SONAR PROCESSING

George W.P. York, PhD, PE
Department of Electrical and Computer Engineering
U.S. Air Force Academy, CO 80840

INTRODUCTION

Sonar, radar, and medical ultrasound imaging share many of the same signal processing stages; only the frequencies and the type of wave (sound versus electro-magnetic) are different. Since sonar is at the lowest frequency of the three it is much more practical to work with. A lower frequency permits less expensive equipment due to a much lower sample rate and processing rate, driving down the cost of all stages. Working in this audio range also has the added benefit of allowing the student to hear the pulses, which can help students gain further intuition about the signals. The sonar application serves as an excellent example of DSP found in many systems. This reading will cover the basic signal processing stages in sonar/radar including phased-array transmit/receive, calibration, time-gain compensation, noise filtering, upsampling, beamforming, quadrature demodulation, scan conversion, image processing, contrast enhancement, and Doppler frequency estimation.

BACKGROUND

Dr Musselman's *Radar in the Electrical Engineering Major* reading [1] (posted on the ece434 website) provides a good background on the basics of radar/sonar. You probably already know that radar basically works by transmitting pulses which travel through the air until reflected by objects. Based on how long it takes the pulse to travel out and back we know precisely how far away it is (range) assuming sound or electromagnetic waves travel at a constant velocity. The frequency of the wave and number of cycles transmitted per pulse determine the range resolution. These range/resolution equations are summarized in Dr Musselman's article.

How do we steer the pulse or determine the angle the object is away from us? The first systems had transducers that transmitted/received pulses that traveled along a narrow beam. If the transducers were mounted on a stepper motor, we could transmit/receive a pulse along one beam line, then move the motor one step (change the angle) and transmit/receive a pulse along another beam line. Thus the position of the stepper motor determined the angle of the object and the finer the transducers beam, the better resolution (in angle) that could be achieved.

Later systems used a *phased array* to electrically steer the beams. The mechanical stepper motor has a disadvantage of moving parts which tend to have more maintenance and calibration problems compared to pure electrical systems. Additionally, the phased array could switch angle faster. Phased arrays are a linear array of transducers that basically work by changing the timing (phase delay) of when each transducer transmitted/received a pulse relative to the other transducers. Beams were then formed were the waves constructively interfered (destructively interfering everywhere else) [2]. If the transducers are equally spaced at a ½ wavelength apart

(or less) then only one beam is formed. See the demo **beamsteer4_halfwave.m** on the EE434 website. At other spacings, multiple beams would form (known as grating-lobes) making it impossible to resolve which beam angle a pulse returned from. See **beamsteer4_3halfwave.m** on the EE434 website. For a good explanation of beamforming and phased arrays see Tyler Gilbert's EE499 final report [3] (posted on the EE434 website). The technique of *beamforming* we will use is a simple algorithm call delay-sum [2] and is discussed later.

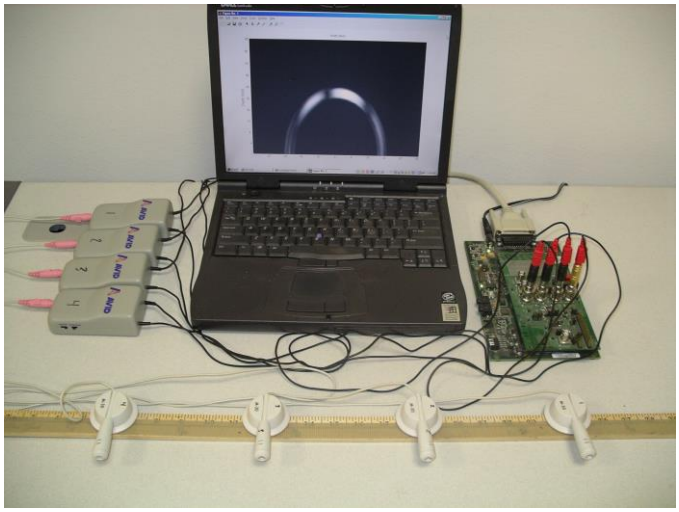

Figure 1. Our sonar system consists of the C6711 DSK board, the THS1206 4-channel ADC daughter card (right), 4 preamplifiers (left), and 4 microphone array spaced at ½ wavelength for a 1 kHz signal. (front). MATLAB is used to display the final echo image (B-Mode) on the laptop.

HARDWARE

The hardware platform we use is shown in Figure 1. This includes the Texas Instruments C6711 DSK board, which makes use of the VLIW/SIMD architecture of the TMS320C6711 DSP microprocessor. [3] For the DSK, TI offers a 12-bit, four-channel, simultaneous sampling (Fs = 150 kHz/channel) ADC daughter-card compatible with the C6711 DSK, known as the THS1206 Evaluation Module (EVM). This is over-designed for our audio range (well above Nyquist) since we are using 10 kHz sonic pulses. Four small battery-powered preamplifiers provide the correct signal level from the microphones to the ADC. Four microphones are used as the input transducers. Figure 1 shows them spaced at a half-wavelength for a 1000 Hz pulse (They must be 10 times closer together for 10 kHz pulse used in CPX3). We do not currently have 4 output channels/transducers to support beamsteering on transmit, so instead we attempt to transmit a less optimal omni-directional pulse using one output transducer (speaker). MATLAB running on the laptop controls the speakers and 4 channel input array, collects the data from the DSK board, and then does all the sonar processing, resulting in less than real-time performance. In the future we plan to move some of the processing down to the 6711 DSP chip (where it belongs) for real-time performance requiring coding in the C programming language.

SONAR PROCESSING STAGES

The Sonar processing stages/algorithms are summarized in a block diagram in Figure 2 (see **sonar_block_v4.pdf**). Initially all these processing stages can seem confusing, especially as the data transforms from the initial data collected from the 4 channel ADCs, $X_i[n]$, into beamformed data, $Beams[\theta,r]$, then into the scan converted echo image, $P[x,y]$. These data structures are illustrated in Figure 3. (see **sonar_data_structures.pdf**).

1) Omni-Directional Transmitter: With a single speaker available, we assume we transmit an omni-directional pulse. An omni-directional pulse radiates equally in all directions. *Is this a good assumption?* In this paper we will compare the result of using two different pulses

(1) **Signal A**: a simulated ideal 2.5-cycle 1 kHz sinusoid (range resolution ~2.5 ft)

(2) **Signal B**: a 6-cycle 10 kHz sinusoid (range resolution ~0.6 ft) transmitted live on our actual sonar system for each processing stage.

You can generate Signal B's figures by running `cpx3 sonar v2.m` (posted on the ECE434 website in cpx3_sonar.zip) which will look clearer than these printed images. To visualize what a 2.5-cycle 1 kHz sinusoid looks like see Figure 4. *Can you calculate the spatial resolution using the formula's from Dr Musselman's article? The speed of sound in air is approximately 1136 ft/sec.*

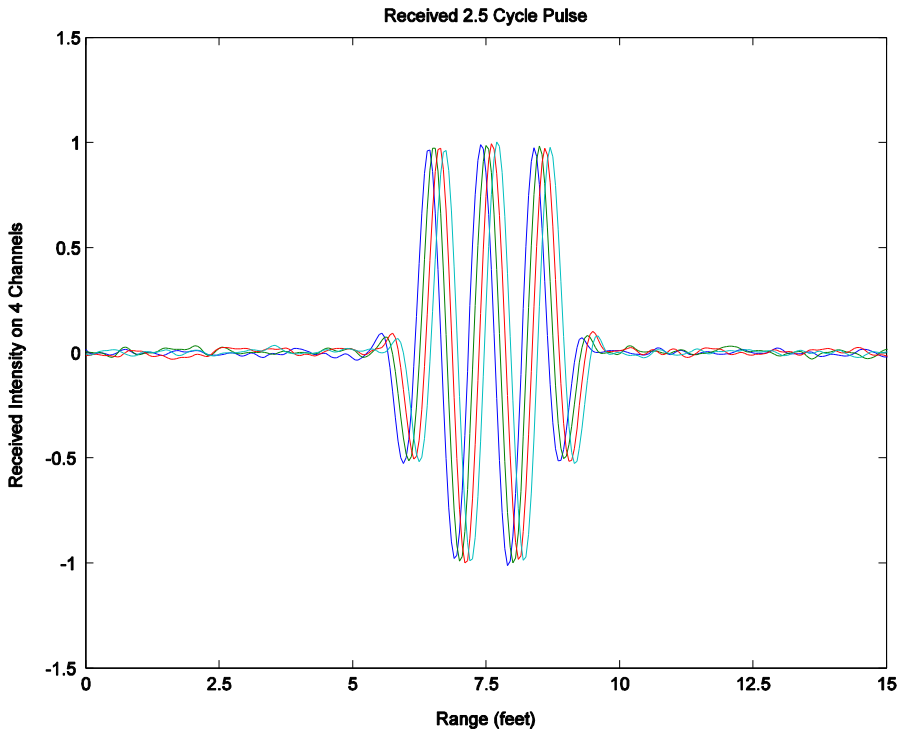

Figure 4. Signal A. Simulated reflected 2.5-cycle 1 kHz pulse received on the 4-channels after blanking, calibrating, low pass filtering and TGC.

