Delivered Audio Quality Verification for Rescue 21

1/c Roland T. Orr

Project Advisors: Dr. Keith Gross, CDR Thomas Routhier

Projects in Electrical & Computer Engineering
United States Coast Guard Academy
May 2005

Delivered Audio Quality Verification for Rescue 21

1/c Roland T. Orr

Project Advisors: Dr. Keith Gross, CDR Thomas Routhier

Projects in Electrical & Computer Engineering
United States Coast Guard Academy
May 2005

ABSTRACT

The Coast Guard needs its new VHF equipment to meet a certain standard of speech clarity. The subjective industry standard called Delivered Audio Quality (DAQ) was chosen in the requirement specification to establish what is acceptable in the new equipment and what is not. The new radios, provided by General Dynamics, must deliver speech with a DAQ of 3.4 or greater in order to satisfy our requirements, and my project consists of verifying that it meets this specification.

I decided to equate the subjective DAQ standard to an objective one called SINAD. To that end, I recorded perfect speech files, made noisy versions, and played the noisy versions for audiences to give me DAQ values while calculating SINAD on the same files. This yielded a relationship between the two that I hoped would map a SINAD to a DAQ, eliminating the subjective nature of testing the specification. However, it is discovered that the medium of testing has a large impact on the relationship, and that any relationship that I might have realized would have been specific to the hardware system used in the communication.

INTRODUCTION

The U.S. Coast Guard has awarded General Dynamics Corporation a 19-year, $611 million contract to replace the Coast Guard's outdated Very High Frequency (VHF) radio equipment as part of a project called "Rescue 21." Rescue 21's goal is two-fold: improve the Delivered Audio Quality (DAQ) of radio transmissions to the Coast Guard, and pinpoint a direction (bearing) that callers are calling from. The project's requirement specification document uses two technical terms to describe their requirements for these goals: Delivered Audio Quality (DAQ) and Direction-Finding (DF). Direction finding is the focus of 1/c Jayme Dubinsky's 2004-2005 senior design project, and more information on DF can be found in her work. My project's focus is measuring the DAQ of General Dynamics' VHF communications equipment.

DAQ is a subjective standard, published by the Telecommunications Industry Association (TIA)[1], of speech quality rated on a scale of one to five, with one being unusable and five being easily understandable. Being a subjective standard implies that there is no one true correct DAQ for a given sound file. Five people that listen to the same sound file could evaluate the DAQ to be five different values, and no one would necessarily be more correct than the other.

The Rescue 21 staff at Headquarters has asked the Coast Guard Academy to develop a testing methodology to verify the DAQ of General Dynamics' VHF Communications equipment. The requirements specification states that the DAQ of received radio calls must be heard with a DAQ of at least 3.4, but the DAQ standard's subjective nature makes it hard to verify that General Dynamics has actually complied with this requirement. Thus, the project members have decided to measure DAQ by equating it to an objective standard, also published by the TIA[1], called SINAD.

The TIA standard for SINAD states that it stands for Signal-to-Noise Ratio, but it is not simply the ratio between signal and noise. The actual calculation is covered in more detail later. For now, simply realize is that SINAD is *objective*. In other words, given a method to undisputedly calculate the portion of a signal that is "signal" and the portion that is "noise", calculating SINAD becomes trivial, and there is only one exact answer. The question now becomes how to distinguish the signal from the noise, which is not hard

if the system is designed well. The details of my algorithm are beyond the scope of the Introduction, but the end result is a correct SINAD value, which is not contestable like DAQ is.

TIA publishes both SINAD and DAQ in the same standard, and actually proposes a relationship between the two that maps DAQ values to ranges of SINAD values[1]. However, I did not blindly use their relationship - I verified it myself so there would be no question of its validity. The method I used to verify this relationship is quite elaborate, so the explanation is left for the System Design section.

Work has been done during 2002-2004 on this project by 1/c Brian Warner[2] and 1/c Jen Jojola[3]. A vast majority of this previous work has been built on the assumption that the signal and noise cannot be easily separated from a signal. In Fall 2004, we started operating under the assumption that I can transmit a pre-recorded noiseless file, and compare it to the received version to separate the signal from the noise. The project's goal is still to assess General Dynamics' VHF communications equipment, but this is a completely new approach to the problem, so much of the past work is not applicable.

OBJECTIVE

The overall goal of the project was to assess the DAQ of General Dynamics' VHF equipment for the Coast Guard's Rescue 21 project (G-AND). The project's success was dependent on my accurate assessment of whether or not the equipment could produce a radio call from a 1-watt radio located 20 nautical miles away and six feet above the ocean surface with a DAQ of at least 3.4. I developed a relationship between SINAD (objective) and DAQ (subjective) to make the testing simpler. My goal was to establish this relationship by February 14, 2005, after which I would conduct field testing on General Dynamics' radios and submit an analysis.

