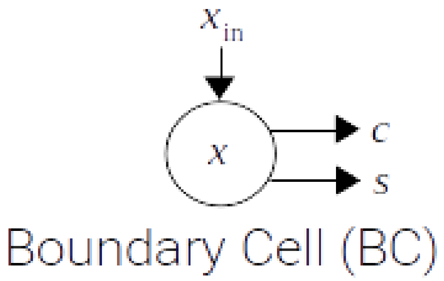


Figure 3.2: Boundary Cell SFG

Algorithm 1 Boundary Cell Operations

```
if $x_{in} = 0$ then
    $c \leftarrow 1$
    $s \leftarrow 0$
    $x \leftarrow \lambda^{1/2} x$
else
    $x' \leftarrow \sqrt{\lambda x^2 + |x_{in}|^2}$
    $c \leftarrow \frac{\lambda^{1/2} x}{x'}$
    $s \leftarrow \frac{x_{in}}{x'}$
    $x \leftarrow x'$
end if
```

In the processing cell equations, x is used to denote a processing cell's internal memory (e.g. register in logic) which maintains value from a previous cycle. λ is the *forgetting factor* which aids in numerical stability of RLS viewing statistical variations over time, as samples in the distant past are "forgotten", and for adaptive purposes here, is usually set to a value close to 1 (e.g. 0.99) [3]. Having no forgetting factor (e.g. $\lambda = 1$) for some systems is admissible depending on the situation [3], [9]. Specifically to the BC, x' is an intermediate value for ease of notation, and c, s are the cosine and sine values respectively corresponding to the Givens Rotations.

It can already be seen that the equations for the BC require some complex operations, namely a square root and division. There are other distinct implementations that can eliminate the square-root operations within the Boundary processing cells like in the Squared Givens Rotation (SGR) algorithm [2], [8], however the logic still needs to support arbitrary integer division operations in SGR [2] which- while possible with methods such as pre-quantized Look-up-Tables (LUT), as done in [11], or multi-cycle, iterative cores- was not pursued for this research due to added complexity and fabric resources. There are other algorithms that are also "division free", however they mainly put off the division operations until the very end and incur extra processing penalties within the systolic array's processing elements, so these algorithms were also not pursued [2]. The ideal method of vectoring- from a complexity, relative performance and resource utilization perspective- uses *CORDIC* engines in programmable logic. This will be covered in detail in the next section.

The Internal Cell (square node in Figure 3.1) performs the *rotation* operation on complex input values using the rotation angles received from that row's Boundary Cell as shown in Figure 3.3 [2], [5], [9].

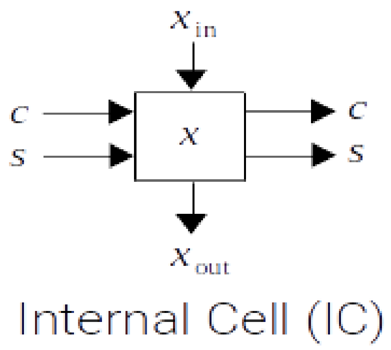


Figure 3.3: Internal Cell SFG

Algorithm 2 Internal Cell Operations

```
$x_{out} \leftarrow cx_{in} - s\lambda^{1/2}x$
$x \leftarrow sx_{in} + c\lambda^{1/2}x$
```

The nodes in the linear array section of 3.1 receive the upper-triangular matrix from the systolic array and performs the back-substitution to finally derive the adaptive beamforming weights [2], [9]. These cells, and their mathematical functions, are shown below:

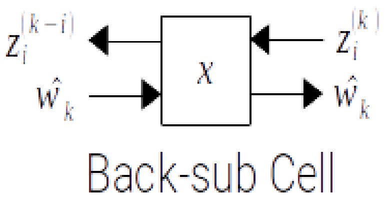


Figure 3.4: Back-Substitution Cell SFG

$$\hat{w}_i = \frac{p_i - z_i^{(i)}}{x_{ii}} \tag{3.9}$$

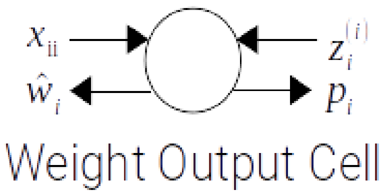


Figure 3.5: Back-Substitution Output Cell SFG

$$z_i^{(k-1)} = z_i^{(k)} + x_{ik}^* \hat{w}_k \tag{3.10}$$

Since back-substitution is an iterative operation, several implementations perform the back-substitution with an embedded processor, such as in [7]. Another approach is to perform *weight flushing*, where the output weights are extracted from the final lower triangular cell by appending sets of zeros after the input matrix has been fully consumed, however this approach requires extra logic and incurs extra latency [1], [8]. As well, since back-substitution is mainly the bottleneck for these QRD-RLS architectures, some approaches look to forgo QRD and instead calculate the adaptive weights using *Cholesky factorization* on the covariance matrix **M**; however while valid, Cholesky is not numerically robust in fixed-point logic [6] (e.g. best to perform in floating point) and thus usually involves shipping off the covariance matrix calculated in FPGA logic to a CPU with a floating-point unit (FPU), which induces further latency. Thus, this approach was not pursued in this research as well.

3.1.2 IQRD Systolic Array Implementation

To overcome the throughput limitations of back-substitution in the de facto QRD-RLS systolic array implementation, another extended QRD-RLS architecture has been implemented which adds a second *lower-triangular downdating array* interfaced to the upper-triangular QRD array to directly extract the final adaptive beamforming weights through a simple multiplication and addition operation [10]. This new systolic array architecture is also known as the *Inverse QR Decomposition (IQRD)* Systolic Array, and provides much lower latency for weight extraction compared to linear back-substitution [10]. The reader is advised to review the works in [12]–[15] for more details on the numerical analysis and proof of Inverse QR Decomposition for Recursive Least Squares Filtering. The IQRD SFG can be seen in Figure 3.6 and is the chosen digital architecture that we will be developing with Very-High Speed Integrated Circuit Hardware Description Language (VHDL) components.

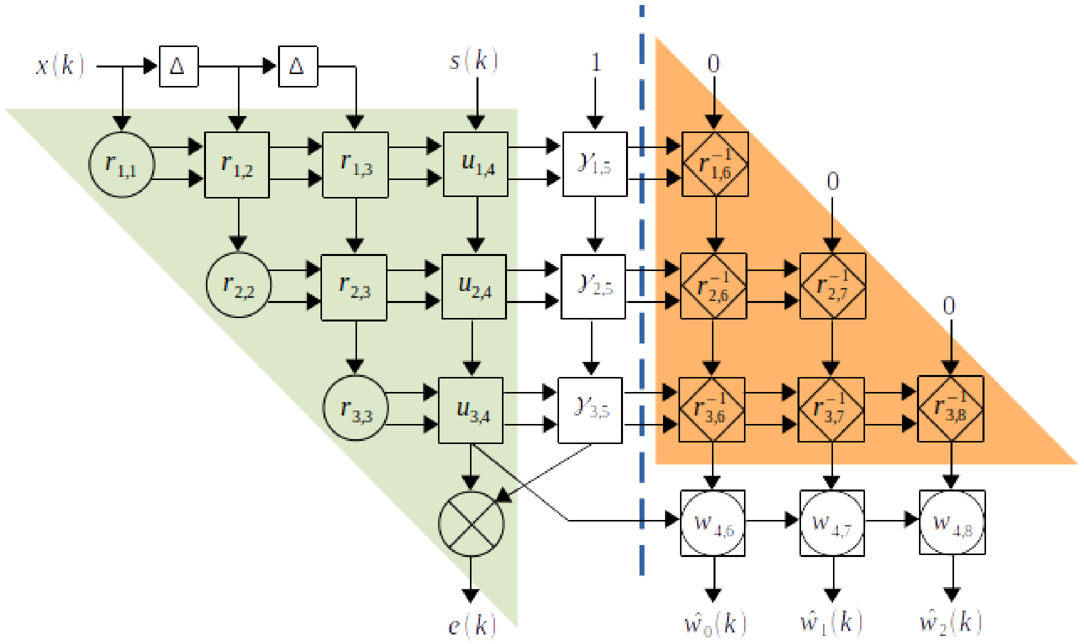


Figure 3.6: Inverse QRD Systolic Array SFG

The lower-triangle array rotates the matrix $\mathbf{R}^{-1}$ stored in the downdating cells using null input vectors [10]. Two new processing cells are added to the systolic array as well: a *downdating cell* and a *weigh extraction cell*.

The downdating cell is very similar to the QRD Internal Cell from previous implementations, with the only difference being the use of a $1/\lambda$ forgetting factor in the internal operation- thus the downdating cell has also been called the *Inverse Internal Cell*.

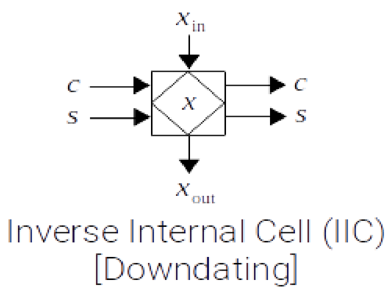


Figure 3.7: Inverse Internal Cell (Downdating) SFG

Algorithm 3 Inverse Internal Cell Operations

```
$x_{out} \leftarrow cx_{in} - s\lambda^{-1/2}x$
$x \leftarrow sx_{in} + c\lambda^{-1/2}x$
```

The weight extraction cell uses the final sample output from the upper-triangular array, and the final sample outputs from its respective column of the lower-triangular inverse array, to form each of the adaptive output weights $\hat{w}_i(k)$ using simple arithmetic operations as shown in Equation 3.11 [10].

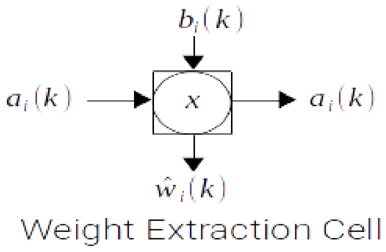


Figure 3.8: Weight Extract Cell SFG

$$\hat{w}_i(k) = \hat{w}_i(k-1) - a_i(k)b_i(k) \tag{3.11}$$

3.2 IQRD HDL Design Details

Given the ideal implementation for Adaptive Beamforming in FPGA programmable logic as the IQRD Systolic Array, we here show the detailed implementation and the HDL architecture of the individual processing cells, as well as the top-level architecture of the array.

Some implementations use High-Level Synthesis (HLS) for complicated DSP designs such as Adaptive Beamforming, such as in [4]. While HLS is a great tool for rapid prototyping, the particular language and/or toolchain is often vendor-specific (e.g. cannot easily port to another vendor's device) and sometimes does not infer the optimal solution that HDL done by-hand can acheive. As such, the IQRD architecture here is being developed in vendor-neutral VHDL without usage of any vendor-specific IP cores.

3.2.1 Covariance Matrix Calculation

The first component developed calculates the covariance matrix $\mathbf{M}$ over a snapshot of K training samples from a set of N parallel input channels, $\mathbf{X}$, as in 3.12 [6]

$$\mathbf{M} = \frac{1}{K}\mathbf{X}^H\mathbf{X} \tag{3.12}$$

On first glance of the Equation in 3.12, it would seem logical that we must buffer K samples of data across all channels to form the sample data matrix $\mathbf{X}$. While correct, needing to store the $N \times K$ matrix could result in a fairly large memory requirement if using a high value of K and/or a high-channel count system. Instead, the covariance matrix can actually be calculated on-the-fly requiring very little resources. This is due to the fact that the covariance matrix multiplication has all information required to form the partial matrix product at each sample time k, since both the $N \times 1$ column-vector $\mathbf{x}(k)$ and its complex conjugate transpose $1 \times N$ row-vector $\mathbf{x}^H(k)$ are known based on simple data reordering and conjugate operations.

As well, we can reduce the number of simultaneous multiply-accumulate (MAC) operations by half since the covariance matrix $\mathbf{M}$ is always Hermitian positive semi-definite, meaning the lower-triangle always equals the complex conjugate of the upper-triangle; because of this, only the lower-triangle values of the covariance matrix need to be calculated, at which point the output can simply copy and complex conjugate the lower-triangle into the upper-triangle elements, which is a simple data reordering and sign change on the imaginary parts.

One last optimization can be found with the $\frac{1}{K}$ division operation at the end of the matrix multiply; if given the constraint that the sample snapshot size K is some power-of-2 value (e.g. $1024 = 2^{10}$), we can avoid direct integer division by K and instead bit-shift the output values right by $\log_2(K)$, which is a simple, singe-cycle operation in digital logic.

The optimized covariance matrix calculation algorithm can be seen below:

Algorithm 4 Optimized Covariance Matrix Calculation

```
for $i \leftarrow 1$ to $N$ do
  for $j \leftarrow 1$ to $i + 1$ do
    for $k \leftarrow 1$ to $M$ do
      $m_{i,j} \leftarrow x_{i,k}x^*_{j,k} + m_{i,j}$
      if $i \neq j$ then
        $m_{j,i} \leftarrow m^*_{i,j}$
      end if
    end for
  end for
end for
$\mathbf{M} \leftarrow \mathbf{M} \gg \log_2(K)$
```

The resulting VHDL code for the entity that calculates the covariance matrix is shown below:

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
library work;
```

```vhdl
  use work.util_pkg.all;

-- #TODO: use find first set bit in MSB (largest across matrix) to dynamically scale elements to output bitwidth?

entity sample_covar_matrix is
  generic (
    G_DATA_WIDTH : natural := 16; -- real & imag part sample bitwidth
    G_ACC_WIDTH  : natural := 48; -- covariance matrix internal accumulator data width
    G_N          : natural :=  4  -- number of channels (rows)
  );
  port (
    clk          : in  std_logic;
    reset        : in  std_logic;
    num_est_samp : in  unsigned; -- number of estimation samples (columns), M
    din_valid    : in  std_logic; -- din_real & din_imag valid (assumed all rows are aligned)
    din_real     : in  T_signed_2D(G_N - 1 downto 0)(G_DATA_WIDTH - 1 downto 0);
    din_imag     : in  T_signed_2D(G_N - 1 downto 0)(G_DATA_WIDTH - 1 downto 0);
    dout_valid   : out std_logic;
    dout_real    : out T_signed_3D(G_N - 1 downto 0)(G_N - 1 downto 0)(G_DATA_WIDTH - 1 downto 0);
    dout_imag    : out T_signed_3D(G_N - 1 downto 0)(G_N - 1 downto 0)(G_DATA_WIDTH - 1 downto 0)
  );
end sample_covar_matrix;

architecture rtl of sample_covar_matrix is

  component complex_multiply_mult4 is
    generic (
      G_AWIDTH : natural := 16;     -- size of 1st input of multiplier
      G_BWIDTH : natural := 18;     -- size of 2nd input of multiplier
      G_CONJ_A : boolean := false; -- take complex conjugate of arg A
      G_CONJ_B : boolean := false  -- take complex conjugate of arg B
    );
    port (
      clk       : in  std_logic;
      ab_valid  : in  std_logic; -- A & B complex input data valid
      ar        : in  signed(G_AWIDTH - 1 downto 0); -- 1st input's real part
      ai        : in  signed(G_AWIDTH - 1 downto 0); -- 1st input's imaginary part
      br        : in  signed(G_BWIDTH - 1 downto 0); -- 2nd input's real part
      bi        : in  signed(G_BWIDTH - 1 downto 0); -- 2nd input's imaginary part
      p_valid   : out std_logic; -- Product complex output data valid
      pr        : out signed(G_AWIDTH + G_BWIDTH downto 0); -- real part of output
      pi        : out signed(G_AWIDTH + G_BWIDTH downto 0)  -- imaginary part of output
    );
  end component;

-- #TODO: double-buffered covar matrix reg's so one can be read out while another is calculated with inputs?

  signal sig_covar_re, sig_covar_im : T_signed_3D(G_N - 1 downto 0)
                                                 (G_N - 1 downto 0)
                                                 (G_ACC_WIDTH - 1 downto 0);
  signal sig_pr, sig_pi             : T_signed_3D(G_N - 1 downto 0)
                                                 (G_N - 1 downto 0)
                                                 (2*G_DATA_WIDTH downto 0);

  constant K_PIPE_DELAY : integer := 3; -- # clk cycles of pipeline delay through component
  signal sig_valid_sr    : std_logic_vector(K_PIPE_DELAY - 1 downto 0) := (others => '0');
  signal sig_end_of_est : std_logic; -- # of estimation samples complete
  signal sig_samp_cnt    : unsigned(num_est_samp'range);

begin

  dout_valid <= sig_end_of_est;
```

```
dout_real  <= sig_covar_re;
dout_imag  <= sig_covar_im;

S_dvalid_count: process(clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      sig_valid_sr    <= (others => '0');
      sig_samp_cnt    <= (others => '0');
      sig_end_of_est <= '0';
    else
      -- shift register to delay data valid to match pipeline delay of cmult
      sig_valid_sr <= sig_valid_sr(K_PIPE_DELAY - 2 downto 0) & din_valid;
      if sig_valid_sr(sig_valid_sr'high) = '1' then
        if sig_samp_cnt >= num_est_samp then
          sig_samp_cnt    <= (others => '0');
          sig_end_of_est <= '1';
        else
          sig_samp_cnt    <= sig_samp_cnt + 1;
        end if;
      end if;

      if sig_end_of_est = '1' then
        sig_end_of_est <= '0';
      end if;
    end if;
  end if;
end process S_dvalid_count;

-- create triangular, fused, complex multiply of input and its complex transpose
UG_gen_rows: for i in 0 to G_N - 1 generate
  UG_gen_columns: for j in 0 to i generate
    -- Perform z[i,k]*conj(z[j,k])
    U_cmplx_mult: complex_multiply_mult4
      generic map (
        G_AWIDTH => G_DATA_WIDTH, -- size of 1st input of multiplier
        G_BWIDTH => G_DATA_WIDTH, -- size of 2nd input of multiplier
        G_CONJ_A => false,
        G_CONJ_B => true          -- take complex conjugate of B arg
      )
```

```
        port map (
          clk      => clk,
          ab_valid => '0',           -- not used, see S_dvalid_count
          ar       => din_real(i), -- 1st input's real part
          ai       => din_imag(i), -- 1st input's imaginary part
          br       => din_real(j), -- 2nd input's real part
          bi       => din_imag(j), -- 2nd input's imaginary part
          p_valid  => open,          -- not used, see S_dvalid_count
          pr       => sig_pr(i)(j),
          pi       => sig_pi(i)(j)
        );

      -- Since output is always Hermitian positive semi-definite, the calculated
      -- lower triangle can be copied to the upper triangle by its diagonal
      -- complex conjugate
      UG_upper_hermitian: if i /= j generate
        sig_covar_re(j)(i) <=  sig_covar_re(i)(j);
        sig_covar_im(j)(i) <= -sig_covar_im(i)(j);
      end generate UG_upper_hermitian;

      S_accumulate: process(clk)
      begin
        if rising_edge(clk) then
          -- reset accumulator at end of estimation cycle (number of samples hit)
          if (reset = '1') or (sig_end_of_est = '1') then
            sig_covar_re(i)(j) <= (others => '0');
            sig_covar_im(i)(j) <= (others => '0');
          else
            if sig_valid_sr(sig_valid_sr'high) = '1' then
              sig_covar_re(i)(j) <= resize( sig_pr(i)(j), G_ACC_WIDTH ) + sig_covar_re(i)(j);
              sig_covar_im(i)(j) <= resize( sig_pi(i)(j), G_ACC_WIDTH ) + sig_covar_im(i)(j);
            end if;
          end if;
        end if;
      end process S_accumulate;
    end generate UG_gen_columns;
  end generate UG_gen_rows;

end rtl;
```

Listing 3.1: Sample Covariance Matrix Calculation Component

When dealing with the direct implementation of the covariance matrix calculation, it should also be noted that the difficulty of *power domain* algorithms such as QRD-RLS is the word length of the multiply-accumulate in the covariance matrix estimation block is somewhat derived by the largest and smallest eigenvalues of the covariance matrix; on the high end, this is related to the max signal power seen from the input sample matrix and on the low end, its related to the thermal noise floor of the receiver [6]. The *MUSE* approach taken by [6] is a *voltage domain* approach which needs not directly calculate the covariance matrix and instead operates directly on the input samples, however this approach takes a fair amount of iterations and so was not considered as part of the architectural trades. As well, we'd like to reuse this covariance matrix estimation block as the input for our Neural Network, though this will be covered in detail in the next chapter.

3.2.2 CORDIC Internal and Boundary Cells

As shown in the previous sections, the Internal and Boundary Cells (including Inverse Internal Cells) perform the Givens Rotations on the $\mathbf{R}$ and $\mathbf{R}^{-1}$ matrices and functions including square-root and division operations. Mentioned previously, digital logic is not well suited for these operations so instead, we will utilize the *Coordinate Rotation Digital Computer (CORDIC)* method to iteratively perform the vectoring and rotation operations for the Boundary and Internal Cells respectively [6], [9], [10].

CORDIC employs only addition/subtraction, bit-shifting and look-up table (LUT) operations to calculate transcendental functions, so the CORDIC engines can be easily pipelined to meet high clock rates with very little expense to fabric resource utilization [9]. CORDIC also has two main modes: *rotation* and *vectoring*.

In rotation mode, the CORDIC block takes an X/Y magnitude value and an input angle, θ, to calculate the trigonometric functions such as $X\sin(\theta)$ and $X\cos(\theta)$. It does this by iteratively decimating the input angle to 0 in successively smaller angle steps, while adding or subtracting the X/Y value at each stage based on the sign of the phase at that step. The angle value the CORDIC engine uses at the i-th stage relates to the function $\arctan(2^{-i})$, and these arctan values are pre-computed and stored in a simple LUT. For every iteration stage, the output values gain one bit of precision; so a 16 stage CORDIC can produce 16-bit output values. A pipelined CORDIC rotator block can be seen in the following VHDL component, 'cordic':

```vhdl
-- Core logic inspired by Verilog example: https://github.com/cebarnes/cordic

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;

entity cordic is
  generic (
    G_ITERATIONS : natural := 16 -- also equates to output precision
  );
  port (
    clk           : in  std_logic;
    reset         : in  std_logic := '0'; -- (optional) sync reset for *valid's
    valid_in      : in  std_logic;
    x_in          : in  signed(G_ITERATIONS - 1 downto 0);
    y_in          : in  signed(G_ITERATIONS - 1 downto 0);
    angle_in      : in  unsigned(31 downto 0);             -- 32b phase_in (0-360deg)

    valid_out    : out std_logic;
    cos_out      : out signed(G_ITERATIONS - 1 downto 0); -- cosine/x_out
    sin_out      : out signed(G_ITERATIONS - 1 downto 0)  -- sine/y_out
  );
end entity cordic;

architecture rtl of cordic is

  type T_sign_iter  is array (integer range<>) of signed(G_ITERATIONS downto 0);
  type T_unsign_32b is array (integer range<>) of unsigned(31 downto 0);

  function F_init_atan_LUT return T_unsign_32b is
    variable V_return : T_unsign_32b(30 downto 0);
  begin
    -- +/-90deg angle rotation already accounted for in S_quad input stage
    V_return( 0) := "00100000000000000000000000000000"; -- 45.000 degrees -> atan(2^0)
    V_return( 1) := "00010010111001000000010100011101"; -- 26.565 degrees -> atan(2^-1)
    V_return( 2) := "00001001111110110011100001011011"; -- 14.036 degrees -> atan(2^-2)
    V_return( 3) := "00000101000100010001000111010100"; -- atan(2^-3)
    V_return( 4) := "00000010100010110000110101000011"; -- ...
    V_return( 5) := "00000001010001011101011111100001";
    V_return( 6) := "00000000101000101111011000011110";
    V_return( 7) := "00000000010100010111110001010101";
    V_return( 8) := "00000000001010001011111001010011";
    V_return( 9) := "00000000000101000101111100101110";
    V_return(10) := "00000000000010100010111110011000";
    V_return(11) := "00000000000001010001011111001100";
    V_return(12) := "00000000000000101000101111100110";
    V_return(13) := "00000000000000010100010111110011";
    V_return(14) := "00000000000000001010001011111001";
    V_return(15) := "00000000000000000101000101111100";
    V_return(16) := "00000000000000000010100010111110";
    V_return(17) := "00000000000000000001010001011111";
    V_return(18) := "00000000000000000000101000101111";
    V_return(19) := "00000000000000000000010100010111";
    V_return(20) := "00000000000000000000001010001011";
    V_return(21) := "00000000000000000000000101000101";
    V_return(22) := "00000000000000000000000010100010";
    V_return(23) := "00000000000000000000000001010001";
    V_return(24) := "00000000000000000000000000101000";
    V_return(25) := "00000000000000000000000000010100";
    V_return(26) := "00000000000000000000000000001010";
    V_return(27) := "00000000000000000000000000000101";
    V_return(28) := "00000000000000000000000000000010";
    V_return(29) := "00000000000000000000000000000001";
    V_return(30) := "00000000000000000000000000000000";
    return V_return;
  end F_init_atan_LUT;

  signal atan_LUT : T_unsign_32b(30 downto 0) := F_init_atan_LUT;

  signal x, y      :  T_sign_iter(G_ITERATIONS - 1 downto 0) := (others => (others => '0'));
  signal z         : T_unsign_32b(G_ITERATIONS - 1 downto 0) := (others => (others => '0'));

  signal sig_valid_sr : std_logic_vector(G_ITERATIONS - 1 downto 0) := (others => '0');

begin

  -- valid pulse output after input pulse passes through shift reg
  valid_out <= sig_valid_sr(sig_valid_sr'high);
  -- sign extend outputs
  cos_out   <= resize( x(G_ITERATIONS - 1), cos_out'length );
  sin_out   <= resize( y(G_ITERATIONS - 1), sin_out'length );

  S_shift_reg_valid: process(clk)
  begin
    if rising_edge(clk) then
      -- shift register to delay data valid to match pipeline delay
      if reset = '1' then
        sig_valid_sr <= (others => '0');
      else
        sig_valid_sr <= sig_valid_sr(G_ITERATIONS - 2 downto 0) & valid_in;
      end if;
    end if;
  end process S_shift_reg_valid;

  -- Pre-CORDIC rotations to normalize input angle & X/Y to +/- 90 deg (Quad I & IV)
```

```vhdl
-- These initial rotations are zero-gain, just sign adjustments
S_quad: process(clk)
begin
  if rising_edge(clk) then
    case angle_in(31 downto 30) is -- account for angles in different quads
      when "00" | "11" => -- (270:90)deg: no changes needed for these quadrants
        x(0) <= resize( x_in, G_ITERATIONS + 1 );
        y(0) <= resize( y_in, G_ITERATIONS + 1 );
        z(0) <= angle_in;
      when "01" => -- (90:180)deg: Quad II
        x(0) <= -resize( y_in, G_ITERATIONS + 1 );
        y(0) <=  resize( x_in, G_ITERATIONS + 1 );
        z(0) <= "00" & angle_in(29 downto 0); -- subtract pi/2 for angle in this quad
      when "10" => -- (180:270)deg: Quad III
        x(0) <=  resize( y_in, G_ITERATIONS + 1 );
        y(0) <= -resize( x_in, G_ITERATIONS + 1 );
        z(0) <= "11" & angle_in(29 downto 0); -- add pi/2 for angle in this quad
      when others =>
    end case;
  end if;
end process S_quad;

-- generate each pipelined stage for CORDIC rotations
UG_CORDIC_rotations: for i in 0 to G_ITERATIONS - 2 generate
  S_add_sub: process(clk) -- add/subtract shifted data based on phase
  begin
    if rising_edge(clk) then
      if z(i)(31) = '1' then -- Negative Phase: rotate clockwise by CORDIC angle
        x(i + 1) <= x(i) + shift_right( y(i), i );
        y(i + 1) <= y(i) - shift_right( x(i), i );
        z(i + 1) <= z(i) + atan_LUT(i);
      else -- Positive Phase: rotate counter-clockwise by CORDIC angle
        x(i + 1) <= x(i) - shift_right( y(i), i );
        y(i + 1) <= y(i) + shift_right( x(i), i );
        z(i + 1) <= z(i) - atan_LUT(i);
      end if;
    end if;
  end process S_add_sub;
end generate UG_CORDIC_rotations;

end architecture rtl;
```
Listing 3.2: Basic CORDIC Rotator

Note that as part of the CORDIC iterative process, each iteration stage not only provides an extra bit of precision, but also changes the processing gain experienced at the outputs of the CORDIC engine. The processing gain can be cancelled out by multiplying the CORDIC outputs by a scale factor a, as computed in Equation 3.13 given n CORDIC processing stages [16].

$$a = \prod_{i=0}^{n-1} \frac{1}{\sqrt{1+2^{-2i}}} \tag{3.13}$$

The other form of the CORDIC function is that of the vectoring mode; in this mode, the CORDIC block takes some X/Y cartesian coordinates and calculates the magnitude and phase using a similar iterative approach as in the rotation mode. This form can also be seen as a complex (rectangular I/Q coordinates) to magnitude and phase (polar coordinates) conversion process. The VHDL entity that achieves this vectoring process can be seen below:

```
-- inspired by https://github.com/ZipCPU/cordic/blob/master/rtl/
    topolar.v
--    ^ since GPL, this component shall be GPL licensed as well
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;

entity cordic_vec is
  generic (
    G_ITERATIONS : natural := 16 -- also equates to output
    precision
  );
  port (
    clk           : in  std_logic;
    reset         : in  std_logic := '0'; -- (optional) sync reset
    for *valid's
    valid_in      : in  std_logic;
    x_in          : in  signed(G_ITERATIONS - 1 downto 0);
    y_in          : in  signed(G_ITERATIONS - 1 downto 0);

    valid_out     : out std_logic;
    phase_out     : out unsigned(31 downto 0); -- 32b phase (0-360 deg)
    mag_out       : out signed(G_ITERATIONS - 1 downto 0)
  );
end entity cordic_vec;

architecture rtl of cordic_vec is

  type T_sign_iter  is array (integer range<>) of signed(G_ITERATIONS downto 0);
  type T_unsign_32b is array (integer range<>) of unsigned(31 downto 0);

  function F_init_atan_LUT return T_unsign_32b is
    variable V_return : T_unsign_32b(29 downto 0);
  begin
    -- +/-45deg angle rotation already accounted for in S_pre_cordic input stage
    V_return( 0) := "00010010111001000000010100011101"; -- 26.565 degrees -> atan(2^-1)
    V_return( 1) := "00001001111110110011100001011011"; -- 14.036 degrees -> atan(2^-2)
    V_return( 2) := "00000101000100010001000111010100"; -- atan(2^-3)
    V_return( 3) := "00000010100010110000110101000011"; -- ...
    V_return( 4) := "00000001010001011101011111100001";
    V_return( 5) := "00000000101000101111011000011110";
    V_return( 6) := "00000000010100010111110001010101";
    V_return( 7) := "00000000001010001011111001010011";
    V_return( 8) := "00000000000101000101111100101110";
    V_return( 9) := "00000000000010100010111110011000";
    V_return(10) := "00000000000001010001011111001100";
    V_return(11) := "00000000000000101000101111100110";
    V_return(12) := "00000000000000010100010111110011";
    V_return(13) := "00000000000000001010001011111001";
    V_return(14) := "00000000000000000101000101111100";
    V_return(15) := "00000000000000000010100010111110";
    V_return(16) := "00000000000000000001010001011111";
    V_return(17) := "00000000000000000000101000101111";
    V_return(18) := "00000000000000000000010100010111";
    V_return(19) := "00000000000000000000001010001011";
    V_return(20) := "000000000000000000000000101000101";
    V_return(21) := "00000000000000000000000010100010";
    V_return(22) := "0000000000000000000000001010001";
    V_return(23) := "0000000000000000000000000101000";
    V_return(24) := "0000000000000000000000000010100";
    V_return(25) := "0000000000000000000000000001010";
    V_return(26) := "0000000000000000000000000000101";
    V_return(27) := "0000000000000000000000000000010";
    V_return(28) := "0000000000000000000000000000001";
    V_return(29) := "0000000000000000000000000000000";
    return V_return;
  end F_init_atan_LUT;

  signal atan_LUT : T_unsign_32b(29 downto 0) := F_init_atan_LUT;

  signal x, y      :  T_sign_iter(G_ITERATIONS - 1 downto 0) := (
   others => (others => '0'));
  signal ph        : T_unsign_32b(G_ITERATIONS - 1 downto 0) := (
   others => (others => '0'));

  signal sig_valid_sr : std_logic_vector(G_ITERATIONS - 1 downto
   0) := (others => '0');

begin

  -- valid pulse output after input pulse passes through shift reg
  valid_out <= sig_valid_sr(sig_valid_sr'high);
  phase_out <= ph(G_ITERATIONS - 1);
  -- sign extend magnitude output
  mag_out   <= resize( x(G_ITERATIONS - 1), mag_out'length );

  S_shift_reg_valid: process(clk)
  begin
    if rising_edge(clk) then
      -- shift register to delay data valid to match pipeline
   delay
      if reset = '1' then
        sig_valid_sr <= (others => '0');
      else
        sig_valid_sr <= sig_valid_sr(G_ITERATIONS - 2 downto 0) &
   valid_in;
      end if;
    end if;
  end process S_shift_reg_valid;

```