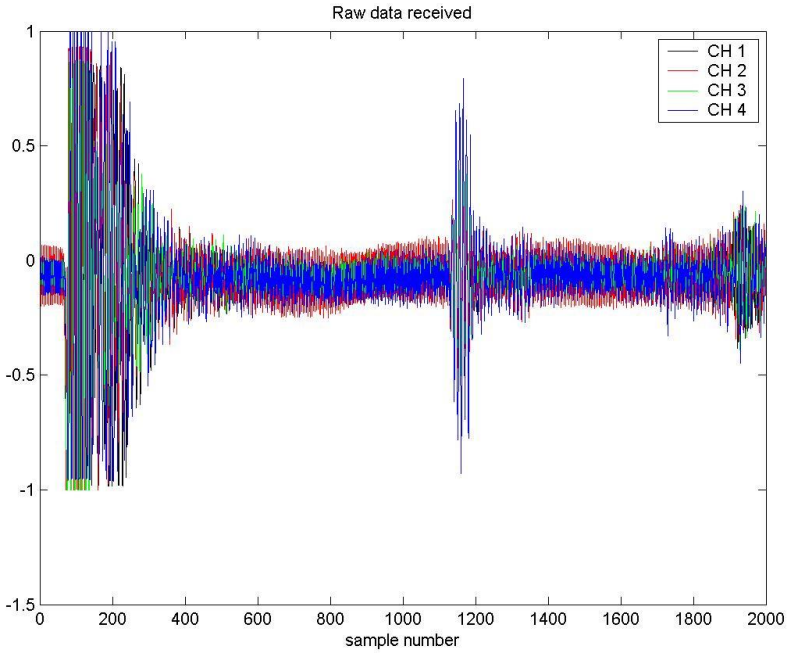

Figure 5. Signal B. Actual transmitted and reflected 6-cycle 10 kHz pulse received on the 4-channels before any processing. The transmitted pulse plus ringing is from samples 25 to 325. There are two objects causing reflections in this data at around samples 1150 and 1900.

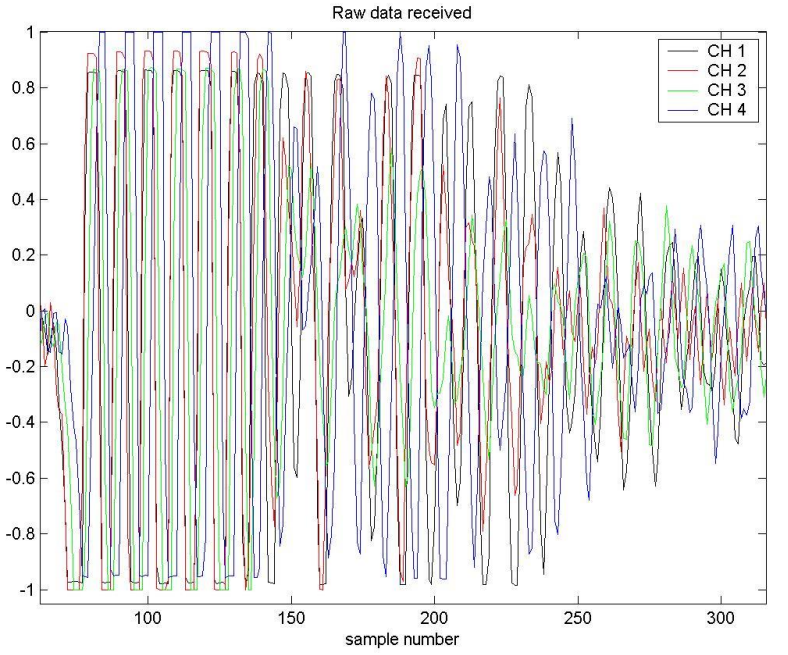

Figure 6. Signal B. Zooming in on Figure 5 to see the 6 cycles of the transmitted pulse (ending about sample 150) and ringing it caused in the system.

2) Self Calibration: We do not have a very precise means of controlling the timing of the start of the 4 channel ADCs sampling relative to when the pulse is transmitted. We start the ADCs collecting samples about the same time we ask the laptop to send a pulse out the audio port, collecting a total of 2000 samples for each channel at F_s = 100 kHz. Therefore the transmitted pulse is recorded in our samples with a slight delay. Since we only want to see reflected pulses, not transmitted pulses, we first detect when the transmitted pulse occurs in our samples (*How would you do this?*). We then blank out the transmitted pulse, then shift the data after this to time zero (*n=0*). In figure 5 and 6, even though we sent a 6 cycle pulse to the speaker, many more oscillations appear (*Why? How could we prevent this? The best solution is building better transducers that do not ring when hit with an impulse, i.e., have a good impulse response. Since we are stuck with this, we could use DSP to fix this like the Hubble telescope warped images were fixed. Hmmmm, deconvolving the pulse we want to transmit with the impulse response of the system? How does that work? How would we get the impulse response of this system? This could be one way to improve this system*). During the time of the transmitted pulse and the oscillations, our sonar system is blind (cannot see reflected pulses). Thus we blank out the samples during this time by setting them to zero. *If we blank out the first 300 samples, how far in distance from the array are we blind?* The result of blanking and shifting for the data in Figure 5 is shown in Figure 7. Note that just replacing these values with zeros causes a harsh transition similar to a rectangular window. (*Is this a problem? How could you improve this?*)

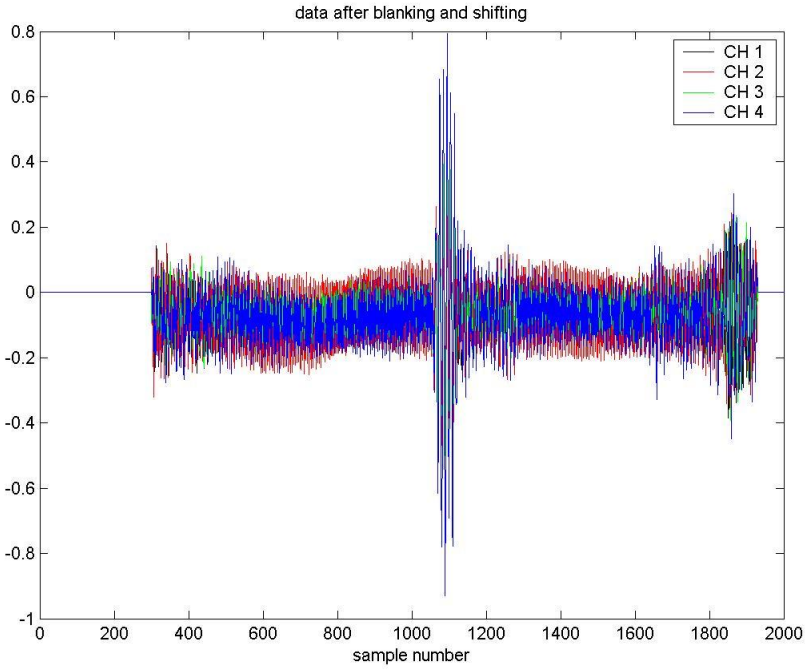

Figure 7. Signal B. Actual 6-cycle 10 kHz pulse data after blanking the transmitted pulse and shifting.

In a real radar/sonar system, the 4 channels sensors are calibrated to have the same amplitude (amplification) and no D.C. bias. (*if this is not done, could it be a problem?)* As you can see in Figure 6, some of our channels amplify the same receive pulse more than others and some have a D.C. bias (i.e., the wave is not centered about the y-axis). So for each channel *i*, we remove its

D.C. bias (*how would you do this?*). We then need to normalize the amplitude across the channels, we make the assumption that they should have all received the same signal (except time shifted), so we normalize them such that they all have the same average power. One method is to normalize each channel by the maximum root mean square (RMS).

$$RMS_i = \sqrt{\frac{\sum x_n^2}{N}} \tag{1}$$

$$Max\_RMS = max(RMS_i) \tag{2}$$

$$RMS_i = RMS_i / Max\_RMS \tag{3}$$

$$X_i[n] = X_i[n] / RMS_i \tag{4}$$

How would you test this DC bias removal and calibration with synthetic data? You would need to create test data for each channel that has the same basis function, but only differ by amplitude and bias. A sinusoid would work,

$$\text{test\_data}_i = A \cdot \cos\left(2\pi \frac{f}{Fs} n\right) + Vavg$$

where *A* and *Vavg* are slightly different for each channel. After DC removal and auto-calibration, the 4 sinusoids should be identical.

The result of this for signal B is shown in Figure 8.

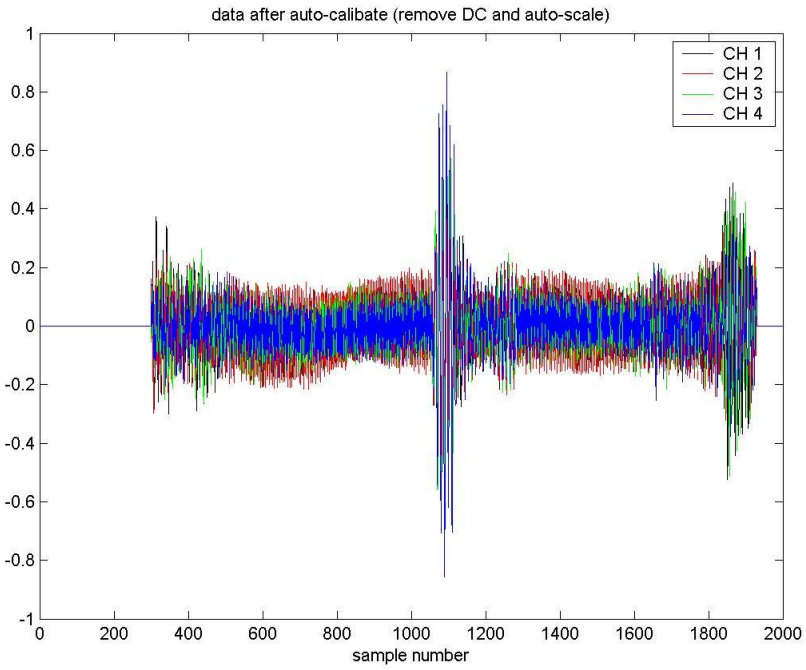

Figure 8. Signal B. Actual 6-cycle 10 kHz pulse data after calibrating for D.C Bias and equalizing the power in across the 4 channels.