SYSTEM DESIGN

The ultimate goal is to assess the DAQ of General Dynamics' VHF equipment. As previously discussed, DAQ is a subjective standard. One possible way to measure DAQ would be to record a received call from one of General Dynamics' VHF radios and play it for a committee of human testers, who would come to a consensus on the DAQ. In fact, this is General Dynamics' proposed method to measure the DAQ of their radios. Unfortunately, this method makes DAQ measurements time-consuming and expensive. Finding neutral subjects would also be a challenge, especially if General Dynamics is paying their salary.

I propose a different method altogether. To remove the subjectivity from the measurement, I have to come up with a relationship between SINAD and DAQ. This was projected to be my first task, but it actually wound up taking up all of my time. As mentioned before, the TIA includes such a relationship in the DAQ and SINAD standard[1]. Table 1 shows the TIA standard for DAQ, and how the TIA asserts DAQ values map to SINAD values.

Table 1 - DAQ standard with TIA's DAQ-SINAD relationship

DAQ	Subjective Performance Description	SINAD equivalent
1	Unusable, Speech present but unreadable	<8 dB
2	Understandable with considerable effort. Frequent repetition due to noise/distortion	12 +/- 4 dB
3	Speech understandable with slight effort. Occasional repetition required due to noise/distortion	17 +/- 5 dB
3.4	Speech understandable with repetition only rarely required. Some noise/distortion	20 +/- 5 dB
4	Speech easily understood. Occasional noise/distortion	25 +/- 5 dB
4.5	Speech easily understood. Infrequent noise/distortion	30 +/- 5 dB
5	Speech easily understood.	>33 dB

Oddly enough, TIA includes a footnote that warns not to use SINAD in this capacity. Thus, I did not want to use this relationship without first verifying it. Equating a subjective speech quality standard to an objective sound quality standard is no easy task, because there are subjective elements inherent in speech that are difficult to quantize into objective values. The way to mitigate such error is to calibrate the relationship using large sample sizes. Specifically, large groups of people to rate sound files are necessary, as well as a large group of sound samples. These were both factors that I incorporated into my approach. Once this SINAD-DAQ relationship was established, I could simply take received calls from General Dynamics' VHF radios, compare them to the noiseless versions to calculate the SINAD, and thus attain the DAQ.

The first consideration was how we envisioned the VHF airwaves. Figure 1 shows a block diagram that illustrates what we thought would happen to a VHF

transmission. The transmitted benchmark is amplified or attenuated along the path by antennas, amplifiers, or any number of things, and noise is added. The gain value (A) and noise combine to yield the received noisy file. We thought that it was logical to assume that A was frequency independent – in other words, any amplifying or attenuating effects on the transmitted signal would affect all frequencies the same. We would find in the end that this was not the case, but at the time of our design, we felt it was a safe assumption.

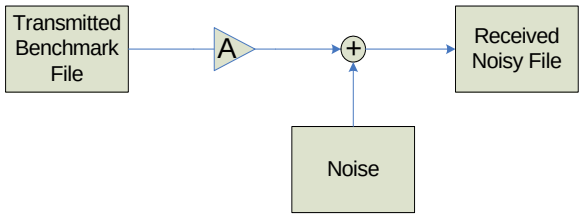

Figure 1: My prediction of the effect of VHF airwaves

To facilitate making noisy files and measure SINADs, I created two m-files in MATLAB®: `SINAD.m` and `noisify.m`. The `SINAD` m-file is a script that takes a noiseless file and a noisy version of that file, and computes an exact SINAD value for the noisy file. The code for `SINAD.m` is included as Appendix A. `Noisify.m` is a function makes a sound file noisy, with certain parameters. Once a wave file is read into MATLAB®, `noisify()` can be used to add noise scaled by one of the parameters and a time offset. I originally coded `noisify()` to add uniformly distributed noise, and then changed it to Gaussian noise at the recommendation of one of my advisors. The noise was filtered to a bandwidth of 8 kHz to reflect what we thought was the typical bandwidth of an audio signal. Again, this turned out to be a poor representation.

Once these scripts were completed, I recorded a set of "benchmark" audio files, which were recorded with high quality equipment, in a quiet environment to have minimal noise, and with a variety of voices to try to capture as much subjectivity in the experiments as possible. I then corrupted these files using noisify(), and saved the noisy versions to separate wave files. A noisy sound file could be played to a test audience to measure DAQ. Used in tandem with its benchmark version, I could also use the noisy sound file to calculate an exact SINAD. *X* number of benchmarks would produce *X* data points to use in mapping SINAD to DAQ. This was the plan for establishing the relationship.

RESULTS

Initial Results

I played the noisy sound files to a group of 24 test listeners, who marked on a DAQ scale what they thought the DAQ was. Averaging the results for each file gave me an estimate of the DAQ for each noisy file. I then used my `SINAD` script to calculate the SINAD values for all of the noisy sound files by using the benchmarks. This process yielded, for each benchmark and noisy pair, a SINAD and a DAQ. I plotted the results, shown in Figure 1 below. Since I had 16 pairs of noisy files and benchmark, there are 16 data points on the chart. I added horizontal error bars to my data points to indicate 90% confidence intervals, under the assumption that the test audience's DAQ values were normally distributed (since σ was unknown, the intervals are t-distribution based). I also added a hyperbolic fit to the data. Figure 2 also shows the TIA relationship on the same chart, with vertical error bars to represent their SINAD uncertainties listed in Table 1.