```vhdl
-- Pre-CORDIC rotations to map input angle to +/- 45deg based on X/Y input quadrant
-- NOTE: use hex(degree_to_signed_fx()) function in Python to help with angle conversions
S_pre_cordic: process(clk)
begin
  if rising_edge(clk) then
    -- Quad IV: rotate by -315deg (so set initial phase to 315 deg)
    if (x_in(x_in'left) = '0') and (y_in(y_in'left) = '1') then
       x(0) <= resize( x_in, G_ITERATIONS + 1 ) - resize( y_in, G_ITERATIONS + 1 );
       y(0) <= resize( x_in, G_ITERATIONS + 1 ) + resize( y_in, G_ITERATIONS + 1 );
      ph(0) <= X"e000_0000";
    -- Quad II: rotate by -135deg (init phase = 135deg)
    elsif (x_in(x_in'left) = '1') and (y_in(y_in'left) = '0') then
       x(0) <= -resize( x_in, G_ITERATIONS + 1 ) + resize( y_in, G_ITERATIONS + 1 );
       y(0) <= -resize( x_in, G_ITERATIONS + 1 ) - resize( y_in, G_ITERATIONS + 1 );
      ph(0) <= X"6000_0000";
    -- Quad III: rotate by -225deg (init phase = 225deg)
    elsif (x_in(x_in'left) = '1') and (y_in(y_in'left) = '1') then
       x(0) <= -resize( x_in, G_ITERATIONS + 1 ) - resize( y_in, G_ITERATIONS + 1 );
       y(0) <=  resize( x_in, G_ITERATIONS + 1 ) - resize( y_in, G_ITERATIONS + 1 );
      ph(0) <= X"a000_0000";
    else -- Quad I ["00"]: rotate by -45deg (init phase = 45deg)
       x(0) <=  resize( x_in, G_ITERATIONS + 1 ) + resize( y_in, G_ITERATIONS + 1 );
       y(0) <= -resize( x_in, G_ITERATIONS + 1 ) + resize( y_in, G_ITERATIONS + 1 );
      ph(0) <= X"2000_0000";
    end if;
  end if;
end process S_pre_cordic;

-- generate each pipelined stage for CORDIC rotations
UG_CORDIC_rotations: for i in 0 to G_ITERATIONS - 2 generate
  -- CORDIC process for rectangular -> polar rotates the Y value to 0 and
```

```
    -- gives the magnitude of the vector as our x value and the phase as the
    -- angle it took to rotate the Y component to 0
    S_add_sub: process(clk)
    begin
      if rising_edge(clk) then
        if y(i)(y(i)'left) = '1' then -- Negative Y val: rotate by CORDIC angle in (+) direction
           x(i + 1) <=  x(i) - shift_right( y(i), i+1 );
           y(i + 1) <=  y(i) + shift_right( x(i), i+1 );
          ph(i + 1) <= ph(i) - atan_LUT(i);
        else -- Positive Y val: rotate by CORDIC angle in (-) direction
           x(i + 1) <=  x(i) + shift_right( y(i), i+1 );
           y(i + 1) <=  y(i) - shift_right( x(i), i+1 );
          ph(i + 1) <= ph(i) + atan_LUT(i);
        end if;
      end if;
    end process S_add_sub;
  end generate UG_CORDIC_rotations;

end architecture rtl;
```

Listing 3.3: Basic CORDIC Vectoring

For the Boundary Cell, we need to calculate two angles for Givens Rotations, ϕ and θ. The first CORDIC vectoring engine essentially transforms the complex input sample into its phase and magnitude components, and as such, the first angle ϕ is calculated by Equation 3.14 [5], [9].

$$\phi = \arctan\left(\frac{\Im(x_{in})}{\Re(x_{in})}\right) \tag{3.14}$$

The angle ϕ is seen as the rotational angle the CORDIC engine took to eliminate the y-axis (imaginary) component of the complex input vector, which equates to the input sample's phase. This phase value ϕ is then utilized by the Internal Cells within the same row as the Boundary Cell to rotate given input samples within the array [9]. The second angle θ to be calculated in the BC annihilates an element of the input matrix, which culminates into the upper-triangular matrix form of **R** in Givens Rotations [5], [9]. The value of θ can be found by Equation 3.15 [9].

$$\theta = \arctan\left(\frac{x_{in}e^{-j\phi}}{x}\right) \tag{3.15}$$

The block diagram of the CORDIC Boundary Cell unit can be seen as in Figure 3.9.

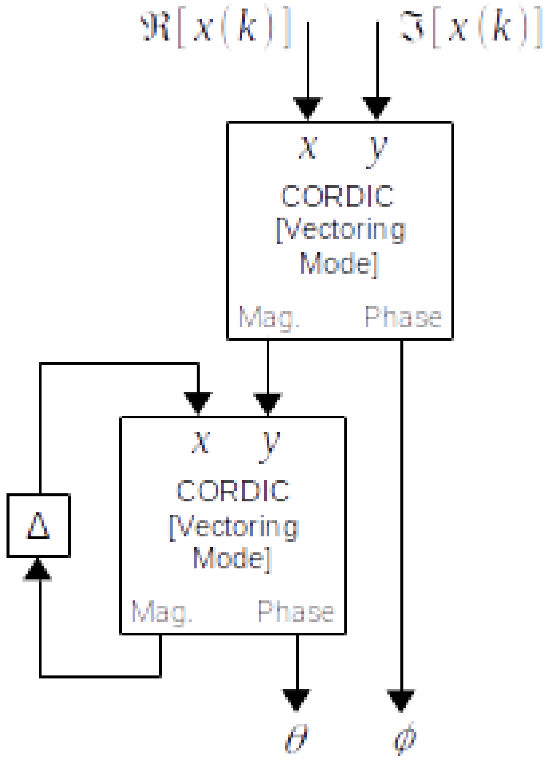


Figure 3.9: CORDIC Boundary Cell

The VHDL entity designed for the Boundary Cell can be seen below:

```
--
-- Implements the boundary cell (BC) of the QR architecture using
   two vector-mode
```

```vhdl
-- CORDIC engines to perform the "vectoring" on complex input samples to
-- nullify their imaginary parts and form rotation angles used by internal cells.
--
-- Inputs:
-- =======
-- - `CORDIC_scale`: scale factor to counteract CORDIC gain on magnitude from
--     vectoring engines
-- - `lambda`: (optional) forgetting factor applied to feedback magnitude. This
--     value is often selected to be slightly less than 1 (e.g. 0.99). When the
--     generic `G_USE_LAMBDA` == false, this forgetting factor is ignored and no
--     multiplier is used.
--

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;

entity boundary_cell is
  generic (
    G_DATA_WIDTH : natural := 16;    -- operational bitwidth of datapath (in & out)
    G_USE_LAMBDA : boolean := false -- use forgetting factor (lambda) in BC calc
  );
  port (
    clk          : in  std_logic;
    reset        : in  std_logic;
    CORDIC_scale : in  signed(G_DATA_WIDTH - 1 downto 0) := X"4DBA";
    lambda       : in  signed(G_DATA_WIDTH - 1 downto 0) := X"7EB8";

    x_real       : in  signed(G_DATA_WIDTH - 1 downto 0); -- real
    x_imag       : in  signed(G_DATA_WIDTH - 1 downto 0); -- imag
    x_valid      : in  std_logic;
    x_ready      : out std_logic;

    -- Current CORDIC/trig blocks use 32b unsigned angles, so keep to that
    -- since this will directly feed the Internal Cell CORDIC Rotators
    phi_out      : out unsigned(31 downto 0);
    theta_out    : out unsigned(31 downto 0);
    bc_valid_out : out std_logic;
    ic_ready     : in  std_logic  -- downstream internal cell (IC) ready to consume
  );
end entity boundary_cell;

architecture rtl of boundary_cell is

  component cordic_vec_scaled is
    generic (
      G_ITERATIONS : integer := 16 -- also equates to output precision
    );
    port (
      clk          : in  std_logic;
      reset        : in  std_logic := '0'; -- (optional) sync reset for *valid's
      valid_in     : in  std_logic;
      x_in         : in  signed(G_ITERATIONS - 1 downto 0);
      y_in         : in  signed(G_ITERATIONS - 1 downto 0);
      CORDIC_scale : in  signed(G_ITERATIONS - 1 downto 0) := X"4DBA";

      valid_out    : out std_logic;
      phase_out    : out unsigned(31 downto 0);
      mag_out      : out signed(G_ITERATIONS - 1 downto 0)
    );
  end component;

  type T_bc_fsm is (S_IDLE, S_WAIT_PHI, S_WAIT_THETA, S_OUT_VALID);
  signal sig_bc_state : T_bc_fsm := S_IDLE;

  -- related to U_input_vectoring
  signal sig_x_valid_gated   : std_logic;
  signal sig_input_vec_valid : std_logic := '0';
  signal sig_phi_out         : unsigned(31 downto 0);
  signal sig_input_vec_mag   : signed(G_DATA_WIDTH - 1 downto 0);

  -- related to U_output_vectoring
  signal sig_output_vec_valid_out : std_logic := '0';
  signal sig_theta_out            : unsigned(31 downto 0);
  signal sig_output_vec_mag       : signed(G_DATA_WIDTH - 1 downto
    0);
  signal sig_feedback_mag         : signed(G_DATA_WIDTH - 1 downto
    0);
  signal sig_feedback_mag_valid   : std_logic := '0';
  signal sig_output_vec_valid_in  : std_logic := '0';

  -- forgetting factor scaling
  signal sig_lambda_mag_valid      : std_logic := '0';
  signal sig_lambda_mag            : signed((2*G_DATA_WIDTH) - 1
   downto 0);

  -- output registers of theta & phi
  signal sig_phi_out_q   : unsigned(31 downto 0) := (others =>
   '0');
  signal sig_theta_out_q : unsigned(31 downto 0) := (others =>
   '0');

begin

  x_ready      <= '1' when (sig_bc_state = S_IDLE) and (reset =
   '0') else '0';
  phi_out      <= sig_phi_out_q;
  theta_out    <= sig_theta_out_q;
  bc_valid_out <= '1' when sig_bc_state = S_OUT_VALID else '0';

  sig_x_valid_gated <= x_valid when sig_bc_state = S_IDLE else
   '0';

  U_input_vectoring: cordic_vec_scaled
    generic map (
      G_ITERATIONS => G_DATA_WIDTH
    )
    port map (
      clk          => clk,
      reset        => reset,
      valid_in     => sig_x_valid_gated,
      x_in         => x_real,
      y_in         => x_imag,
      CORDIC_scale => CORDIC_scale,

      valid_out    => sig_input_vec_valid,
      phase_out    => sig_phi_out,       -- phi = atan2(Q, I)
      mag_out      => sig_input_vec_mag -- mag = sqrt(I**2 + Q**2)
    );

```

```vhdl
-- we need only care about input vectoring magnitude valid as feedback magnitude
-- will _always_ be valid and stable before this point, due to being calculated
-- from previous cycle (or from reset, default value). Thus the signal
-- ‘sig_feedback_mag_valid‘ is purely for informational/debug value, and will
-- get optmized out as a dead-path in synthesis as nothing reads it
sig_output_vec_valid_in <= sig_input_vec_valid;

U_output_vectoring: cordic_vec_scaled
  generic map (
    G_ITERATIONS => G_DATA_WIDTH
  )
  port map (
    clk          => clk,
    reset        => reset,
    valid_in     => sig_output_vec_valid_in,
    x_in         => sig_feedback_mag,
    -- scaled magnitude output from input vectoring
    y_in         => sig_input_vec_mag,
    CORDIC_scale => CORDIC_scale,

    valid_out    => sig_output_vec_valid_out,
    phase_out    => sig_theta_out,
    mag_out      => sig_output_vec_mag
  );


UG_apply_forgetting_factor: if G_USE_LAMBDA generate
  S_scale_lambda: process(clk)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        -- feedback magnitude's zero'ed on reset
        sig_lambda_mag_valid   <= '0';
        sig_lambda_mag         <= (others => '0');
        sig_feedback_mag       <= (others => '0');
        sig_feedback_mag_valid <= '0';
      else
        -- apply lambda scaling/forgetting factor for feedback magnitude
        if sig_output_vec_valid_out = '1' then
          sig_lambda_mag     <= sig_output_vec_mag * lambda;
        end if;
        sig_lambda_mag_valid <= sig_output_vec_valid_out;

        -- scale back down to operational data width
        if sig_lambda_mag_valid = ’1’ then
          sig_feedback_mag     <= resize( shift_right(
 sig_lambda_mag,

 G_DATA_WIDTH - 1 ),
                                          sig_feedback_mag’
 length );
        end if;
        sig_feedback_mag_valid <= sig_lambda_mag_valid;
      end if;
    end if;
  end process S_scale_lambda;
end generate UG_apply_forgetting_factor;

UG_no_forgetting_factor: if not G_USE_LAMBDA generate
  S_no_lambda: process(clk)
  begin
    if rising_edge(clk) then
      if reset = ’1’ then
        -- feedback magnitude’s zero’ed on reset
        sig_feedback_mag       <= (others => ’0’);
        sig_feedback_mag_valid <= ’0’;
      else
        if sig_output_vec_valid_out = ’1’ then
          sig_feedback_mag     <= sig_output_vec_mag;
        end if;
        sig_feedback_mag_valid <= sig_output_vec_valid_out;
      end if;
    end if;
  end process S_no_lambda;
end generate UG_no_forgetting_factor;


S_output_FSM: process(clk)
begin
  if rising_edge(clk) then
    if reset = ’1’ then
      sig_bc_state <= S_IDLE;
    else
      case sig_bc_state is
        when S_IDLE =>
          if x_valid = ’1’ then
```

```
              sig_bc_state <= S_WAIT_PHI;
            end if;

          when S_WAIT_PHI =>
            if sig_input_vec_valid = '1' then
              sig_phi_out_q <= sig_phi_out;
              sig_bc_state  <= S_WAIT_THETA;
            end if;

          -- since theta needs second CORDIC vectoring operation,
   it will always
          -- take longer than input/first CORDIC vectoring
   operation
          when S_WAIT_THETA =>
            if sig_output_vec_valid_out = '1' then
              sig_theta_out_q <= sig_theta_out;
              sig_bc_state    <= S_OUT_VALID;
            end if;

          when S_OUT_VALID =>
            -- wait till downstream internal cell is ready to
   consume theta & phi
            if ic_ready = '1' then
              sig_bc_state <= S_IDLE;
            end if;

          when others => sig_bc_state <= S_IDLE;
        end case;
      end if;
    end if;
  end process S_output_FSM;

end architecture rtl;
```

Listing 3.4: CORDIC-based Boundary Cell

The Internal Cell rotates each input sample x_{in} by the angles ϕ and θ given from that row's Boundary Cell [9]. These rotated samples are then passed to the next row via x_{out}. The block diagram of the CORDIC Internal Cell made up of CORDIC rotation engines can be seen in Figure 3.10.

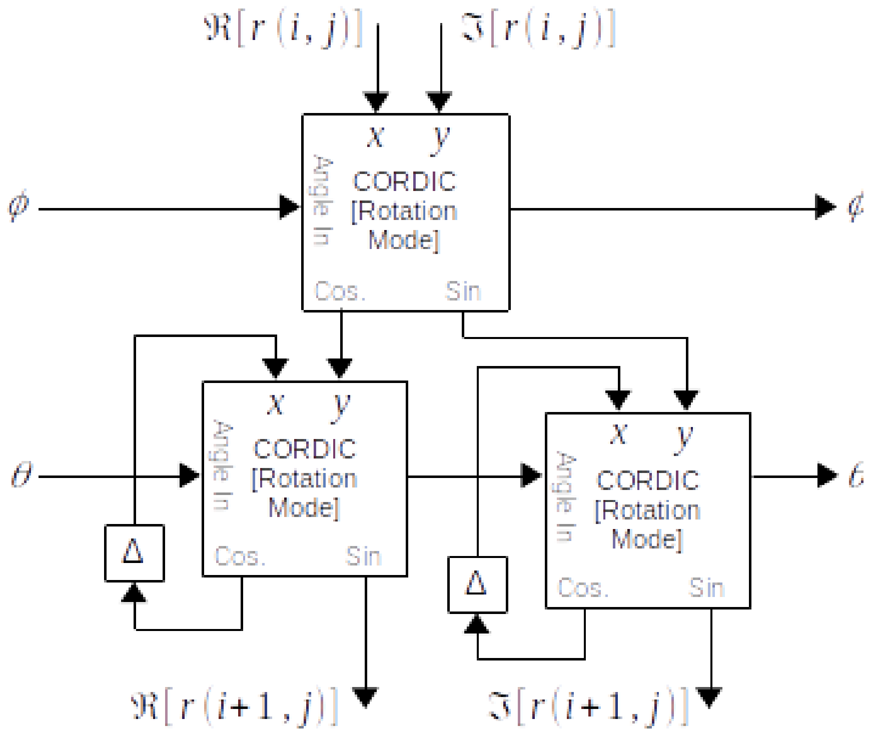


Figure 3.10: CORDIC Internal Cell

The VHDL entity designed for the Internal Cell can be seen below:

```
--
-- Implements the internal cell (IC) of the QR architecture using four rotation-mode
-- CORDIC engines
--
-- Inputs:
-- =======
-- - ‘CORDIC_scale‘: scale factor to counteract CORDIC gain on magnitude from
--    vectoring engines
-- - ‘lambda‘: (optional) forgetting factor applied to feedback magnitude. This
--    value is often selected to be slightly less than 1 (e.g. 0.99). When the
--    generic ‘G_USE_LAMBDA‘ == false, this forgetting factor is ignored and no
--    multiplier is used. For inverse internal cells, set this value to 1/lambda
--
```

```vhdl

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;

entity internal_cell is
  generic (
    G_DATA_WIDTH : natural := 16;    -- operational bitwidth of datapath (in & out)
    G_USE_LAMBDA : boolean := false -- use forgetting factor (lambda) in BC calc
  );
  port (
    clk          : in  std_logic;
    reset        : in  std_logic;
    CORDIC_scale : in  signed(G_DATA_WIDTH - 1 downto 0) := X"4DBA";
    lambda       : in  signed(G_DATA_WIDTH - 1 downto 0) := X"7EB8";

    xin_real     : in  signed(G_DATA_WIDTH - 1 downto 0);
    xin_imag     : in  signed(G_DATA_WIDTH - 1 downto 0);
    xin_valid    : in  std_logic;
    xin_ready    : out std_logic;
    -- Current CORDIC/trig blocks use 32b unsigned angles, so keep to that
    -- since this is directly feed from Boundary Cell CORDIC Vector engines
    phi_in       : in  unsigned(31 downto 0);
    theta_in     : in  unsigned(31 downto 0);
    bc_valid_in  : in  std_logic; -- connected to BC on first IC in row, else connected to angles valid from previous IC in row
    ic_ready     : out std_logic; -- this internal cell (IC) ready to consume (only needed for first IC connected to BC)


    xout_real    : out signed(G_DATA_WIDTH - 1 downto 0);
    xout_imag    : out signed(G_DATA_WIDTH - 1 downto 0);
    xout_valid   : out std_logic;
    xout_ready   : in  std_logic;

    -- These are registered copies, propogated to next IC in row, to prevent
    -- high fan-out of 32b angle signals (no handshaking needed, since ICs not
    -- connected directly to a BC have handshaking/timing with
   rotations)
    phi_out      : out unsigned(31 downto 0);
    theta_out    : out unsigned(31 downto 0);
    angles_valid : out std_logic
  );
end entity internal_cell;

architecture rtl of internal_cell is

  component cordic_rot_scaled is
    generic (
      G_ITERATIONS : natural := 16 -- also equates to output
   precision
    );
    port (
      clk          : in  std_logic;
      reset        : in  std_logic := '0'; -- (optional) sync
   reset for *valid's
      valid_in     : in  std_logic;
      x_in         : in  signed(G_ITERATIONS - 1 downto 0);
      y_in         : in  signed(G_ITERATIONS - 1 downto 0);
      angle_in     : in  unsigned(31 downto 0);             -- 32b
    phase_in (0-360deg)
      CORDIC_scale : in  signed(G_ITERATIONS - 1 downto 0) := X"4
   DBA";

      valid_out    : out std_logic;
      cos_out      : out signed(G_ITERATIONS - 1 downto 0); --
   cosine/x_out
      sin_out      : out signed(G_ITERATIONS - 1 downto 0)  --
   sine/y_out
    );
  end component cordic_rot_scaled;

  type T_ic_fsm is (S_IDLE, S_CONSUME, S_WAIT_ROTATIONS,
   S_OUT_VALID);
  signal sig_ic_state : T_ic_fsm := S_IDLE;

  signal sig_inputs_valid         : std_logic;

  -- Input Rotator
  signal sig_in_rot_valid_out     : std_logic;
  signal sig_in_rot_cos_out       : signed(G_DATA_WIDTH - 1 downto
   0);
  signal sig_in_rot_sin_out      : signed(G_DATA_WIDTH - 1 downto
   0);

  -- Real Rotator
  signal sig_real_rot_valid      : std_logic;
  signal sig_real_x_feedback     : signed(G_DATA_WIDTH - 1 downto
   0);
  signal sig_real_x_out          : signed(G_DATA_WIDTH - 1 downto
   0);
  signal sig_real_y_out          : signed(G_DATA_WIDTH - 1 downto
   0);

  -- Imag Rotator
  signal sig_imag_rot_valid      : std_logic;
  signal sig_imag_x_feedback     : signed(G_DATA_WIDTH - 1 downto
   0);
  signal sig_imag_x_out          : signed(G_DATA_WIDTH - 1 downto
   0);
  signal sig_imag_y_out          : signed(G_DATA_WIDTH - 1 downto
   0);

  -- Registered outputs
  signal sig_xout_real        : signed(G_DATA_WIDTH - 1 downto 0);
  signal sig_xout_imag        : signed(G_DATA_WIDTH - 1 downto 0);
  signal sig_phi_out          : unsigned(31 downto 0);
  signal sig_theta_out        : unsigned(31 downto 0);

begin

  -- assert ready once able to consume both x/sample & BC inputs
  -- due to difference in timing between datapaths
  xin_ready    <= '1' when sig_ic_state = S_CONSUME   else '0';
  ic_ready     <= '1' when sig_ic_state = S_CONSUME   else '0';
  xout_valid   <= '1' when sig_ic_state = S_OUT_VALID else '0';
  angles_valid <= '1' when sig_ic_state = S_OUT_VALID else '0';

  xout_real <= sig_xout_real;
  xout_imag <= sig_xout_imag;
  phi_out   <= sig_phi_out;
  theta_out <= sig_theta_out;

  -- gated valid signal, only propagate through once we've
   consumed a sample
  sig_inputs_valid <= '1' when sig_ic_state = S_CONSUME else '0';

  U_input_rotator: cordic_rot_scaled
```

```vhdl
    generic map (
      G_ITERATIONS => G_DATA_WIDTH
    )
    port map (
      clk          => clk,
      reset        => reset,
      valid_in     => sig_inputs_valid,
      x_in         => xin_real,
      y_in         => xin_imag,
      angle_in     => phi_in,
      CORDIC_scale => CORDIC_scale,

      valid_out    => sig_in_rot_valid_out,
      cos_out      => sig_in_rot_cos_out,
      sin_out      => sig_in_rot_sin_out
    );

  U_real_rotator: cordic_rot_scaled
    generic map (
      G_ITERATIONS => G_DATA_WIDTH
    )
    port map (
      clk          => clk,
      reset        => reset,
      valid_in     => sig_in_rot_valid_out,
      x_in         => sig_real_x_feedback,
      y_in         => sig_in_rot_cos_out,
      angle_in     => theta_in,
      CORDIC_scale => CORDIC_scale,

      valid_out    => sig_real_rot_valid,
      cos_out      => sig_real_x_out,
      sin_out      => sig_real_y_out
    );

  U_imag_rotator: cordic_rot_scaled
    generic map (
      G_ITERATIONS => G_DATA_WIDTH
    )
    port map (
      clk          => clk,
      reset        => reset,
      valid_in     => sig_in_rot_valid_out,
      x_in         => sig_imag_x_feedback,
      y_in         => sig_in_rot_sin_out,
      angle_in     => theta_in,
      CORDIC_scale => CORDIC_scale,

      valid_out    => sig_imag_rot_valid,
      cos_out      => sig_imag_x_out,
      sin_out      => sig_imag_y_out
    );


  UG_no_lambda: if not G_USE_LAMBDA generate
    S_X_feedbacks: process(clk)
    begin
      if rising_edge(clk) then
        if reset = '1' then
          sig_real_x_feedback <= (others => '0');
          sig_imag_x_feedback <= (others => '0');
        else
          if sig_real_rot_valid = '1' then
            sig_real_x_feedback <= sig_real_x_out;
          end if;

          if sig_imag_rot_valid = '1' then
            sig_imag_x_feedback <= sig_imag_x_out;
          end if;
        end if;
      end if;
    end process S_X_feedbacks;
  end generate UG_no_lambda;

  S_output_FSM: process(clk)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        sig_ic_state <= S_IDLE;
      else
        case sig_ic_state is
          when S_IDLE =>
            if (xin_valid = '1') and (bc_valid_in = '1') then
              sig_ic_state <= S_CONSUME;
            end if;

          when S_CONSUME =>
            sig_ic_state <= S_WAIT_ROTATIONS;

          when S_WAIT_ROTATIONS =>
            -- Real & Imag rotations should take exactly the same
   amount of time
```

```
            if (sig_real_rot_valid = '1') and (sig_imag_rot_valid
   = '1') then
              sig_xout_real <= sig_real_y_out;
              sig_xout_imag <= sig_imag_y_out;
              sig_ic_state  <= S_OUT_VALID;
            end if;

          when S_OUT_VALID =>
            -- wait till downstream internal/boundary cell in next
    row is ready
            if xout_ready = '1' then
              sig_ic_state <= S_IDLE;
            end if;

          when others => sig_ic_state <= S_IDLE;
        end case;
      end if;
    end if;
  end process S_output_FSM;

  S_pipeline_angles: process(clk)
  begin
    if rising_edge(clk) then
      if bc_valid_in = '1' then -- reg angles whenever valid to
   hold until output
        sig_phi_out   <= phi_in;
        sig_theta_out <= theta_in;
      end if;
    end if;
  end process S_pipeline_angles;

end architecture rtl;
```

Listing 3.5: CORDIC-based Internal Cell

The weight extract cell uses simple multiply and subtraction, so the VHDL entity can be directly shown here:

```
-- Weight Extraction Cell:
-- W_{i,j}(k) =  W_{i,j}(k - 1) - a_{i}(k)b_{i}(k)

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;

entity weight_extract_cell is
```

```
  generic (
    G_DATA_WIDTH : natural := 16
  );
  port (
    clk           : in  std_logic;
    reset         : in  std_logic;

    -- no ‘ready‘ signal as a is updated across final row
    ain_real      : in  signed(G_DATA_WIDTH - 1 downto 0);
    ain_imag      : in  signed(G_DATA_WIDTH - 1 downto 0);
    ain_valid     : in  std_logic;

    -- pipelined ‘a‘ to be passed to next weight extract cell
    aout_real     : out signed(G_DATA_WIDTH - 1 downto 0);
    aout_imag     : out signed(G_DATA_WIDTH - 1 downto 0);
    aout_valid    : out std_logic;

    b_real        : in  signed(G_DATA_WIDTH - 1 downto 0);
    b_imag        : in  signed(G_DATA_WIDTH - 1 downto 0);
    b_valid       : in  std_logic;
    b_ready       : out std_logic;

    w_real        : out signed(G_DATA_WIDTH - 1 downto 0);
    w_imag        : out signed(G_DATA_WIDTH - 1 downto 0);
    w_valid       : out std_logic;
    w_ready       : in  std_logic
  );
end entity weight_extract_cell;

architecture rtl of weight_extract_cell is

  component complex_multiply_mult4 is
    generic (
      G_AWIDTH : natural := 16;     -- size of 1st input of
   multiplier
      G_BWIDTH : natural := 18;     -- size of 2nd input of
   multiplier
      G_CONJ_A : boolean := false; -- take complex conjugate of
   arg A
      G_CONJ_B : boolean := false  -- take complex conjugate of
   arg B
    );
    port (
      clk      : in  std_logic;
      reset    : in  std_logic := ’0’; -- (optional) sync reset
   for *valid’s
```

```
      ab_valid : in  std_logic; -- A & B complex input data valid
      ar       : in  signed(G_AWIDTH - 1 downto 0); -- 1st input's real part
      ai       : in  signed(G_AWIDTH - 1 downto 0); -- 1st input's imaginary part
      br       : in  signed(G_BWIDTH - 1 downto 0); -- 2nd input's real part
      bi       : in  signed(G_BWIDTH - 1 downto 0); -- 2nd input's imaginary part
      p_valid  : out std_logic; -- Product complex output data valid
      pr       : out signed(G_AWIDTH + G_BWIDTH downto 0); -- real part of output
      pi       : out signed(G_AWIDTH + G_BWIDTH downto 0)  -- imaginary part of output
    );
  end component;

  type T_weight_fsm is (S_IDLE, S_CONSUME, S_WAIT_CALC, S_OUT_VALID);
  signal sig_weight_state : T_weight_fsm := S_IDLE;

  signal sig_input_valid : std_logic;

  signal sig_ab_valid : std_logic := '0';
  signal sig_ab_real  : signed((2*G_DATA_WIDTH) downto 0);
  signal sig_ab_imag  : signed((2*G_DATA_WIDTH) downto 0);

  signal sig_weight_z_real : signed(G_DATA_WIDTH downto 0);
  signal sig_weight_z_imag : signed(G_DATA_WIDTH downto 0);

  signal sig_aout_real    : signed(G_DATA_WIDTH - 1 downto 0);
  signal sig_aout_imag    : signed(G_DATA_WIDTH - 1 downto 0);
  signal sig_aout_valid   : std_logic;

begin

  aout_real    <= sig_aout_real;
  aout_imag    <= sig_aout_imag;
  aout_valid   <= sig_aout_valid;

  sig_input_valid <= '1' when sig_weight_state = S_CONSUME else '0';
  b_ready         <= '1' when sig_weight_state = S_CONSUME else '0';

```

```
w_real  <= resize( shift_right( sig_weight_z_real, 1 ), w_real'
 length );
w_imag  <= resize( shift_right( sig_weight_z_imag, 1 ), w_imag'
 length );
w_valid <= '1' when sig_weight_state = S_OUT_VALID else '0';

-- register 'a' to next weight extract cell
S_reg_a: process(clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      sig_aout_valid <= '0';
    else
      if ain_valid = '1' then
        sig_aout_real  <= ain_real;
        sig_aout_imag  <= ain_imag;
      end if;
      sig_aout_valid <= ain_valid;
    end if;
  end if;
end process S_reg_a;

U_cmult_AB: complex_multiply_mult4
  generic map (
    G_AWIDTH => G_DATA_WIDTH,
    G_BWIDTH => G_DATA_WIDTH,
    G_CONJ_A => false,
    G_CONJ_B => false
  )
  port map (
    clk      => clk,
    reset    => reset,
    ab_valid => sig_input_valid,
    ar       => ain_real,
    ai       => ain_imag,
    br       => b_real,
    bi       => b_imag,
    p_valid  => sig_ab_valid,
    pr       => sig_ab_real,
    pi       => sig_ab_imag
  );

S_weight_diff: process(clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
```

```
        sig_weight_z_real <= (others => '0');
        sig_weight_z_imag <= (others => '0');
      else
        if sig_ab_valid = '1' then
          sig_weight_z_real <= sig_weight_z_real - resize(
   shift_right( sig_ab_real,

         G_DATA_WIDTH + 1 ),

   G_DATA_WIDTH + 1 );
          sig_weight_z_imag <= sig_weight_z_imag - resize(
   shift_right( sig_ab_imag,

         G_DATA_WIDTH + 1 ),

   G_DATA_WIDTH + 1 );
        end if;
      end if;
    end if;
  end process S_weight_diff;

  S_output_FSM: process(clk)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        sig_weight_state <= S_IDLE;
      else
        case sig_weight_state is
          when S_IDLE =>
            -- only care about b_valid to continue, since a should
    always be updated
            -- before b value since it comes from a preceeding QRD
    column output
            if b_valid = '1' then
              sig_weight_state <= S_CONSUME;
            end if;

          when S_CONSUME =>
            sig_weight_state <= S_WAIT_CALC;

          when S_WAIT_CALC =>
            if sig_ab_valid = '1' then
              sig_weight_state <= S_OUT_VALID;
            end if;

          when S_OUT_VALID =>
```

```
            if w_ready = '1' then
              sig_weight_state <= S_IDLE;
            end if;

          when others => sig_weight_state <= S_IDLE;
        end case;
      end if;
    end if;
  end process S_output_FSM;

end rtl;
```

Listing 3.6: Weight Extract Cell

3.2.3 IQRD Top-Level

Tied together structurally, the top-level IQRD component matches the Signal Flow Graph in Figure 3.6. The design also uses VHDL *generics* to parameterize the bit width of the internal data samples, as well as to support an arbitrary number of channels and samples.