3) Time Gain Compensation: Sound waves attenuate over distance similar to electromagnetic waves (see Dr Musselman's reading). [1] Assuming little absorption in air the intensity of our sound waves are primarily reduced by geometric spreading. Assuming our speaker (transducer) is omni-directional, the sound will lose intensity following the inverse-square law ($I \propto 1/r^2$). However, our speakers have some degree of directivity, so the loss is not that extreme and is best determined experimentally. *How would you do this?*

The time gain compensation (TGC) stage function is to compensate for the attenuation of the pulse and amplifies the channel data with respect to *n* (depth or time). Figure 9 is Signal B after TGC. Figure 10 is zoomed in on the first reflected pulse and Figure 11 is zoomed in on the second reflected pulse. (*Looking at the relative phase between the 4 channels, what is different about the objects generating the two reflected pulses?*)

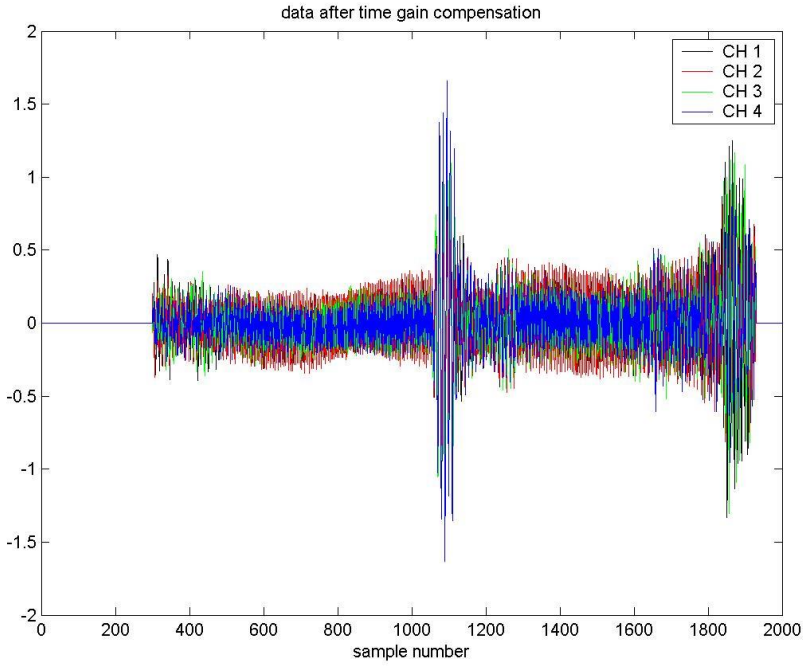

Figure 9. Signal B after Time Gain Compensation.

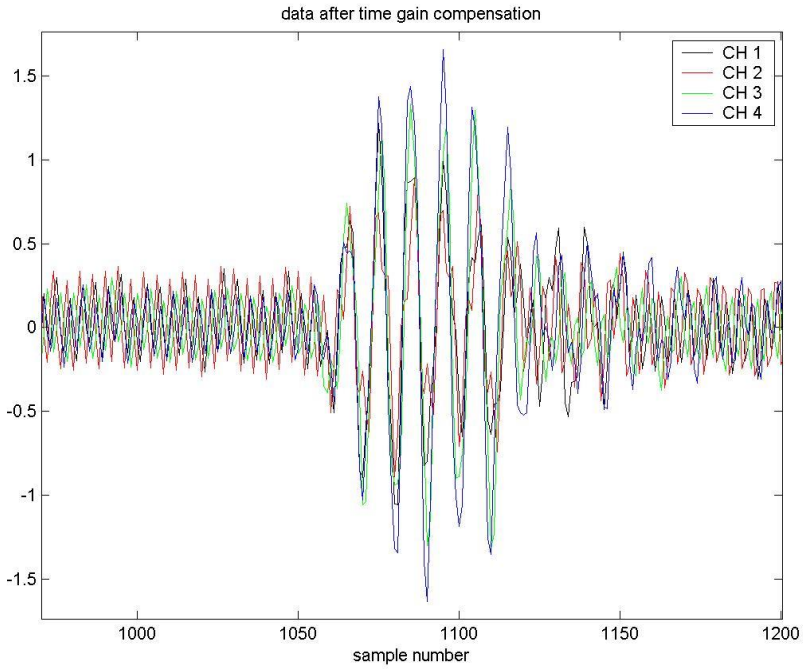

Figure 10. Signal B after Time Gain Compensation, zoomed in on the first reflected pulse. Note how the 6 cycles now appear.

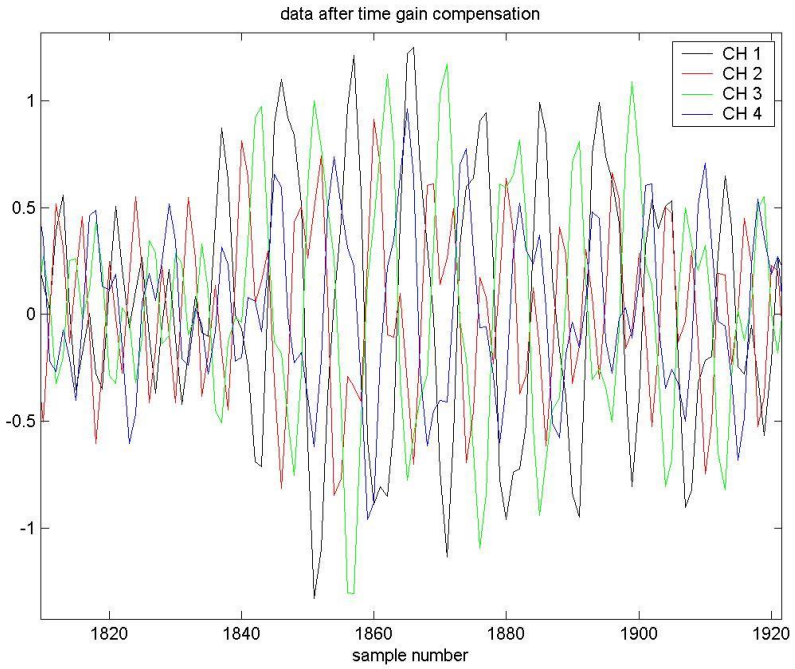

Figure 11. Signal B after Time Gain Compensation, zoomed in on the second reflected pulse.

4) Noise Removal and Bandwidth Limitation: Sonic images often have noise which is noticeable in Figure 10. Filters are applied to remove the noise and smooth the data, but without removing your signal. *Where is the signal in Figure 14?* Your signal is basically an AM modulated signal with a carrier at f_c with an upper and lower sideband. You want to preserve this and get rid of everything else (noise). Figure 12 and 13 show signal B after my noise smoothing. Figures 14 and 15 show the spectrum of Signal B before and after my noise smoothing. *Can you do better?*

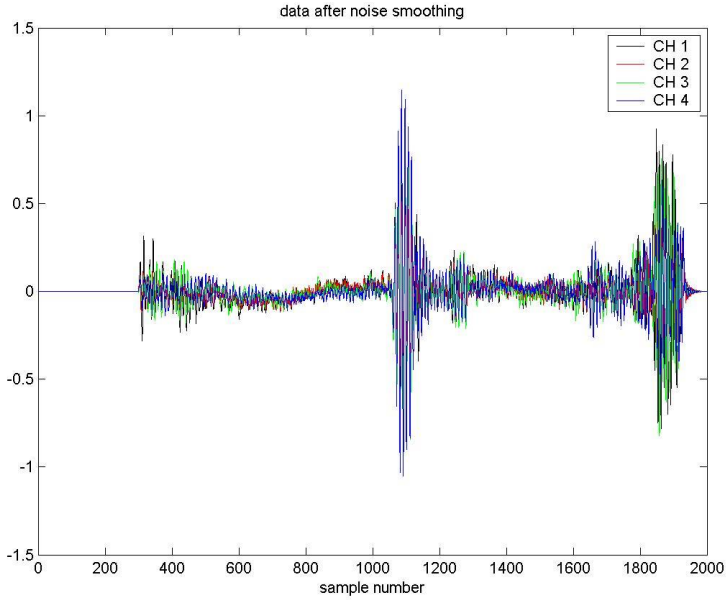

Figure 12. Signal B after Noise Smoothing.

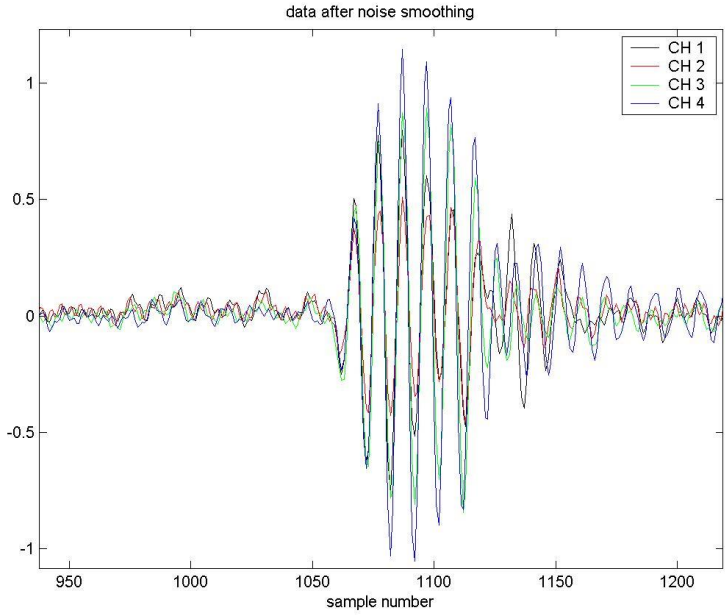

Figure 13. Signal B after Noise Smoothing, zoomed in on the first reflected pulse.