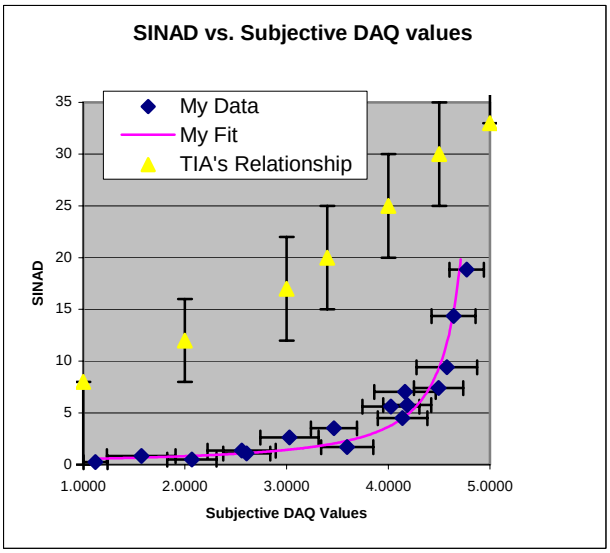

Figure 2 - My initial results with 90% confidence intervals, plotted with TIA's relationship from Table 1

These initial results clearly did not support TIA's relationship. For a known SINAD value, TIA's relationship suggested a much lower DAQ value than my results indicated. In other words, my test subjects were hearing the speech better than TIA asserted they would.

Furthermore, we are most concerned with DAQ values around 3.4. If General Dynamics' radios are supplying very poor audio, the DAQs will be very low and we will know they did not meet the requirement.

Similarly, if the speech is nearly perfect, the DAQs will be high, and we will know that they did fulfill the requirement. Thus, the area around DAQ=3.4 has the most potential for dispute, and should therefore be the area with the most accuracy in the SINAD-DAQ relationship. The fit in figure 2 does not even touch any of the 90% confidence intervals in the area around DAQ=3.4 – this was a serious problem.

These were initial results, so we did not expect them to be perfect, but they were so far off that we had to take a step back and analyze our approach. We decided that there were several things we could do to improve results:

- Increase the number of people recording DAQ values (test audience size)
- Increase the number of files
- Include more files with mid-range DAQ values
- Research VHF noise and add "smarter" noise to the files to better emulate VHF transmissions

Second set of results

We focused on the top three improvements first. I expanded my benchmark library by 14 audio files to include 30. When I corrupted the 14 new files, I tried to focus them in the 3 - 4 DAQ range to help fill in the gap present in the initial results. Finally, I played all 30 files for a group of 45 people instead of 24. The results from the improvements are shown in Figure 3.

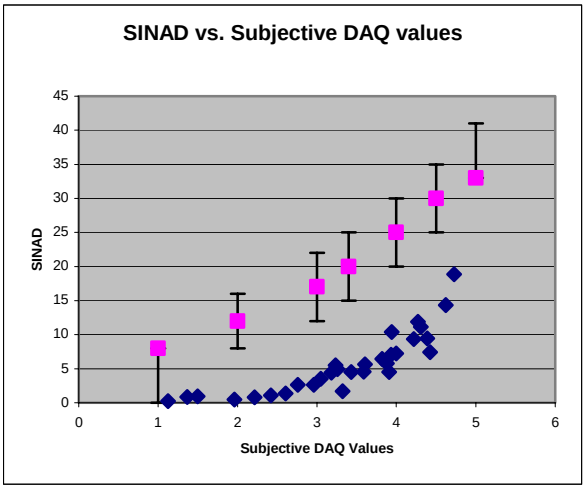

Figure 3 - My second group of data, plotted with TIA's relationship from Table 1

The new data looks less hyperbolic, and the new files filled in the 3 - 4 DAQ range nicely, but my results still did not reflect TIA's results. At this point, we decided to study VHF transmission noise more closely.

The system for the VHF Noise study

The first step in studying VHF noise is acquiring the noise. We decided to use two marine radios: one to transmit a benchmark, and another to receive the benchmark and record it. After about a week of trying to use existing equipment to no avail, I decided to invest in some new radios. Standard Horizon was the manufacturer of the model we eventually decided on: the HX371. We chose this model for several reasons:

- It had variable transmission power (1 Watt, 2.5 Watts, 5 Watts), which would hopefully enable us to get multiple noise levels while transmitting from the same location.
- It had a output that doubled as a microphone and speaker
- It came with a "clone cable", which used the microphone/speaker jack to connect to another radio for the purpose of moving saved settings and memory to a new radio. We did not intend on using it for this feature, but thought that it might serve some purpose so we wanted the capability.
- The antenna was detachable, allowing us to attach the receiving radio to a roof antenna
- The radio unit is handheld, making it easy to create a portable laptop/handheld transmitting unit

A picture of the unit is shown in Figure 4.