```
-- Inverse QR Decomposition (IQRD)
--   Solves the linear equation Ax = b for x, where:
--     Âů A = complex input matrix, of size (M,N), where M âĽě N
--     Âů b = complex input vector, of size (M,1)
--     Âů x = complex output vector to solve for, of size (N,1)
--
-- NOTE: For ready/valid handshaking, components with multiple
   input dependencies
--       (such as this top-level component) expect data producers
   (e.x. A & b
--       driven inputs) to assert ‘valid‘ before this component
   asserts ‘ready‘
--       which then signals to the driving component(s) that input
    data aligned
--       to that ‘valid‘ has been successfully consumed.
--

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
library work;
```

```vhdl
  use work.util_pkg.all;

entity IQRD is
  generic (
    G_DATA_WIDTH : positive := 16;     -- operational bitwidth of datapath (in & out)
    G_USE_LAMBDA : boolean  := false; -- use forgetting factor (lambda) in BC calc
    G_M          : positive := 4;
    G_N          : positive := 3
  );
  port (
    clk          : in  std_logic;
    reset        : in  std_logic;
    CORDIC_scale : in  signed(G_DATA_WIDTH - 1 downto 0) := X"4DBA";
    lambda       : in  signed(G_DATA_WIDTH - 1 downto 0) := X"7EB8"; -- 0.99
    inv_lambda   : in  signed(G_DATA_WIDTH - 1 downto 0) := X"814A"; -- 1.01

    A_real       : in  T_signed_3D(G_M - 1 downto 0)
                                  (G_N - 1 downto 0)
                                  (G_DATA_WIDTH - 1 downto 0);
    A_imag       : in  T_signed_3D(G_M - 1 downto 0)
                                  (G_N - 1 downto 0)
                                  (G_DATA_WIDTH - 1 downto 0);
    A_valid      : in  std_logic;
    A_ready      : out std_logic;

    b_real       : in  T_signed_2D(G_M - 1 downto 0)
                                  (G_DATA_WIDTH - 1 downto 0);
    b_imag       : in  T_signed_2D(G_M - 1 downto 0)
                                  (G_DATA_WIDTH - 1 downto 0);
    b_valid      : in  std_logic;
    b_ready      : out std_logic;

    x_real       : out T_signed_2D(G_N - 1 downto 0)
                                  (G_DATA_WIDTH - 1 downto 0);
    x_imag       : out T_signed_2D(G_N - 1 downto 0)
                                  (G_DATA_WIDTH - 1 downto 0);
    x_valid      : out std_logic;
    x_ready      : in  std_logic
  );
end IQRD;
```

```vhdl
architecture rtl of IQRD is

  type T_IQRD_FSM is (S_IDLE, S_CONSUME, S_WAIT_X, S_OUT_VALID);
  signal sig_iqrd_state : T_IQRD_FSM := S_IDLE;
  -- counts how many valid weights have been extracted to know
   when final
  -- weight vector is completed
  signal sig_w_valid_cntr : integer range 0 to G_M - 1 := 0;

  signal sig_A_real : T_signed_3D(G_M - 1 downto 0)
                                 (G_N - 1 downto 0)
                                 (G_DATA_WIDTH - 1 downto 0);
  signal sig_A_imag : T_signed_3D(G_M - 1 downto 0)
                                 (G_N - 1 downto 0)
                                 (G_DATA_WIDTH - 1 downto 0);

  -- indexes A matrix, for each column, as it is consumed into the
     systolic array
  type T_2D_idx is array (integer range <>) of unsigned( F_clog2(
   G_M) - 1 downto 0 );
  signal sig_A_idx   : T_2D_idx(G_N - 1 downto 0);
  signal sig_A_valid : std_logic_vector(G_N - 1 downto 0);
  signal sig_A_ready : std_logic_vector(G_N - 1 downto 0);

  signal sig_b_real  : T_signed_2D(G_M - 1 downto 0)
                                  (G_DATA_WIDTH - 1 downto 0);
  signal sig_b_imag  : T_signed_2D(G_M - 1 downto 0)
                                  (G_DATA_WIDTH - 1 downto 0);
  -- indexes b vector as it is consumed into the systolic array
  signal sig_b_idx   : unsigned( F_clog2(G_M) - 1 downto 0 );
  signal sig_b_valid : std_logic;
  signal sig_b_ready : std_logic;

  -- cmplx samples & handshaking from row -> row (up/down)
  --   +1 row extra to map outputs to weight extract cells
  --   -1 column since most right/last IIC cell in systolic array
   is always fed null
  --   dim: (row index)(column index)
  signal sig_X_real, sig_X_imag   : T_signed_3D(G_N     downto 0)
                                               (G_N + 1 downto 0)
                                               (G_DATA_WIDTH - 1
   downto 0);
  signal sig_X_valid, sig_X_ready :    T_slv_2D(G_N     downto 0)
                                               (G_N + 1 downto 0);

  -- rotation angles (phi & theta) across rows & columns
  --   dim: (row index)(column index*)
  --         * column indexing ‘downto 1‘ to match other indexing
   in array
  signal sig_phi, sig_theta : T_unsigned_3D(G_N - 1 downto 0)
                                          (G_N + 2 downto 1)
                                          (31 downto 0);
  signal sig_angles_valid   :      T_slv_2D(G_N - 1 downto 0)
                                          (G_N + 2 downto 1);
  -- ‘ready‘ signal for angles only needed between BC & fist IC of
    each row
  signal sig_bc_ic_ready    : std_logic_vector(G_N - 1 downto 0);

  -- weight extract signals (indexing to match absolute column
   indexing)
  -- "G_N + 2" is extra column index for final output cell but is
   not consumed
  signal sig_w_a_real, sig_w_a_imag : T_signed_2D(2 to G_N + 2)
                                                (G_DATA_WIDTH - 1
    downto 0);
  signal sig_w_a_valid              : std_logic_vector(2 to G_N +
   2);
  signal sig_w_w_real, sig_w_w_imag : T_signed_2D(2 to G_N + 1)
                                                (G_DATA_WIDTH - 1
    downto 0);
  signal sig_w_w_valid              : std_logic_vector(2 to G_N +
   1);

  -- reg output vector X from weight extract cells
  signal sig_out_x_real, sig_out_x_imag : T_signed_2D(G_N - 1
   downto 0)
                                                    (G_DATA_WIDTH
    - 1 downto 0);

begin

  A_ready <= '1' when sig_iqrd_state = S_CONSUME else '0';
  b_ready <= '1' when sig_iqrd_state = S_CONSUME else '0';

  x_real  <= sig_out_x_real;
  x_imag  <= sig_out_x_imag;
  x_valid <= '1' when sig_iqrd_state = S_OUT_VALID else '0';


  UG_index_A_matrix_for_each_column: for col_idx in 0 to (G_N - 1)
    generate
    S_index_A_input_matrix: process(clk)
    begin
      if rising_edge(clk) then
        if (reset = '1') or (sig_iqrd_state = S_IDLE) then
          sig_A_idx  (col_idx) <= (others => '0');
          sig_A_valid(col_idx) <= '0';
        else
          if (sig_A_ready(col_idx) = '1') and (sig_A_valid(col_idx
) = '1') then
            -- if we're at the end of indexing the A matrix,
 samples are no longer valid
            if sig_A_idx(col_idx) = (G_M - 1) then
              sig_A_valid(col_idx) <= '0';
            else
              -- increment A-matrix index when systolic array
 consumes a sample
              sig_A_idx(col_idx) <= sig_A_idx(col_idx) + 1;
            end if;
          end if;

          if sig_iqrd_state = S_CONSUME then
            sig_A_valid(col_idx) <= '1';
          end if;
        end if;
      end if;
    end process S_index_A_input_matrix;
  end generate UG_index_A_matrix_for_each_column;

  S_index_b_input_vector: process(clk)
  begin
    if rising_edge(clk) then
      if (reset = '1') or (sig_iqrd_state = S_IDLE) then
        sig_b_idx   <= (others => '0');
        sig_b_valid <= '0';
      else
        if (sig_b_ready = '1') and (sig_b_valid = '1') then
          -- if we're at the end of indexing the b vector, samples
 are no longer valid
          if sig_b_idx = (G_M - 1) then
            sig_b_valid <= '0';
          else
            -- increment b-vector index when systolic array
 consumes a sample
            sig_b_idx <= sig_b_idx + 1;
          end if;
        end if;

        if sig_iqrd_state = S_CONSUME then
          sig_b_valid <= '1';
        end if;
      end if;
    end if;
  end process S_index_b_input_vector;

  UG_map_inputs_to_first_row: for col_idx in 0 to (G_N + 1) generate
    UG_input_from_matrix_A: if col_idx < G_N generate
      -- Indexes registed A matrix:        | index into M dim.  | samp column |
      sig_X_real (0)(col_idx) <= sig_A_real( to_integer(sig_A_idx(col_idx)) )(col_idx);
      sig_X_imag (0)(col_idx) <= sig_A_imag( to_integer(sig_A_idx(col_idx)) )(col_idx);
      sig_X_valid(0)(col_idx) <= sig_A_valid(col_idx);
         sig_A_ready(col_idx) <= sig_X_ready(0)(col_idx);
    end generate UG_input_from_matrix_A;

    UG_input_from_vector_b: if col_idx = G_N generate
      sig_X_real (0)(col_idx) <= sig_b_real( to_integer(sig_b_idx) );
      sig_X_imag (0)(col_idx) <= sig_b_imag( to_integer(sig_b_idx) );
      sig_X_valid(0)(col_idx) <= sig_b_valid;
      sig_b_ready             <= sig_X_ready(0)(col_idx);
    end generate UG_input_from_vector_b;

    UG_input_const_1: if col_idx = (G_N + 1) generate
      sig_X_real (0)(col_idx) <= to_signed( 1, G_DATA_WIDTH);
      sig_X_imag (0)(col_idx) <= to_signed( 0, G_DATA_WIDTH);
      sig_X_valid(0)(col_idx) <= '1';
      -- since giving constant 1 + 0j, d/c about ready signal, always valid
    end generate UG_input_const_1;

    -- right-most IIC cell fed NULL samples in below systolic array generate clauses
  end generate UG_map_inputs_to_first_row;

  -- Number of rows = size N
  UG_systolic_array_rows: for row_idx in 0 to (G_N - 1) generate
    -- Number of columns = size N + 3, where the first (left-most) processing
    --   element within a row is the BC
```

```vhdl
 UG_systolic_array_columns: for col_idx in 0 to (G_N + 2)
generate

   -- Boundary Cell is always left-most/first in column
   UG_left_BC: if col_idx = 0 generate
     U_BC: entity work.boundary_cell
       generic map (
         G_DATA_WIDTH => G_DATA_WIDTH,
         G_USE_LAMBDA => G_USE_LAMBDA
       )
       port map (
         clk          => clk,
         reset        => reset,
         CORDIC_scale => CORDIC_scale,
         lambda       => lambda,

         x_real       => sig_X_real (row_idx)(col_idx),
         x_imag       => sig_X_imag (row_idx)(col_idx),
         x_valid      => sig_X_valid(row_idx)(col_idx),
         x_ready      => sig_X_ready(row_idx)(col_idx),

         phi_out      =>        sig_phi  (row_idx)(col_idx+1),
         theta_out    =>        sig_theta(row_idx)(col_idx+1),
         bc_valid_out => sig_angles_valid(row_idx)(col_idx+1),
         ic_ready     =>  sig_bc_ic_ready(row_idx)
       );
   end generate UG_left_BC;

   UG_internal_cells: if (col_idx > 0) and (col_idx < (G_N + 2
- row_idx) ) generate
     -- the first IC needs the BC/IC ready handshaking signal
     UG_first_IC: if col_idx = 1 generate
       U_IC_BC: entity work.internal_cell
         generic map (
           G_DATA_WIDTH => G_DATA_WIDTH,
           G_USE_LAMBDA => G_USE_LAMBDA
         )
         port map (
           clk          => clk,
           reset        => reset,
           CORDIC_scale => CORDIC_scale,
           lambda       => lambda,

           xin_real     => sig_X_real (row_idx)(col_idx),
           xin_imag     => sig_X_imag (row_idx)(col_idx),
           xin_valid    => sig_X_valid(row_idx)(col_idx),
           xin_ready    => sig_X_ready(row_idx)(col_idx),

           phi_in       =>        sig_phi  (row_idx)(col_idx),
           theta_in     =>        sig_theta(row_idx)(col_idx),
           bc_valid_in  => sig_angles_valid(row_idx)(col_idx),
           ic_ready     => sig_bc_ic_ready(row_idx),

           -- X sample to next row, but shifted one column left
 (triangular array)
           xout_real    => sig_X_real (row_idx+1)(col_idx-1),
           xout_imag    => sig_X_imag (row_idx+1)(col_idx-1),
           xout_valid   => sig_X_valid(row_idx+1)(col_idx-1),
           xout_ready   => sig_X_ready(row_idx+1)(col_idx-1),

           phi_out      => sig_phi  (row_idx)(col_idx+1),
           theta_out    => sig_theta(row_idx)(col_idx+1),
           angles_valid => sig_angles_valid(row_idx)(col_idx+1)
         );
     end generate UG_first_IC;

     -- other (non-first) ICs are interconnected within a row
     UG_other_ICs: if col_idx /= 1 generate
       U_IC: entity work.internal_cell
         generic map (
           G_DATA_WIDTH => G_DATA_WIDTH,
           G_USE_LAMBDA => G_USE_LAMBDA
         )
         port map (
           clk          => clk,
           reset        => reset,
           CORDIC_scale => CORDIC_scale,
           lambda       => lambda,

           xin_real     => sig_X_real (row_idx)(col_idx),
           xin_imag     => sig_X_imag (row_idx)(col_idx),
           xin_valid    => sig_X_valid(row_idx)(col_idx),
           xin_ready    => sig_X_ready(row_idx)(col_idx),

           phi_in       =>        sig_phi  (row_idx)(col_idx),
           theta_in     =>        sig_theta(row_idx)(col_idx),
           bc_valid_in  => sig_angles_valid(row_idx)(col_idx),
           ic_ready     => open, -- not needed for other ICs

           -- X sample to next row, but shifted one column left
 (triangular array)
           xout_real    => sig_X_real (row_idx+1)(col_idx-1),
           xout_imag    => sig_X_imag (row_idx+1)(col_idx-1),
           xout_valid   => sig_X_valid(row_idx+1)(col_idx-1),
           xout_ready   => sig_X_ready(row_idx+1)(col_idx-1),

           phi_out      => sig_phi  (row_idx)(col_idx+1),
           theta_out    => sig_theta(row_idx)(col_idx+1),
           angles_valid => sig_angles_valid(row_idx)(col_idx+1)
         );
     end generate UG_other_ICs;
   end generate UG_internal_cells;

   UG_inverse_internal_cells: if (row_idx > 0) and
                                 (col_idx >= (G_N + 2 - row_idx)) and
                                 (col_idx <  (G_N + 2)) generate
     U_IIC: entity work.internal_cell
       generic map (
         G_DATA_WIDTH => G_DATA_WIDTH,
         G_USE_LAMBDA => G_USE_LAMBDA
       )
       port map (
         clk          => clk,
         reset        => reset,
         CORDIC_scale => CORDIC_scale,
         lambda       => inv_lambda,

         xin_real     => sig_X_real (row_idx)(col_idx),
         xin_imag     => sig_X_imag (row_idx)(col_idx),
         xin_valid    => sig_X_valid(row_idx)(col_idx),
         xin_ready    => sig_X_ready(row_idx)(col_idx),

         phi_in       =>        sig_phi  (row_idx)(col_idx),
         theta_in     =>        sig_theta(row_idx)(col_idx),
         bc_valid_in  => sig_angles_valid(row_idx)(col_idx),
         ic_ready     => open, -- not needed for other ICs

         -- X sample to next row, but shifted one column left (triangular array)
         xout_real    => sig_X_real (row_idx+1)(col_idx-1),
         xout_imag    => sig_X_imag (row_idx+1)(col_idx-1),
         xout_valid   => sig_X_valid(row_idx+1)(col_idx-1),
         xout_ready   => sig_X_ready(row_idx+1)(col_idx-1),

         phi_out      => sig_phi  (row_idx)(col_idx+1),
         theta_out    => sig_theta(row_idx)(col_idx+1),
          angles_valid => sig_angles_valid(row_idx)(col_idx+1)
        );
    end generate UG_inverse_internal_cells;

    -- Inverse Interal Cell fed by null-sample is always right-most/last in column
    UG_right_IIC: if col_idx = (G_N + 2) generate
      U_null_IIC: entity work.internal_cell
        generic map (
          G_DATA_WIDTH => G_DATA_WIDTH,
          G_USE_LAMBDA => G_USE_LAMBDA
        )
        port map (
          clk          => clk,
          reset        => reset,
          CORDIC_scale => CORDIC_scale,
          lambda       => inv_lambda,

          xin_real     => to_signed( 0, G_DATA_WIDTH),
          xin_imag     => to_signed( 0, G_DATA_WIDTH),
          xin_valid    => '1',  -- always NULL input
          xin_ready    => open, -- d/c

          phi_in       =>        sig_phi  (row_idx)(col_idx),
          theta_in     =>        sig_theta(row_idx)(col_idx),
          bc_valid_in  => sig_angles_valid(row_idx)(col_idx),
          ic_ready     => open, -- ready signaling not needed here

          xout_real    => sig_X_real (row_idx+1)(col_idx-1),
          xout_imag    => sig_X_imag (row_idx+1)(col_idx-1),
          xout_valid   => sig_X_valid(row_idx+1)(col_idx-1),
          xout_ready   => sig_X_ready(row_idx+1)(col_idx-1),

          -- last cell, these angles not needed
          phi_out      => open,
          theta_out    => open,
          angles_valid => open
        );
    end generate UG_right_IIC;

  end generate UG_systolic_array_columns;
end generate UG_systolic_array_rows;


```

```
--// Start Final Extraction of X Vector //////////////////////////////////////
-- first IC from last row in systolic array feeds ‘a‘ sample to weight extract cells
-- and since these cells don’t have a ‘ready‘ signal for ‘a‘, assert it here
sig_X_ready(G_N)(0) <= ’1’;
-- same thing for next IC, always ‘ready‘ so the IC is not held up
sig_X_ready(G_N)(1) <= ’1’;

-- map last row, first IC data output -> weight extract ‘a‘ input row
sig_w_a_real (2) <= sig_X_real (G_N)(0);
sig_w_a_imag (2) <= sig_X_imag (G_N)(0);
sig_w_a_valid(2) <= sig_X_valid(G_N)(0);

-- first two output column’s (0 & 1) samples can be multiplied togeter to form
-- error function e(k)
UG_output_vector: for col_idx in 2 to (G_N + 1) generate
  U_calc_x: entity work.weight_extract_cell
    generic map (
      G_DATA_WIDTH => G_DATA_WIDTH
    )
    port map (
      clk          => clk,
      reset        => reset,

      ain_real     => sig_w_a_real (col_idx),
      ain_imag     => sig_w_a_imag (col_idx),
      ain_valid    => sig_w_a_valid(col_idx),

      aout_real    => sig_w_a_real (col_idx+1),
      aout_imag    => sig_w_a_imag (col_idx+1),
      aout_valid   => sig_w_a_valid(col_idx+1),

      -- use final/output row from IC/IICs in systolic array
      b_real       => sig_X_real (G_N)(col_idx),
      b_imag       => sig_X_imag (G_N)(col_idx),
      b_valid      => sig_X_valid(G_N)(col_idx),
      b_ready      => sig_X_ready(G_N)(col_idx),

      w_real       => sig_w_w_real (col_idx),
      w_imag       => sig_w_w_imag (col_idx),
      w_valid      => sig_w_w_valid(col_idx),
      w_ready       => '1' -- for now, final ready from
 downstream is handled in FSM
    );

  S_reg_weight_outputs_to_x: process(clk)
  begin
    if rising_edge(clk) then
      if sig_w_w_valid(col_idx) then
        sig_out_x_real(col_idx-2) <= sig_w_w_real (col_idx);
        sig_out_x_imag(col_idx-2) <= sig_w_w_imag (col_idx);
      end if;
    end if;
  end process S_reg_weight_outputs_to_x;
end generate UG_output_vector;
--// End Final Extraction of X Vector
 //////////////////////////////////////


--// Start FSM that coordinates array timing
 ///////////////////////////////
S_main_FSM: process(clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      sig_w_valid_cntr <= 0;
      sig_iqrd_state   <= S_IDLE;
    else
      case sig_iqrd_state is
        when S_IDLE =>
          if (A_valid = '1') and (b_valid = '1') then
            sig_A_real     <= A_real;
            sig_A_imag     <= A_imag;
            sig_b_real     <= b_real;
            sig_b_imag     <= b_imag;
            sig_iqrd_state <= S_CONSUME;
          end if;

        when S_CONSUME =>
          sig_iqrd_state   <= S_WAIT_X;

        when S_WAIT_X =>
          -- when last/right-most weight is valid, count up
          -- once we've hit G_M - 1, we know this is the last
 weight and
          -- can move to show this as final x vector output
          if sig_w_w_valid(G_N+1) = '1' then
            if sig_w_valid_cntr >= (G_M - 1) then
              sig_w_valid_cntr <= 0;
              sig_iqrd_state    <= S_OUT_VALID;
            else
              sig_w_valid_cntr <= sig_w_valid_cntr + 1;
            end if;
          end if;

        when S_OUT_VALID =>
          if x_ready = '1' then
            sig_iqrd_state <= S_IDLE;
          end if;

        when others => sig_iqrd_state <= S_IDLE;
      end case;
    end if;
  end if;
end process S_main_FSM;
--// End FSM that coordinates array timing
  /////////////////////////////////

end rtl;
```

Listing 3.7: Top-Level IQRD Design

3.2.4 Application of Beamforming Weights

As shown in previous chapters, the application of the complex, adaptive beamforming weights is simply a *dot product* of the column vector of each spatial sample at time k ($\mathbf{x}(k)$) and the complex conjugate of the weight vector $\hat{\mathbf{w}}$, as $\mathbf{y}(k) = \hat{\mathbf{w}}^H\mathbf{x}(k)$.

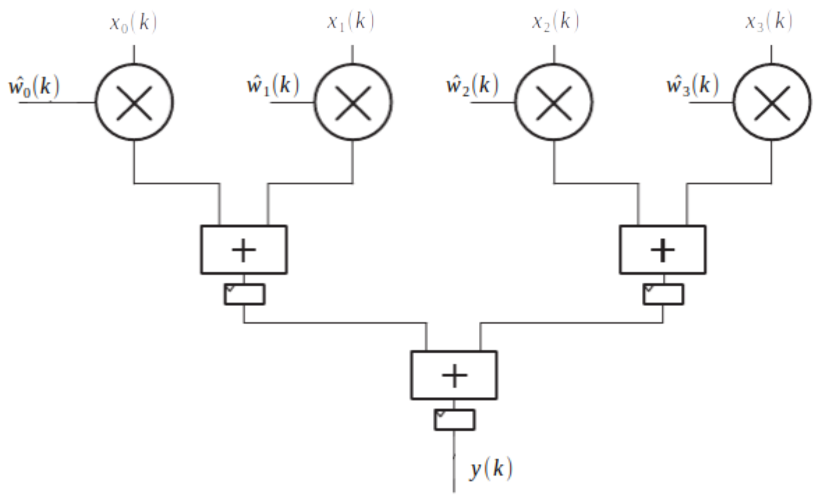


Figure 3.11: Application of Beamforming Weights, $N = 4$

The dot-product VHDL design developed below utilizes a recursive adder tree implementation to adapt to varying vector lengths through generics:

```
-- Computes dot-product of two complex, signed input vectors (uses
    parallel adder tree)
-- If G_CONJ is TRUE, the complex transpose product a^{H}b is
   computed, else does a^{T}b
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
library work;
  use work.util_pkg.all;

entity dot_product_cmplx is
  generic (
    G_AWIDTH   : natural := 16;    -- input vector bitwidth
    G_BWIDTH   : natural := 16;    -- input vector bitwidth
    G_VEC_LEN  : natural :=  8;    -- number of input samples in
   each vector
    G_CONJ     : boolean := true   -- if true, do complex conjugate
   on input vector a
  );
  port (
    clk           : in  std_logic;
    reset         : in  std_logic := '0'; -- (optional) sync reset
   for *valid's
    -- input data valid across input row vectors
    din_valid    : in  std_logic := '1';
    din_a_real   : in  T_slv_2D(G_VEC_LEN - 1 downto 0)(G_AWIDTH -
     1 downto 0);
    din_a_imag   : in  T_slv_2D(G_VEC_LEN - 1 downto 0)(G_AWIDTH -
     1 downto 0);
    din_b_real   : in  T_slv_2D(G_VEC_LEN - 1 downto 0)(G_BWIDTH -
     1 downto 0);
    din_b_imag   : in  T_slv_2D(G_VEC_LEN - 1 downto 0)(G_BWIDTH -
     1 downto 0);

    dout_valid   : out std_logic;
    dout_real    : out std_logic_vector(F_clog2(G_VEC_LEN) +
   G_AWIDTH + G_BWIDTH downto 0);
    dout_imag    : out std_logic_vector(F_clog2(G_VEC_LEN) +
   G_AWIDTH + G_BWIDTH downto 0)
  );
end dot_product_cmplx;

architecture rtl of dot_product_cmplx is

  component adder_tree is
    generic (
      G_DATA_WIDTH : natural := 16; -- sample bitwidth
      G_NUM_INPUTS : natural :=  8  -- number of input samples in
   vector
    );
    port (
      clk          : in  std_logic;
      reset        : in  std_logic := '0'; -- (optional) sync
   reset for *valid's
      -- input data valid across input row vector
      din_valid    : in  std_logic := '1';
      -- NOTE: input samples not registered
      din          : in  T_slv_2D(G_NUM_INPUTS - 1 downto 0)(
   G_DATA_WIDTH - 1 downto 0);

      dout_valid   : out std_logic;
      dout         : out std_logic_vector(F_clog2(G_NUM_INPUTS) +
   G_DATA_WIDTH - 1 downto 0)
    );
  end component adder_tree;

  component complex_multiply_mult4 is
    generic (
      G_AWIDTH : natural := 16;    -- size of 1st input of
   multiplier
```

```
    G_BWIDTH : natural := 18;     -- size of 2nd input of multiplier
    G_CONJ_A : boolean := false; -- take complex conjugate of arg A
    G_CONJ_B : boolean := false  -- take complex conjugate of arg B
  );
  port (
    clk      : in  std_logic;
    reset    : in  std_logic := ’0’; -- (optional) sync reset for *valid’s
    ab_valid : in  std_logic; -- A & B complex input data valid
    ar       : in  signed(G_AWIDTH - 1 downto 0); -- 1st input’s real part
    ai       : in  signed(G_AWIDTH - 1 downto 0); -- 1st input’s imaginary part
    br       : in  signed(G_BWIDTH - 1 downto 0); -- 2nd input’s real part
    bi       : in  signed(G_BWIDTH - 1 downto 0); -- 2nd input’s imaginary part
    p_valid  : out std_logic; -- Product complex output data valid
    pr       : out signed(G_AWIDTH + G_BWIDTH downto 0); -- real part of output
    pi       : out signed(G_AWIDTH + G_BWIDTH downto 0)  -- imaginary part of output
  );
end component complex_multiply_mult4;

-- registered product outputs -> adder tree
signal sig_product_real : T_signed_2D(G_VEC_LEN - 1 downto 0)(G_AWIDTH + G_BWIDTH downto 0)
                        := (others => (others => ’0’));
signal sig_product_imag : T_signed_2D(G_VEC_LEN - 1 downto 0)(G_AWIDTH + G_BWIDTH downto 0)
                        := (others => (others => ’0’));

signal sig_product_slv_real : T_slv_2D(G_VEC_LEN - 1 downto 0)(G_AWIDTH + G_BWIDTH downto 0)
                            := (others => (others => ’0’));
signal sig_product_slv_imag : T_slv_2D(G_VEC_LEN - 1 downto 0)(G_AWIDTH + G_BWIDTH downto 0)
                            := (others => (others => ’0’));

signal sig_product_valid : std_logic := ’0’;
signal dout_valid_real   : std_logic := ’0’;
```

```vhdl
  signal dout_valid_imag   : std_logic := '0';

begin

  -- NOTE: since initial complex products are enforced to be valid contiguously
  --       the output valid need only come from one of the adder tress since they
  --       have equal pipeline delay
  dout_valid <= dout_valid_real; -- and dout_valid_imag;

  UG_index_input_vectors: for i in 0 to G_VEC_LEN - 1 generate
    U_cmplx_mult: complex_multiply_mult4
      generic map (
        G_AWIDTH => G_AWIDTH,
        G_BWIDTH => G_BWIDTH,
        G_CONJ_A => G_CONJ,
        G_CONJ_B => false
      )
      port map (
        clk      => clk,
        reset    => reset,
        ab_valid => din_valid,
        ar       => signed( din_a_real(i) ),
        ai       => signed( din_a_imag(i) ),
        br       => signed( din_b_real(i) ),
        bi       => signed( din_b_imag(i) ),
        p_valid  => sig_product_valid,
        pr       => sig_product_real(i),
        pi       => sig_product_imag(i)
      );

      sig_product_slv_real(i) <= std_logic_vector( sig_product_real(i) );
      sig_product_slv_imag(i) <= std_logic_vector( sig_product_imag(i) );
  end generate UG_index_input_vectors;

  U_adder_tree_real: adder_tree
    generic map (
      G_DATA_WIDTH => G_AWIDTH + G_BWIDTH + 1,
      G_NUM_INPUTS => G_VEC_LEN
    )
    port map (
      clk          => clk,
      reset        => reset,
      din_valid    => sig_product_valid,
      din          => sig_product_slv_real,
      dout_valid   => dout_valid_real,
      dout         => dout_real
    );