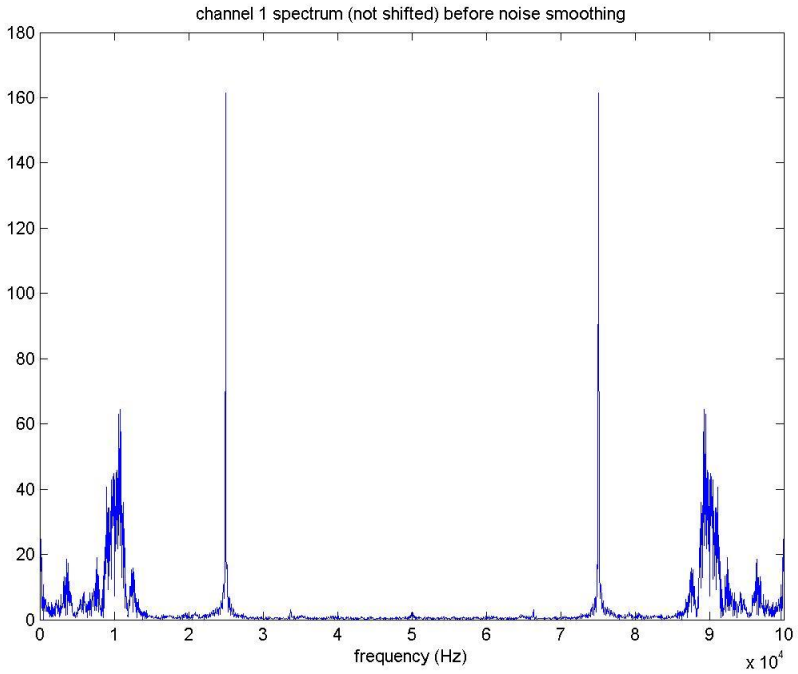

Figure 14. The spectrum of Signal B before Noise Smoothing.

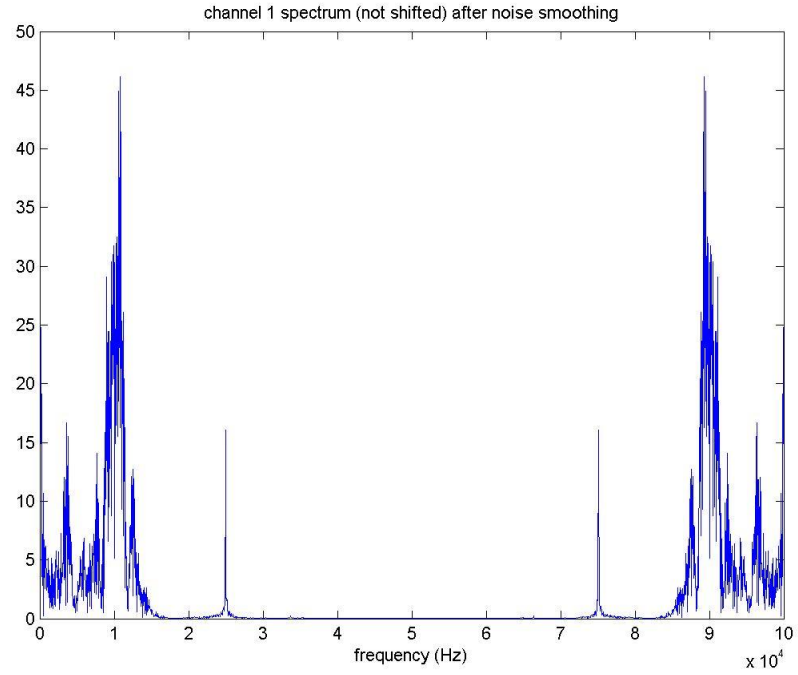

Figure 15. The spectrum of Signal B after Noise Smoothing.

.

5) Upsampling: In the next stage, beamforming, we will find that the number of beams we can create with delay-sum beamforming is a function of our sample rate. Doubling our sample rate approximately doubles the number of beams. Since our ADC's are set at a fixed sample rate, *how can we get a higher sample rate?* One area of DSP is known as *multi-rate signal processing.* We often have signals obtained at different sample rates and need to combine them. To do this we must change them to the same sample rate. We can reduce our signal's sample rate through a process known as *downsampling (or decimation)* and we can increase our sample rate through *upsampling (or interpolation).* See your textbook by Ifeachor and Jervis, section 9.2.1 explains downsampling and section 9.2.2 explains upsampling. After reading this, you now realize to upsample your single by factor L, you just need to add $L - 1$ zeros between each sample, then do a LPF with a $f_c = F_s/2$.

Note: You may not use MATLAB's *interp* or *upsample* functions. You must code your own upsampler!

Our simulated Signal A is already at a sample rate for the 21 beams we want in the next section, so no upsampling was required. However for Signal B to have 21 beams requires upsampling by 2 (i.e., $L = 2$). The result of upsampling Signal B to 200 kHz is shown in Figure 16 and the resulting spectrum in Figure 17. *From your understanding of upsampling (especially after seeing I&J Figure 9.4), in Figure 17 what are two little humps near 100 kHz caused by?)*

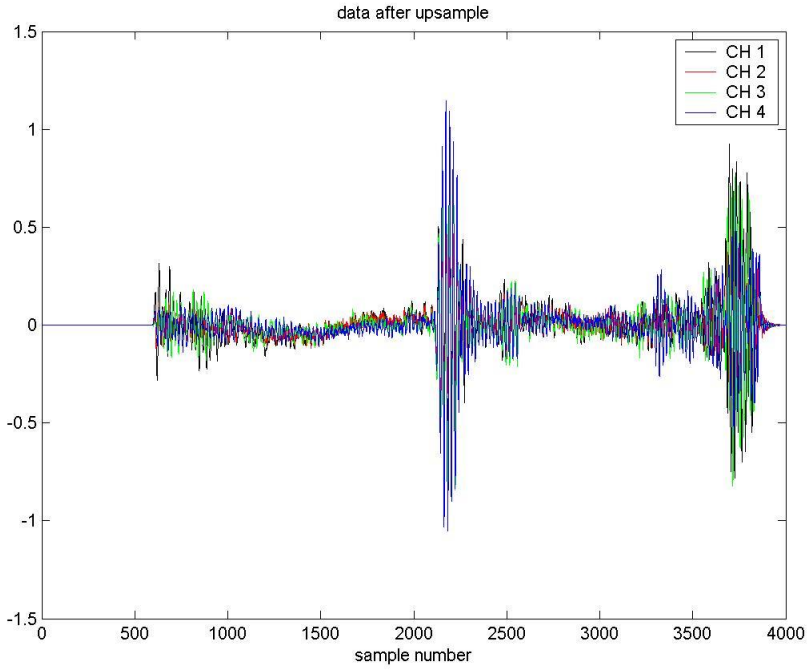

Figure 16. Signal B after upsampling by 2 to Fs = 200 kHz.

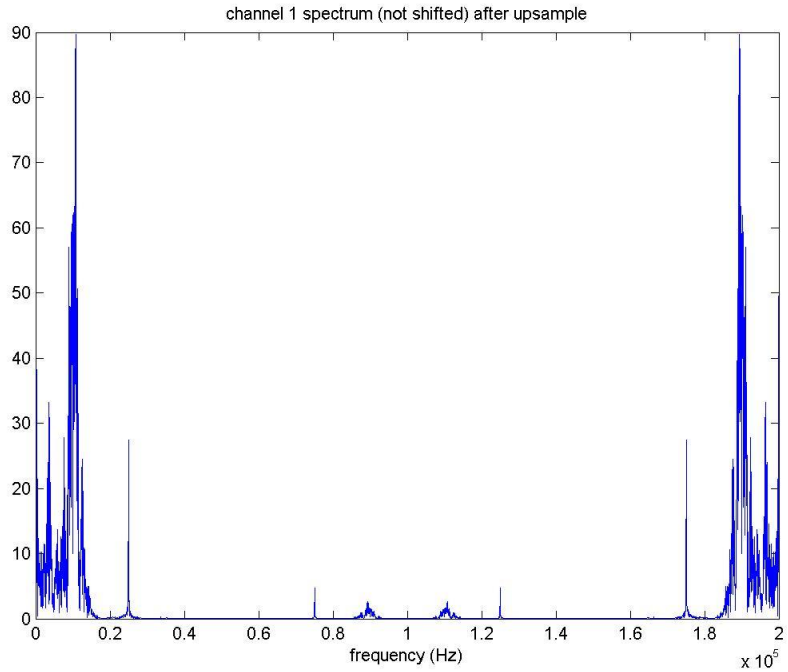

Figure 17. The spectrum of Signal B after upsampling by 2.