Figure 4 - Standard Horizon HX-370 radio

One option with the HX370 was to buy a speaker/microphone, which allowed a user to clip the radio to a belt and use the "speaker-mic" to communicate. We also purchased three of these. I modified one speaker-mic to be the transmitter, and the other a receiver. For the transmitter, I unsoldered the connections to the physical microphone inside the speaker-mic, and soldered them to an RCA cord. The RCA cord could then be plugged into a laptop to transmit recorded benchmark files. We kept the push-to-talk button in tact to allow easy keying of the unit when it came time to transmit. Figure 5 shows a close-

up view of the modifications we made to the transmitting speaker-mic.

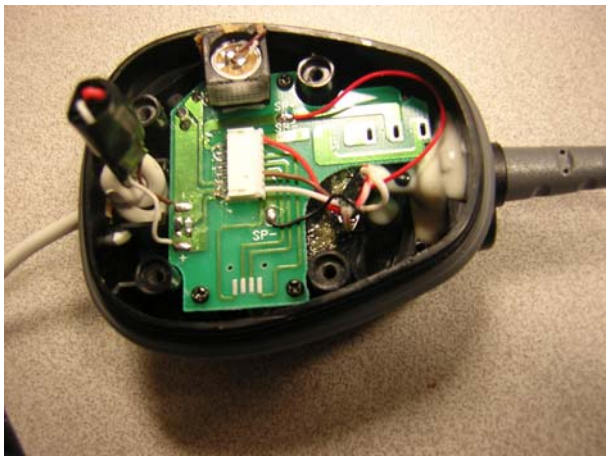

Figure 5 - Transmitting speaker-mic

For the receiver, we unsoldered the connections to the physical speaker, and soldered them to another RCA cord. The RCA cord plugged into the line-in jack of a computer sound card to record the received benchmark. Figure 6 shows a close-up view of the modifications we made to the receiving speaker-mic.

Figure 6 - Receiving speaker-mic

At this point, we had a working system to study VHF noise. The transmitting unit consisted of the transmitting speaker-mic plugged into a HX371 radio, with the added RCA cord plugged into the speaker jack of a laptop. The receiving unit consisted of the receiving speaker-mic plugged into the other HX371 radio, with the added RCA cord plugged into the line-in jack on a desktop's sound card. This system is illustrated in Figure 7.

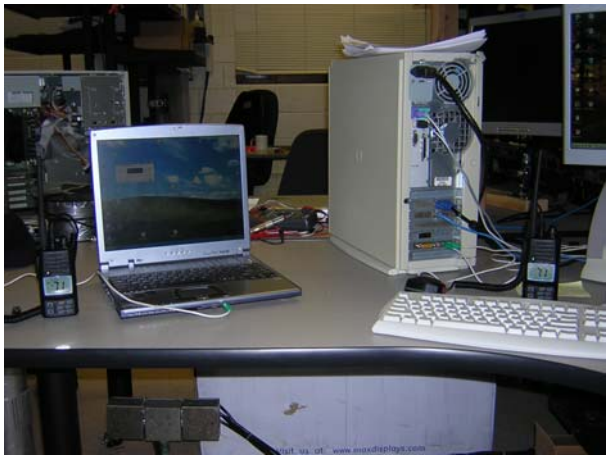

Figure 7 - System used to study VHF noise

The actual study of VHF noise

The VHF noise study was not as easy as I had hoped. I envisioned transmitting a benchmark, recording its received version, and simply subtracting the benchmark from the received version to get the noise. Doing this repeatedly, I hoped, would yield a noise distribution function which I could implement into `noisify()` to replace the apparently inaccurate Gaussian noise. Things are not always what they seem, and so it turned out that there was much more going on between a VHF transmission and the reception of that transmission than a gain and some additive noise.

My SINAD.m file used equation (1) to estimate the value of A (see Appendix A). This equation did a poor job for very noisy signals.

$$A = \frac{Benchmark^T Received}{Benchmark^T Benchmark} \quad (1)$$

In fact, it seemed as though whenever voice was in a transmitted file, the A-value would be *so* far off that I would wind up with negative SINAD values; but SINAD values cannot be less than 0dB.

Instead of using benchmark speech files, I decided to transmit a pure tone sine wave. The results with the pure tone were much better. The A-value was estimated flawlessly by the algorithm. Figure 8 shows the results. The top plot is the sine wave that was transmitted. The middle plot is the received sine wave. The bottom plot is the bottom plot subtracted from the top plot, yielding noise.