  U_adder_tree_imag: adder_tree
    generic map (
      G_DATA_WIDTH => G_AWIDTH + G_BWIDTH + 1,
      G_NUM_INPUTS => G_VEC_LEN
    )
    port map (
      clk          => clk,
      reset        => reset,
      din_valid    => sig_product_valid,
      din          => sig_product_slv_imag,
      dout_valid   => dout_valid_imag,
      dout         => dout_imag
    );

end architecture rtl;
```

Listing 3.8: Beamforming Weights- Dot Product

3.3 IQRD Performance

The parameterize-able IQRD core has differing amounts of processing latency and FPGA resource utilization depending on the size of the matrix inversion supported. Since this is directly driven by the number of spatial channels to support, table 3.1 shows a comparison of latency and resource utilization based on a few common channel count values, N, using the implemented design in a 100MHz synchronous clock domain, and using a pre-computed covariance matrix (since the snapshot length K may dominate latency in applications where this value is high):

N	Latency (μs)	FF	LUT	DSP48	BRAM
3	4.80	3889 (27.6%)	3901 (54.9%)	90 (25.0%)	0 (0.0%)
4	6.34	63657 (45.1%)	64149 (90.4%)	144 (40.0%)	0 (0.0%)
8	12.50	231199 (164%)	252446 (356%)	272 (75.6%)	0 (0.0%)
16	24.82	895377 (635%)	1040992 (1466%)	144 (40.0%)	0 (0.0%)

Table 3.1: IQRD Performance: Latency and Resource Utilization vs Channel Count for XCZU3EG FPGA

The values in Table 3.1 were generated post-synthesis using Xilinx Vivado 2020.1, targeting a Xilinx XCZU3EG Zynq UltraScale+ FPGA. Note that differing FPGA architectures/devices, synthesis settings, and effects of placement and routing (PaR) can alter the resource utilization of the design, however for this architecture, the differences will not be much. Interestingly, the $N = 16$ case showed a massive increase in slice utilization- generally related to flip-flop (FF) and look-up-table (LUT) logic building blocks, though depends on the exact FPGA architecture- but a decrease in DSP resources; the exact cause is not known, however the vendor tools might try to rearrange logic to utilize less DSP resources since this device does not have many available [17].

Even though this part is not very large, it is a modern FPGA SoC which, due to its size and power, would be of common size to an edge deployed device. Thus, the main takeaway from this exploration of existing architectures is that fully FPGA-accelerated adaptive beamforming using QRD systolic arrays may not fit in some devices, as seen above where the current part can really only support up to a 4-channel implementation. As discussed before, there are other FPGA architectures which utilize folded arrays or weight-flushing to save on resources, however they trade these resource savings for processing latency, in which case, it may be faster to just compute the covariance matrix in FPGA logic and then transfer the matrix to an embedded processor (as is found in the XCZU3EG SoC device here which has 4x ARM A53 cores locally attached) to do the matrix inversion and back substitution to get the adaptive weights (which can then be applied to incoming data streams in the FPGA). It should be noted that the CORDIC IQRD implementation used here could be slightly improved by replacing the CORDIC Boundary and Internal Cells with direct multiplier/LUT equations as in [2], however this will cause an increase in DSP and BRAM/LUT utilization.

References

[1] C. R. Ward, P. J. Hargrave, and J. G. McWhirter, "A novel algorithm and architecture for adaptive digital beamforming," *IEEE Transactions On Antennas And Propagation*, vol. AP-34, no. 3, 1986.

[2] M. Karkooti and J. R. Cavallaro, "Fpga implementation of matrix inversion using qrd-rls algorithm," *Conference Record of the Thirty-Ninth Asilomar Conference onSignals, Systems and Computers*, 2005.

[3] S. Haykin, *Adaptive Filter Theory*, 5th ed. pearson, 2014, ISBN: 978-0-132-67145-3.

[4] "Adaptive beamforming for radar: Floating-point qrd+wbs in an fpga," Xilinx, Tech. Rep. WP452 (v1.0), 2014.

[5] "Application note 506: Qr matrix decomposition," Altera, Tech. Rep. 2.0, 2008. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an506.pdf.

[6] C. Rader, "Vlsi systolic arrays for adaptive nulling," *IEEE Signal Processing Magazine*, 1996.

[7] "Implementation of cordic-based qrd-rls algorithm on altera stratix fpga with embedded nios soft processor technology," Altera, Tech. Rep. 1.0, 2004.

[8] J. McWhirter and T. Shepherd, "Systolic array processor for mvdr beamforming," *IEEE Proceedings*, vol. 136, no. 2, 1989.

[9] C. Dick, F. Harris, M. Pajic, and D. Vuletic, "Real-time qrd-based beamforming on an fpga platform," in *2006 Fortieth Asilomar Conference on Signals, Systems and Computers*, 2006, pp. 1200–1204. DOI: 10.1109/ACSSC.2006.354945.

[10] Q. Gao, L. Crockett, and R. Stewart, "Coarse angle rotation mode cordic based single processing element qr-rls processor," in *2009 17th European Signal Processing Conference*, 2009, pp. 1279–1283.

[11] R. Irfan, H. ur Rasheed, and W. A. Toor, “Fpga-based low latency inverse qrd architecture for adaptive beamforming in phased array radars,” *Radioengineering*, vol. 26, no. 3, 2017.

[12] S. T. Alexander and A. L. Ghirnikar, “A method for recursive least squares filtering based upon an inverse qr decomposition,” *IEEE Transactions on Signal Processing*, vol. 41, no. 1, 1993.

[13] A. L. Ghirnikar, S. T. Alexander, and R. Plemmons, “A parallel implementation of the inverse qr adaptive filter,” *Computer Elect. Engineering*, vol. 18, no. 3, 1991.

[14] M. Harteneck, R. Stewart, J. McWhirter, and I. Proudler, “Algorithmic engineering applied to the qr-rls adaptive algorithm,”

[15] S.-J. Chern and C.-Y. Chang, “Adaptive linearly constrained inverse qrd-rls beamforming algorithm for moving jammers suppression,” *IEEE Transactions on Antennas and Propagation*, vol. 50, no. 8, 2002.

[16] R. Andraka, “A survey of cordic algorithms for fpga based computers,” *ACM*, 1998.

[17] “Zynq ultrascale+ mpsoc product tables and product selection guide,” Xilinx, Tech. Rep. 2.5.1, 2021. [Online]. Available: https://www.xilinx.com/support/documentation/selection-guides/zynq-ultrascale-plus-product-selection-guide.pdf.

Chapter 4

Machine Learning Applied to Adaptive Beamforming

Now that we have a baseline, optimized implementation for the closed-form solution of generating adaptive beamforming weights using IQRD-RLS, we can explore the potential of using advances in the field of Machine Learning to possibly generate these adaptive weights in a more efficient, or more performant, manner. We will first give a background on the current body of knowledge in Machine Learning, specifically in the subset class of *Deep Learning* where a *model* uses multiple *layers* to progressively extract, and infer, features from a given set of data inputs [1].

After a deep learning model for the adaptive beamforming case is developed, we will walk through the FPGA-based implementation of the model to show practical deployment of Machine Learning at the edge.

The developed approach takes the covariance matrix and steering vector as a two-dimensional input layer and generates the adaptive weights directly at the output of the *Convolutional Neural Network (CNN)*. To the writer's understanding, this approach, and application, is fairly unique. The current body of knowledge of Deep Learning applied to RF systems is relatively small compared to the large amount of works applied to optimizations for computer vision or classification applications. Furthermore, much of the example literature and code tools for CNNs are skewed towards classification taks, in which the output of the CNN is one or more *confidence* values that a certain input matches a pre-defined, discrete *label*. This can even be seen in RF signal classification research works, as in [2] and [3] where CNNs are trained to classify input signals as a certain modulation type. Instead, the adaptive beamforming case, in which we desire to have the CNN give us weight values directly, we instead look to have a system essentially perform *multi-output regression* [1]. As well, we do not have discrete labels for training, but rather a numerical optimization problem (maximizing SINR) for each training scenario, thus a custom training methodology also had to be developed.

One work by Lin and Zhu [4] took a similar approach to this work by training a *neural network (NN)* for mmWave MIMO systems, however the phase shift control was assumed to be part of a hybrid beamforming system- where discrete analog phase shifters perform the shift in discrete, quantized steps- as well assumed specific channel state information (CSI) was available for training and inference. The closest approach to our adaptive beamforming application is in [5] where the authors used a CNN for generating adaptive beamforming weights for an ultrasound imaging application; the input pre-processing and network layers differed from our implementation, but the authors' results of 96.4% processing efficiency provided the impetus to further explore the application of Adaptive Beamforming CNNs in resource-constrained edge devices, specifically in FPGAs [5].

4.1 Deep Learning Background

Deep Learning, and largely the broader field of Machine Learning, is extremely dense and pulls from many mathematical and scientific disciplines. Moreover, the field is growing quite literally everyday due to the popularity of implementation in a variety of application areas, so the "state-of-the-art" in Machine Learning is constantly changing. As such, the reader is suggested to explore comprehensive texts, as in [1], to gain a deeper understanding of Machine Learning as a whole, as well as some of the background math behind these models. A comprehensive explanation here is beyond the scope of this specific research work.

4.1.1 CNN Architecture

A popular architecture for deep neural networks is that of the Convolutional Neural Network (CNN). The premise of the CNN is that it models the interaction of neurons within the human brain [1], where some set of input data, $\mathbf{x}$, is *convolved* with a set of weights- also known as a *kernel*- to produce an output *feature map* [1].

The discrete convolution operation that forms the basis of CNNs can be seen in Equation 4.1 [1].

$$s(t) = (x * w)(t) = \sum_{a==\infty}^{\infty} x(a)w(t-a) \tag{4.1}$$

For a two-dimensional, $m \times n$ input $\mathbf{X}$, a two-dimensional kernel $\mathbf{K}$ can also be used to perform 2D convolution, as in Equation 4.2 [1]:

$$S(i,j) = (X * K)(i,j) = \sum_m \sum_n X(i-m, j-n)K(m,n) \tag{4.2}$$

This convolution operation at the macro level starts with the individual neuron- also called *perceptrons* [6]- functionality. In a standard, one-dimensional perceptron structure, a certain number of inputs are multiplied by a corresponding number of weights, and then summed together, in effect performing a dot-product of the two. After which, an *activation function* is performed on the output to mimic the activity filtering of real neurons, as well as aid in network training [1], [6]. This perceptron can be seen in Figure 4.1.

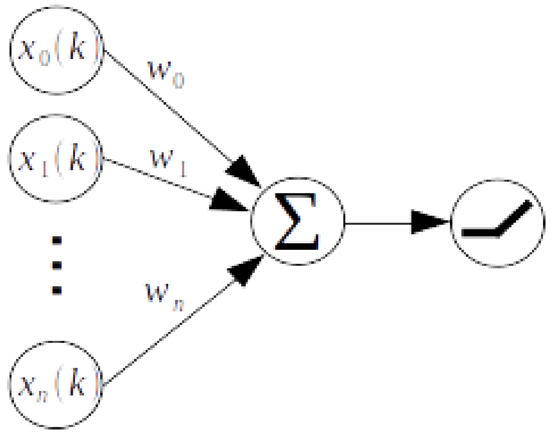


Figure 4.1: Perceptron Building Block in Neural Networks

The popular *Rectified Linear Unit (ReLU)* activation function was used for this research work; ReLU as an activation function not only mimics the biological neuron, but has proved to enable better training of deep neural networks than previous logistic sigmoid or hyperbolic tangent activation functions of the past [1]. The ReLU is so called due to the fact that it returns 0 for negative input values, and directly returns the input value for positive values, similar to electrical rectification circuits [1], as shown in Equation 4.3.

$$ReLU(x) = x^{+} = \max(0, x) \tag{4.3}$$

The interconnection of a number of inputs to a number of neurons, where each neuron is fed all inputs, is known as a *fully connected (FC) layer* in CNNs [1], [7]. The fully connected layer has the property that a change in one input value has a corresponding effect on all output values of the next layer, as shown in Figure 4.2. A sequential combination of multiple layers creates a *network*, and is the basis of the overall CNN.

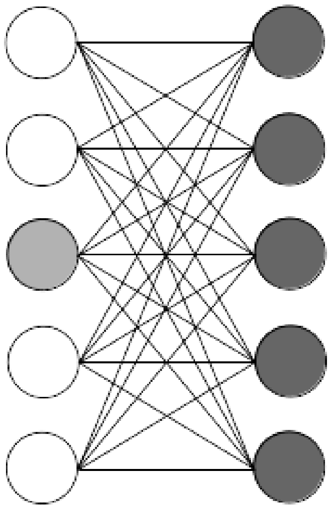


Figure 4.2: Fully Connected Layer

In Figure 4.2, the nodes on the left side represent the inputs, and the nodes on the right side represent the perceptions, each with a unique set of weights and an activation function. If this is the first layer of a network, the inputs are the data directly fed into the CNN. Correspondingly if this is the last layer of the network, the output of the perceptrons are the final values computed by the CNN.

The fully connected layer depicted here is indicative of a one-dimensional layer, however layers may be made of any arbitrary dimension, for instance a 2D layer which performs 2D convolution on input data, as shown in Equation 4.2. In this case, most software tools used to develop neural networks, like TensorFlow [8], operate generally on *tensors*, which is broad term for a matrix of variable dimension [1]. There are other layer types as well, which can perform different operations other than convolution, however for brevity a further catalog of layer types is not covered in this work. Again, the reader is encouraged to view comprehensive Machine Learning texts like [1] for more information on different layers and their applicability to a certain neural network.

4.1.2 CNN Model Development in TensorFlow

Given the background on CNNs, we can now start to develop a CNN specifically designed and trained for computing adaptive beamforming weights using the open-source software tool TensorFlow [8]. The decision to use TensorFlow over other tools was simply due its wide popularity- which gives it a broad support base- and its constant development and updates.

Starting with the input layer of the CNN model, the decision was made to use the generated covariance matrix as a preprocessing step prior to the CNN, rather than some collection of unprocessed sampled data. Besides it being relatively easy to implement in FPGA logic and light on resources- as shown in the previous chapter-, if input preprocessing was not used, the input layer would be very large to sample a sufficient time-series window across all channels in order to extract meaningful features in the CNN. If the time-series approach was chosen, a different NN structure would be used, such as a *Long Short-Term Memory (LSTM)* network, which is a class of *Recurrent Neural Networks (RNNs)* that has internal feedback connections to support inference over time [1]. This is compared to *feedforward* networks, like CNNs, which have no memory and infer a result based only on the current input sample set, thus being time invariant. The approach to pre-process the input samples before input to the CNN is similar to the approach of [3] and [2] where they use a Short-time Fourier Transform to create an image retaining the time and frequency properties of a set of input sample data. In this case, we only care to keep the spatial properties of the input sample data.

Since we needed to provide the CNN a way to calculate adaptive weights while not nulling the intended SOI, the steering vector was included as part of the CNN's input dataset, similar to the previous adaptive beamforming algorithms. To accomplish this, the steering vector was appended to the input covariance matrix as an added row to the input layer. As well, since the covariance matrix is complex-valued, and most CNN tools assume real-valued data and operations, we define the input layer as 2D in an approach similar to

CNNs designed for image data input; similar to how computer vision CNNs utilize an added dimension to represent the three color channels of an image (red, green and blue), we add a dimension to represent the two "channels" of our complex covariance matrix: one for the real, and one for the imaginary part. Thus, given a targeted implementation support N spatial channels, we create an input layer tensor of dimension $N + 1 \times N \times 2$.

As stated previously, since we are looking for the CNN to compute the adaptive beamforming weights directly, our output layer is designed to be of size $2N$; the reason for this size is that in the developed CNN model, one of the *hidden layers* (a layer within a deep neural network) flattens the 2D fully connected layer to a 1D representation, and as such, the output is designed to give the real part of the adaptive beamforming weights in the lower half of the output vector, and the imaginary part in the upper half.

Through experimentation, a CNN structure was chosen with one 2D convolutional layer, a flattened hidden layer, and a final fully connected layer. The approach looked to balance performance and number of filter parameters (driven by dimensionality and number of hidden layers) since the goal was to implement a CNN directly in FPGA fabric. This structure is shown in Figure 4.3.

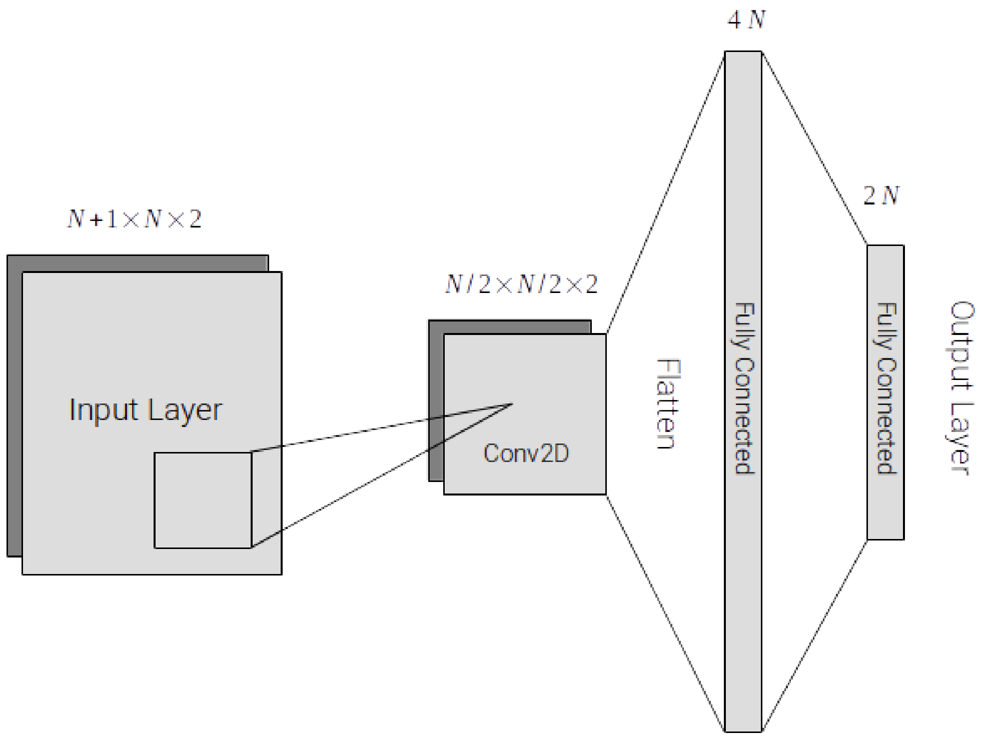


Figure 4.3: Adaptive Beamforming CNN Structure

After CNN model definition, we look to *train* the model, which is the process of deriving weights for each layer based on a desired output function [1], [7]. To accomplish this, we created a synthetic test data set in Python which generated a large amount of test scenarios to feed the model during training. Each test data scenario was generated to vary the number of interference sources- up to the $N-1$ theoretical limit for nulling-, interference and SOI incidence angles, interference and SOI center frequencies, and the desired and SOI signal-to-noise ratios (SNRs). The large number of test scenarios, and the large number of varied parameters, attempt to avoid the issue of *overfitting* during model training, in which the model essentially memorizes test datasets instead of creating real associations to input features [1]; this problem is further exacerbated by very deep, or high dimensional, neural networks with relatively small training datasets, or datasets that don't fully represent the full space of input features.

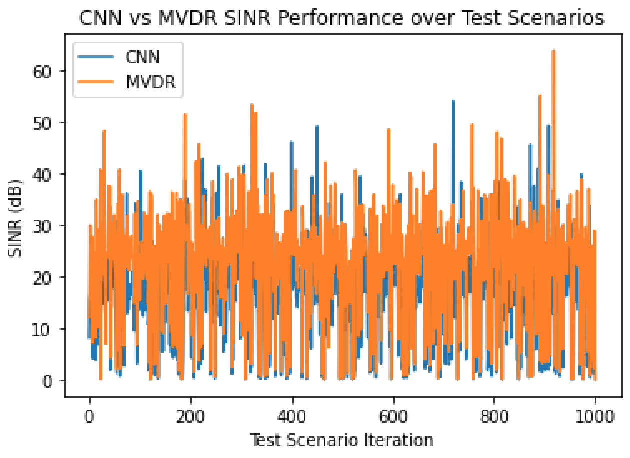
In the approach of *supervised learning*, we feed the network test input data, where each scenario has an associated *label*, and check the output of the network for how close it inferred towards the expected label. A label can be any set of values- as in multi-output regression where we look to match a set of input values with a set of expected output vales-, or something like an enumerated integer value- as in the case for classification tasks which look to decide an input to a discrete set of output labels. For tasks like regression and classification, the statistical error between the set of inferred (Y) and expected ($\hat{Y}$) output values can be computed using methods such as *Mean Squared Error (MSE)* [1], [7]:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{4.4}$$

The specific calculation of error in Machine Learning terms is the *loss function*, and the job of the training tool is to derive neural network weights which minimizes loss [1], [7]. Finding the minimal loss is done through *stochastic gradient descent* and *backpropagation (a.k.a. "backprop")*, which essentially finds the analytic derivative of the loss function to find a local minima [1], [7].

In this work, we used MSE to calculate the loss during training between the inferred weights during a training batch, and the ideal set of weights derived from the standard MVDR calculation process. Ideally, we looked to actually create a custom loss function based on calculating $1/SINR$, such that the tools could possibly derive a better solution, however TensorFlow does not currently support complex-valued loss functions, which were necessary for the SINR computation.

As such, TensorFlow was able to at least perform multi-output regression with the ideal, MVDR results for comparison and loss calcuations, but the relative performance suffered slightly, with the average SINR difference being almost 5 dB worse from the CNN implementation versus MVDR, as shown in Figure 4.4:



```
Average SINR from CNN: 17.76 dB
Average SINR from MVDR: 22.67 dB
Average SINR performance of CNN compared to the MVDR approach: -4.91 dB
```

Figure 4.4: CNN vs MVDR SINR over Validation Test Dataset

Future developments of TensorFlow, or perhaps even a different tool set altogether which supports complex-valued core operations, should be pursued for future work to better develop a neural network for adaptive beamforming. For this initial research work however, where we will next implement this CNN in FPGA logic, this slight under-performance will suffice to prove the concept.

4.2 FPGA Implementation of CNN

Now that we have a working CNN trained to perform adaptive beamforming, we can look to port the layers, and structure, to FPGA logic. However, before porting the CNN, we should take some steps to optimize the model for embedded targets like FPGAs. An extension of the TensorFlow distribution called TensorFlow Lite contains utilities to perform these optimization steps while aiming to maintain model accuracy [9].

First, we can perform an operation known as *pruning* to remove (zero-out) nodes within a model that have little effect to the model's overall accuracy [9]. At a basic level, pruning can be thought of as a process that removes existing connections between layers where the weights are at, or nearly, zero; in this sense creating a sparsely connected layer where those connections simply don't exist can reduce model size and processing latency, while theoretically having little effect on overall model operation. However, this should be verified through testing and usually pruning is best done in the latter, less critical layers of a neural net [9].

Even more critical to efficient implementation in hardware accelerators is the process of *quantization* [9]. As the name implies, quantization is a process of converting model weights from a full-precision numeric representation- which is often single precision, 32-bit floating point- to a smaller numeric representation such as 16-bit floating point (FP16) or integer data types. While floating point operations are supported in most major FPGA vendors by now, the coding and implementation is often vendor-specific- through vendor supplied Intellectual Property (IP) cores- or best done through High-Level

Synthesis (HLS) toolchains. As such, integer quantization is preferred for our vendor-neutral FPGA implementation goal, as well integer multiplication is cheaper in FPGA DSP fabric resources than the equivalent floating point operations. It further makes sense to use integer weights given our input covariance matrix is assumed to be in fixed point for this model. To support this, TensorFlow Lite supports a post-training integer quantization model where the same model we trained with floating point weights can be directly converted to quantized weights and the accuracy checked within TensorFlow before final deployment [9]. Furthermore, a relatively new (at the time of this writing) mode of quantization was used for this research called "16x8 quantization mode" [9]; in this mode weights are quantized to 8-bit integer values while activations are converted to larger 16-bit values. This gives us better model dynamic range than the traditional, full 8-bit quantization modes where all model parameters were quantized down to 8-bit integer sizes, while still drastically reducing model resource utilization [9]. Quantization of a set of floating-point values, b, to a signed, n bit integer data type can be found by multiplying each value by a scaling coefficient k, as shown in the Equation 4.5 below, and then rounding to the nearest integer:

$$k = \frac{2^{n-1}}{\max\limits_{x \in b} |x|} \tag{4.5}$$

Given the set of quantized weights, we can start to build up the neural network; starting with the basic perceptron building block, we created a VHDL component that matches the dot-product operation shown in Figure 4.1. However, instead of taking the direct form of the perceptron and calculating all connection weight products and summing in parallel (which could use our previous dot-product component), we can attain great DSP resource savings by using a single multiplier in the perceptron, and then iterating through each connection and multiplying by the associated weight stored in local Read-only memory (ROM). Given our CNN with over 2000 unique weights for an $N = 8$ configuration, and an embedded device with only 360 multipliers- as in the XCZU3EG- this DSP resource savings is critical. The amount of memory space the weights take up is minimal (since we have 8-bit quantized weights) and, for overall latency, fully-connected layers with small to medium dimensions will not see much of a performance hit. Given a Python export of quantized filter weights (see Python code in Appendix), we create per-node weight files which we can point to with this component to load a node's associated weights per input connection.

```vhdl
-- Implements a perceptron with N-connections
-- For quantizations like 16b data w/8b weights, we can simply
   keep the 24b product and accumulate
--   to something like a 32b/48b value (large adders are cheap
   nowadays) and then shift at very end
--   to keep relative precision
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
  use std.textio.all;
library work;
  use work.util_pkg.all;

entity perceptron is
  generic (
    G_DATA_WIDTH   : integer := 16;
    G_WEIGHT_WIDTH : integer :=  8;
    -- number of connections from previous layer (== # of weights)
    G_NUM_CONNECT  : integer := 32;
    -- accumulator register word size
    G_ACCUM_WIDTH  : integer := 24;
    G_WEIGHT_PATH  : string  := "../scripts/coef.txt"
  );
  port (
    clk             : in  std_logic;
    reset           : in  std_logic;

    din_valid       : in  std_logic;
    din             : in  T_signed_2D(G_NUM_CONNECT - 1 downto 0)(
   G_DATA_WIDTH - 1 downto 0);

    dout_valid      : out std_logic;
    dout            : out signed(G_DATA_WIDTH - 1 downto 0)
  );
end entity perceptron;

architecture rtl of perceptron is

  type T_percep_fsm is (S_IDLE,
                        S_ITER_MAC,
                        S_FINAL_ACC,
                        S_OUT_VALID);
  signal sig_percep_state : T_percep_fsm := S_IDLE;

  type T_rom_type is array (G_NUM_CONNECT - 1 downto 0) of
   std_logic_vector(G_WEIGHT_WIDTH - 1 downto 0);

  -- Reads an ASCII file with bit-vector patterns on each line
   where:
  --   + each line has a single binary value of length ‘slv_length
   ‘
  --   + reads up to ‘dim_length‘ lines of file
  -- e.x. a file with values ‘0‘, ‘1‘, and ‘7‘ is:
  --      00000000
  --      00000001
  --      00000111
  -- similar to Vivado RAM file init VHDL template
  function F_read_from_file( file_path  : string ) return
   T_rom_type is
    file     fd       : text is in file_path;
    variable V_line   : line;
    variable V_bitvec : bit_vector(G_WEIGHT_WIDTH - 1 downto 0);
    variable V_return : T_rom_type;
  begin
    for i in T_rom_type'range loop
      readline( fd, V_line );
      read( V_line, V_bitvec );
      V_return(i) := to_stdlogicvector( V_bitvec );
    end loop;
    return V_return;
  end F_read_from_file;


  -- infers as ROM by synthesis tools (LUTRAM vs BRAM left to
    tooling, could
  -- explicitly specificy here as an attribute) and initial values
     are weights
  -- from passed weight file path
  signal sig_weight_array : T_rom_type := F_read_from_file(
    G_WEIGHT_PATH );

  signal sig_idx  : unsigned( F_clog2(G_NUM_CONNECT) - 1 downto 0
    );
  signal sig_prd  : signed(G_DATA_WIDTH + G_WEIGHT_WIDTH - 1
    downto 0);
  signal sig_acc  : signed(G_ACCUM_WIDTH - 1 downto 0);

begin

  dout_valid <= '1' when sig_percep_state = S_OUT_VALID else '0';
  -- given large accumulator register, and we've been shift/
    scaling
  -- after each multiplication, we can simply take the LSBs for
    our
  -- final data output
  dout <= sig_acc(G_DATA_WIDTH - 1 downto 0);

  S_output_FSM: process(clk)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        sig_idx <= (others => '0');
        sig_acc <= (others => '0');

        sig_percep_state <= S_IDLE;
      else
        case sig_percep_state is
          when S_IDLE =>
            if din_valid = '1' then
              -- perform 1st lookup/mult here
              sig_prd <= din(to_integer(sig_idx)) *
                         signed( sig_weight_array(to_integer(
    sig_idx)) );
              sig_idx <= sig_idx + 1;
```

```vhdl

              sig_percep_state <= S_ITER_MAC;
            end if;


          -- iterate through weights & connections and accumulate result
          when S_ITER_MAC =>
            -- accumulate scaled/RSH product from last cycle
            sig_acc <= sig_acc + resize( shift_right( sig_prd, G_WEIGHT_WIDTH ),
                                         sig_acc'length );
            sig_prd <= din(to_integer(sig_idx)) *
                       signed( sig_weight_array(to_integer(sig_idx)) );

            if sig_idx = G_NUM_CONNECT - 1 then
              sig_percep_state <= S_FINAL_ACC;
            end if;
            sig_idx <= sig_idx + 1;


          when S_FINAL_ACC =>
            -- accumulate product from last cycle
            sig_acc <= sig_acc + resize( shift_right( sig_prd, G_WEIGHT_WIDTH ),
                                         sig_acc'length );
            sig_percep_state <= S_OUT_VALID;


          when S_OUT_VALID =>
            -- clear index & accumulator registers for next use
            sig_idx <= (others => '0');
            sig_acc <= (others => '0');
            -- since feed-forward, no need to wait for 'ready'
            sig_percep_state <= S_IDLE;

          when others => sig_percep_state <= S_IDLE;
        end case;
      end if;
    end if;
  end process S_output_FSM;

end rtl;
```

Listing 4.1: Perceptron Component

After the perceptron was built and verified, we created the simple ReLU activation function in VHDL, as shown below:

```vhdl
-- Pipelined ReLU activation: max(0, x)
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;

entity ReLU is
  generic (
    G_DATA_WIDTH : integer := 16
  );
  port (
    clk          : in  std_logic;
    din_valid    : in  std_logic;
    din          : in  signed(G_DATA_WIDTH - 1 downto 0);
    dout_valid   : out std_logic;
    dout         : out signed(G_DATA_WIDTH - 1 downto 0)
  );
end entity ReLU;

architecture rtl of ReLU is

  signal sig_dout   : signed(G_DATA_WIDTH - 1 downto 0);
  signal sig_dvalid : std_logic := '0';

begin

  dout_valid <= sig_dvalid;
  dout       <= sig_dout;

  S_relu: process(clk)
  begin
    if rising_edge(clk) then
      if din > 0 then
        sig_dout <= din;
      else
        sig_dout <= (others => '0');
      end if;
      sig_dvalid <= din_valid;
    end if;
  end process S_relu;

end architecture rtl;
```

Listing 4.2: ReLU Activation Component

Now that we have parameterize-able perceptron components, we can create the fully-connected (dense) layer by pointing to a set of weight files and connecting the input and output array of signed values:

```vhdl
-- Implements a Fully-Connected (Dense) layer of perceptrons and activations
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
  use ieee.std_logic_misc.all;
  use std.textio.all;
library work;
  use work.util_pkg.all;

entity FC is
  generic (
    G_DATA_WIDTH   : integer := 16;
    G_WEIGHT_WIDTH : integer :=  8;
    G_NUM_INPUTS   : integer := 50;
    G_NUM_OUTPUTS  : integer := 32;
    -- accumulator register word size
    G_ACCUM_WIDTH  : integer := 24;
    G_LAYER_IDX    : integer :=  0;
    -- base file system path to weight files for this FC layer, also uses
    -- layer index from above to match file pattern for node's weight file
    G_BASE_PATH    : string  := "/home/jgentile/src/jhu-masters-thesis/src/hdl-lib/DSP/ML/neural/sim/FC_weights_layer_";
    -- choice of activation function post-perceptron: ["NONE", "RELU"]
    G_ACTIVATION   : string  := "RELU"
  );
  port (
    clk            : in  std_logic;
    reset          : in  std_logic;

    -- only one valid required, since all nodes from previous layer need to be valid before moving here
    din_valid      : in  std_logic;
    din            : in  T_signed_2D(G_NUM_INPUTS - 1 downto 0)(G_DATA_WIDTH - 1 downto 0);
    -- no handshaking/ready signaling required either, since we only do simple feed-forward operation
    dout_valid     : out std_logic;
    dout            : out T_signed_2D(G_NUM_OUTPUTS - 1 downto 0)(
   G_DATA_WIDTH - 1 downto 0)
  );
end entity FC;

architecture rtl of FC is

  signal sig_percep_out   : T_signed_2D(G_NUM_OUTPUTS - 1 downto
   0)
                                       (G_DATA_WIDTH - 1 downto 0)
   ;
  signal sig_percep_valid : std_logic_vector(G_NUM_OUTPUTS - 1
   downto 0);
  signal sig_activ_out    : T_signed_2D(G_NUM_OUTPUTS - 1 downto
   0)
                                       (G_DATA_WIDTH - 1 downto 0)
   ;
  signal sig_activ_valid  : std_logic_vector(G_NUM_OUTPUTS - 1
   downto 0);

begin

  -- all nodes should have equal delay so arbitrarily use just one
    activation's
  -- valid output, but pruned layers or other things might change
   this
  -- better than 'and_reduce' right now too, as thats unnecessary
   logic
  dout_valid <= sig_activ_valid(0);
  dout       <= sig_activ_out;

  UG_gen_nodes: for i in 0 to G_NUM_OUTPUTS - 1 generate
    U_percep_x: entity work.perceptron
      generic map (
        G_DATA_WIDTH   => G_DATA_WIDTH,
        G_WEIGHT_WIDTH => G_WEIGHT_WIDTH,
        -- number of connections from previous layer (== # of
   weights)
        G_NUM_CONNECT  => G_NUM_INPUTS,
        -- accumulator register word size
        G_ACCUM_WIDTH  => G_ACCUM_WIDTH,
        -- build path to each weight file here
        G_WEIGHT_PATH  => G_BASE_PATH &
                          integer'image(G_LAYER_IDX) &
                          "_node_" &
                          integer'image(i) &
                          ".txt"
      )
      port map (
        clk             => clk,
        reset           => reset,
        din_valid       => din_valid,
        din             => din,
        dout_valid      => sig_percep_valid(i),
        dout            => sig_percep_out(i)
      );

      UG_ReLU: if G_ACTIVATION = "RELU" generate
        U_ReLU_x: entity work.ReLU
          generic map (
            G_DATA_WIDTH => G_DATA_WIDTH
          )
          port map (
            clk           => clk,
            din_valid     => sig_percep_valid(i),
            din           => sig_percep_out(i),
            dout_valid    => sig_activ_valid(i),
            dout          => sig_activ_out(i)
          );
      end generate UG_ReLU;

      UG_no_activation: if G_ACTIVATION = "NONE" generate
        sig_activ_valid(i) <= sig_percep_valid(i);
        sig_activ_out(i)   <= sig_percep_out(i);
      end generate UG_no_activation;
  end generate UG_gen_nodes;

end rtl;
```

Listing 4.3: Fully-Connected Layer Component

Given our 2D convolutional input layer in our proposed CNN, we must also have a component which can perform the 2D convolution of a given set of filter weights across the 2D input signal. The component developed multiplies all kernel weights by a certain input data window in one cycle, and then uses a two-dimensional, adder tree to perform the accumulation to a final output value; the multiply-accumulate logic is pipelined such that as soon as one window's product is complete, we immediately slide to the next window coordinates and repeat. This gives us a great balance between DSP/resource utilization and low processing latency:

```vhdl
-- Implements 2D convolutional filter given a set of input kernel
   weights
-- of size K_HEIGHT x K_WIDTH and an input signal size of I_HEIGHT
    x I_WIDTH
-- resulting in a configurable output signal size of O_HEIGHT x
   O_WIDTH
-- assumes a single stride and spacing of 0 around input signal
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
  use ieee.std_logic_misc.all;
library work;
  use work.util_pkg.all;

entity conv2D is
  generic (
    G_DATA_WIDTH   : integer := 16;
    G_WEIGHT_WIDTH : integer :=  8;
    G_I_HEIGHT     : integer :=  9;
    G_I_WIDTH      : integer :=  8;
    G_K_HEIGHT     : integer :=  5;
    G_K_WIDTH      : integer :=  4;
    G_O_HEIGHT     : integer :=  5;
    G_O_WIDTH      : integer :=  5
  );
  port (
    clk            : in  std_logic;
    reset          : in  std_logic;

    conv_kern      : in  T_signed_3D(G_K_HEIGHT - 1 downto 0)
                                    (G_K_WIDTH  - 1 downto 0)
                                    (G_WEIGHT_WIDTH - 1 downto 0);

    din_valid      : in  std_logic;
    din            : in  T_signed_3D(G_I_HEIGHT - 1 downto 0)
                                    (G_I_WIDTH  - 1 downto 0)
                                    (G_DATA_WIDTH - 1 downto 0);

    dout_valid     : out std_logic;
    dout           : out T_signed_3D(G_O_HEIGHT - 1 downto 0)
                                    (G_O_WIDTH  - 1 downto 0)
                                    (G_DATA_WIDTH - 1 downto 0)
  );
end entity conv2D;

architecture rtl of conv2D is

  type T_conv2D_fsm is (S_IDLE,
                        S_CALC_KERN,
                        S_WAIT_FINAL_ACC,
                        S_OUT_VALID);
  signal sig_conv2D_state : T_conv2D_fsm := S_IDLE;

  signal sig_row_offst : integer range 0 to G_O_HEIGHT;
  signal sig_col_offst : integer range 0 to G_O_WIDTH;

  signal sig_final_row_offst : integer range 0 to G_O_HEIGHT;
  signal sig_final_col_offst : integer range 0 to G_O_WIDTH;

  constant K_POST_MULT_SZ    : integer := G_DATA_WIDTH +
   G_WEIGHT_WIDTH;
  constant K_POST_ROW_ADD_SZ : integer := K_POST_MULT_SZ + F_clog2
   (G_K_WIDTH);
  constant K_POST_COL_ADD_SZ : integer := K_POST_ROW_ADD_SZ +
   F_clog2(G_K_HEIGHT);

  signal sig_conv_kern_prd_valid : std_logic;
  signal sig_conv_kern_prd : T_signed_3D(G_K_HEIGHT - 1 downto 0)
                                        (G_K_WIDTH  - 1 downto 0)
                                        (K_POST_MULT_SZ - 1 downto
    0);
  signal sig_conv_kern_prd_slv : T_slv_3D(G_K_HEIGHT - 1 downto 0)
                                         (G_K_WIDTH  - 1 downto 0)
                                         (K_POST_MULT_SZ - 1
   downto 0);

  signal sig_kern_prd_row_acc_slv     : T_slv_2D(G_K_HEIGHT - 1
   downto 0)
                                                (K_POST_ROW_ADD_SZ
    - 1 downto 0);
  signal sig_kern_prd_row_acc_vld_vec : std_logic_vector(
   G_K_HEIGHT - 1 downto 0);
  signal sig_kern_prd_row_acc_vld     : std_logic;

  signal sig_kern_prd_final_acc     : std_logic_vector(
   K_POST_COL_ADD_SZ - 1 downto 0);
  signal sig_kern_prd_final_acc_vld : std_logic;

  signal sig_dout      : T_signed_3D(G_O_HEIGHT - 1 downto 0)
                                    (G_O_WIDTH  - 1 downto 0)
                                    (G_DATA_WIDTH - 1 downto 0);

begin

  dout_valid <= '1' when sig_conv2D_state = S_OUT_VALID else '0';
  dout       <= sig_dout;

  -- create 2D adder tree which adds in parallel across rows, then adds
  -- the column-sum vector to a single output. design is pipelined so
  -- that we can start throwing 2D products to it, and a seperate valid
  -- counter indexes into the final registered 2D signal
  UG_parallel_adder_tree_rows: for i in 0 to G_K_HEIGHT - 1 generate
    -- convert to T_slv_3D type for adder tree component use
    UG_map_slv: for j in 0 to G_K_WIDTH - 1 generate
      sig_conv_kern_prd_slv(i)(j) <= std_logic_vector( sig_conv_kern_prd(i)(j) );
    end generate UG_map_slv;

    U_row_adder: entity work.adder_tree
      generic map (
        G_DATA_WIDTH => K_POST_MULT_SZ,
        G_NUM_INPUTS => G_K_WIDTH
      )
      port map (
        clk          => clk,
        reset        => reset,
        din_valid    => sig_conv_kern_prd_valid,
        din          => sig_conv_kern_prd_slv(i),
        dout_valid   => sig_kern_prd_row_acc_vld_vec(i),
        dout         => sig_kern_prd_row_acc_slv(i)
      );
  end generate UG_parallel_adder_tree_rows;

  -- just need to use one of the valids since all should complete at the same time
  --sig_kern_prd_row_acc_vld <= and_reduce( sig_kern_prd_row_acc_vld_vec );
  sig_kern_prd_row_acc_vld <= sig_kern_prd_row_acc_vld_vec(0);

```

```
U_col_adder: entity work.adder_tree -- final add across rows
  generic map (
    G_DATA_WIDTH => K_POST_ROW_ADD_SZ,
    G_NUM_INPUTS => G_K_HEIGHT
  )
  port map (
    clk          => clk,
    reset        => reset,
    din_valid    => sig_kern_prd_row_acc_vld,
    din          => sig_kern_prd_row_acc_slv,
    dout_valid   => sig_kern_prd_final_acc_vld,
    dout         => sig_kern_prd_final_acc
  );



S_build_output_matrix: process(clk)
begin
  if rising_edge(clk) then
    if (reset = '1') or (sig_conv2D_state = S_OUT_VALID) then
      sig_final_row_offst <= 0;
      sig_final_col_offst <= 0;
    else
      if sig_kern_prd_final_acc_vld = '1' then
        sig_dout(sig_final_row_offst)(sig_final_col_offst) <=
          signed( sig_kern_prd_final_acc(G_DATA_WIDTH - 1 downto
  0) );

        if sig_final_col_offst = G_O_WIDTH - 1 then
          sig_final_row_offst <= sig_final_row_offst + 1;
          sig_final_col_offst <= 0;
        else
          sig_final_col_offst <= sig_final_col_offst + 1;
        end if;
      end if;
    end if;
  end if;
end process S_build_output_matrix;



S_main_FSM: process(clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      sig_row_offst    <= 0;
```

```
        sig_col_offst    <= 0;
        sig_conv2D_state <= S_IDLE;

        sig_conv_kern_prd_valid <= '0';
      else
        case sig_conv2D_state is
          when S_IDLE =>
            sig_conv_kern_prd_valid <= '0';

            sig_row_offst <= 0;
            sig_col_offst <= 0;
            if din_valid = '1' then
              sig_conv2D_state <= S_CALC_KERN;
            end if;

          when S_CALC_KERN =>
            sig_conv_kern_prd_valid <= '1';
            -- parallel products of 2D kernel and current offset into 2D input
            for i in 0 to G_K_HEIGHT - 1 loop
              for j in 0 to G_K_WIDTH - 1 loop
                sig_conv_kern_prd(i)(j) <= conv_kern(i)(j) *
                                           din(i + sig_row_offst)(j + sig_col_offst);
              end loop;
            end loop;

            if sig_col_offst = G_O_WIDTH - 1 then
              if sig_row_offst = G_O_HEIGHT - 1 then
                -- should be at end of output size, wrap things up, change state
                sig_conv2D_state <= S_WAIT_FINAL_ACC;
              end if;
              sig_row_offst <= sig_row_offst + 1;
              sig_col_offst <= 0;
            else
              sig_col_offst <= sig_col_offst + 1;
            end if;


          when S_WAIT_FINAL_ACC =>
            sig_conv_kern_prd_valid <= '0';
            -- #TODO: wait till parallel adder valid goes low? since we should have stuffed that pipeline
            if sig_kern_prd_final_acc_vld = '0' then
              sig_conv2D_state <= S_OUT_VALID;
```

```vhdl
            end if;

          when S_OUT_VALID =>
            sig_conv2D_state <= S_IDLE;

          when others => sig_conv2D_state <= S_IDLE;
        end case;
      end if;
    end if;
  end process S_main_FSM;

end rtl;
```

Listing 4.4: 2D Convolution Component

Now that we have all of our CNN building blocks, we can easily tie together a sequential CNN- similar to the Keras sequential layering process in TensorFlow- with our VHDL components to create our overall Adaptive Beamforming CNN. Since our input is really a 2D "image"- with real and complex channels instead of RGB color channels- we split the 2D convolution across both channels, and then use a generative VHDL statement to flatten the 2D convolved output to feed the two dense, fully-connected layers. This overall CNN structure can be seen below:

```vhdl
-- Example CNN from thesis research of ABF CNN with N=8:
--      Input Size:  9x8x2
--      Output Size: 8x2
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
  use ieee.std_logic_misc.all;
library work;
  use work.util_pkg.all;

entity ABF_CNN_N9x8x2 is
  generic (
    G_DATA_WIDTH    : integer := 16
  );
  port (
    clk             : in  std_logic;
    reset           : in  std_logic;

    -- input from covariance matrix calculation
    din_valid      : in  std_logic;
    din_real       : in  T_signed_3D(8 downto 0)
                                    (7 downto 0)
                                    (G_DATA_WIDTH - 1 downto 0);
    din_imag       : in  T_signed_3D(8 downto 0)
                                    (7 downto 0)
                                    (G_DATA_WIDTH - 1 downto 0);

    -- output adaptive weights from CNN
    dout_valid     : out std_logic;
    dout_real      : out T_signed_2D(7 downto 0)
                                    (G_DATA_WIDTH - 1 downto 0);
    dout_imag      : out T_signed_2D(7 downto 0)
                                    (G_DATA_WIDTH - 1 downto 0)
  );
end entity ABF_CNN_N9x8x2;

architecture rtl of ABF_CNN_N9x8x2 is

  constant K_WEIGHT_WIDTH : integer := 8; -- signed, 8b quantized
   weights throughout

  signal sig_conv_kern_real : T_signed_3D(4 downto 0)
                                         (3 downto 0)
                                         (K_WEIGHT_WIDTH - 1
   downto 0);
  constant K_conv_kern_int_real  : T_int_3D(4 downto 0)
                                             (3 downto 0) :=
                                   (
                                     (-26,  66,  16, -15),
                                     (-62,  -5, -24, -36),
                                     (-39,  29, -44, -38),
                                     (-84,  53,  12,   9),
                                     ( 99,  80, -65, -44)
                                   );
  signal sig_conv_kern_imag : T_signed_3D(4 downto 0)
                                         (3 downto 0)
                                         (K_WEIGHT_WIDTH - 1
   downto 0);
  constant K_conv_kern_int_imag  : T_int_3D(4 downto 0)
                                             (3 downto 0) :=
                                   (
                                     ( -10, -21, -13, -63),
                                     (  -4, -54, -30,  57),
                                    (  24,  10,  10, -32),
                                    (-104,  23,  17, -26),
                                    ( -97,-127,  96, 125)
                                  );

  signal sig_conv2D_out_real : T_signed_3D(4 downto 0)
                                          (4 downto 0)
                                          (G_DATA_WIDTH - 1 downto
    0);
  signal sig_conv2D_out_real_valid : std_logic;
  signal sig_conv2D_out_imag : T_signed_3D(4 downto 0)
                                          (4 downto 0)
                                          (G_DATA_WIDTH - 1 downto
    0);
  signal sig_conv2D_out_imag_valid : std_logic;

  signal sig_FC0_din        : T_signed_2D(49 downto 0)(
   G_DATA_WIDTH - 1 downto 0);
  signal sig_FC0_dout_valid : std_logic;
  signal sig_FC0_dout       : T_signed_2D(31 downto 0)(
   G_DATA_WIDTH - 1 downto 0);
  signal sig_FC1_dout_valid : std_logic;
  signal sig_FC1_dout       : T_signed_2D(15 downto 0)(
   G_DATA_WIDTH - 1 downto 0);

begin

  -- map integer values to signed input
  UG_row_conv2D: for i in 0 to 4 generate
    UG_col_conv2D: for j in 0 to 3 generate
      sig_conv_kern_real(i)(j) <= to_signed( K_conv_kern_int_real(
   i)(j), K_WEIGHT_WIDTH );
      sig_conv_kern_imag(i)(j) <= to_signed( K_conv_kern_int_imag(
   i)(j), K_WEIGHT_WIDTH );
    end generate UG_col_conv2D;
  end generate UG_row_conv2D;

  U_real_conv2D: entity work.conv2D
    generic map (
      G_DATA_WIDTH   => G_DATA_WIDTH,
      G_WEIGHT_WIDTH => K_WEIGHT_WIDTH,
      G_I_HEIGHT     => 9,
      G_I_WIDTH      => 8,
      G_K_HEIGHT     => 5,
      G_K_WIDTH      => 4,
      G_O_HEIGHT     => 5,
      G_O_WIDTH      => 5
    )
    port map (
      clk            => clk,
      reset          => reset,
      conv_kern      => sig_conv_kern_real,
      din_valid      => din_valid,
      din            => din_real,
      dout_valid     => sig_conv2D_out_real_valid,
      dout           => sig_conv2D_out_real
    );

  U_imag_conv2D: entity work.conv2D
    generic map (
      G_DATA_WIDTH   => G_DATA_WIDTH,
      G_WEIGHT_WIDTH => K_WEIGHT_WIDTH,
      G_I_HEIGHT     => 9,
      G_I_WIDTH      => 8,
      G_K_HEIGHT     => 5,
      G_K_WIDTH      => 4,
      G_O_HEIGHT     => 5,
      G_O_WIDTH      => 5
    )
    port map (
      clk            => clk,
      reset          => reset,
      conv_kern      => sig_conv_kern_imag,
      din_valid      => din_valid,
      din            => din_imag,
      dout_valid     => sig_conv2D_out_imag_valid,
      dout           => sig_conv2D_out_imag
    );

  -- flatten 2Dx2 outputs to wide 2D signal for input to first
   dense hidden layer
  --   goes from 5x5x2 -> 50x1
  UG_row_flatten: for i in 0 to 4 generate
    UG_col_flatten: for j in 0 to 4 generate
      sig_FC0_din( (i*10) + (j*2) )     <= sig_conv2D_out_real(i)(
   j);
      sig_FC0_din( (i*10) + (j*2) + 1 ) <= sig_conv2D_out_imag(i)(
   j);
    end generate UG_col_flatten;
  end generate UG_row_flatten;

  U_hidden_layer_4N: entity work.FC
```

```
    generic map (
      G_DATA_WIDTH   => G_DATA_WIDTH,
      G_WEIGHT_WIDTH => K_WEIGHT_WIDTH,
      G_NUM_INPUTS   => 50,
      G_NUM_OUTPUTS  => 32,
      G_ACCUM_WIDTH  => 24,
      G_LAYER_IDX    => 0,
      G_BASE_PATH    => "/home/jgentile/src/jhu-masters-thesis/src
  /hdl-lib/DSP/ML/neural/sim/FC_weights_layer_",
      G_ACTIVATION   => "RELU"
    )
    port map (
      clk            => clk,
      reset          => reset,
      din_valid      => sig_conv2D_out_real_valid, -- could've
  used *imag too, doesn't matter
      din            => sig_FC0_din,
      dout_valid     => sig_FC0_dout_valid,
      dout           => sig_FC0_dout
    );

  U_output_layer_2N: entity work.FC
    generic map (
      G_DATA_WIDTH   => G_DATA_WIDTH,
      G_WEIGHT_WIDTH => K_WEIGHT_WIDTH,
      G_NUM_INPUTS   => 32,
      G_NUM_OUTPUTS  => 16,
      G_ACCUM_WIDTH  => 24,
      G_LAYER_IDX    => 1,
      G_BASE_PATH    => "/home/jgentile/src/jhu-masters-thesis/src
  /hdl-lib/DSP/ML/neural/sim/FC_weights_layer_",
      G_ACTIVATION   => "NONE" -- no activation for final layer
  that gives weights
    )
    port map (
      clk            => clk,
      reset          => reset,
      din_valid      => sig_FC0_dout_valid,
      din            => sig_FC0_dout,
      dout_valid     => sig_FC1_dout_valid,
      dout           => sig_FC1_dout
    );


  dout_valid <= sig_FC1_dout_valid;
  dout_real  <= sig_FC1_dout( 7 downto 0);
  dout_imag   <= sig_FC1_dout(15 downto 8);

end rtl;
```
Listing 4.5: Adaptive Beamforming CNN

The result is a dramatic decrease in both processing latency and resource utilization for an 8-channel adaptive beamforming implementation, as shown in Table 4.1 using an equivalent 100 MHz clock domain as used in IQRD testing:

N	Latency (μs)	FF	LUT	DSP48	BRAM
8	1.20	3347 (2.4%)	16422 (23.1%)	87 (24.2%)	0 (0.0%)

Table 4.1: FPGA CNN Performance: Latency and Resource Utilization for XCZU3EG FPGA, $N = 8$

Note that, while the CNN architecture was made to be somewhat general, the process from model development to training to FPGA implementation is somewhat tailored for a certain channel count, thus only an attempt for an $N = 8$ case was made and completed. For a specific system where channel count is uncertain, or changing, this could be a reason to use a closed-form QRD-RLS solution that more easily scales for differing channel counts. However if performance and resource constraints are key, the CNN solution looks very attractive.

We can attain even further logic/DSP savings by "folding" or reusing the same multipliers for multiple layers, though similar to folded QRD-RLS architectures, the latency hit may increase beyond system requirements. Similarly at some point, the number of resources required for the FPGA CNN implementation may exceed available space for a given target device, especially for large, complex neural nets with hundreds of thousands (or more) weight parameters. In this case, a certain system may look at other hardware accelerator architectures like Google's Tensor Processing Unit (TPU) which uses high-bandwidth memory and a large, 256×256 wide systolic array to perform 65,536 multiply-accumulate (MAC) operations per cycle, which gives parallel matrix multiplications as shown below [10]:

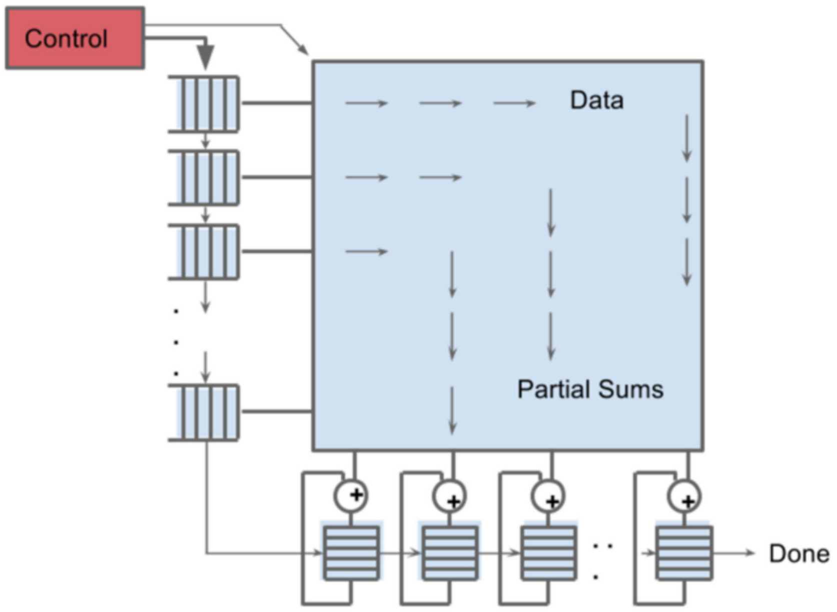


Figure 4.5: Google Tensor Processing Unit- Matrix Multiplier

Source: Adapted from [10]

A detailed comparison of different deep learning accelerator hardware architectures can be found in [11].

4.3 Future Work

The results provided here are likely by no means complete, and more areas of advanced research could likely be applied to squeeze more performance out of this CNN implementation. For instance, in the generated training data, we could provide more complex input signaling, such as different modulation types, instead of simple narrowband input signals.

Specifically to the need of providing a steering vector as part of the input layer, the training scenarios could also include situations where there are slight errors in the actual/ideal steering direction (e.g. the intended SOI is actually a couple degrees off spatially than thought when creating the steering vector) and the model must compensate for those errors to still produce the best SINR; this would be a significant contribution to the field of adaptive beamforming where steering vector errors could lead to the SOI being treated as an interference source, and effectively nulled out, as shown in [12].

One other future area of future optimization may also be in the approach of using *Generative Adversarial Networks (GANs)*, developed in [13], to train the CNN as opposed to simulated data from Python/MATLAB code. This approach was used by [14] with success to train a Massive MIMO's antenna parameters for differing users. Furthermore, using real world sample data would be advantageous in both training and validation of inference of the adaptive beamforming CNN for further confidence in real-world deployments.

It would also be valuable to research the field of *Complex-Valued Neural Networks (CVNNs)* in future works, as shown by Hirose in [15], since RF data is, by nature, complex-valued. However, at the time of this writing, the current set of popular open-source toolsets, like TensorFlow, do not natively support CVNNs yet, and the performance characteristics of using complex-valued layers and activation functions would need to be weighed against traditional, real-valued methods as used in this research process.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[2] T. J. O'Shea, J. Corgan, and T. C. Clancy, *Convolutional radio modulation recognition networks*, 2016. arXiv: 1602.04105 [cs.LG].

[3] Y. Shi, K. Davaslioglu, Y. E. Sagduyu, W. C. Headley, M. Fowler, and G. Green, *Deep learning for rf signal classification in unknown and dynamic spectrum environments*, 2019. arXiv: 1909.11800 [cs.NI].

[4] T. Lin and Y. Zhu, "Beamforming design for large-scale antenna arrays using deep learning," *IEEE Wireless Communications Letters*, vol. 9, no. 1, pp. 103–107, 2020. DOI: 10.1109/LWC.2019.2943466.

[5] B. Luijten, R. Cohen, F. J. de Bruijn, H. A. Schmeitz, M. Mischi, Y. C. Eldar, and R. J. van Sloun, "Deep learning for fast adaptive beamforming," *ICASSP*, 2019.

[6] "Deep learning with int8 optimization on xilinx devices," Xilinx, Tech. Rep. 1.0.1, 2017. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf.

[7] D. A. Teman, *Lecture series on hardware for deep learning part 1: Introduction to deep learning*, https://www.eng.biu.ac.il/temanad/files/2020/09/Part-5-The-DL-Landscape.pdf, 2020.

[8] *Tensorflow*, https://www.tensorflow.org/learn.

[9] *Tensorflow lite- model optimization guide*, https://www.tensorflow.org/model_optimization/guide.

[10] *An in-depth look at googles first tensor processing unit (tpu)*, https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

[11] D. A. Teman, *Lecture series on hardware for deep learning part 5: The deep learning acceleration landscape*, `https://www.eng.biu.ac.il/temanad/files/2020/09/Part-5-The-DL-Landscape.pdf`, 2020.

[12] D. Li, Q. Yin, P. Mu, and W. Guo, "Robust mvdr beamforming using the doa matrix decomposition," *IEEE-ISAS*, 2011.

[13] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, *Generative adversarial networks*, 2014. arXiv: `1406.2661 [stat.ML]`.

[14] T. Maksymyuk, J. Gazda, O. Yaremko, and D. Nevinskiy, "Deep learning based massive mimo beamforming for 5g mobile network," *IEEE International Symposium on Wireless Systems within the International Conferences on Intelligent Data Acquisition and Advanced Computing Systems*, 2018.

[15] A. Hirose, *Complex-Valued Neural Networks*, 2nd ed. Springer, 2012.

Chapter 5

Conclusion

In this research work, we've covered the background knowledge of RF array processing in the context of current, and next generation, MIMO systems. We then explored the current state-of-the-art in Adaptive Beamforming processes for embedded FPGA devices. After creating a baseline implementation for performance and resource comparisons, we explored a novel Deep Learning model to solve Adaptive Beamforming weights in a more efficient process then the current closed-form, statistical solution.

The exciting result was both a proof-of-concept of applying advances in Machine Learning to complicated RF Signal processing tasks, as well as the development of a deployable, vendor-neutral FPGA VHDL design code base that showed drastic performance improvements, as well as reductions in resource utilization for the same FPGA target and channel count, as seen in Table 5.1.

	Latency (μs)	**FF**	**LUT**	**DSP48**
IQRD-RLS	12.50	231199 (164%)	252446 (356%)	272 (75.6%)
CNN	1.20	3347 (2.4%)	16422 (23.1%)	87 (24.2%)
Reduction	**10.4x**	**69.1x**	**15.4x**	**3.13x**

Table 5.1: IQRD vs CNN FPGA Performance: Latency and Resource Utilization for XCZU3EG FPGA, $N = 8$

With this great decrease in FPGA resources for the $N = 8$ channel-count case, we can now comfortably fit in our resource-constrained FPGA device. This decrease in resources also means we can use even lower power FPGA devices, and use lower power for the logic device overall, which allows even more deeply embedded deployment environments. A reduction in FPGA fabric resources can also mean we can now fit other logic in the same programmable-logic space for other important acceleration blocks, such as modulation/demodulation cores for communication systems.

The large decrease in latency of processing time means we can more readily support a "real-time" adaptive beamforming system in an edge FPGA, where we need only buffer- or delay- $1\mu s$ worth of incoming RF data, rather than more than $10\times$ as much data in the IQRD implementation case.

The advantages of Deep Learning applied to array signal processing is also of use beyond RF communication arrays, such as active noise cancellation (ANC) audio applications, which may be extremely SWaP constrained for headphone-style devices. Due to these research findings, it's of the writer's belief that Deep Learning applied to Digital Signal Processing applications will not only grow over time, but may be necessary to meet some future systems' requirements that are currently not met with traditional methods.

Appendix A

Software Code

A.1 MATLAB Code

```
%% X = 2D input sample array(samples x element), sv = steering
    vector,
%  Y = output beam, w = MVDR weights
function [Y, w] = MVDR_beamform(X, sv)
    % form covariance matrix of input samples
    Ecx = X.'*conj(X);

    % compute weight vector using steering vector
    % NOTE: the MATLAB '\' operator is a 2-3x more efficient inv()
     operation
    %       for solving systems of linear equations than inv(Ecx)*
    sv
    wp = Ecx\sv;

    % normalize response
    w = wp/(sv'*wp);

    % form output beam
    Y = X*conj(w);
end
```

Listing A.1: MVDR Process

```
function x=backSubstitution(U,b,n)
% Solving an upper triangular system by back-substitution
% Input matrix U is an n by n upper triangular matrix
% Input vector b is n by 1
% Input scalar n specifies the dimensions of the arrays
% Output vector x is the solution to the linear system
% U x = b
% K. Ming Leung, 01/26/03, from http://cis.poly.edu/~mleung/CS3734
    /s03/ch02/backSubstitutionU.htm

x=zeros(n,1);
for j=n:-1:1
    if (U(j,j)==0)
        error('Matrix is singular!')
    end
    x(j)=b(j)/U(j,j);
    b(1:j-1)=b(1:j-1)-U(1:j-1,j)*x(j);
end
```

Listing A.2: Back-Substitution

```
clear; close('all');
%% Deterministic Digital Beamformer
% givens/user defined values
N       = 16;       % number of elements in ULA (more elements =
    tighter mainlobe & more gain (SNR gain = M))
fc      = 300e6;    % carrier frequency (Hz)
fs      = 1e9;      % sampling frequency (Hz)
theta   = 5;        % wave Angle of Arrival (AoA) in degrees
SNR     = 1;        % element SNR (linear)
noiseP  = 1;        % noise power (linear)
spacing = 0.5;      % d/wavelength element spacing (0.5 = half-
    wavelength)

% calculated constants & vectors
c           = physconst('LightSpeed');
wavelength  = fc/c;
antPos      = (0:1:N-1)*wavelength*spacing; % antenna element
    positions
% create spatial response vector at each ULA element
d = exp(1i*2*pi/wavelength*antPos'*sind(theta)); % phase shift
    over ULA
s = sqrt(SNR*noiseP)*d;


%% compute hypothesis of steering vectors from -1<>+1 (sine space)
     for quiescent response
% sine space is same as sin(-90:90deg)
numHyp = 400; % number of hypothesis to compute
u = linspace(-1,1,numHyp);
v = exp(1i*2*pi/wavelength*antPos'*u);
```

```
% create matched filter (beam weights) for quiescent case (no
   interference)
wq = v;
% unit normalize filter weights
mag = sum(wq .* conj(wq));
wq  = wq./mag;
% compute array response to incoming signal across ULA
yq = wq'*s;

% plot quiescent response in sine space
figure
plot(u*spacing, 20*log10(abs(yq)));
xlabel('Normalized angle, $\frac{d}{\lambda}\sin(\theta)$','
   Interpreter','latex')
ylabel('Normalized Amplitude (dB)')
grid on; ylim([-60 0]);
title('Quiescent ULA Response $\frac{d}{\lambda}=0.5$','
   Interpreter','latex')


%% additive noise response to quiescent beamformer
Nperiod = 1000;
xn = sqrt(noiseP/2)*(randn(N,Nperiod) + 1i*randn(N,Nperiod));
x  = repmat(s,1,Nperiod) + xn;
% apply quiescent beamformer
yn = wq'*x;

figure
plot(u*spacing, 20*log10(abs(yn(:,1))), u*spacing, 20*log10(mean(
   abs(yn.^2),2)));
xlabel('Normalized angle, $\frac{d}{\lambda}\sin(\theta)$','
   Interpreter','latex')
ylabel('Normalized Amplitude (dB)')
grid on; ylim([-60 10]);
title('Quiescent ULA Response with Noise $\frac{d}{\lambda}=0.5$',
   'Interpreter','latex')
legend('Single Period','Average over Periods','Location','
   southwest')


%% Create example received signal w/additive noise & interference
thetaInf      = 30;     % interference wave Angle of Arrival (AoA)
    in degrees
fInf          = 0.9*fc; % interference wave frequency
lambdaInf     = fInf/c;

```

```
dInf = exp(1i*2*pi/lambdaInf*antPos'*sind(thetaInf)); % phase
    shift over ULA

M    = N*128; % M received samples, where M âĽě N channels to form
    MxN sample matrix
t    = (1:1:M)/fs;
rxd = sqrt(SNR*noiseP)*exp(1i*2*pi*fc*t) .* ... % fundamental cw
    pulse
      d;                                        % phase over array
infNoise = sqrt(noiseP/2)*(randn(N,M) + 1i*randn(N,M));
infRx    = sqrt(SNR*noiseP)*exp(1i*2*pi*fInf*t).*dInf; %
    interference wave

% add interference to RX waveform (only for section of time)
rx = rxd + infRx + infNoise;

nonDBF = zeros(1,M);
for i = 1:N % perform non-DBF (weighted sum average) across array
    to show effect
    nonDBF = nonDBF + (rx(i,:)/N);
end

% apply quiescent beamformer using weights matching intended
    incident AoA
[~,uIdx] = min(abs(u-sind(theta))); % find array position of sine
    space
qDBF = wq(:,uIdx)'*rx;

figure
freqBin = (1:1:M)*(fs/M);
subplot(211)
plot(freqBin, 20*log10(abs(fft(nonDBF))))
title('Weighted-Sum Average Spectrum')
xlabel('Frequency (Hz)'); ylabel('Magnitude (dB)');
axis tight
subplot(212)
plot(freqBin, 20*log10(abs(fft(qDBF))))
xline(fc,'g--'); xline(fInf,'r--');
title('Quiescent Beamformer Spectrum')
legend('RX Spectrum', 'f_{c}', 'f_{Inf}')
xlabel('Frequency (Hz)'); ylabel('Magnitude (dB)');
axis tight


%% MVDR weight calculation
```

```
% the desired response or steering vector, repeated to create size
    (m,1)
b = d; % matched filter response of ULA phase shift
%  the complex received sample matrix, size (m,n) where m âĽě n
A = rx.'; % nonconjugate transpose of signal matrix to get correct
    dimensions
[ymv, wmv] = MVDR_beamform(A, b);
figure
plot(freqBin, 20*log10(abs(fft(ymv))))
xline(fc,'g--'); xline(fInf,'r--');
title('MVDR Beamformer Spectrum')
legend('RX Spectrum', 'f_{c}', 'f_{Inf}')
xlabel('Frequency (Hz)'); ylabel('Magnitude (dB)');

% calculate SINR
Rd = (rxd*rxd')/M;
intr_noise = infNoise + infRx;
Rin = (intr_noise*intr_noise')/M;
SINR = 20*log10(abs((wmv'*Rd*wmv)/(wmv'*Rin*wmv)));

sin_mvdr = wmv'*wq;
figure
plot(u*spacing, 20*log10(abs(sin_mvdr)));
xline(sind(theta)*spacing/wavelength,'g--');
xline(sind(thetaInf)*spacing/lambdaInf,'r--');
xlabel('Normalized angle, $\frac{d}{\lambda}\sin(\theta)$','
    Interpreter','latex')
ylabel('Amplitude (dB)')
title('MVDR Response Sine Space $\frac{d}{\lambda}=0.5$','
    Interpreter','latex')
legend('MVDR', '\theta_{c}', '\theta_{Inf}')