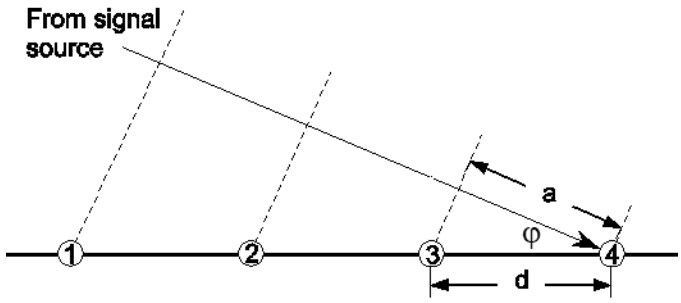

Figure 18. A linear sensor array consisting of four omni-directional microphones, labeled 1-4. $d = \lambda/2$

6) Phased-Array Receiver: Beamforming: Beamforming was bsriefly introduced in the beginning of this paper and is explained in detail in Tyler Gilbert's paper on the ECE434 website. [3] This section will focus on how we derive the number of Beams, how we implement delay-sum beamforming, and the results of beamforming on Signal's A and B.

With a four-channel system, a uniformly spaced linear sensor array consisting of four omni-directional microphones can be depicted as in Figure 18. We will use a simple beamforming algorithm known as delay-and-sum [2]. Assuming the sensor array is in the far field region of the signal source, the arriving wave fronts may be assumed to be linear. For each sample integer delay *k* (discrete phase shifts), the respective beam angle θ (angle normal to the sensor array axis in Figure 18) sensed by the array can be computed from

$$\sin\theta = \frac{a}{d} = 2k\frac{f_c}{F_s} \tag{5}$$

where f_c is the frequency of the signal and F_s is the sample frequency [2]. The number of beams is equal to $(F_s/f_c) + 1$. For example, for Signal A, we selected f_c = 1 kHz and F_s = 20 kHz, which leads to an ability to form 21 different beams for $-90° < \theta < 90°$. For Signal B (Figure 5) we selected f_c = 10 kHz and F_s = 100 kHz, then upsampled the data to F_s = 200 kHz which leads to an ability to form 21 different beams.

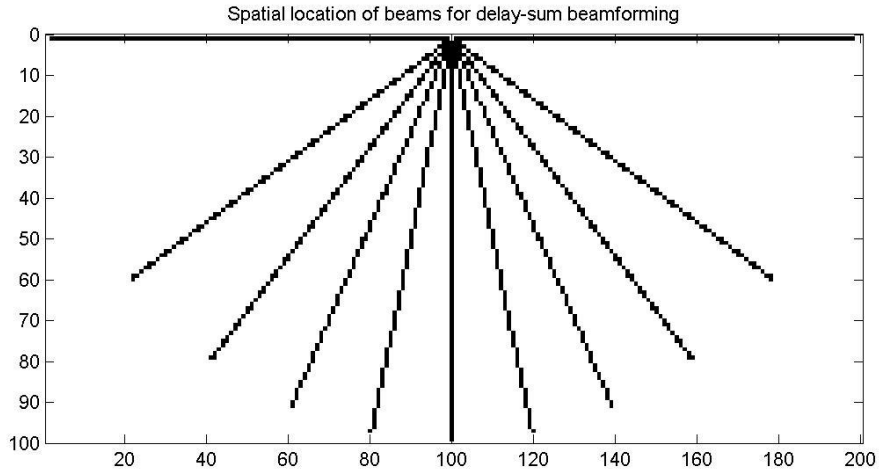

Figure 19. Plot of where the beams spatially lie for delay-sum beamforming, for the case when there are 11 beams. Note the angle spacing varies non-linearly, with finer resolution broadside to the array.

Figure 19 illustrates that due to the non-linear sine function, our beams are not equi-angle apart, but instead we have a finer resolution broadside to the array, and very poor resolution along the array axis. See Delay_Sum_Beams.m on the ECE434 website.

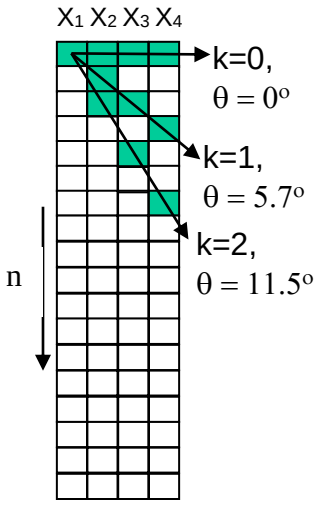

Figure 20. Illustration of the data structure for the 4 channels X_i (not all samples n shown) and the relative samples added at fixed delays (k) for three of the beams, k = 0 to 2.

Figure 20 helps explain how we calculate the beams from our 4 channels of sampled data. It is fairly obvious that to calculate the broadside beam (where $\theta = 0^o$ or k = 0), the sound hitting the microphones have no phase delay. So we just add the four samples together that were sampled at the same time, or

$$Beams[0^0, n] = X_1[n] + X_2[n] + X_3[n] + X_4[n] \quad \text{for all n}$$

Note: after beamforming we often refer to the index *r* (for range) instead of *n* (sample number). Now to calculate the 1st beam away from broadside (where k = 1 or $\theta = 5.7^o$), we delay each channel X_i by one sample (k = 1) relative to the previous channel (i.e., by the same phase delay), and add them together

$$Beams[5.7^0, n] = X_1[n] + X_2[n+1] + X_3[n+2] + X_4[n+4] \quad \text{for all n}$$

Now to calculate the 2nd beam away from broadside (where k = 2 or $\theta = 11.5^o$), we delay each channel X_i by two samples (k = 2) relative to the previous channel (i.e., by the same phase delay), and add them together

$$Beams[11.5^0, n] = X_1[n] + X_2[n+2] + X_3[n+4] + X_4[n+6] \quad \text{for all n}$$

Now to generalize this for all beams (and using the beam index *k* instead of angle θ), we get

$$Beams[k, n] = X_1[n] + X_2[n+k] + X_3[n+2*k] + X_4[n+3*k] \quad \text{for all n and all k} \tag{6}$$

Note: For the beams to the right of broadside (in Figure 18), k is a positive index reaching a maximum of $(F_s/f_c)/2$ or 10 for Signals A and B. For the beams to the left of broadside, k is a negative index reaching a minimum of $-(F_s/f_c)/2$ or -10 for Signals A and B.

Here are some practical hints for programming this delay-sum beamforming algorithm.

1. In the Beamforming equation (equation 6), for the upsampled Signal B, the input data X only has a range from n = 1:4000, however, when calculating Beams (which has output location n = 1:4000), the equation n+3k can go outside of n=1:4000 for the calculated Beams, which causes an index out of range error for X... or in other words, those Beam output locations are "undefined". The largest value for k is +/- 10. The smallest output n location that can be "defined" or calculated is when n+3(-10) = 1. The largest output n location that can be calculated is when n+3(10) = 4000. So what is the range of n that can be calculated for Beams? [n = smallest:largest]. What do you as the system designer do about the Beam output locations for n that are undefined, or are outside this range n = smallest:largest? What is a safe value you could write to these locations?
2. While you need to use k in the above beamforming equation 6 for the math, you cannot use k as an index to the 2D array called Beams in MATLAB, since k can be negative and all indexes must be greater than or equal to 1. So, I defined a new variable for the index call kk, where kk = k - 11. So kk's range is 1 to 21, while k's range is -10 to +10.

For Signal A's simulated 2.5-cycle 1 kHz pulse we limited sampling to 300 samples of the received echo signal (corresponding to a range depth of about 15 ft). This data is then beamformed from the data if Figure 4 into the image of the 21 beams (*x*-axis) versus depth (*y*-axis) before demodulation as shown in Figure 21. You can see the "hot spot" of the return wave pulse around range of 7.5 feet and k = 2 (or beam number 13). For Signal B the beamform plot is shown in Figure 22. The hotspot for one of the reflected pulses is around sample number 2200 and k = 0 (beam #11) and the other around sample number 3700 and k = 4 (beam #15). Note how must larger the 1 kHz pulse in Figure 21 is compared to the much finer resolution of the 10 kHz pulse in Figure 22.

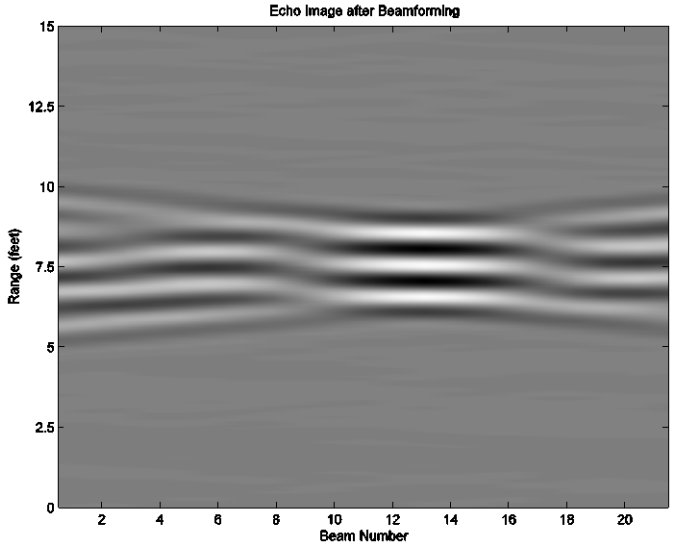

Figure 21. Image of reflected 2.5-cycle pulse for Signal A after beamforming and before demodulation. X-axis is 21 beams representing different angle θ; y-axis depth in feet. Beam number 11 is k=0. Beam number 21 is k=10. Beam number 1 is k = -10.