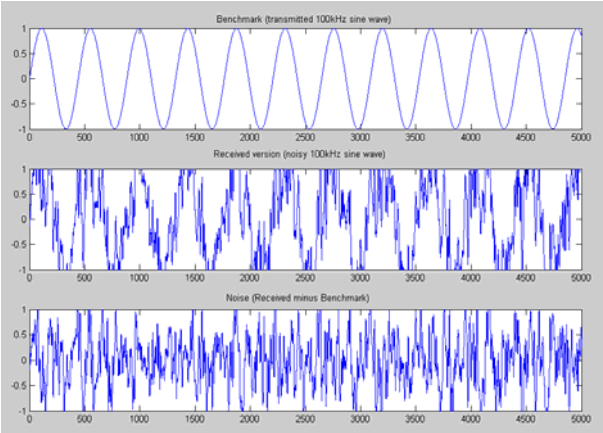

Figure 8 – Time domain results for the 100kHz sine wave experiment

Suppose that the A-value was estimated incorrectly. If the received wave and the original wave were not on the same vertical scale, the noise would still have a 100 kHz sine wave in it, and this would show up in the autocorrelation sequence as repeating spikes every 10^{-5} seconds. A plot of the autocorrelation sequence (generated using the `xcorr()` function in MATLAB®) shows that this is not the case in Figure 9. There is only one spike, corresponding to no shift. A look at the power spectral density (using `pwelch()`) of the noise in Figure 10 shows that some 100 kHz component actually did slip through, and that the 3dB bandwidth of the noise is about 3 kHz, not 8 kHz like we originally thought.

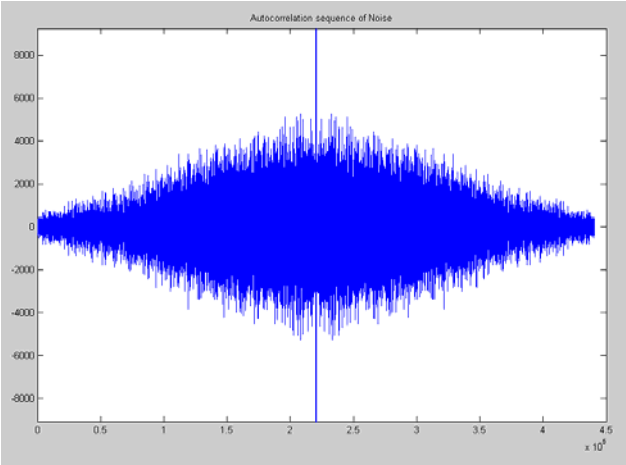

Figure 9 – Autocorrelation sequence of the VHF noise taken from the sine wave

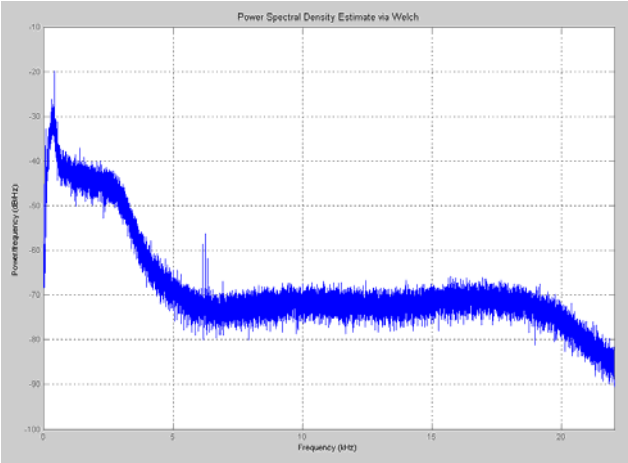

Figure 10 – Power spectral density of the VHF noise from the pure tone experiment.

We also found that the noise is very close to a normal distribution. Using the `histfit()` and `normplot()` commands in MATLAB®, we compared the noise to normal data. Figure 10 shows the noise plotted on a histogram with a normal distribution overlay, and figure 11 shows a plot that measures how normal a data is. In figure 11, normal data would be completely along the dotted line down the middle, and any data that varies from that line is not normal. Figure 10's normal density overlay shows that the data is almost completely normal, and figure 11 confirms this.