%% QR MATLAB
%Acovar = A.'*conj(A);
Acovar = (rx*rx')/M; % when M = pow2, can use simple lsh bitwise
    op (FXP)
                     % since Hermitian positive semi-definite
    output, need
                     % only compute upper or lower triangle of
    values, then
                     % copy conj in other triangle for output
% the desired response or steering vector, repeated to create size
    (m,1)
%b = repmat(d,M/N,1); % matched filter response of ULA phase shift
[Q,R] = qr(Acovar); % perform QR decomp of input sample matrix
```

```
c_qr = Q'*b;
% perform back substituion to solve Rx = Q'b, where x = weights
w_qr = backSubstitution(R, c_qr, N);
%[~,R] = qr(A,0); % perform Q-less QR decomp of input sample
    matrix
%w_qr  = R\R'\b;
% form output beam from complex weights
y_qr = A*conj(w_qr);
figure
plot(freqBin, 20*log10(abs(fft(y_qr))))
xline(fc,'g--'); xline(fInf,'r--');
title('QRD Beamformer Spectrum')
legend('RX Spectrum', 'f_{c}', 'f_{Inf}')
xlabel('Frequency (Hz)'); ylabel('Magnitude (dB)');

sin_qr = w_qr'*wq;
figure
plot(u*spacing, 20*log10(abs(sin_qr)));
xline(sind(theta)*spacing/wavelength,'g--');
xline(sind(thetaInf)*spacing/lambdaInf,'r--');
xlabel('Normalized angle, $\frac{d}{\lambda}\sin(\theta)$','
    Interpreter','latex')
ylabel('Amplitude (dB)')
title('QR Decomposition Response Sine Space $\frac{d}{\lambda}=0.5
    $','Interpreter','latex')
legend('QRD', '\theta_{c}', '\theta_{Inf}')


%% Modified Gram Schmidt
% Q = zeros(N,N);
% R = zeros(N,N);
% for i = 1:N
%     Q(:,i) = A(:,i);
%
%     for j = 1:i-1
%         R(j,i) = Q(:,j)'*Q(:,i);
%         Q(:,i) = Q(:,i) - (R(j,i)*Q(:,j));
%     end
%
%     R(i,i) = norm(Q(:,i));
%     Q(:,i) = Q(:,i)/R(i,i);
% end
% c_qr = Q'*b;
% w_qr = zeros(N,1);
% for i = N:-1:1 % perform back substitution to find weights
%     for j = i+1:N
%         w_qr(i) = R(i,j)*w_qr(j) + w_qr(i);
%     end
%     w_qr(i) = (c_qr(i)-w_qr(i))/R(i,i);
% end
```

Listing A.3: ULA Calculations and MVDR Process

```
clear; close('all');
%% Generates fixed point signed 16bit (S15.0) for HW tests
% givens/user defined values
N         = 4;        % number of elements in ULA (more elements = tighter mainlobe & more gain (SNR gain = N))
fc        = 300e6;    % carrier frequency (Hz)
fs        = 1e9;      % sampling frequency (Hz)
thetaD    = 0;        % desired wave Angle of Arrival (AoA) in degrees
thetaInf = 30;        % interference wave Angle of Arrival (AoA) in degrees
fInf      = 0.9*fc;   % interference wave frequency
SNR       = 1;        % element SNR (linear)
noiseP    = 1;        % noise power (linear)
spacing   = 0.5;      % d/wavelength element spacing (0.5 = half-wavelength)

% calculated constants & vectors
c           = physconst('LightSpeed');
wavelength  = fc/c;
antPos      = (0:1:N-1)*wavelength*spacing; % antenna element positions
% create spatial response vector at each ULA element
d = exp(1i*2*pi/wavelength*antPos'*sind(thetaD)); % phase shift over ULA

%% Create example received signal w/additive noise & interference
lambdaInf = fInf/c;
dInf      = exp(1i*2*pi/lambdaInf*antPos'*sind(thetaInf)); % phase shift over ULA

M = 1024;
t  = (1:1:M)/fs;
rx = sqrt(SNR*noiseP)*exp(1i*2*pi*fc*t) .* ... % fundamental cw pulse
    d +                                    ... % phase over array
    sqrt(noiseP/2)*(randn(N,M) + 1i*randn(N,M)); % random noise

infRx = sqrt(SNR*noiseP)*exp(1i*2*pi*fInf*t).*dInf; % interference wave
```

```
rx     = rx + infRx; % add interference to RX waveform

%% Convert Data to Signed 16b Fixed Point
maxRxValReal = max(abs(real(rx(:))));
maxRxValImag = max(abs(imag(rx(:))));
if maxRxValReal > maxRxValImag
    maxRxVal = maxRxValReal;
else
    maxRxVal = maxRxValImag;
end
scaleVal = ((2^15)/maxRxVal)/2; % scale value for signed 16bit (/2
     for less gain)
rxs16    = round(rx*scaleVal); % scale and round to create signed
    16b values
figure
subplot(211)
plot(t,real(rxs16))
title('Fixed-Point Data')
subplot(212)
freqBin = (1:M)*(fs/M);
plot(freqBin, 20*log10(abs(fft(rxs16(1,:)))))
title('Fixed-Point Spectrum')
xline(fc,'g--'); xline(fInf,'r--');
legend('RX Spectrum', 'f_{Desired}', 'f_{Interference}')
axis tight

%% Convert Steering Vector to 16b Fixed Point
maxDval  = max(abs(d));
scaleVal = floor((2^15)/maxDval)/4; % scale value for signed 16bit
     (/4 to not overflow)
Ds16     = round(d*scaleVal); % scale and round to create signed
    16b values

%% Write steering vector to text file
fId = fopen('steering.txt', 'w');
for i = 1:length(Ds16)
    % each row is: I Q
    I = real(Ds16(i));
    Q = imag(Ds16(i));
    fprintf(fId, '%d %d\n', I, Q);
end
fclose(fId);

%% Write FXP data to text file
fId = fopen('input.txt', 'w');
for sample = 1:length(rxs16(:,1)):length(rxs16(1,:))
```

```
    for ch = 1:length(rxs16(:,1))
        for ch_s = sample:sample+length(rxs16(:,1))-1
            % pre-build square 2D matrix, by printing out 1 of M channels
            % and M samples at a time, then reiterating
            % each row is: I Q
            I = real(rxs16(ch, ch_s));
            Q = imag(rxs16(ch, ch_s));
            fprintf(fId, '%d %d\n', I, Q);
        end
    end
end
fclose(fId);

%% compute hypothesis of steering vectors from -1<>+1 (sine space) for quiescent response
% sine space is same as sin(-90:90deg)
numHyp = 400; % number of hypothesis to compute
u = linspace(-1,1,numHyp);
v = exp(1i*2*pi/wavelength*antPos'*u);
% create matched filter (beam weights) for quiescent case (no interference)
wq = v;
% unit normalize filter weights
mag = sum(wq .* conj(wq));
wq  = wq./mag;

%% QR MVDR Process
Acovar = (rxs16*rxs16')/M; % when M = pow2, can use simple lsh bitwise op (FXP)
                     % since Hermitian positive semi-definite output, need
                     % only compute upper or lower triangle of values, then
                     % copy conj in other triangle for output
maxCovarValReal = max(abs(real(Acovar(:))));
maxCovarValImag = max(abs(imag(Acovar(:))));
if maxCovarValReal > maxCovarValImag
    maxCovarVal = maxCovarValReal;
else
    maxCovarVal = maxCovarValImag;
end
scaleVal  = ((2^15)/maxCovarVal)/2; % scale value for signed 16bit (/2 for less gain)
Acovars16 = round(Acovar*scaleVal); % scale and round to create signed 16b values
```

```
% the desired response or steering vector, repeated to create size
    (m,1)
[Q,R] = qr(Acovars16); % perform QR decomp of input sample matrix
c_qr = Q'*Ds16;
% perform back substituion to solve Rx = Q'b, where x = weights
w_qr = backSubstitution(R, c_qr, N);
A = rxs16.'; % nonconjugate transpose of signal matrix to get
   correct dimensions
% form output beam from complex weights
y_qr = A*conj(w_qr);
figure
plot(freqBin, 20*log10(abs(fft(y_qr))))
xline(fc,'g--'); xline(fInf,'r--');
title('QRD Beamformer Spectrum')
legend('RX Spectrum', 'f_{c}', 'f_{Inf}')
xlabel('Frequency (Hz)'); ylabel('Magnitude (dB)');

sin_qr = w_qr'*wq;
figure
plot(u*spacing, 20*log10(abs(sin_qr)));
xline(sind(thetaD)*spacing/wavelength,'g--');
xline(sind(thetaInf)*spacing/lambdaInf,'r--');
xlabel('Normalized angle, $\frac{d}{\lambda}\sin(\theta)$','
   Interpreter','latex')
ylabel('Amplitude (dB)')
title('QR Decomposition Response Sine Space $\frac{d}{\lambda}=0.5
   $','Interpreter','latex')
legend('QRD', '\theta_{c}', '\theta_{Inf}')
```

Listing A.4: ULA Fixed-Point Test Data Generation

```
clear; close('all');
%% Demonstrate the effect of beamsquint by calculating the
   response of a
% linear array with a design frequency of 3.0 GHz and half
   wavelength
% element spacing. The ULA is 1m long. The system has a bandwidth
   of 600
% Mhz. Show the response when the array is focused at 0ř, 30ř, 45ř
    and 60ř
% off broadside and at 0%, 5%, 10%, 25%, and 50% of the bandwidth
   above the
% design frequency. Assume uniform weighting. The phase shift
   between
% elements should be calculated at the design frequency for the
   antenna
```

```
% beamforming regardless of the calculation frequency.

c        = 3e8;        % speed of light (m/s)
fc       = 3e9;        % given design center frequency (Hz)
lambda0  = c/fc;       % wavelength of center design frequency (m)
elmSpc   = lambda0/2; % half-wavelength element spacing (m)
arrLen   = 1;          % given array length (m)
BW       = 600e6;      % given system bandwidth (Hz)
bwAbvVec = 0.5;        % BW % > design frequency

N = round(arrLen/elmSpc); % # of array elements based on Length &
    spacing
fTest = -90:0.1:90;        % frequencies to test response of array
    over (deg)

figure
hold on;

fBw = fc + (bwAbvVec*BW); % calculate highest freqency at BW
lambda = c/fBw; % wavelength of current BW over design frequency

for pntAng = [0 30 45 60] % pointing angles off broadside (deg)
     % from POMR eq. 9.22, we can calculate the angle/wavelength
    for
     % the set of all incoming RX angles over the wavlength
    determined
     % by the current BW, and the current steering angle with a
    phase
     % shift/wavelength determined by the design center frequency
    giving
     % the incoming waveform angle to pointing angle ratio:
     rx2pnt_ratio = (sind(fTest)/lambda) - (sind(pntAng)/lambda0);
     E = 0; % initialize total antenna directivity pattern
     for n = 1:N % perform summation over each element as eq. 9.22
         E = E + exp(-1i*2*pi*n*elmSpc*rx2pnt_ratio);
     end
     % element directivity pattern Ee(?,?) = 1 for uniform
    weighting
     Ee = 1;
     E = Ee*E/N; % final calc of total antenna directivity pattern
     legendStr = ['\theta_{s} = ', num2str(pntAng), '^{\circ}'];
     plot(fTest, 20*log10(abs(E)), 'DisplayName', legendStr)
     aa = gca; aa.YLim = [-40 5]; aa.XLim = [-90 90];
     title(['Frequency Response @ ',num2str(fBw/1e9),' GHz for ULA
    designed for ', ...
         num2str(fc/1e9),'GHz']);
     xlabel('Angle (\theta^{\circ})');
     ylabel('Directivity (dB)');
 end

 hold off;
 legend('Location','southwest')
```

Listing A.5: Beam Squint Calculation

A.2 Python Code

```
#!/usr/bin/python3
import numpy as np

ang_bitwidth  = 32 # based on atan2 LUT
data_bitwidth = 16 # also the number of CORDIC rotations to
    perform
# CORDIC processing gain: https://en.wikipedia.org/wiki/CORDIC#
    Rotation_mode
processing_gain = 1
for i in range(data_bitwidth):
    processing_gain *= np.sqrt(1.0 + (2.0**(-2.0*i)))

print('CORDIC Processing Gain of component: %0.8f' %
    processing_gain)
u_scale_factor = int(np.floor((1/processing_gain)*(2**
    data_bitwidth)))
s_scale_factor = int(np.floor((1/processing_gain)*(2**(
    data_bitwidth-1))))
print('\tTo cancel gain (scale of %0.8f) for %d bit outputs:' %
    (1/processing_gain, data_bitwidth))
print('\t\t- Multiply by 0x%X (%d unsigned)' % (u_scale_factor,
    u_scale_factor))
print('\t\t\t- Then shift right (>>) by %d bits' % (data_bitwidth)
    )
print('\t\t- Or multiply by 0x%X (%d signed)' % (s_scale_factor,
    s_scale_factor))
print('\t\t\t- Then shift right (>>) by %d bits' % (data_bitwidth
    - 1))

# Convert angle (in degrees) to unsigned integer value for input
    to CORDIC block
def degree_to_unsigned_fxp( angle, bitwidth ):
    # Python mod operator works with FP and constrains to positive
     values:
    #   e.x. -45deg input angle -> 315deg wrapped angle
    wrapped_angle = angle % 360.0
    return int(np.floor( (wrapped_angle/360.0) * (2**bitwidth) ))

# Rotation Mode Tests
    ---------------------------------------------------------
# https://en.wikipedia.org/wiki/CORDIC#Rotation_mode
# this is an efficient way to compute trigonometric functions &
    rotations of a vector
#   Mag/Phase -> I/Q (https://en.wikipedia.org/wiki/
    Polar_coordinate_system#
    Converting_between_polar_and_Cartesian_coordinates)
#     X = r*cos(theta)
#     Y = r*sin(theta)
print('Testing Rotation Mode: Polar format (Mag & Phase) ->
    Rectangular (X & Y)')
magnitudes  = [19429, 5000]
test_angles = [45, 60]
for x_in, ang in zip(magnitudes, test_angles):
    print('%d deg input angle value: %d' % (ang,
        degree_to_unsigned_fxp(ang, ang_bitwidth)) )
    cos_est = round(processing_gain*x_in*np.cos(np.deg2rad(ang)))
    sin_est = round(processing_gain*x_in*np.sin(np.deg2rad(ang)))
    print('\t%d*Cos(%d) [X] ~= %d' % (x_in, ang, cos_est))
    print('\t%d*Sin(%d) [Y] ~= %d' % (x_in, ang, sin_est))


# Vectoring Mode Tests
    --------------------------------------------------------
# https://en.wikipedia.org/wiki/CORDIC#Vectoring_mode
# this is an efficient way to compute magnitude and phase of a
    complex signal
#   I/Q -> Mag/Phase (https://en.wikipedia.org/wiki/
    Polar_coordinate_system#
    Converting_between_polar_and_Cartesian_coordinates)
#     where Mag = sqrt(X**2 + Y**2)
#         Phase = atan2(Y,X)

# CORDIC processing gain: https://www.xilinx.com/support/
    documentation/ip_documentation/cordic/v6_0/pg105-cordic.pdf

print('Testing Vectoring Mode: Rectangular (X & Y) -> Polar format
     (Mag & Phase)')
test_x = [5000]
test_y = [2000]
for x_in, y_in in zip(test_x, test_y):
    print('X: %d, Y: %d' % (x_in, y_in))
    print('Mag: %d' % round(processing_gain*np.sqrt(x_in**2 + y_in
    **2)))
    phase = degree_to_unsigned_fxp(np.rad2deg(np.arctan2(y_in,
    x_in)), ang_bitwidth)
    print(phase)
    print( hex(phase) )
```

Listing A.6: CORDIC Numerical Validation

```
import numpy as np
import random

N = 3 # number of channels
M = 5 # samples per channel to estimate, where M âĽě N
# form MxN complex sample matrix
x = np.matrix( np.arange(N*M).reshape((N,M)) )
z = x - 1j*x
# create differing imag() parts to show Hermitian response
for i in range(0, N):
    for j in range(0, M):
        z[i,j] = x[i,j] - (random.randint(-5,5)*1j*x[i,j])
print("Sample Data & Complex Transpose:")
print(z)
print()
print(z.H)
print()
# Sample covariance matrix estimation (https://en.wikipedia.org/
    wiki/Estimation_of_covariance_matrices)
covar = np.matmul(z, z.H)/M
print("Direct Covariance Response:")
print(covar)
print()

# show manual model of covariance calc (for HDL implementation)
ct = np.zeros((3,3), dtype=np.complex_)
for i in range(0, N):         # rows
    for j in range(0, i+1):   # columns
        for k in range(0, M): # sample in row
            # MAC input sample vector at each time step based on
    output position
            # Only need to calculate lower triangle of covariance
    matrix since
            # output is always Hermitian positive semi-definite (
    lower == upper
            # triangle)
            ct[i,j] = z[i,k]*np.conjugate( z[j,k] ) + ct[i,j]
            # copy conj() in upper triangle for output
            if i != j:
                ct[j,i] = np.conjugate( ct[i,j] )

# when M = pow2, can use simple lsh bitwise op (FXP) for divide-by-M, and
# use seperate wrapper component to do /div & give option to either stream/double-buffer
# output covar matrix, or just output 3D signed array directly for use somewhere else
ct = ct/M
print("Dataflow model output:")
print(ct)
```

Listing A.7: Sample Covariance Matrix Validation

```
#!/usr/bin/env python3
#
# takes exported np-array weights from Netron and converts to binary string files
# for use by VHDL components. could also do this direct from *.tflite file like in:
# https://stackoverflow.com/questions/52111699/how-can-i-view-weights-in-a-tflite-file
#

import numpy as np

def int_to_bin_string(value, bitWidth):
    # convert to signed-8b twos-complement value
    twos_cmplt = value & ((2**bitWidth)-1)
    # write out as padded binary string (always fixed character width)
    return str((bin(twos_cmplt)[2:].zfill(bitWidth)))

# assumes FC weights are simple 2D numpy matrix of size (output, input)
def write_FC_weight_files(weights, layerID, bitWidth):
    for node_idx in range(len(weights)):
        # write individual weight file per perceptron/neural-node
        fd = open("FC_weights_layer_%d_node_%d.txt" % (layerID, node_idx), "w")
        for weight_val in weights[node_idx]:
            fd.write( int_to_bin_string(weight_val, bitWidth) + "\n" )
        fd.close()


```

```python
if __name__ == "__main__":
    # execute only if run as a script
    nBits      = 8  # bitwidth of quantized integer weights
    FClayerIdx = 0  # layer index to help identify sets of weight
    files

    # Netron easily export layer weights directly as NumPy array
    files
    conv2D_weights = np.load("./sequential_conv2d_0")
    FC0_weights    = np.load("./sequential_dense_MatMul_FC0")
    FC1_weights    = np.load("./sequential_dense_MatMul_FC1")

    print("2D-convolutional filter dimensions:")
    print("\tLayer 0: 2D convolution weights of size {}".format(
    conv2D_weights.shape))
    print("Fully-connected dimensions = (output, input)")
    print("\tLayer 1: Fully-connected weights of size {}".format(
    FC0_weights.shape))
    print("\tLayer 2: Fully-connected weights of size {}".format(
    FC1_weights.shape))

    write_FC_weight_files(FC0_weights, FClayerIdx, nBits)
    FClayerIdx += 1
    write_FC_weight_files(FC1_weights, FClayerIdx, nBits)
```

Listing A.8: TFLite CNN Weights Binary String Files

A.3 TensorFlow Jupyter Notebook

ML Adaptive Beamformer for ULA

Quiescent Beamforming

First as a background, the quiescent (e.g. static) case of a linear array will be considered. The non-dynamic beamforming weights will be derived for a given steering direction. We will also show the basis of Digital Beamforming (DBF) with these static weights.

Lets also import the necessary Python packages and libraries now too:

In [1]:
```
import os
import pathlib
import random
from tqdm.notebook import trange, tqdm
from IPython.display import Image, display

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.constants

import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras import models
# used for model pruning
import tensorflow_model_optimization as tfmot
```

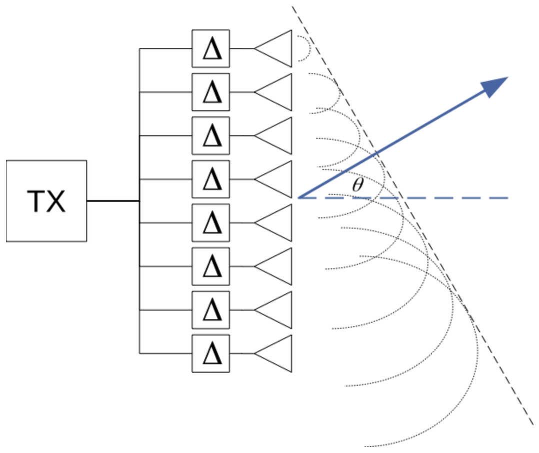
Set the random seeds for the Python libraries for experiment reproducibility

In [2]:
```
seed = 17
random.seed(seed)
tf.random.set_seed(seed)
np.random.seed(seed)
```

Uniform Linear Array Parameters & System Constants

Here we are dealing with a linear phased array system with distances between source and emitters assumed to be in the far field ($\geq \frac{2D^2}{\lambda}$) so phase shift across array elements are equal to the same angle θ.

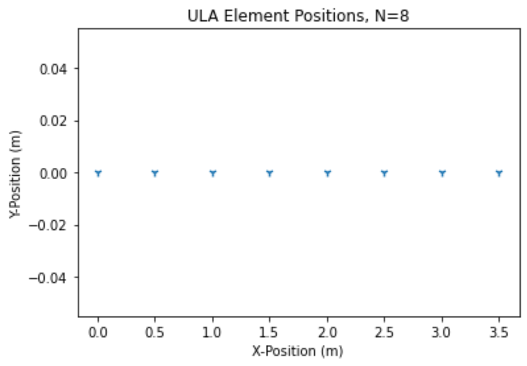
In [3]:
```
display(Image(filename='../../02_abf_background/phased_array.png', width=400))
```



Set the parameters and system constants for the Uniform Linear Array (ULA), including the number of antenna elements, N, the operating/carrier frequency, f_c, and the desired plane wave *angle of arrival* (AoA) relative to boresight, θ_0:

> NOTE: to prevent grating lobes, the antenna element spacing should be $\frac{d}{\lambda} \leq 0.5$

In [4]:
```
N       = 8     # number of elements in ULA
fc      = 300e6 # RF carrier frequency (assuming narrowband here)
fs      = 1e9   # IF/direct sampling frequency
theta   = -10   # desired signal Angle of Arrival (AoA) in degrees
SNR     = 1     # element SNR (linear units)
noiseP  = 1     # noise power (linear units)
spacing = 0.5   # d/wavelength element spacing (0.5 = half-wavelength spacing)

wavelength = fc/scipy.constants.c
# generate array of antenna element positions
antPos = np.linspace(0,N-1,N)*wavelength*spacing
plt.scatter(antPos, np.zeros(N), marker='1')
plt.title("ULA Element Positions, N=%i" % N)
plt.ylabel("Y-Position (m)")
plt.xlabel("X-Position (m)")
plt.show()
```



Spatial Response

For narrowband signals, the complex spatial response vector is formed from the baseband envelope phasor at each ULA element, which is a function of AoA θ_0, operating wavelength λ , and the elemental spacing d [2]:

$$s_n = e^{j2\pi(n-1)\frac{d}{\lambda}\sin\theta_0} \quad 0 \leq n \leq N-1$$

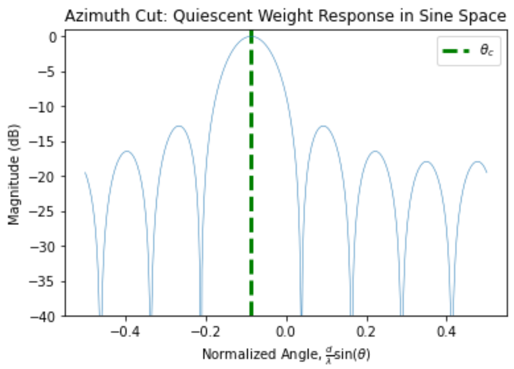
In [5]:
```
# given wavelength (same units as ula_pos_vec), azimuth direction of wave impinc
def narrowband_spatial_phasor(wavelen, theta_deg, ula_pos_vec):
    cmplx_pos = (1j*2*np.pi/wavelen)*ula_pos_vec.T
    sn = np.exp(cmplx_pos*np.sin(np.deg2rad(theta_deg)))
    return sn

s = narrowband_spatial_phasor(wavelength, theta, antPos)
```

Compute hypothesis of steering vectors in sine space for quiescent beamforming weights. Plot the weight response over sine space by testing weight magnitude at each look direction in vector u , from -90° to 90°

In [6]:
```
# numHyp = number of direction hypothesis to compute
def calc_quiescent_weights(wavelen, ula_pos_vec, numHyp=400):
    u = np.linspace(-1, 1, numHyp)
    wq = np.exp(np.outer((1j*2*np.pi/wavelen)*ula_pos_vec.T, u))
    # normalize quiescent filter weights to unity (0dB) @ boresight
    mag = wq * wq.conj()
    wq = wq / mag.sum(axis=0) # sum over columns (each channel, per hypothesis)
    return wq, u

wq, u = calc_quiescent_weights(wavelength, antPos, 400)
```

In [7]:
```
def plot_az_cut(weights,                # weights to plot
                thetas,                 # list of tuples (angles, wavelengths)
                                        # to plot (1st is desired angle)
                plt_title='Azimuth Cut', # plot title
                lims=[-40,1]):          # plot limits (b/c resp -> 0 near edges
```

```
    # convolve quiescent ULA response w/given spatial weights
    conv_weights = np.inner(wq.conj().T, weights)
    fig, ax = plt.subplots()
    ax.plot(u*spacing, 20*np.log10(np.abs(conv_weights)), linewidth=0.5)
    ax.set(xlabel=r'Normalized Angle, $\frac{d}{\lambda} \sin(\theta)$',
           ylabel='Magnitude (dB)',
           title=plt_title)
    ax.set_ylim(lims)
    for idx, angle in enumerate(thetas):
        if idx == 0:
            lin_color = 'green'
            plt_lbl = r'$\theta_{c}$'
        elif idx == 1:
            lin_color = 'red'
            plt_lbl = r'$\theta_{Inf}$'
        else:
            lin_color = 'red'
            plt_lbl = ''
        # NOTE: since interference tone is at different frequency than desired
        #       tone, the normalized sine space plot scales the incidence angles
        #       by its wavelength
        norm_angle = np.sin(np.deg2rad(angle[0]))*spacing/angle[1]
        # wrap normalized angle for wavelengths >>/<< desired
        while norm_angle < -spacing:
            norm_angle += 2*spacing
        while norm_angle > spacing:
            norm_angle -= 2*spacing
        ax.vlines(norm_angle,
                  lims[0], lims[1],
                  colors=lin_color,
                  linestyles='dashed',
                  label=plt_lbl,
                  linewidth=(3 if idx == 0 else 1))
    ax.legend()
    plt.show()

# create matched filter (beam weights) for quiescent case (no interference)
plot_az_cut(s,
            [(theta, wavelength)],
            plt_title='Azimuth Cut: Quiescent Weight Response in Sine Space',
            lims=[-40, 1])
```



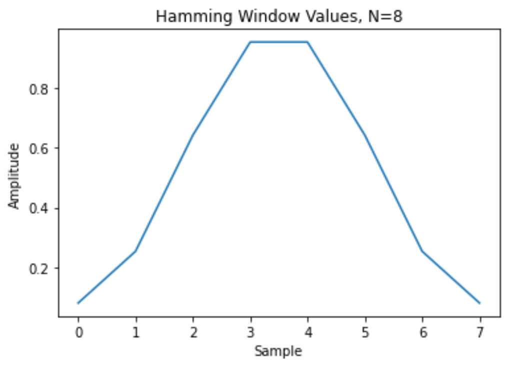
Windowing Quiescent Weights

Windowing can be applied to weights as well to adjust sidelobes, such as applying an M-point Hamming window which is weighted by:

$$w(n) = 0.54 - 0.46\cos(\frac{2\pi n}{M-1}) \quad 0 \le n \le M-1$$

In [8]:

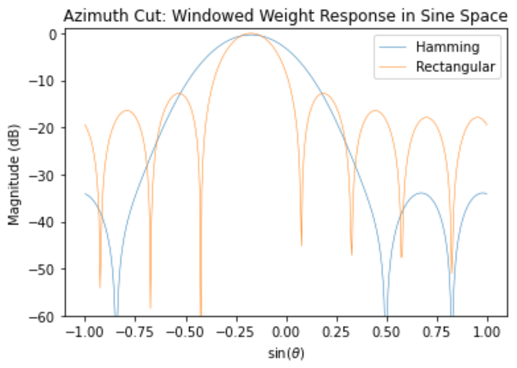
```
window = np.hamming(N)
plt.plot(window)
plt.title("Hamming Window Values, N=%i" % N)
plt.ylabel("Amplitude")
plt.xlabel("Sample")
plt.show()
```



As expected, the great reduction in sidelobes causes an increase in mainlobe width compared to non-windowed weights. Windowing is always a balance between sidelobe performance and mainlobe width.

In [9]:

```
w_ham = window * s
# normalize Hamming windowed weights to 0dB by giving 6dB gain (mainlobe is atte
w_ham *= 10.0**(6.0/20.0)
y_ham = np.inner(wq.conj().T, w_ham)
fig, ax = plt.subplots()

yq = np.inner(wq.conj().T, s) # quiescent response for reference
ax.plot(u, 20*np.log10(np.abs(y_ham)),
        u, 20*np.log10(np.abs(yq)), linewidth=0.5)
ax.set(xlabel=r'$\sin(\theta)$',
       ylabel='Magnitude (dB)',
       title='Azimuth Cut: Windowed Weight Response in Sine Space')
ax.set_ylim([-60, 1]) # set sensible magnitude limits since edges near 0 gain
ax.legend(['Hamming', 'Rectangular'])
plt.show()
```



The non-windowed (rectangular) null-to-null width of a ULA can be found by [2]:

$$\theta_{MB} = 2\sin^{-1}\left[\frac{\lambda}{Nd} - \sin(\theta_0)\right]$$

In [10]:
```
theta_MB = 2*np.rad2deg( np.arcsin( 1/(N*spacing) - np.sin(np.deg2rad(theta)) ))
print('Mainlobe null-to-null width of rectangular window is %f degrees' % theta_
```

```
Mainlobe null-to-null width of rectangular window is 50.130255 degrees
```

Response to Off-Angle Interference

Here we create an interference signal at a different angle of arrical than our intended signal.

In [11]:
```
def shifted_tone(amplitude, freq, time_vec, spatial_phasor_vec):
    phasor_tone = amplitude * np.exp(1j*2*np.pi*freq*time_vec)
    phasor_shft = np.outer(phasor_tone, np.matrix(spatial_phasor_vec)).T
    return phasor_shft
```

Create an interference signal and additive gaussian noise.

In [12]:
```
thetaInf = 30 # intereference signal Angle of Arrival (AoA) in degrees
fInf = 0.9*fc # interference signal carrier frequency (assuming narrowband)
wavelengthInf = fInf/scipy.constants.c

M = N*128 # M received samples/ch, where M ≥ N channels to form MxN sample matri
# time vector based on sampling frequency
t = np.linspace(1,M,M)/fs
t.shape = (1,M) # force transpose

# phase shift received ideal waveform w/o noise or interference
rx_shft = shifted_tone(np.sqrt(SNR*noiseP), fc, t, s)

# calculate phase shift of interefence wave over ULA
sInf = narrowband_spatial_phasor(wavelengthInf, thetaInf, antPos)
# interference waveform with interference phase shift based on its frequency (ag
infRx_shft = shifted_tone(np.sqrt(SNR*noiseP), fInf, t, sInf)
```

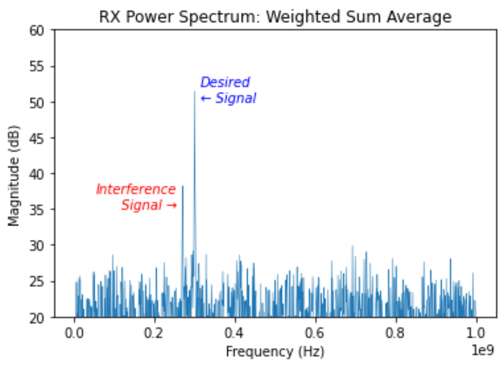
```
# create random RX noise (thermal, environmental, etc.) across array which is Ga
# in distribution: https://en.wikipedia.org/wiki/Johnson%E2%80%93Nyquist_noise
infNoise = np.sqrt(noiseP/2) * (np.random.randn(N,M) + 1j*np.random.randn(N,M))

# add interference & noise to create combined, synthetic RX waveform
rx_all = rx_shft + infNoise + infRx_shft
```

Next, we show a comparison between no digital beamforming and a simple DBF with quiescent weights in the presence of an interferer and noise.

Notice that since the interference is off angle, and the basic sum averaging case is the same as having quiescent weights for a look directly at boresight (0 degrees), the 30 degree interference is seen but at a lower power since it falls within a sidelobe (see previous weight plots above).

In [13]:

```
# perform basic sum averaging across all channels for "no DBF" case
nonDBF = np.sum(rx_all, axis=0)/N