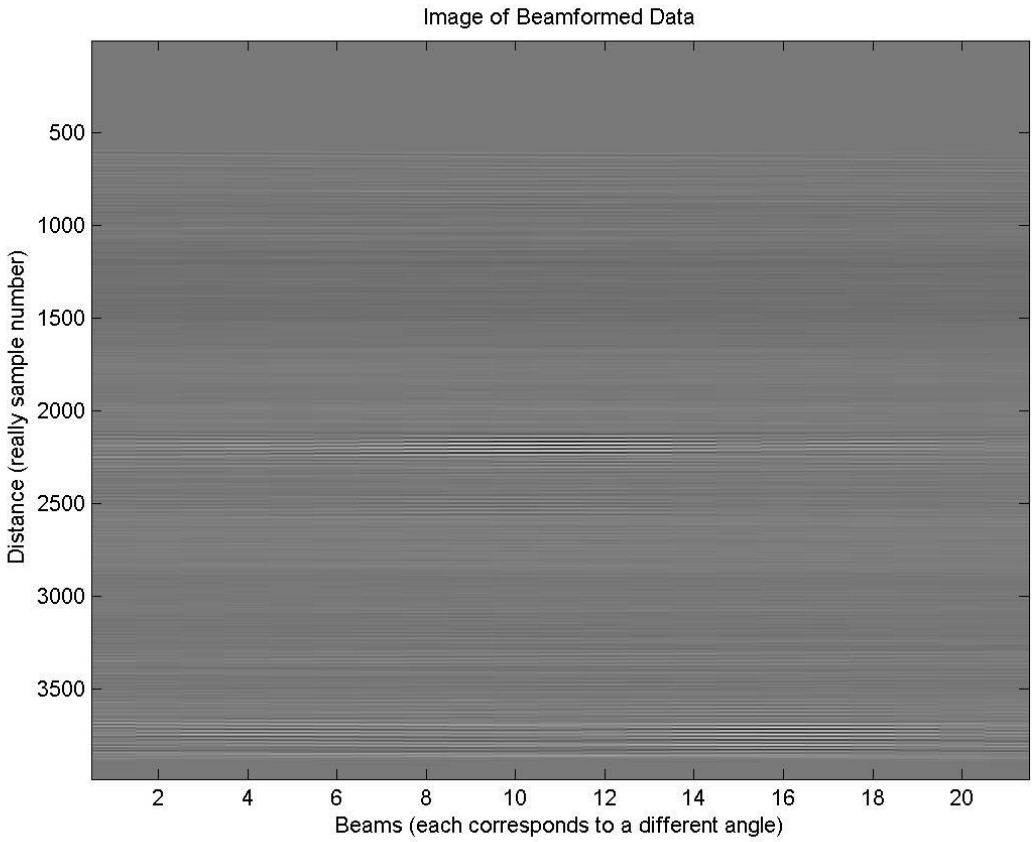

Figure 22. Image of Signal B after beamforming and before demodulation. X-axis is 21 beams representing different angle θ; y-axis depth in sample number. Beam number 11 is k=0. Beam number 21 is k=10. Beam number 1 is k = -10. (hard to see ripples due to printer quality… easier to see in MATLAB demo)

Another way to look at the beamformed data is shown in Figure 23. Figure 23 is a plot of just the two beams were the two reflected pulses like, corresponding to k = 0 and k = 4. *(what two angles θ do these correspond too?)*

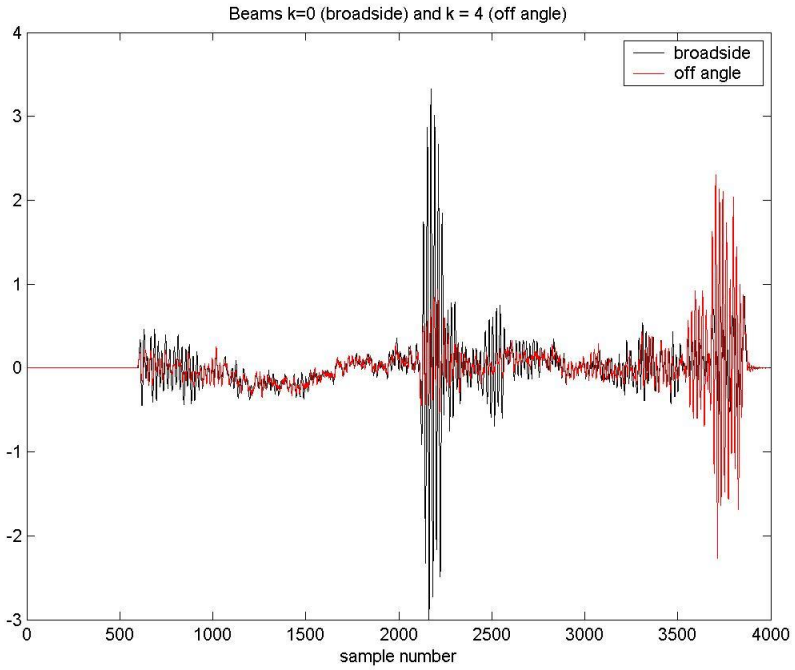

Figure 23. After beamforming, plot of Signal B's beam k=0 (black) and k=4 (red) where most of the energy for the two reflected pulses lie.

7) Demodulation: Demodulation is then used to remove the carrier frequency (1 kHz for Signal A; 10 kHz for Signal B) to recover the echo signal, very similar to AM demodulation (envelope detection) you may have learned about in ECE447. Our information signal is stored on the envelope of this carrier pulse. We use quadrature demodulation. In quadrature demodulation the received signal is multiplied by $\cos(2\pi f_c t) + j\sin(2\pi f_c t)$ resulting in a complex signal $I[\theta,r] + jQ[\theta,r]$, retaining both magnitude and phase of the signal [4]. You may have heard of synchronous demodulation (just multiplying by the cosine term). We cannot use synchronous demodulation because we cannot guarantee the returning echo pulse will be in-phase with this cosine. Quadrature demodulation overcomes this problem. After getting the complex signal $I[\theta,r] + jQ[\theta,r]$, A low pass filter is then used to remove the duplicate signal at $2f_c$, leaving the echo signal as the DC component.

The echo image is computed by taking the magnitude of the signal, $B[\theta,r] = \sqrt{I^2[\theta,r] + Q^2[\theta,r]}$, as shown for Signal A in Figure 24. Note the poor resolution using only 2.5 cycles of 1 kHz. The improved range resolution of Signal A's higher frequency pulse (6 cycles at 10 kHz) can be seen in Figure 25. The envelope of the k=0 and k=4 beams can be seen in Figure 26 (*compare to Figure 23).* The spectrum of beam k=0 after demodulation is shown in Figure 27, with the 10 kHz carrier removed.

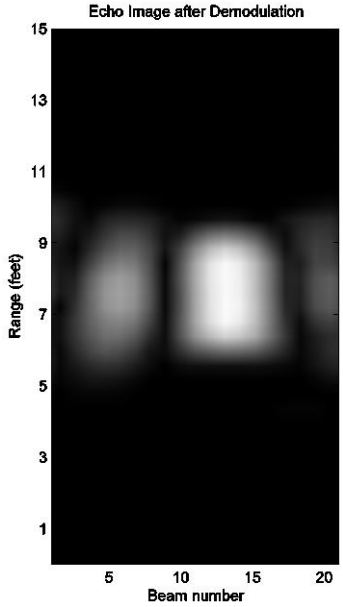

Figure 24. Image of Signal A's 2.5-cycle 1 kHz pulse after demodulation and 8:1 decimation in range. X-axis is 21 beams; y-axis is depth in feet.

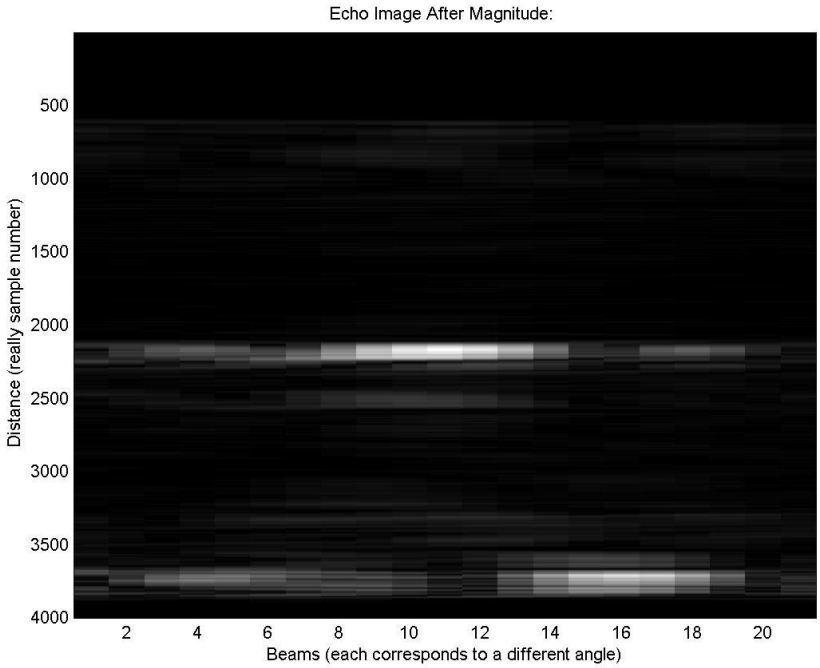

Figure 25. Image of Signal B's 6-cycle 10 kHz pulse after demodulation. X-axis is 21 beams; y-axis is the sample number, corresponding to range. The ECE434 students will convert the y-axis units to feet.