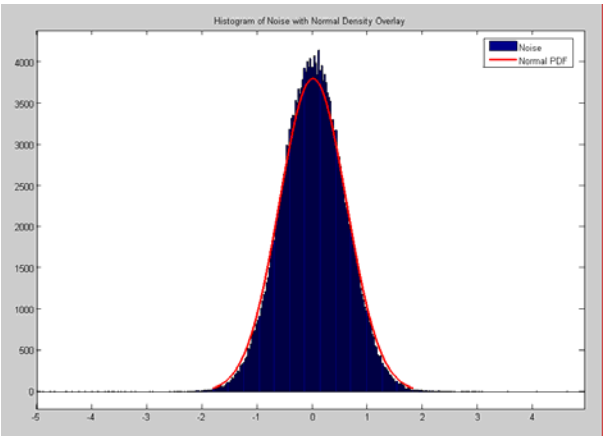

Figure 10: Histogram of the VHF noise with a normal density function overlay

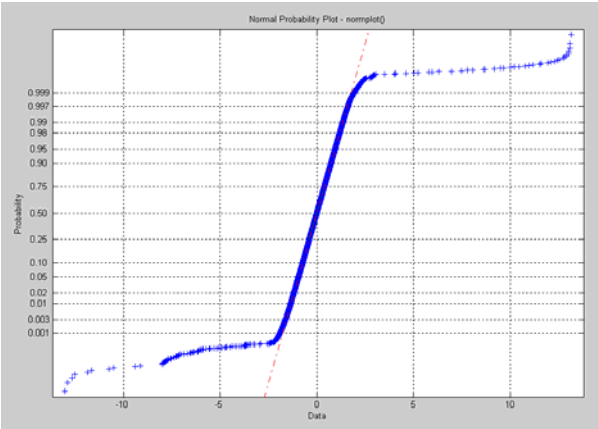

Figure 11: `normplot()` of the data

Figures 10 and 11 don't necessarily show that VHF channel noise is normal. There was more going on here than VHF noise alone. Internal hardware in the radios was affecting the signal also. One option was to cater our noise to reflect the system we were using – the desktop, laptop, radios, etc. However, since the type of noise added has such an effect on results, this would make our SINAD-DAQ relationship specific to the particular hardware we were using whenever we took data. Since the specification for Rescue 21 doesn't narrow down any one type of radio used on a civilian's boat, a relationship dependent on hardware would be mostly useless. This realization was a serious obstacle for the project.

The other serious obstacle was the discovery that the A-value was in reality a function of frequency. After the pure tone experiment, I tested the system with a periodic chirp. This let me look at the effect of the system on all frequencies. Figure 12 shows the results of the test.

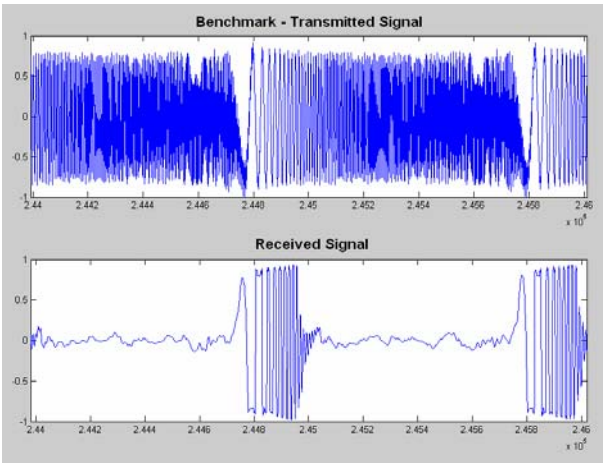

Figure 12: Time domain results of the periodic chirp experiment

As you can see, the periodic chirp's higher frequencies are heavily attenuated, while the lower frequencies are left alone. This shows that the system has a low-pass filter for a frequency response, due to internal hardware limitations inside the radios like amplifiers. The cutoff frequency for the magnitude response would thus depend on the hardware used. The only fair way to handle this consideration would be to filter both the benchmarks and the added noise using a filter that looks like this frequency response. Once again, we would have to accommodate the specifics of our system's hardware, so the relationship is even more restricted in its application.

At this point in the project, I ran out of time to push further. However, it was a good stopping point. From the results, it seems that a universal DAQ-SINAD relationship is nearly nonexistent. It would also be difficult to prove that ones exists for just the VHF airwaves. My results indicate that the hardware chosen for the experiments has a large impact on the noise and gain. Since the noise and gain have a large impact on any such DAQ-SINAD relationship, this suggests that there may only exist DAQ-SINAD relationships for individual systems of airwaves and hardware. In other words, any relationship that I might have pulled from this data would be specific to my two HX-370 radios, the two speaker-mics, the laptop and desktop sound cards, and channel 71 VHF. Needless to say, this is very limiting. The Rescue 21 specification does not allow us to pinpoint exactly what hardware will be used to transmit calls to the General Dynamics radios, so the system is undefined. If my analysis in this paragraph is correct, then perhaps no DAQ-SINAD relationship can be proven accurate for the purpose of Rescue 21 DAQ verification.

CONCLUSIONS

A DAQ-SINAD relationship is going to be harder to develop than originally conceived. We found that the noise we were adding to the benchmarks was too high a bandwidth for VHF communications, and also that the magnitude response in our system was a function of frequency. The noise bandwidth issue was a simple setback that can be avoided by simply altering a value in my `noisify()` function. The magnitude response can be accounted for by filtering both the benchmarks and noisy versions with a filter similar to the magnitude response. However, both of these steps account for specific properties of the system I was using to develop this relationship – the radios, the computers, the VHF channels. This suggests that any DAQ-SINAD relationship I could glean would be non-transferable to other systems, including the Rescue 21 system.

Perhaps there is an inherent flaw in trying to correlate an objective standard to a subjective one.

The next step in this project is to decide whether this is a valid approach. I think my research suggests that it is not, but many more avenues remain to be investigated. For example, nothing I did incorporated the use of phonemes, which are individual sounds that make up language. Phonemes are commonly used in speech analysis and speech synthesis to bridge the gap between objective computer logic and the subjective human ear.

ACKNOWLEDGEMENTS

The author wishes to acknowledge the contributions of:

- Project advisors – Dr. Gross and CDR Routhier, for countless hours of assistance.
- JJMA employees – Christian Oates and Jim LoBianco, for their assistance in modifying the speaker mics and collecting data.
- USCGA EE Chief's Mess – for their assistance in modifying the speaker mics.
- Rescue 21 project manager – CAPT Mobley, for taking time out to come to the Academy and meet with 1/c Dubinsky and myself about Rescue 21.