# plot FFT power spectrum of RX signals
freqBin = np.linspace(1,M,M)*(fs/M)
nonDBF_PSD = 20*np.log10(np.abs(np.fft.fft(nonDBF)))
fig, ax = plt.subplots()
ax.plot(freqBin, nonDBF_PSD, linewidth=0.5)
ax.set(xlabel='Frequency (Hz)',
       ylabel='Magnitude (dB)',
       title='RX Power Spectrum: Weighted Sum Average')
ax.text(fInf*.95, 35, 'Interference\nSignal →', horizontalalignment='right', col
ax.text(fc*1.05, 50, 'Desired\n← Signal', color='blue', fontstyle='italic')
ax.set_ylim([20, 60])
plt.show()
```



Calculate Signal-to-interference-plus-noise ratio (SINR) using simple procedure since desired and interference signals known a priori.

In [14]:

```
def calc_SINR_simple(X,      # power spectrum vector
                     Fs,     # sample frequency
                     freqs): # list of >=2 frequencies (1st is desired, others i
    nBins = len(X)
    desired_fbin_idx = int(round(freqs[0]*nBins/Fs))
    desired_power    = X[desired_fbin_idx]
    intfr_fbin_idx   = int(round(freqs[1]*nBins/Fs))
    intfr_power      = X[intfr_fbin_idx]
    for infFreq in freqs[2:]:
        intfr_fbin_idx = int(round(infFreq*nBins/Fs))
        temp = X[intfr_fbin_idx]
        if temp > intfr_power:
            intfr_power = temp
    return desired_power - intfr_power

print("Non-DBF SINR: %0.2f dB" % calc_SINR_simple(nonDBF_PSD, fs, [fc, fInf]))
```

```
Non-DBF SINR: 15.38 dB
```

MVDR Beamforming

Minimum Variance Distortionless Response (MVDR) minimizes the total array power while maintaining unity gain for signals in the desired direction [1]. Essentially this process minimizes noise and off-angle interference signals' powers by placing spatial nulls at certain array angles (also known as "null steering"). The MVDR beamforming weights, w, can be calculated by:

$$w = \frac{\boldsymbol{S}^{-1}\boldsymbol{v_0}}{\boldsymbol{v_0}^H\boldsymbol{S}^{-1}\boldsymbol{v_0}}$$

where $\boldsymbol{S} = XX^H$ is the spatial sample covariance matrix of the N element by M input sample matrix X, and $\boldsymbol{v_0}$ is the complex steering vector (of size N) representing the phase shifts across the array to form the desired steering direction.

In [15]:

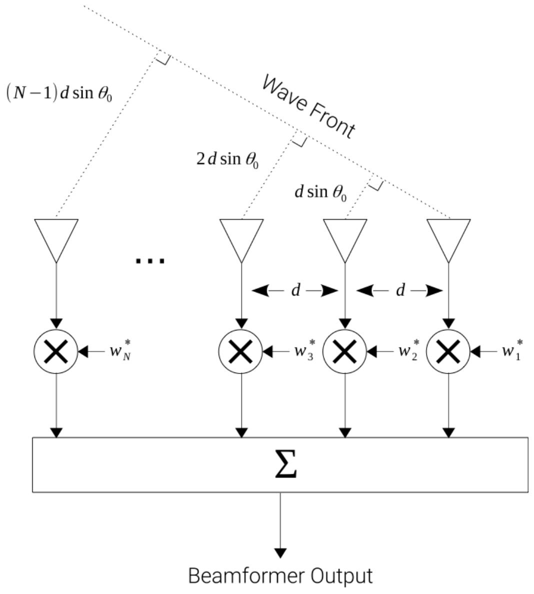
```
# Given N channels, with M samples per channel, and M ≥ N
#   input args: X = 2D input sample array(N elements x M samples)
#      returns: S = sample covariance matrix (N x N)
def calc_covar_matrix(X):
    # form covariance matrix of input samples
    S = X @ X.T.conj()
    return S

# Given N channels, with M samples per channel, and M ≥ N
#   input args: S = sample covariance matrix (N x N), sv = steering vector(N x 1
#      returns: w = MVDR weight col vector(N x 1)
def MVDR_beamform(S, sv):
    # compute weight vector using steering vector and inverting matrix
    # solves for x in Ax=b, where A is the covariance sample matrix, and b is th
    wp, resid, rank, s = np.linalg.lstsq(S, sv, rcond=None)
    # normalize weight response
    w = wp/(np.matrix(sv).H * wp)
    return w

# input args: X = 2D input sample array(N samples x M elements), w = weight col
# returns: Y = output beam row vector(1 x M)
def DBF_apply(X, w):
    # form output beam with MVDR computed weights
    # TODO: why does below work but not -> Y = X * np.conjugate(w)
    Y = np.zeros(M) + 1j*np.zeros(M)
    for i in range(N):
        for j in range(M):
            Y[j] += X[i,j] * np.conjugate(w[i])
    return Y
```

Adaptive Beamforming Application

We can now apply MVDR beamforming to our combined RX data set (containing noise and the desired + intereference signal sources) by calculating the adaptive weights and then applying them to each element channel:

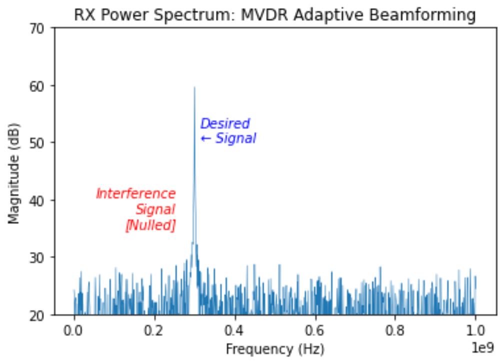
In [16]:

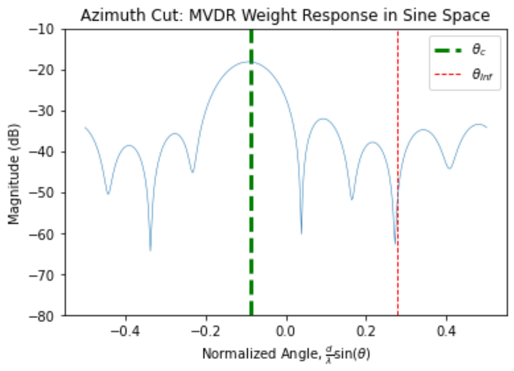
```
display(Image(filename='../../02_abf_background/ula_beamformer.png', width=400))
```



In [17]:

```
steer_vec = np.matrix(s).T
w_MVDR = MVDR_beamform( calc_covar_matrix(rx_all), steer_vec )
Y_MVDR = DBF_apply(rx_all, w_MVDR)

MVDR_PSD = 20*np.log10(np.abs(np.fft.fft(Y_MVDR)))
fig, ax = plt.subplots()
ax.plot(freqBin, MVDR_PSD, linewidth=0.5)
ax.set(xlabel='Frequency (Hz)',
       ylabel='Magnitude (dB)',
       title='RX Power Spectrum: MVDR Adaptive Beamforming')
ax.text(fInf*.95, 35, 'Interference\nSignal\n[Nulled]', horizontalalignment='rig
ax.text(fc*1.05, 50, 'Desired\n← Signal', color='blue', fontstyle='italic')
ax.set_ylim([20, 70])
plt.show()

plot_az_cut(w_MVDR.T,
            [(theta, wavelength),
             (thetaInf, wavelengthInf)],
            plt_title='Azimuth Cut: MVDR Weight Response in Sine Space',
            lims=[-80, -10])

print("Post-MVDR SINR: %0.2f dB" % calc_SINR_simple(MVDR_PSD, fs, [fc, fInf]))
```





```
Post-MVDR SINR: 35.64 dB
```

Here we can see that the off-angle interference wave is nulled out by seeing that the SINR is much improved over non-DBF case. The desired signal gain is also improved over the previous weighted-sum average spectrum as the steering direction is optimizing directionality.

```
In [18]: def calc_SINR_sine(weights, # weights to plot
                            thetas): # list of tuples (angles, wavelengths) to plot (1st
             # convolve quiescent ULA response w/given spatial weights
             conv_weights = np.inner(wq.conj().T, weights)
             #w_spectrum = 20*np.log10(np.abs(conv_weights))
             #ax.plot(u*spacing, w_spectrum, linewidth=0.5)
             desired_pwr = 0
             intfr_pwr   = 0
             for idx, angle in enumerate(thetas):
                 norm_angle = np.sin(np.deg2rad(angle[0]))*spacing/angle[1]
                 # wrap normalized angle for wavelengths >>/<< desired
                 while norm_angle < -spacing:
```

```
            norm_angle += 2*spacing
        while norm_angle > spacing:
            norm_angle -= 2*spacing
        # find index in computed weight spectrum by accounting for -1<>+1 hypthe
        ang_idx = int(round((norm_angle + spacing)*len(u)))
        # bounds check edge condition where we round up to array limit
        if ang_idx >= len(u):
            ang_idx = len(u) - 1
        tmp_pwr = 20*np.log10(np.abs(conv_weights[ang_idx]))
        if idx == 0:
            desired_pwr = tmp_pwr
        elif idx == 1:
            intfr_pwr = tmp_pwr
        else:
            if tmp_pwr > intfr_pwr:
                intfr_pwr = tmp_pwr
    return desired_pwr - intfr_pwr


SINR = calc_SINR_sine(w_MVDR.T,
                      [(theta, wavelength),
                       (thetaInf, wavelengthInf)])
print("Sine-space calculated SINR: %0.2f dB" % SINR)
```

```
Sine-space calculated SINR: 31.90 dB
```

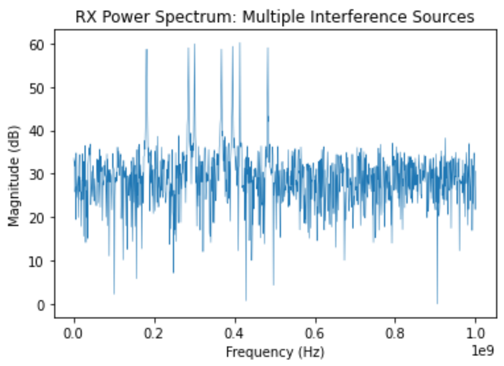
Multiple Interference Sources

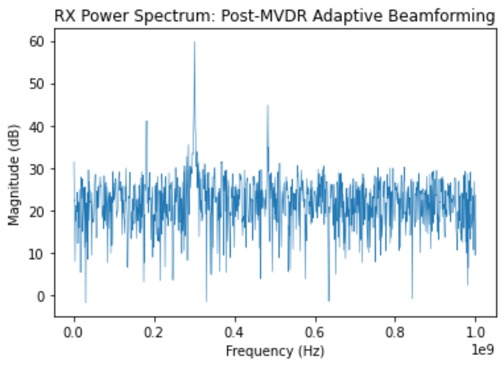
The adaptive nulling scenario can be expanded to multiple sources of interference.

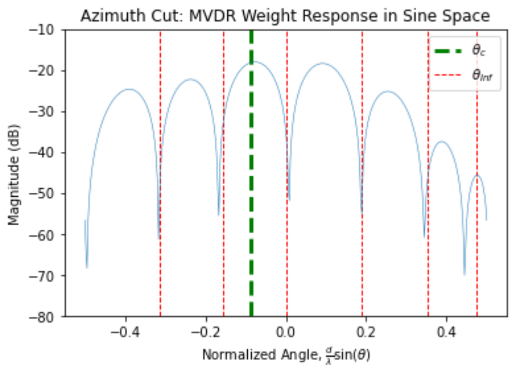
In [19]:

```
num_intfr = N - 2 # number of interference sources to add

# reuse desired signal & noise from before
plt_angles = [(theta, wavelength)]
rx_multi = rx_shft + infNoise
deg_step = 180.0/num_intfr # degree step to use for each interference source
for intfr_idx in range(num_intfr):
    inf_theta = -90.0 + (intfr_idx*deg_step)
    inf_fc    = random.uniform(0.05*fs, 0.5*fs)
    inf_wvlen = inf_fc/scipy.constants.c
    plt_angles += [(inf_theta, inf_wvlen)]
    id_tmp    = narrowband_spatial_phasor(inf_wvlen, inf_theta, antPos)
    rx_multi += shifted_tone(np.sqrt(SNR*noiseP), inf_fc, t, id_tmp)

covar_MVDR = calc_covar_matrix(rx_multi)
w_MVDR     = MVDR_beamform( covar_MVDR, steer_vec )
Y_MVDR     = DBF_apply(rx_multi, w_MVDR)

# arbitrarily plot one channel's spectrum
test_PSD = 20*np.log10(np.abs(np.fft.fft(rx_multi[6,:])))
fig, ax = plt.subplots()
ax.plot(freqBin, test_PSD, linewidth=0.5)
ax.set(xlabel='Frequency (Hz)',
       ylabel='Magnitude (dB)',
       title='RX Power Spectrum: Multiple Interference Sources')
plt.show()

test_MVDR_PSD = 20*np.log10(np.abs(np.fft.fft(Y_MVDR)))
fig, ax = plt.subplots()
```

```
ax.plot(freqBin, test_MVDR_PSD, linewidth=0.5)
ax.set(xlabel='Frequency (Hz)',
       ylabel='Magnitude (dB)',
       title='RX Power Spectrum: Post-MVDR Adaptive Beamforming')
plt.show()

plot_az_cut(w_MVDR.T,
            plt_angles,
            plt_title='Azimuth Cut: MVDR Weight Response in Sine Space',
            lims=[-80, -10])

SINR = calc_SINR_sine(w_MVDR.T,
                      plt_angles)
print("Sine-space calculated SINR: %0.2f dB" % SINR)
```







```
Sine-space calculated SINR: 13.44 dB
```

Note that in some cases (such as the above if the random seed was not changed), MVDR beamforming can not perfectly null all inteference sources. This is mainly due to where interferers fall spatially relative to each other and the desired look direction, which can be seen from the sine space plot; you'll notice most interference angles fall into nulls, but a couple are very close to the desired direction, which cannot be placed into a null (without also nulling the desired direction) due to the lobe width of the given antenna pattern. The takeaway of this effect is that more antenna nulls not only give a system more numerous nulls to place with ABF, but also tighter lobes which can more easily null interference directions with close spacing (relative to each other and/or the desired look direction).

ML Beamforming

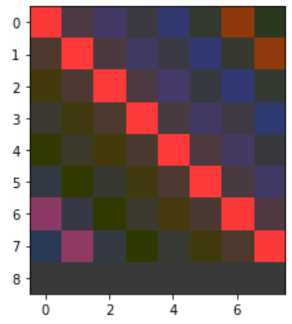
Relevant Current Research

- Beamforming using the Relevance Vector Machine- UCSF: shows Relevance Vector Machine (RVM) to improve standard MVDR with basic sample covariance estimates, for instance better DoA estimation.
- Neural Network Adaptive Beamforming for Robust Multichannel Speech Recognition- Google: uses Long Short Term Memory (LSTM) layers to predict time domain beamforming filter coefficients for time varying speech/acoustic models.
- A Deep Learning Framework for Optimization of MISO Downlink Beamforming- IEEE: uses CNNs to optimize SINR, however with slightly different structure of "exploitation of expert knowledge"

Covariance Matrix Input Layer

We can visualize the covariance matrix as a false color image, along with the deterministic steering vector on the last row; this corollary of the array input preprocessing (e.g. covariance matrix calculation) to 2D images makes an easier transition to interacting with our proposed Machine Learning (ML) model. Though an added dimension is added here to easily plot to standard `pyplot` utilities expecting three color channels (Red, Green, Blue), the input layer to the ML model can be of shape `(batch_size, N+1, N, 2)`, where the added `N+1` dimension allows us to include the intended steering vector, and the last dimension of `2` accounts for the real and imaginary parts of the complex sample covariance matrix.

In [20]:
```
comb_re  = np.zeros((N+1,N))
comb_im  = np.zeros((N+1,N))
comb_re[:N, :] = covar_MVDR.real
comb_re[N, :]  = s.real
comb_im[:N, :] = covar_MVDR.imag
comb_im[N, :]  = s.imag
# create (N+1, N, 3) false color matrix for plotting covariance matrix & steerir
full_test = np.dstack((comb_re, np.zeros((N+1,N)), comb_im))
# normalize covariance values to 0-1 float range for proper plotting
ind_max   = np.unravel_index(np.argmax(full_test, axis=None), full_test.shape)
ind_min   = np.unravel_index(np.argmin(full_test, axis=None), full_test.shape)
full_test = (full_test + np.abs(full_test[ind_min]))/(full_test[ind_max] + np.at
plt.imshow(full_test)
plt.show()
```



Building the covariance sample/estimation matrix as preprocessing before the CNN input layer is likely the best way to map this problem set. Besides it being relatively easy to implement and light on resources (e.g. in HDL and SW code), if input preprocessing was not used, the input layer would be unnecessarilly large to sample some time-series window across all channels (and sample enough data in time to create meaningful associations between channels). If time-series information was needed, a different ML structure such as LTSMs might fit better.

Time-frequency transforms are also popular in applications such as audio recognition

- Two other interesting input preprocessing could also be considered:
  - Using STFT (or overlapped/RTSA to get both time & frequency resolution) BUT instead of taking it across the temporal domain (e.g. across the time domain of each channel's samples like traditional FFT processing), we take it across the spatial dimension (e.g. across all channels, btw could process an 8-channel system with 8-pt FFT per sample, or larger FFT like 64-pt+ by stacking inputs?). We can make a good comparison to STFT used in audio/RF classification tasks (like TF's example and RadioML)

    - https://matplotlib.org/3.1.1/gallery/images_contours_and_fields/specgram_demo.html#sph glr-gallery-images-contours-and-fields-specgram-demo-py
  - Investigate performance with input from CWT, WVD or other time-frequency transform?
  - Investigate any other Lossy Compression preprocessing methods, like DCT or DWT, which can create compressed 2D input layers?

Spatial Dataset Generation

The generated test data sets for training can be built using mathematical RF signal models to create realistic test data. Different parameters should vary to best train the CNN (as well as buck "overfitting" to a specific dataset/test) like:

- Number of interference sources (up to num_channels - 1 theoretical nulling limits)
- Desired & interference signal directions
- Desired & interference signal center frequencies
- Desired & interference signal SNRs
  - In Tim O'Shea's RadioML research, he built an interesting graph which was able to show model accuracy vs input SNR, where expectedly, very low SNR (or signals below noise) led to less probability of modulation classification (different application than us btw, though the RadioML dataset could be another interesting test source)
  - Maybe start to train w/high SNR first, then move to lower and lower SNR levels during training process
- [Advanced] Desired & interference signal modulation and bandwidths (e.x. simple AM/tone, QPSK, QAM, OFDM, LFM/chirp, etc.). This may be somewhat futile given we are generating narrowband weights.

Since we have lots of desired *features* for our training data set, we could also explore using *dimensionality reduction* algorithms in the future for feature extraction.

In [21]:
```
# using current ULA antenna setup, create a matrix of test scenarios
# vary frequency, direction
num_scenarios = 10000   # total size of dataset to generate
train_ratio   = 0.9     # ratio of total dataset to use for training
min_num_intfr = 1       # minimum number of interference sources (0 = no interfe
max_num_intfr = 1       # maximum number of interference sources (N-1 limit arra
min_az_deg    = -89     # minimum look angle (degrees)
max_az_deg    =  89     # maximum look angle (degrees)
min_fc        = 0.05*fs # minimum carrier frequency (Hz, based on given sample r
max_fc        = 0.49*fs # maximum carrier frequency (Hz, based on given sample r

narrowband_test_set = np.zeros((num_scenarios, N+1, N, 2)) # allocate dataset ar
target_weight_set   = np.zeros((num_scenarios, N*2))  # N*2 for flattened output
# array of angle, wavelength pairs [:,0,0] = desired theta, [:,0,1] = desired wa
gen_test_angle_set  = np.zeros((num_scenarios, max_num_intfr + 1, 2))

train_size = int(round(num_scenarios*train_ratio))
pred_size  = int(round(num_scenarios*(1-train_ratio)))

train_test_set   = np.zeros((train_size, N+1, N, 2)) # allocate training array
train_weight_set = np.zeros((train_size, N*2))  # N*2 for flattened output
```

```
pred_test_set    = np.zeros((pred_size, N+1, N, 2)) # allocate prediction array
pred_weight_set  = np.zeros((pred_size, N*2))

# covariance matrices for loss function
Rd = np.zeros((num_scenarios, N, N)) + 1j*np.zeros((num_scenarios, N, N))
Ri = np.zeros((num_scenarios, N, N)) + 1j*np.zeros((num_scenarios, N, N))
for samp_idx in trange(num_scenarios, desc='Generating Dataset'):
    # first start w/additive gaussian noise (~0 dB)
    rx_inf = (10.0**(-40.0/20.0))*(np.random.randn(N,M) + 1j*np.random.randn(N,N

    num_intfr     = random.randint(min_num_intfr, max_num_intfr)
    desired_theta = random.uniform(min_az_deg, max_az_deg)
    desired_fc    = random.uniform(min_fc, max_fc)
    desired_wvlen = desired_fc/scipy.constants.c

    gen_test_angle_set[samp_idx, 0, 0] = desired_theta
    gen_test_angle_set[samp_idx, 0, 1] = desired_wvlen

    # add desired signal
    sd_tmp  = narrowband_spatial_phasor(desired_wvlen, desired_theta, antPos)
    d_SNR   = 20
    d_amp   = 10.0**((-65.0+d_SNR)/20.0)
    # signal of interest
    rx_SOI  = shifted_tone(d_amp, desired_fc, t, sd_tmp)
    # add interference sources
    for intfr_idx in range(num_intfr):
        inf_theta = random.uniform(min_az_deg, max_az_deg)
        inf_fc    = random.uniform(min_fc, max_fc)
        inf_wvlen = inf_fc/scipy.constants.c
        id_tmp    = narrowband_spatial_phasor(inf_wvlen, inf_theta, antPos)
        rx_inf   += shifted_tone(d_amp, inf_fc, t, id_tmp)

        gen_test_angle_set[samp_idx, intfr_idx + 1, 0] = inf_theta
        gen_test_angle_set[samp_idx, intfr_idx + 1, 1] = inf_wvlen

    rx_tot = rx_SOI + rx_inf

    # preprocessing: covariance matrix estimation & steering vector concatenatic
    sv = np.matrix(sd_tmp).T
    covar_tmp = calc_covar_matrix(rx_tot)
    target_weights = MVDR_beamform( covar_tmp, sv ).T

    # we also calculate the desired & interference+noise covariance matrices to
    # function during training to calculate SINR for a batch of infered adaptive
    Rd_tmp = calc_covar_matrix(rx_SOI)
    Ri_tmp = calc_covar_matrix(rx_inf)
    Rd[samp_idx, :, :] = Rd_tmp
    Ri[samp_idx, :, :] = Ri_tmp

    # turn complex covariance matrix & steering vectors -> multi-dim layer
    narrowband_test_set[samp_idx, :N, :, 0] = covar_tmp.real
    narrowband_test_set[samp_idx, :N, :, 1] = covar_tmp.imag
    narrowband_test_set[samp_idx,  N, :, 0] = sv.T.real
    narrowband_test_set[samp_idx,  N, :, 1] = sv.T.imag
    target_weight_set[samp_idx, :N]  = target_weights.real
    target_weight_set[samp_idx,  N:] = target_weights.imag

train_test_set   = narrowband_test_set[:train_size,:,:,:]
train_weight_set = target_weight_set[:train_size,:]
pred_test_set    = narrowband_test_set[train_size:,:,:,:]
pred_weight_set  = target_weight_set[train_size:,:]
```

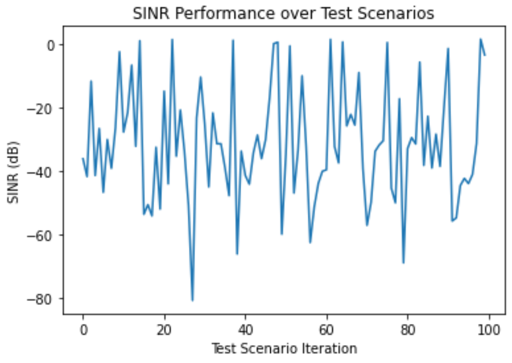
```
train_angle_set  = gen_test_angle_set[:train_size,:,:]
pred_angle_set   = gen_test_angle_set[train_size:,:,:]
print('Spatial Dataset shape: ', narrowband_test_set.shape)
print('Target ABF weights shape: ', target_weight_set.shape)
print('Training Dataset shape: ', train_test_set.shape)
print('Training ABF weights shape: ', train_weight_set.shape)
print('Training angles shape: ', train_angle_set.shape)
print('Prediction Dataset shape: ', pred_test_set.shape)
print('Prediction ABF weights shape: ', pred_weight_set.shape)
print('Prediction angles shape: ', pred_angle_set.shape)
```

```
Spatial Dataset shape:  (10000, 9, 8, 2)
Target ABF weights shape:  (10000, 16)
Training Dataset shape:  (9000, 9, 8, 2)
Training ABF weights shape:  (9000, 16)
Training angles shape:  (9000, 2, 2)
Prediction Dataset shape:  (1000, 9, 8, 2)
Prediction ABF weights shape:  (1000, 16)
Prediction angles shape:  (1000, 2, 2)
```

In [22]:

```
trgt_weights = np.matrix( target_weight_set[:,:N] + 1j*target_weight_set[:,N:] )
print(trgt_weights.shape)
print(Rd.shape)
print(Ri.shape)

#SINR = (w_test.conj() * rd * w_test.T)/(w_test.conj() * ri * w_test.T)
MVDR_SINRs = np.zeros(num_scenarios)
for i in range(num_scenarios):
    #MVDR_SINRs[i] = 20*np.log10( np.abs( (trgt_weights[i,:].conj() * Rd[i,:,:]
    MVDR_SINRs[i] = 20*np.log10( np.abs( (trgt_weights[i,:] * Rd[i,:,:] * trgt_w

plt.plot(MVDR_SINRs[:100])
plt.xlabel('Test Scenario Iteration')
plt.ylabel('SINR (dB)')
plt.title('SINR Performance over Test Scenarios')
plt.show()
```

```
(10000, 8)
(10000, 8, 8)
(10000, 8, 8)
```



CNN Design

The input layer is the 2D, complex covariance sample matrix (with steering vector appended as an extra row). The input 2D convolutional layer (Conv2d) is defined by the Keras API.

The amount of hidden layers is derived from experimentation and resource constraints; more layers can not only can cause overfitting and take longer to train, but many layers blow our eventual resource utilization out of the water.

A max pooling layer MaxPool2D after `Conv2D` can be used to reduce dimensionality, however for our spatial weight derivation, this actually works against us (though great for applications that need to reduce dimensionality, such as single-output regression).

The output layer should be 2D for complex weights (N channels x 2 per I/Q weight), since the goal is to have a CNN which can directly create weights for beamforming "weight and sum"/MAC application in PL logic; specifically for TensorFlow implementation, the output layer is a flattened vector of `2*N` samples, where the first `N` samples are the real part, and the last `N` samples are the imaginary part.

The popular ReLU activation function) is used as the rectification of the neural layers; note other papers have used other activation functions like the Antirectifier since it can keep negative part, however its not necessary for this application since weights can still produce negative output values for final output weights.

In [23]:
```
# since 1x input tensor and 1x output tensor, can use simple sequential layering
# look at https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D
#   http://d2l.ai/chapter_convolutional-neural-networks/lenet.html
model = models.Sequential([
    # 2D dim filter output (keep real & imag) [filter dim/conv is +1 in row to e
    # NOTE: Don't use aditional Conv2D & MaxPooling2D layers since we actually w
    #       Weirdly even by decimating input spatial dimensions into depth-wise
    #       in a layer below), the output performance is terrible...
    layers.Conv2D(2, (int(np.floor(N/2))+1, int(np.floor(N/2))), activation='rel
    # dropout is good for randomnly dropping out weights to prevent overfit but
    #layers.Dropout(0.25),
    layers.Flatten(),
    # interestingly, even though relu gets rid of negative parts, this still giv
    # and results in a better trained network (vs no activation)
    layers.Dense(N*4, activation='relu'),
    layers.Dense(N*2),
])

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 5, 5, 2)           82        
_________________________________________________________________
flatten (Flatten)            (None, 50)                0         
_________________________________________________________________
dense (Dense)                (None, 32)                1632      
_________________________________________________________________
dense_1 (Dense)              (None, 16)                528       
=================================================================
Total params: 2,242
```

```
Trainable params: 2,242
Non-trainable params: 0
_________________________________________________________________
```

Compile & Fit using Keras API

Since doing multi-output regression, not using loss functions meant for classification, but can use basic statistical functions like Mean Squared Error

The Adam optimization algorithm is a currently popular stochastic gradient descent method that is computationally efficient and has many modern benefits.

TODO:

- So it looks like regression to MVDR weights works, so since MVDR isn't necessarilly perfect, should we instead make a custom loss function based on calculating SINR on the fly? (like https://neptune.ai/blog/keras-loss-functions) This could lead to inferring a *better* ABF algorithm, and less expensive
  - There should be a way to pad extra data columns in your input tensor to provide the info necessary for calculating SINR as a loss function https://stackoverflow.com/questions/55445712/custom-loss-function-in-keras-based-on-the-input-data

In [24]:
```
EPOCHS     = 30
# default batch size is 32: higher batch sizes decrease training time, but small
batch_size = 1

#TODO: adapt to calculate SINR (or inverse since loss is looking to be
#      minimized by training optimization function) as new loss fn
def invSINR_loss(train_angle, y_pred):
    pred_SINR = 0
    #for i in range(batch_size):
        #predicted_weights = np.matrix(y_pred[i,:N] + 1j*y_pred[1,N:])
        #pred_SINR += calc_SINR_sine(predicted_weights,
        #                [(train_angle[i,0,0], train_angle[i,0,1]),
        #                 (train_angle[i,1,0], train_angle[i,1,1])])
    #pred_SINR /= batch_size # average SINR for batch
    predicted_weights = np.matrix(y_pred[:N] + 1j*y_pred[N:])
    pred_SINR += calc_SINR_sine(predicted_weights,
                   [(train_angle[0,0], train_angle[0,1]),
                    (train_angle[1,0], train_angle[1,1])])
    return 1/pred_SINR


#loss=MSE_ex(4),
#def MSE_ex(i):
#    def loss(y_true, y_pred):
#        squared_diff = tf.square(y_true - y_pred) + i
#        return tf.reduce_mean(squared_diff, axis=-1)
#    return loss

model.compile(
    optimizer=tf.keras.optimizers.Adam(),
    #loss=invSINR_loss,
    # MSE is good for basic regression/matching
    loss=tf.keras.losses.MeanSquaredError(),
)

# https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit
history = model.fit(
    train_test_set,
    train_weight_set, # standard for regression
    #train_angle_set, # set for eventual custom loss function?
    batch_size=batch_size,
    epochs=EPOCHS,
    verbose=1
)
```

```
Epoch 1/30
9000/9000 [==============================] - 4s 472us/step - loss: 0.0029
Epoch 2/30
9000/9000 [==============================] - 4s 474us/step - loss: 5.0312e-04
Epoch 3/30
9000/9000 [==============================] - 4s 477us/step - loss: 4.4402e-04
Epoch 4/30
9000/9000 [==============================] - 4s 482us/step - loss: 4.3002e-04
Epoch 5/30
9000/9000 [==============================] - 4s 474us/step - loss: 4.1494e-04
Epoch 6/30
9000/9000 [==============================] - 4s 473us/step - loss: 3.9901e-04
Epoch 7/30
9000/9000 [==============================] - 4s 477us/step - loss: 3.9162e-04
Epoch 8/30
9000/9000 [==============================] - 4s 475us/step - loss: 3.9474e-04
Epoch 9/30
9000/9000 [==============================] - 4s 481us/step - loss: 3.9544e-04
Epoch 10/30
9000/9000 [==============================] - 4s 479us/step - loss: 3.9070e-04
Epoch 11/30
9000/9000 [==============================] - 4s 478us/step - loss: 3.7549e-04
Epoch 12/30
9000/9000 [==============================] - 4s 470us/step - loss: 3.7896e-04
Epoch 13/30
9000/9000 [==============================] - 4s 464us/step - loss: 3.7347e-04
Epoch 14/30
9000/9000 [==============================] - 4s 464us/step - loss: 3.7372e-04
Epoch 15/30
9000/9000 [==============================] - 4s 466us/step - loss: 3.7589e-04
Epoch 16/30
9000/9000 [==============================] - 4s 474us/step - loss: 3.6998e-04
Epoch 17/30
9000/9000 [==============================] - 4s 472us/step - loss: 3.6914e-04
Epoch 18/30
9000/9000 [==============================] - 4s 474us/step - loss: 3.6735e-04
Epoch 19/30
9000/9000 [==============================] - 4s 473us/step - loss: 3.6580e-04
Epoch 20/30
9000/9000 [==============================] - 4s 469us/step - loss: 3.6501e-04
Epoch 21/30
9000/9000 [==============================] - 4s 473us/step - loss: 3.7036e-04
Epoch 22/30
9000/9000 [==============================] - 4s 474us/step - loss: 3.6429e-04
Epoch 23/30
9000/9000 [==============================] - 4s 473us/step - loss: 3.6137e-04
Epoch 24/30
9000/9000 [==============================] - 4s 469us/step - loss: 3.6063e-04
Epoch 25/30
9000/9000 [==============================] - 4s 476us/step - loss: 3.5744e-04
Epoch 26/30
9000/9000 [==============================] - 4s 470us/step - loss: 3.6383e-04
```

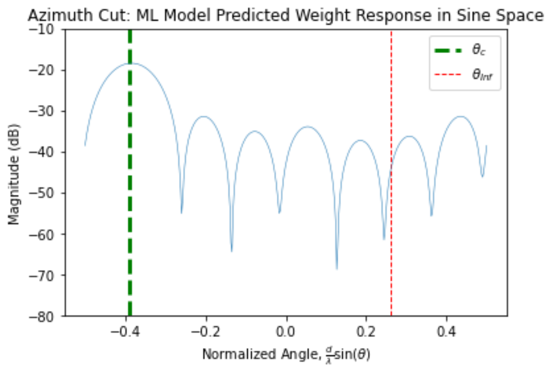
```
Epoch 27/30
9000/9000 [==============================] - 4s 470us/step - loss: 3.5341e-04
Epoch 28/30
9000/9000 [==============================] - 4s 471us/step - loss: 3.5375e-04
Epoch 29/30
9000/9000 [==============================] - 4s 472us/step - loss: 3.5577e-04
Epoch 30/30
9000/9000 [==============================] - 4s 469us/step - loss: 3.4591e-04
```

Once trained, we now use the prediction API to estimate outputs from the designated prediction scenarios.

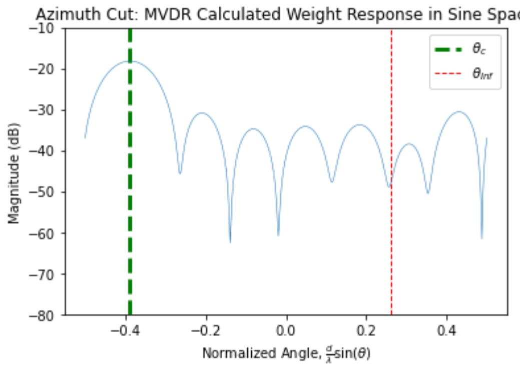
Here's an example cut of one of the predictive scenarios and the performance of the CNN and the traditional MVDR weight calculation process:

In [25]:

```
# use prediction set and get output weights
pred_idx = 28
cnn_pred_weights       = model.predict(pred_test_set[pred_idx:pred_idx+1,:,:,:])
# unflatten to complex numbers for plotting in sine space
cnn_pred_weights_cmplx = cnn_pred_weights[0,:N] + 1j*cnn_pred_weights[0,N:]
verif_pred_weights     = pred_weight_set[pred_idx,:N] + 1j*pred_weight_set[pred_

plot_az_cut(np.matrix(cnn_pred_weights_cmplx),
            [(pred_angle_set[pred_idx,0,0], pred_angle_set[pred_idx,0,1]),
             (pred_angle_set[pred_idx,1,0], pred_angle_set[pred_idx,1,1])],
            plt_title='Azimuth Cut: ML Model Predicted Weight Response in Sine S
            lims=[-80, -10])
SINR = calc_SINR_sine(np.matrix(cnn_pred_weights_cmplx),
            [(pred_angle_set[pred_idx,0,0], pred_angle_set[pred_idx,0,1]),
             (pred_angle_set[pred_idx,1,0], pred_angle_set[pred_idx,1,1])])
print("CNN Output SINR: %0.2f dB" % SINR)

plot_az_cut(np.matrix(verif_pred_weights),
            [(pred_angle_set[pred_idx,0,0], pred_angle_set[pred_idx,0,1]),
             (pred_angle_set[pred_idx,1,0], pred_angle_set[pred_idx,1,1])],
            plt_title='Azimuth Cut: MVDR Calculated Weight Response in Sine Spac
            lims=[-80, -10])
SINR = calc_SINR_sine(np.matrix(verif_pred_weights),
            [(pred_angle_set[pred_idx,0,0], pred_angle_set[pred_idx,0,1]),
             (pred_angle_set[pred_idx,1,0], pred_angle_set[pred_idx,1,1])])
print("MVDR SINR: %0.2f dB" % SINR)
```



```
CNN Output SINR: 24.48 dB
```

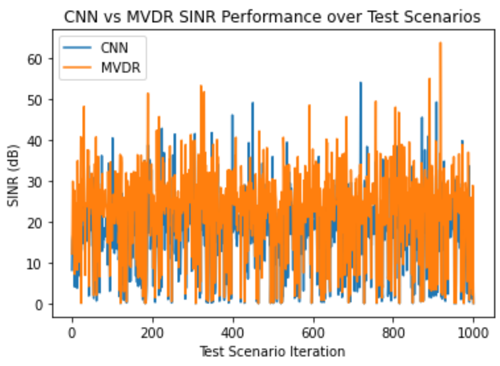


```
MVDR SINR: 28.38 dB
```

Iterate over training scenarios to calculate average SINR for ML model and MVDR approach.

In [26]:

```
avg_CNN_SINR  = 0
avg_MVDR_SINR = 0
CNN_SINRs  = np.zeros(pred_size)
MVDR_SINRs = np.zeros(pred_size)
for test_idx in trange(pred_size, desc='Calculating SINR over Test Scenarios and
    cnn_pred_weights       = model.predict(pred_test_set[test_idx:test_idx+1,:,:
    # unflatten to complex numbers for plotting in sine space
    cnn_pred_weights_cmplx = cnn_pred_weights[0,:N] + 1j*cnn_pred_weights[0,N:]
    verif_pred_weights     = pred_weight_set[test_idx,:N] + 1j*pred_weight_set[t

    CNN_SINRs[test_idx] = calc_SINR_sine(np.matrix(cnn_pred_weights_cmplx),
                                       [(pred_angle_set[test_idx,0,0],
                                         pred_angle_set[test_idx,0,1]),
                                        (pred_angle_set[test_idx,1,0],
                                         pred_angle_set[test_idx,1,1])])
    avg_CNN_SINR += CNN_SINRs[test_idx]

    MVDR_SINRs[test_idx] = calc_SINR_sine(np.matrix(verif_pred_weights),
                                        [(pred_angle_set[test_idx,0,0],
                                          pred_angle_set[test_idx,0,1]),
                                         (pred_angle_set[test_idx,1,0],
                                          pred_angle_set[test_idx,1,1])])
    avg_MVDR_SINR += MVDR_SINRs[test_idx]


plt.plot(np.linspace(1,pred_size,pred_size), CNN_SINRs, label='CNN')
plt.plot(np.linspace(1,pred_size,pred_size), MVDR_SINRs, label='MVDR')
plt.xlabel('Test Scenario Iteration')
plt.ylabel('SINR (dB)')
plt.title('CNN vs MVDR SINR Performance over Test Scenarios')
plt.legend()
plt.show()

avg_CNN_SINR  /= pred_size
avg_MVDR_SINR /= pred_size
print('Average SINR from CNN: %0.2f dB' % avg_CNN_SINR)
print('Average SINR from MVDR: %0.2f dB' % avg_MVDR_SINR)
print('Average SINR performance of CNN compared to the MVDR approach: % 0.2f dB'
      % (avg_CNN_SINR - avg_MVDR_SINR))
```



```
Average SINR from CNN: 17.76 dB
Average SINR from MVDR: 22.67 dB
Average SINR performance of CNN compared to the MVDR approach: -4.91 dB
```

Adaptation to FPGA FW

- After quantization, we do not need to quantize to `0.0 - 1.0` constraint on input data like with floats anymore, nor have negative values get thrown, the model is made to use signed 8-bit integer input and output with the inference input/output type value
- The "why?" of optimization has great TF documentation at -> https://www.tensorflow.org/lite/performance/model_optimization and in Post-training quantization

Pruning

Prune using TF API

In [27]:

```
model_for_pruning = tfmot.sparsity.keras.prune_low_magnitude(model)
model_for_pruning.summary()
```

```
/home/jgentile/.local/lib/python3.8/site-packages/tensorflow/python/keras/engin
e/base_layer.py:2281: UserWarning: `layer.add_variable` is deprecated and will b
e removed in a future version. Please use `layer.add_weight` method instead.
  warnings.warn('`layer.add_variable` is deprecated and '
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
prune_low_magnitude_conv2d ( (None, 5, 5, 2)           164       
_________________________________________________________________
prune_low_magnitude_flatten  (None, 50)                1         
_________________________________________________________________
prune_low_magnitude_dense (P (None, 32)                3234      
_________________________________________________________________
prune_low_magnitude_dense_1  (None, 16)                1042
=================================================================
Total params: 4,441
Trainable params: 2,242
Non-trainable params: 2,199
_________________________________________________________________
```

Quantization

Tools like TF Lite intended for embedded deployment should output a fixed point model for comparison as well (since this would be easiest/most resource optimized for FPGA logic implementation).

To scale a set of float values (b) to signed, integer values, each value can be multiplied by a scaling coefficient k and then rounded to the nearest integer:

$$k = \frac{2^{N-1}}{\max\limits_{x \in b} |x|}$$

Quantize in 16x8 mode which gives 16-bit Integer weights and 8-bit Integer quantized data.

In [28]:

```python
# check if 16x8 (16b weights, 8b quantized values) quanitzation is supported
tf.lite.OpsSet.EXPERIMENTAL_TFLITE_BUILTINS_ACTIVATIONS_INT16_WEIGHTS_INT8
converter = tf.lite.TFLiteConverter.from_keras_model(model)
# use default `optimizations`
converter.optimizations = [tf.lite.Optimize.DEFAULT]
# to use 16x8 mode use the OpsSet flag
converter.target_spec.supported_ops = [tf.lite.OpsSet.EXPERIMENTAL_TFLITE_BUILTI
#converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
# Set inputs & outputs of quantized model to be 8b Integers
#  https://www.tensorflow.org/lite/performance/post_training_quantization#full_i
# thus entire model (inputs, outputs, weights, biases, etc.) are integers
#converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int16

# TODO: is this OK for representative data gen for quantization? It may be just
#  https://www.tensorflow.org/lite/performance/post_training_integer_quant_16x8
def representative_data_gen():
    for _ in range(100):
        data = np.random.rand(1, 9, 8, 2)
        #yield [data.astype(np.int8)]
        yield [data.astype(np.float32)]

converter.representative_dataset = representative_data_gen

tflite_16x8_model = converter.convert()
```

```
INFO:tensorflow:Assets written to: /tmp/tmp4ann_fb5/assets
```

Can save/export Keras model weights to an `h5` file format, which can then be read with basic Python using the h5py package; we can dynamically traverse the h5 structure to pull out the weights for each layer.

Convolutional layer descriptions:

- Convolutional Networks- MIT Deep Learning Book <- tons of great reference here for documentation
- A Comprehensive Introduction to Different Types of Convolutions in Deep Learning
- Intuitively Understanding Convolutions for Deep Learning
- Calculating Parameters of Convolutional and Fully Connected Layers with Keras
- Convolutional Neural Networks- Intel YouTube
- Convolutions in Image Processing- MIT YouTube

Can also manually extract, and prototype, weights using Keras API: Or even using Netron

In [30]:
```
model.save_weights("weights.h5")

tflite_models_dir = pathlib.Path("quantized_tflite_model/")
tflite_models_dir.mkdir(exist_ok=True, parents=True)
tflite_model_16x8_file = tflite_models_dir/"abf_model_quant_16x8.tflite"
tflite_model_16x8_file.write_bytes(tflite_16x8_model)
```

Out[30]:
```
5384
```

References

[1] Van Trees, H.L. *Optimum Array Processing*. New York, NY: Wiley-Interscience, 2002.

[2] Guerci, J.R. *Space-Time Adaptive Processing for Radar*, 2nd ed. Boston, MA: Artech House, 2015.

Appendix B

VHDL Design Source

B.1 Miscellaneous/Support VHDL Entities

```vhdl
-- Package for common utilities
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
  use ieee.math_real.all;
  use std.textio.all;

package util_pkg is

-- // Start: Common Types /////////////////////////////////////////////////////
  -- VHDL-2008 unbounded array definitions
  type T_slv_2D      is array (integer range <>) of std_logic_vector;
  type T_signed_2D   is array (integer range <>) of signed;
  type T_unsigned_2D is array (integer range <>) of unsigned;
  type T_int_2D      is array (integer range <>) of integer;
  type T_slv_3D      is array (integer range <>) of T_slv_2D;
  type T_signed_3D   is array (integer range <>) of T_signed_2D;
  type T_unsigned_3D is array (integer range <>) of T_unsigned_2D;
  type T_int_3D      is array (integer range <>) of T_int_2D;
-- // End: Common Types ///////////////////////////////////////////////////////

-- // Start: File I/O Utilities ///////////////////////////////////////////////
  impure function F_read_file_slv_2D( file_path  : string;
                                      slv_length : integer;
                                      dim_length : integer ) return T_slv_2D;
-- // End: File I/O Utilities /////////////////////////////////////////////////

-- // Start: String Utilities /////////////////////////////////////////////////
  -- Converts a String to std_logic_vector
  function F_string_to_slv( X : string ) return std_logic_vector;
-- // End: String Utilities ///////////////////////////////////////////////////

-- // Start: Number Utilities /////////////////////////////////////////////////
  function F_return_smaller( A : integer;
                             B : integer ) return integer;
  function F_return_larger( A : integer;
                            B : integer ) return integer;

  function   F_clog2( x : real )    return integer;
  function   F_clog2( x : natural ) return integer;
  function F_is_even( x : integer ) return boolean;
  function  F_is_odd( x : integer ) return boolean;

  function F_FFS_bit( x : std_logic_vector ) return integer;
  function F_FFS_bit( x : signed ) return integer;
  function F_FFS_bit( x : unsigned ) return integer;
-- // End: Number Utilities ///////////////////////////////////////////////////

end util_pkg;

package body util_pkg is

-- // Start: File I/O Utilities ///////////////////////////////////////////////
  -- Reads an ASCII file with bit-vector patterns on each line where:
  --   + each line has a single binary value of length ‘slv_length‘
  --   + reads up to ‘dim_length‘ lines of file
  -- e.x. a file with values ‘0‘, ‘1‘, and ‘7‘ is:
  --      00000000
  --      00000001
  --      00000111
  impure function F_read_file_slv_2D( file_path  : string;
                                                 slv_length : integer;
                                                 dim_length : integer )
    return T_slv_2D is
    file     fd       : text;
    variable V_line   : line;
    variable V_bitvec : bit_vector(slv_length - 1 downto 0);
    variable V_return : T_slv_2D(dim_length - 1 downto 0)(
    slv_length - 1 downto 0)
                            := (others => (others => '0'));
  begin
    if file_path /= "" then
      file_open( fd, file_path, read_mode );
      for i in 0 to dim_length - 1 loop
        readline( fd, V_line );
        read( V_line, V_bitvec );
        V_return(i) := to_stdlogicvector( V_bitvec );
      end loop;
    end if;
    return V_return;
  end F_read_file_slv_2D;
-- // End: File I/O Utilities
    ////////////////////////////////////////////////

-- // Start: String Utilities
    ////////////////////////////////////////////////
  function F_string_to_slv( X : string ) return std_logic_vector
    is
    variable V_return : std_logic_vector((X'length*8)-1 downto 0);
  begin
    for i in X'range loop
      V_return(((i+1)*8)-1 downto i*8) :=
        std_logic_vector( to_unsigned( character'pos( X(i) ), 8 )
    );
    end loop;
    return V_return;
  end F_string_to_slv;
-- // End: String Utilities
    //////////////////////////////////////////////////

-- // Start: Number Utilities
    ////////////////////////////////////////////////
  function F_return_smaller( A : integer;
                             B : integer ) return integer is
  begin
    if A < B then
      return A;
    else
      return B;
    end if;
  end F_return_smaller;

  function F_return_larger( A : integer;
                            B : integer ) return integer is
  begin
    if A > B then
      return A;
    else
      return B;
    end if;
  end F_return_larger;

  function F_clog2( x : real ) return integer is
  begin
    return integer(ceil(log2(x)));
  end F_clog2;

  function F_clog2( x : natural ) return integer is
  begin
    return F_clog2(real(x));
  end F_clog2;

  function F_is_even( x : integer ) return boolean is
  begin
    return (x mod 2) = 0;
  end F_is_even;

  function F_is_odd( x : integer ) return boolean is
  begin
    return (x mod 2) = 1;
  end F_is_odd;

  -- Find First Set bit: returns the first set bit, respecting
  --   given SLV range direction (e.g. if x(2 downto 0) := "011",
  --   F_FFS_bit(x) would return index '1', however if defined as
  --   x(0 to 2) := "011", F_FFS_bit(x) returns index '0')
  function F_FFS_bit( x : std_logic_vector ) return integer is
  begin
    for i in x'range loop
      if x(i) = '1' then
        return i;
      end if;
    end loop;
```

```
    -- set bit not found (all 0's), return left-most index since
    this
    -- function is often used to decide how much to shift
    return x'left;
  end F_FFS_bit;

  function F_FFS_bit( x : signed ) return integer is
  begin
    return F_FFS_bit( std_logic_vector( x ) );
  end F_FFS_bit;

  function F_FFS_bit( x : unsigned ) return integer is
  begin
    return F_FFS_bit( std_logic_vector( x ) );
  end F_FFS_bit;
-- // End: Number Utilities
    ///////////////////////////////////////////////////

end util_pkg;
```

Listing B.1: VHDL Common Utilities Package

```
-- Complex Multilier:
--   The following code implements a parameterizable complex
    multiplier
--   The style described uses 4 DSP's to implement the direct
    complex multiply
--   which can be optimized for a given architecture pipeline
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;

entity complex_multiply_mult4 is
  generic (
    G_AWIDTH : natural := 16;     -- size of 1st input of
    multiplier
    G_BWIDTH : natural := 18;     -- size of 2nd input of
    multiplier
    G_CONJ_A : boolean := false; -- take complex conjugate of arg
    A
    G_CONJ_B : boolean := false   -- take complex conjugate of arg
    B
  );
  port (
    clk       : in  std_logic;
    reset     : in  std_logic := '0'; -- (optional) sync reset for
    *valid's
```

```vhdl
    ab_valid : in  std_logic; -- A & B complex input data valid
    ar       : in  signed(G_AWIDTH - 1 downto 0); -- 1st input's real part
    ai       : in  signed(G_AWIDTH - 1 downto 0); -- 1st input's imaginary part
    br       : in  signed(G_BWIDTH - 1 downto 0); -- 2nd input's real part
    bi       : in  signed(G_BWIDTH - 1 downto 0); -- 2nd input's imaginary part
    p_valid  : out std_logic; -- Product complex output data valid
    pr       : out signed(G_AWIDTH + G_BWIDTH downto 0); -- real part of output
    pi       : out signed(G_AWIDTH + G_BWIDTH downto 0)  -- imaginary part of output
  );
end complex_multiply_mult4;

architecture rtl of complex_multiply_mult4 is

  signal ar_q, ai_q                          : signed(G_AWIDTH - 1 downto 0) := (others => '0');
  signal br_q, bi_q                          : signed(G_BWIDTH - 1 downto 0) := (others => '0');
  signal multr0, multr1, multi0, multi1 : signed(G_AWIDTH + G_BWIDTH - 1 downto 0) := (others => '0');
  signal addr, addi                          : signed(G_AWIDTH + G_BWIDTH downto 0) := (others => '0');

  constant K_PIPE_DELAY : integer := 3; -- # clk cycles of pipeline delay through component
  signal sig_valid_sr   : std_logic_vector(K_PIPE_DELAY - 1 downto 0) := (others => '0');

begin

  pr      <= addr;
  pi      <= addi;
  p_valid <= sig_valid_sr(sig_valid_sr'high);

  S_reg_inputs: process(clk)
  begin
    if rising_edge(clk) then
      ar_q <= ar;
      if G_CONJ_A then
        ai_q <= -ai;
      else
```

```
        ai_q <= ai;
      end if;
      br_q <= br;
      if G_CONJ_B then
        bi_q <= -bi;
      else
        bi_q <= bi;
      end if;

      -- shift register to delay data valid to match pipeline
   delay
      if reset = '1' then
        sig_valid_sr <= (others => '0');
      else
        sig_valid_sr <= sig_valid_sr(K_PIPE_DELAY - 2 downto 0) &
   ab_valid;
      end if;
    end if;
  end process S_reg_inputs;

  -- Implements pr = (ar*br) - (ai*bi)
  S_real: process(clk)
  begin
    if rising_edge(clk) then
      multr0 <= ar_q * br_q;
      multr1 <= ai_q * bi_q;
      addr   <= resize( multr0, G_AWIDTH + G_BWIDTH + 1 ) - resize
   ( multr1, G_AWIDTH + G_BWIDTH + 1 );
    end if;
  end process S_real;

  -- Implements pi = (ar*bi) + (ai*br)
  S_imag: process(clk)
  begin
    if rising_edge(clk) then
      multi0 <= ar_q * bi_q;
      multi1 <= ai_q * br_q;
      addi   <= resize( multi0, G_AWIDTH + G_BWIDTH + 1 ) + resize
   ( multi1, G_AWIDTH + G_BWIDTH + 1 );
    end if;
  end process S_imag;

end architecture rtl;
```

Listing B.2: Complex Multiply Block

```
-- Parallel Adder Tree w/recursion (VHDL-2008)
```

```vhdl
--   inspired by: https://stackoverflow.com/a/50002251
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
library work;
  use work.util_pkg.all;

entity adder_tree is
  generic (
    G_DATA_WIDTH : natural := 16; -- sample bitwidth
    G_NUM_INPUTS : natural :=  8  -- number of input samples in vector
  );
  port (
    clk          : in  std_logic;
    reset        : in  std_logic := '0'; -- (optional) sync reset for *valid's
    -- input data valid across input row vector
    din_valid    : in  std_logic := '1';
    -- NOTE: input samples not registered
    din          : in  T_slv_2D(G_NUM_INPUTS - 1 downto 0)(G_DATA_WIDTH - 1 downto 0);

    dout_valid   : out std_logic;
    dout         : out std_logic_vector(F_clog2(G_NUM_INPUTS) + G_DATA_WIDTH - 1 downto 0)
  );
end adder_tree;

architecture rtl of adder_tree is
  constant K_NXT_NUM_INPUTS : natural := (G_NUM_INPUTS/2) + (G_NUM_INPUTS mod 2);

  -- registered adder outputs for next stage (+1 bit growth)
  -- NOTE: arbitrarily adding input slv's as unsigned since addition is same
  -- with sign extension and accounted overflow bit
  signal sig_nxt_din : T_unsigned_2D(K_NXT_NUM_INPUTS - 1 downto 0)(G_DATA_WIDTH downto 0)
                     := (others => (others => '0'));
  signal sig_nxt_slv : T_slv_2D(K_NXT_NUM_INPUTS - 1 downto 0)(G_DATA_WIDTH downto 0);
  signal sig_dvalid  : std_logic := '0';

begin

```

```
UG_unsigned_to_slv_2D: for i in sig_nxt_din'range generate
  sig_nxt_slv(i) <= std_logic_vector( sig_nxt_din(i) );
end generate UG_unsigned_to_slv_2D;

S_adder: process(clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      sig_dvalid <= '0';
    else
      if din_valid = '1' then
        for i in 0 to (G_NUM_INPUTS/2) - 1 loop
          sig_nxt_din(i) <= resize( unsigned( din(i*2) ), G_DATA_WIDTH + 1 ) +
                            resize( unsigned( din((i*2)+1) ), G_DATA_WIDTH + 1);
        end loop;

        if F_is_odd( G_NUM_INPUTS ) then -- account for odd input -> next stage
          sig_nxt_din(sig_nxt_din'high) <= resize( unsigned( din(din'high) ),
                                                   G_DATA_WIDTH + 1 );
        end if;
      end if;
      sig_dvalid <= din_valid;
    end if;
  end if;
end process S_adder;

UG_recurse: if F_clog2( G_NUM_INPUTS ) > 1 generate
  U_next_adder_stage: entity work.adder_tree
    generic map (
      G_DATA_WIDTH => G_DATA_WIDTH + 1,
      G_NUM_INPUTS => K_NXT_NUM_INPUTS
    )
    port map (
      clk          => clk,
      reset        => reset,
      din_valid    => sig_dvalid,
      din          => sig_nxt_slv,
      dout_valid   => dout_valid,
      dout         => dout
    );
end generate UG_recurse;
```

```

  UG_final_stage: if F_clog2( G_NUM_INPUTS ) = 1 generate
    dout_valid <= sig_dvalid;
    dout       <= std_logic_vector( sig_nxt_din(0) );
  end generate UG_final_stage;

end architecture rtl;
```

Listing B.3: Generic Parallel Adder Tree

```
-- CORDIC logic with output scaling to cancel out CORDIC gain (via
    CORDIC_scale)

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;

entity cordic_rot_scaled is
  generic (
    G_ITERATIONS : natural := 16 -- also equates to output
   precision
  );
  port (
    clk          : in  std_logic;
    reset        : in  std_logic := '0'; -- (optional) sync reset
   for *valid's
    valid_in     : in  std_logic;
    x_in         : in  signed(G_ITERATIONS - 1 downto 0);
    y_in         : in  signed(G_ITERATIONS - 1 downto 0);
    angle_in     : in  unsigned(31 downto 0);              -- 32b
   phase_in (0-360deg)
    CORDIC_scale : in  signed(G_ITERATIONS - 1 downto 0) := X"4DBA
   ";

    valid_out    : out std_logic;
    cos_out      : out signed(G_ITERATIONS - 1 downto 0); --
   cosine/x_out
    sin_out      : out signed(G_ITERATIONS - 1 downto 0)  -- sine/
   y_out
  );
end entity cordic_rot_scaled;

architecture rtl of cordic_rot_scaled is

  component cordic is
    generic (
      G_ITERATIONS : natural := 16 -- also equates to output precision
    );
    port (
      clk          : in  std_logic;
      reset        : in  std_logic := ’0’; -- (optional) sync reset for *valid’s
      valid_in     : in  std_logic;
      x_in         : in  signed(G_ITERATIONS - 1 downto 0);
      y_in         : in  signed(G_ITERATIONS - 1 downto 0);
      angle_in     : in  unsigned(31 downto 0);              -- 32b phase_in (0-360deg)

      valid_out    : out std_logic;
      cos_out      : out signed(G_ITERATIONS - 1 downto 0); -- cosine/x_out
      sin_out      : out signed(G_ITERATIONS - 1 downto 0)  -- sine/y_out
    );
  end component cordic;

  signal sig_valid_out : std_logic := ’0’;
  signal sig_cos_out   : signed(G_ITERATIONS - 1 downto 0); -- cosine/x_out
  signal sig_sin_out   : signed(G_ITERATIONS - 1 downto 0); -- sine/y_out

  signal sig_scl_valid : std_logic := ’0’;
  signal sig_cos_scl   : signed((2*G_ITERATIONS) - 1 downto 0);
  signal sig_sin_scl   : signed((2*G_ITERATIONS) - 1 downto 0);

  signal sig_sft_valid : std_logic := ’0’;
  signal sig_cos_sft   : signed(G_ITERATIONS - 1 downto 0);
  signal sig_sin_sft   : signed(G_ITERATIONS - 1 downto 0);

begin

  valid_out <= sig_sft_valid;
  cos_out   <= sig_cos_sft;
  sin_out   <= sig_sin_sft;

  U_CORDIC_rotation: cordic
    generic map (
      G_ITERATIONS => G_ITERATIONS
    )
    port map (
      clk            => clk,
      reset          => reset,
      valid_in       => valid_in,
      x_in           => x_in,
      y_in           => y_in,
      angle_in       => angle_in,
      valid_out      => sig_valid_out,
      cos_out        => sig_cos_out,
      sin_out        => sig_sin_out
    );

  S_scale_magnitudes: process(clk)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        -- NOTE: mostly we need only reset registers related to
   handshaking/dataflow,
        --       which will aid in easing timing (less reset
   routing required than
        --       resetting the wider, data output registers)
        sig_scl_valid <= '0';
        sig_sft_valid <= '0';
      else
        -- normalize/cancel CORDIC gain using given scale factor
        if sig_valid_out = '1' then
          sig_cos_scl <= sig_cos_out * CORDIC_scale;
          sig_sin_scl <= sig_sin_out * CORDIC_scale;
        end if;
        sig_scl_valid <= sig_valid_out;

        -- scale normalized CORDIC magnitude back down to
   operational data width
        if sig_scl_valid = '1' then
          -- since scaling & data are always of same data width,
   can simply shift right
          -- by >> G_ITERATIONS value (-1 data width since given
   signed scale factor)
          sig_cos_sft <= resize( shift_right( sig_cos_scl,
                                              G_ITERATIONS - 1 ),
                                 sig_cos_sft'length );
          sig_sin_sft <= resize( shift_right( sig_sin_scl,
                                              G_ITERATIONS - 1 ),
                                 sig_sin_sft'length );
        end if;
        sig_sft_valid <= sig_scl_valid;

      end if;
    end if;
  end process S_scale_magnitudes;

end architecture rtl;
```

Listing B.4: Gain-Scaled CORDIC Rotator

```
-- CORDIC logic with output scaling to cancel out CORDIC gain (via
     CORDIC_scale)

library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;

entity cordic_vec_scaled is
  generic (
    G_ITERATIONS : natural := 16 -- also equates to output
   precision
  );
  port (
    clk          : in  std_logic;
    reset        : in  std_logic := '0'; -- (optional) sync reset
   for *valid's
    valid_in     : in  std_logic;
    x_in         : in  signed(G_ITERATIONS - 1 downto 0);
    y_in         : in  signed(G_ITERATIONS - 1 downto 0);
    CORDIC_scale : in  signed(G_ITERATIONS - 1 downto 0) := X"4DBA
   ";

    valid_out    : out std_logic;
    phase_out    : out unsigned(31 downto 0); -- 32b phase (0-360
   deg)
    mag_out      : out signed(G_ITERATIONS - 1 downto 0)
  );
end entity cordic_vec_scaled;

architecture rtl of cordic_vec_scaled is

  component cordic_vec is
    generic (
      G_ITERATIONS : natural := 16 -- also equates to output
   precision
    );
    port (
      clk          : in  std_logic;
```

```vhdl
      reset         : in  std_logic := ’0’; -- (optional) sync reset for *valid’s
      valid_in      : in  std_logic;
      x_in          : in  signed(G_ITERATIONS - 1 downto 0);
      y_in          : in  signed(G_ITERATIONS - 1 downto 0);

      valid_out     : out std_logic;
      phase_out     : out unsigned(31 downto 0); -- 32b phase (0-360deg)
      mag_out       : out signed(G_ITERATIONS - 1 downto 0)
    );
  end component cordic_vec;

  signal sig_valid_out : std_logic := ’0’;
  signal sig_mag_out   : signed(G_ITERATIONS - 1 downto 0);
  signal sig_phase_out : unsigned(31 downto 0);

  signal sig_scl_valid : std_logic := ’0’;
  signal sig_mag_scl   : signed((2*G_ITERATIONS) - 1 downto 0);
  signal sig_phase_q   : unsigned(31 downto 0);

  signal sig_sft_valid : std_logic := ’0’;
  signal sig_mag_sft   : signed(G_ITERATIONS - 1 downto 0);
  signal sig_phase_qq  : unsigned(31 downto 0);

begin

  valid_out <= sig_sft_valid;
  phase_out <= sig_phase_qq;
  mag_out   <= sig_mag_sft;

  U_CORDIC_vectoring: cordic_vec
    generic map (
      G_ITERATIONS => G_ITERATIONS
    )
    port map (
      clk          => clk,
      reset        => reset,
      valid_in     => valid_in,
      x_in         => x_in,
      y_in         => y_in,

      valid_out    => sig_valid_out,
      phase_out    => sig_phase_out,
      mag_out      => sig_mag_out
    );

  S_scale_magnitudes: process(clk)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        -- NOTE: mostly we need only reset registers related to
   handshaking/dataflow,
        --       which will aid in easing timing (less reset
   routing required than
        --       resetting the wider, data output registers)
        sig_scl_valid <= '0';
        sig_sft_valid <= '0';
      else
        -- normalize/cancel CORDIC gain using given scale factor
        if sig_valid_out = '1' then
          sig_mag_scl <= sig_mag_out * CORDIC_scale;
        end if;
        sig_scl_valid <= sig_valid_out;
        -- since we don't care about scaling phase (for now,
   interacts with
        -- other CORDIC/trig functions at full 32b width) just
   pipeline to
        -- match delay of scale & shift of magnitude signal
        sig_phase_q   <= sig_phase_out;

        -- scale normalized CORDIC magnitude back down to
   operational data width
        if sig_scl_valid = '1' then
          -- since scaling & data are always of same data width,
   can simply shift right
          -- by >> G_ITERATIONS value (-1 data width since given
   signed scale factor)
          sig_mag_sft <= resize( shift_right( sig_mag_scl,
                                              G_ITERATIONS - 1 ),
                                 sig_mag_sft'length );
        end if;
        sig_sft_valid <= sig_scl_valid;
        sig_phase_qq  <= sig_phase_q;

      end if;
    end if;
  end process S_scale_magnitudes;

end architecture rtl;
```

Listing B.5: Gain-Scaled CORDIC Vectoring