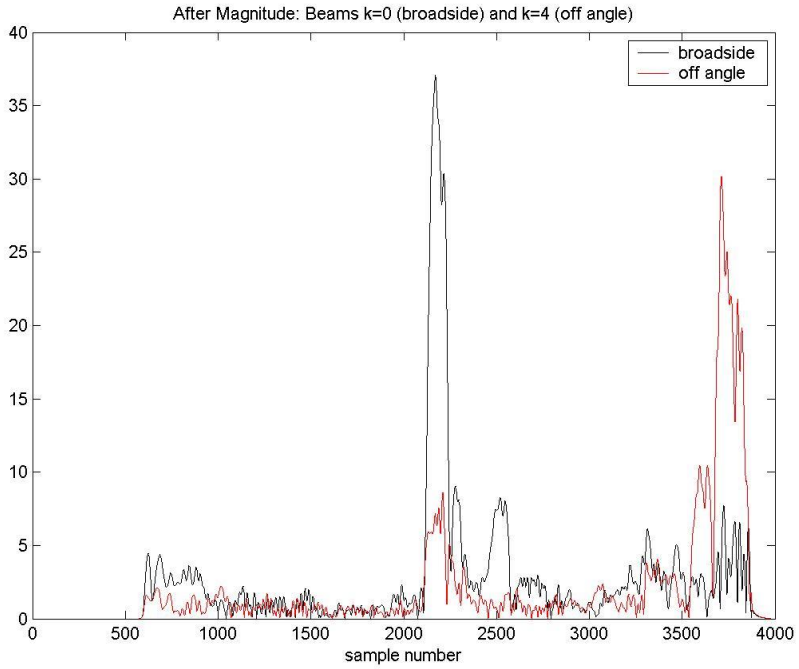

Figure 26. Plot of Signal B's beam k=0 (black) and k=4 (red) where most of the energy for the two reflected pulses lie after demodulation. Compare to Figure 23, and note this is the envelope of Figure 23.

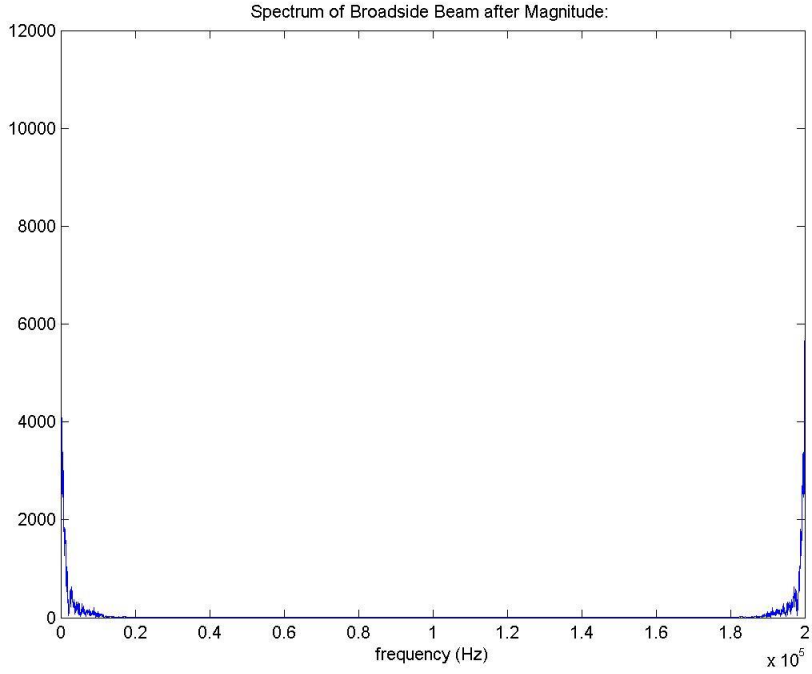

Figure 27. Spectrum of Signal B's beam k=0 after demodulation, with the 10 kHz carrier gone. The pulse signal is now moved down to baseband (D.C.)

How would you test Quad Demod with synthetic data? Since test_data is very noisy, you could create new clean Beam data [initialize to zero: Beams = zeros(21, 4000)] and add in multiple cycles of 10 KHz sinusoid sampled at 200 KHz to simulate a pulse, along the k=0 and k=4 beams, and let the pulse go multiple cycles, possibly from n = 1000:1500 for k = 0 and n = 3000:3500 for k=4.

8) Scan Conversion: Another practical consideration is the difficulty involved in implementing the simple geometric transformation of the beam data (stored in memory in rectangular coordinates $[\theta, r]$, as in Figure 24 and 25) into the proper polar coordinates for displaying the spatial echo image (Figure 28 and 29). Each Cartesian output pixel value *P[x,y]* must be interpolated from its respective surrounding polar vector data $B[\theta, r]$ by (1) calculating the address in memory (or array indexes) of the input data $B[\theta, r]$ (i.e., a polar conversion, requiring $\theta =$ *arctan(x/y)* and $r = \sqrt{x^2 + y^2}$ for each output pixel *P[x,y]*; (2) calculating the appropriate interpolation coefficient weights by the spatial fractional offset between each *P[x,y]* and its neighboring polar $B[\theta, r]$; and (3) computing a simple bi-linear interpolation [6] using the equation

$$P[x, y] = (1-\beta)\left[(1-\alpha)B[\theta, r] + \alpha B[\theta+1, r]\right] + \beta\left[(1-\alpha)B[\theta, r+1] + \alpha B[\theta+1, r+1]\right] \qquad (7)$$

where α is the fractional offset in the angle θ direction and β is the fractional offset in the range r direction, as illustrated in Figure 27. To speed up real-time computation, we recommend precomputing the computationally intense operations in steps 1 and 2 in a lookup table. Figure 28 shows the result of Signal A after scan conversion. Figure 29 is Signal B after scan conversion.

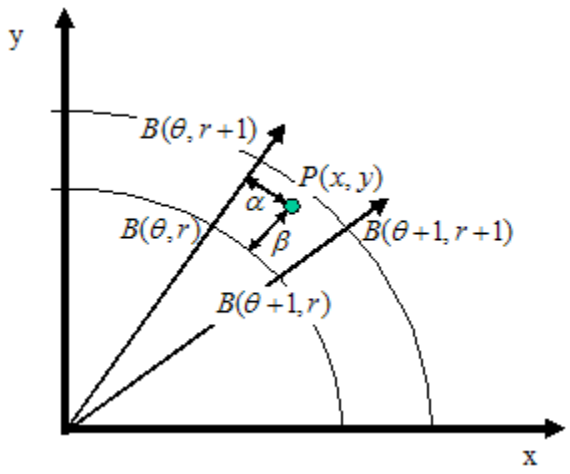

Figure 27. Interpolation for Scan Conversion.
$\theta = 0^o$ is along the y-axis

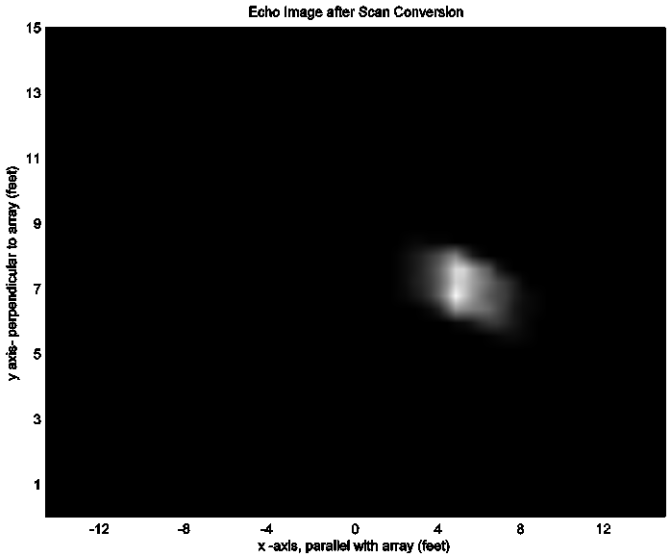

Figure 28. Final Echo Image of Signal A after Scan Conversion and contrast enhancement.

9) Image Processing: Image processing stages are then usually added to enhance the image. For example, enhancing the edges in the image can help with locating targets. Edge enhancement filters are basically high pass filters. (*why?*) A sonic image is often noisy and filters are used to reduce speckle, normally requiring a low pass filter (*why?*). For a good example of low and high pass filtering image, run MATLAB's "`firdemo`." For a good example of finding edges in images, run MATLAB's "`edgedemo`." These are all implemented by simply doing a 2-D convolution of the filter coefficients, h(n,m), over the image. MATLAB's help utility can tell you about the various classic methods such as Sobel, Prewitt, Roberts, Laplacian, etc, and will show you how to implement these yourself using MATLAB's "`edge`" function. A picture of these 2-D filters are on the ECE434 website in edge_filters.pdf.

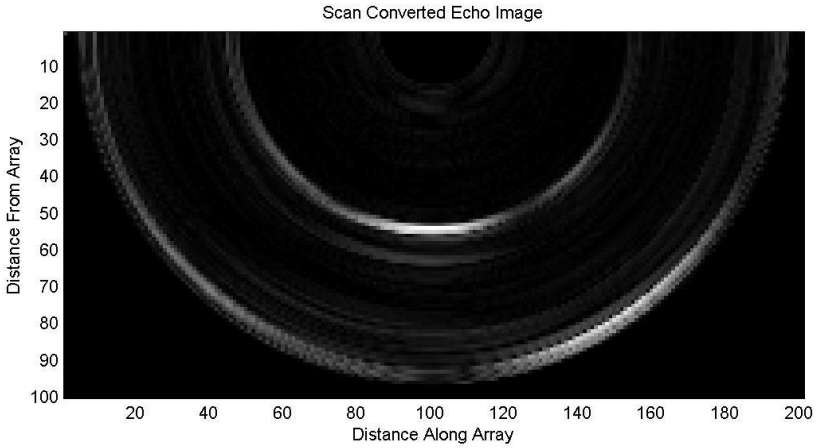

Figure 29. Signal B's echo image after scan conversion.