REFERENCES

[1] "Wireless Communications Systems - Performance in Noise and Interference - Limited Situations - Recommended Methods for Technology - Independent Modeling, Simulation, and Verification - Addendum 1". TSB-88-A-1. *TIA/EIA Telecommunications Systems Bulletin*. January 2002.

[2] Warner, Brian. *Proposed Testing Methodologies For NDRSMP*. April 2003.

[3] Jojola, Jennifer. *Delivered Audio Quality Verification*. April 2004.

APPENDICES

Appendix A: MATLAB® SINAD.m code

Appendix B: MATLAB® noisify.m code

Appendix A

```
%   M-file name: sinad.m

%   Author: 1/c Tim Orr, US Coast Guard Academy
%   Date:   25 October 04
%   Brief overview: takes two wave files, one distorted and one not, and
%       calculates the DAQ of the distorted one with the non-distorted one as
%       a reference.
%
%   Inputs: Since this is a script, not a function, there are no inputs in the
%           function sense.  However, there are inputs that the user specifies:
%           - two names of wave files: one is the "benchmark signal", which is
%             undistorted, and the other is the received signal, which is
%             probably distorted somehow.
%   Output: - the offset, or the sample number in the received signal where the
%             benchmark signal starts, as identified by this algorithm.  An
%             offset of zero means that there is no added noise at the beginning
%             of the received signal.  The variable is called Offset.
%           - the noise signal that this algorithm identifies as being imbedded
%             in the signal.
%           - the SINAD of the noisy file as compared to the benchmark
%
%   Description: this script compares a benchmark signal with its received
%       version.  Using xcorr(), it identifies any time sample offset.  It then
%       calculates any magnitude gain from benchmark to noise.  Then, it
%       identifies noise by subtracting the original signal.  Finally, it
%       calculates the SINAD of the received signal.



% notes to myself:
%       make sure you have the whole known signal in your received signal
%       i don't need to worry about row/column vectors since wavread always
%          reads into column vectors.



%--------------------------%
%                          %
%       User inputs:       %
%                          %
%--------------------------%

disp('This script takes a benchmark signal and a noisy version of the benchmark');
disp('and calculates the SINAD of the noisy signal.  The user must supply the');
disp('filenames of the benchmark and noisy sound files');
disp(' ');
BenchmarkFilename = input('Benchmark signal filename: ', 's') ;
NoisyFilename = input('Received signal filename: ', 's') ;

[Benchmark, fs_b, nbits_b] = wavread(BenchmarkFilename);
                                            % Benchmark is the signal,
                                            % fs_b is the sampling frequency
                                            % nbits_b is the bits per sample
[NoisyBenchmark, fs_n, nbits_n] = wavread(NoisyFilename);
                                            % NoisyBenchmark is the signal,
                                            % fs_n is the sampling frequency
                                            % nbits_n is the bits per sample



%-----------------------------------------------------------------%
%                                                                 %
%       Calculate the offset and un-offset NoisyBenchmark:        %
%                                                                 %
%-----------------------------------------------------------------%

% This algorithm can get confusing, so I'll explain it in detail before the code
% becomes overwhelming.  The xcorr() function is somewhat hard to grasp.
% xcorr(), when passed two vectors, does the following:
%   1. zero-pads the shorter vector to be the same length as the longer
%   2. creates a cross-correlation sequence of length 2n-1, where "n" is the
%       length of the longer of the two input vectors.  The center of the
```

```
%       cross-correlation sequence is located at the "n"th index.  Going one way
%       from the center corresponds to shifting one of the input vectors to the
%       left or right and comparing the two vectors.  Going the opposite way
%       from the center corresponds to shifting the same input vector in the
%       opposite direction (right or left as opposed to left or right), and
%       comparin the two vectors.  I have no idea which vector is shifted, and
%       in which direction it is shifted for a given direction from the center
%       of the cross-correlation sequence.  I don't think this is a problem as
%       long as I consistently pass the input vectors in the same order.
% The cross-correlation sequence is of length 2n-1, and the middle of the
% sequence is at index 'n', where n is the length of the larger of the two
% input vectors.  If I pass the benchmark signal first, then the maximum of the
% cross-correlation sequence occurs to the left of the middle.  Recall that a
% shift of one of the input vectors corresponds to a shift of the same amount
% from the *center* of the cross-correlation sequence.  Thus, to find the
% offset, I simply have to subtract the index of the maximum value in the
% cross-correlation sequence from the center (which, again, is located at n).
[MaxXCorr, IndexOfMaxXCorr] = max( abs( xcorr( Benchmark , NoisyBenchmark ) ) ) ;
CenterOfXCorr = max( length(Benchmark), length(NoisyBenchmark) ) ;
Offset = CenterOfXCorr - IndexOfMaxXCorr ;

% Now, remove the extra samples from the NoisyBenchmark vector.