The objects of interest in the image may tend to lie in certain gray levels in the image (certain levels of brightness). Therefore contrast enhancement algorithms can be applied (similar to contrast and brightness controls on TV). In Figure 29, much noise appears in the image of Signal B after scan conversion. We can remove this noise using a "window-level" algorithm, which can set all gray levels below a "bottom" threshold to zero (hiding the noise), set all gray levels above a "top" threshold to maximum brightness (emphasizing target object above this threshold), and then amplifying all other gray levels between "bottom" and "top" to the full dynamic range of the available gray levels. Figure 30 illustrates this mapping.

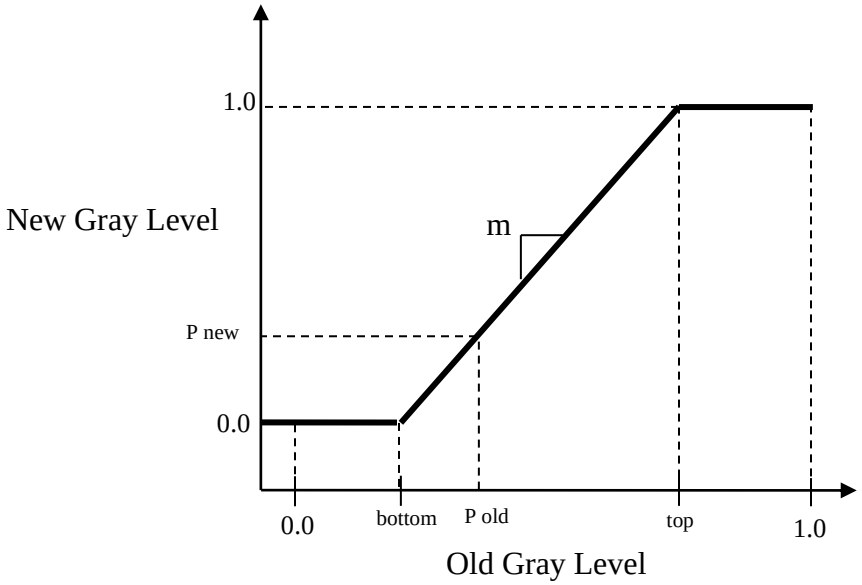

Figure 30. Mapping between the old gray level of the image before window-level to the new gray level for the output image after window-level. For example, a input pixel with gray level value "P old" will map to an output value of "P new". The gray levels between "bottom and top" are stretched to the full dynamic range of 0.0 to 1.0.

The specific algorithm for window/level is

```
m = (1.0 - 0.0)/(top - bottom);   /* slope */
b = 0.0 - m*bottom;    /* y intercept  */
if P(x,y) < bottom
   P(x,y) = 0.0;
else
   if P(x,y) > top
      P(x,y) = 1.0;
   else
      P(x,y) = m*P(x,y) + b;
```

The above algorithm assumes the magnitude of each input pixel ranges from 0.0 to 1.0. *How would you need to change this if your input image's pixels have a different dynamic range?*

Figure 31 shows the contrast enhancement achieved to the image in Figure 29 after Window-Level is applied.

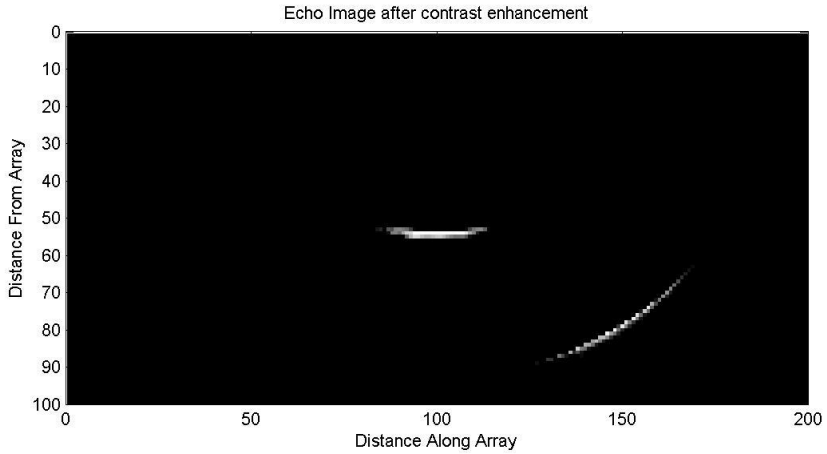

Figure 31. Signal B's echo image after contrast enhancement (window-level), removing most of the noise from the visual range and enhancing the contrast of the targets.

Another speckle reduction technique is known as temporal compounding or persistence [4]. Temporal compounding enhances stationary signals while reducing the time-varying noise by averaging the current unfiltered image, P_{in}, with the previous output image, P_{out}:

$$P_{out}(i) = a \cdot P_{out}(i-1) + (1-a)P_{in}(i) \tag{8}$$

where i is the frame number and a is the weight (or persistence). (*hmmm, this seems like a IIR LPF. How many "taps" does this IIR filter have for each pixel?*) Persistence strengthens stationary signals, but too much persistence causes blurring of fast moving objects. This is difficult to illustrate with static images. You need to see the live demonstration of the sonar system to appreciate persistence, or run `cpx3 sonar v2.m` using **test_data3.bin** which has several image frames of a moving object and a stationary object.

10) Tracking: Automatically tracking moving objects by a radar or sonar system is a very important feature for military sensor systems and there are several ways to do this. If you know a lot about the expected target signature, then *matched filter* or *template matching* techniques can be used. This can be implemented as a *cross-correlation* of our data with the expected pattern (template). Mathematically, c*ross-correlation* is very similar to *convolution*

$$R[n] = \sum_{m=0}^{N-m-1} x^*[m]\,t[n+m] \tag{9}$$

Where *x* is the data and *t* is the template. *R[n]* will peak at the location of max correlation, and identify where the object is in the data (assuming this max point is above your minimum threshold you set to declare this as a match, versus no object found).

Matlab has a 1-D correlation function called *xcorr* and a 2-D correlation function called *xcorr2*. If you want to use 1-D correlation, you have a couple of options. Since your ideal pulse returned is 6-cycle of 10 kHz sinusoid (**signal B**), you could make this your template, and correlate it with each Beam before demodulation. Or you could correlate a template with each Beam after demodulation. *What would this template look like?* It should be the envelope of your 6-cycle sinusoid, which would be a *rect* having the same length as 6 cycles of the pulse.

If you want to use 2-D correlation, you can correlate a 2-D template of what you expect the target (blob) to look like in the sonar image. If the template is a good match, you will get a high peak at the location (x,y) in the image where the object occurs. Again, you have two options. You could correlate your template with your image before scan conversion, *B[θ,r]*, or with your image after scan conversion, *P(x,y). Which is the better option?*

After you find the location of the object(s), you then need to find a meaningful way to display this information to the user. You could calculate its coordinates (x/y or lat/lon), and display it as a text message. Or you could graphically display where you think its location is my overlaying a colored graphic symbol (like cross-hairs or dot) on the object. Then when you use our dataset called **test_data3.bin,** we should see your graphic symbol move with the object when you run `cpx3 sonar v2.m`.

11) Velocity Estimation and Doppler Imaging: Doppler imaging is not required for CPX3, but can be an optional feature a team would like to add. Doppler imaging is one method to computing the velocity of moving objects taking advantage of the Doppler effect. The Doppler effect causes the reflected pulse to change frequency proportional to the velocity of a moving object (only the component in the direction to/from the receiver) as discussed in Dr Musselman's reading [1], and give the classic formula relating velocity to frequency change. Thus to know the speed of each location in our image (e.g., pixel), we need to know the frequency change of the pixel over time. One way to calculate this is to take the FFT **for each pixel** over time (e.g., over many images). Any deviation in frequency from the carrier (e.g., 10 kHz) can be plotted as a color overlay on the image, with color brightness proportional to velocity [4]. However, the FFT has some practical limitations. The FFT is usually too computationally intense to compute for each pixel in real-time. Also, a limited number of samples in time (images) results in very coarse frequency bins, thus not very sensitive velocity measurements.

CONCLUSIONS

This has been a quick summary of the basic signal processing stages commonly used in sonar and radar systems. Hopefully this reading has taken the mystery out of complex algorithms such as beamforming and given you an appreciation for the importance of signal processing in radar/sonar.

REFERENCES

[1] Musselman, R., *Radar in the Electrical Engineering Major,* Department of Electrical Engineering, Spring 2005.

[2] Morrow, M.G., Welch, T.B., Wright, C.H.G., & York G.W.P., "Demonstration Platform for Real-Time Beamforming," *26th International Conference on Acoustics, Speech, and Signal Processing (ICASSP),* Salt Lake City, UT, May 2001.

[3] Gilbert, T., *EE499 final report*, May 2005.

[4] Turley, J and Hakkarainen, H,., "TI's New 'C6x DSP Screams at 1,600 MIPS: Radical Design Offers 8-Way Superscalar Execution, 200 MHz Clock Speed," *Microprocessor Report,* Vol 11-2, February 17, 1997.

[5] York G. & Kim Y., "Ultrasound Processing and Computing: Review and Future Directions," Chapter in *Annual Review of Biomedical Engineering,* Vol. 1, 1999, pp 559-588.

[6] Oppenheim, A.V., *Applications of Digital Signal Processing.* Prentice Hall, 1978.

[7] York G., Basoglu C., and Kim Y., "Real-Time Ultrasound Scan Conversion on Programmable Mediaprocessors", *SPIE Medical Imaging,* Vol. 3335, 1998, pp. 252-262.