NoisyBenchmark = NoisyBenchmark(Offset+1 : end) ; % Notice that I added 1 to
            % Offset.  This is because MATLAB starts vectors at index 1, not 0.

% Ideally, at this point, Benchmark and NoisyBenchmark are the same length.
However,
% this might not be the case.  If they are not the same length, there are two
% possibilities: either NoisyBenchmark is smaller or NoisyBenchmark is larger.
%  -If NoisyBenchmark is now smaller than Benchmark, then the *entire* benchmark
signal
%   wasn't included in the original NoisyBenchmark file.  This will generate an
error
%   message in the console window because we don't think we should judge DAQ
%   based on an imcomplete received signal.
%  -If NoisyBenchmark is now larger than Benchmark, then there is added noise at
the end
%   of the received file, so truncate the end as well.
if length(NoisyBenchmark) < length(Benchmark)       % display error and exit the
script
    disp('ERROR: All of the benchmark signal was not included in the received
signal');
    return
elseif length(NoisyBenchmark) > length(Benchmark) % lop off the additional noise at
the
                                          % end of NoisyBenchmark
    NoisyBenchmark = NoisyBenchmark(1 : length(Benchmark));
end

% % DEBUG
% figure(1) ;
% subplot(211) ;
% plot(Benchmark) ;
% subplot(212) ;
% plot(NoisyBenchmark) ;



%-------------------------------------------------------------%
%       Calculate the scaling factor from Benchmark to       %
%         NoisyBenchmark and descale NoisyBenchmark          %
%                                                             %
%-------------------------------------------------------------%

% This is pretty straightforward.  It simply implements a formula given to me by
% Dr. Gross.  The formula is:
%           benchmark' * noisy
%         ---------------------- , where ' means transpose.
%         benchmark' * benchmark
% Multiplying a vector transposed by itself gives you the sum of the squares
% of the individual values.  This is a measure of the power in the signal.
% Thus, if the numerator was changed to "noisy' * noisy", then the result
```

```
% would be the square of the scale factor.  To get magnitude from power,
% you take the square root.  The reason the formula works is because by
% replacing noisy' with benchmark', you avoid having to take the square root.
% This formula is actually functionally equivalent to:
%         (   noisy   )         (   noisy   )
%      sum( --------- ) / length( --------- ) ,
%         ( benchmark )         ( benchmark )
% but the first formula is more efficient in MATLAB.

A = (Benchmark.'*NoisyBenchmark) / (Benchmark.'*Benchmark) ;

% Descale NoisyBenchmark to have the same overall magnitude as Benchmark:
NoisyBenchmark = NoisyBenchmark ./ A ;

% % DEBUG
% figure(2) ;
% subplot(211) ;
% plot(Benchmark) ;
% subplot(212) ;
% plot(NoisyBenchmark) ;



%-------------------------------------------------------------------------%
%                                                                         %
%        Form noise vector, calculate scaler signal and noise values      %
%                                                                         %
%-------------------------------------------------------------------------%

NoiseVector = NoisyBenchmark - Benchmark ; % This is the noise signal present
                                           % in the benchmark.
% % DEBUG
% figure(3) ;
% plot(NoiseVector) ;

% Recall that multiplying a transposed vector by itself yields the sum of the
% squares of its values.  This is a measure of the total power present in the
% signal.
NoisePlusSignal = NoisyBenchmark.'*NoisyBenchmark ; % NoisyBenchmark is really
% the benchmark plus the noise, so its power is really the noise+signal power.
Noise = NoiseVector.'*NoiseVector ;  % NoiseVector is the noise vector, so its
% power is the noise power.

% % DEBUG
% figure(4) ;
% psd(NoiseVector, 256, fs_b) ;


%-------------------------------%
%                               %
%        CALCULATE SINAD        %
%                               %
%-------------------------------%

SINAD = 10*log10( (NoisePlusSignal) / Noise ) % since NoisePlusSignal and Noise
% are formed from the *squares* of the values of NoisyBenchmark and NoiseVector,
% they are *powers*, not *magnitudes*, so the dB for them is formed by
% multiplying their logarithm by 10, not 20.

Return
```

```
function [noisy, offset, noise] = noisify (benchmark, fs, MaxOffsetPercent,
NoiseStdDev)

%   Function:   noisify( benchmark, fs, MaxOffsetPercent, NoiseStdDev )
%   Filename:  "noisify.m"
%   Author:     1/c Tim Orr, US Coast Guard Academy,
%   Inputs:     benchmark - an array of values that correspond to the magnitude
%                           of a signal.  The signal input to this function will
%                           be made noisy and offset in time.  In other words,
%                           benchmark is the undistorted signal you wish to
%                           randomly distort.
%               fs    -     the sampling frequency of the benchmark signal.
%                           This is used in filtering the noise to a reasonable
%                           bandwidth.
%               MaxOffsetPercent - the maximum percent of the benchmark signal's
%                           length to use as the offset.  Default value is 30%.
%               NoiseStdDev - the standard deviation of the added Gaussian
%                           (Normal) noise.  The larger this number, the more
%                           varied the noise is.  Default value is 10% of the
%                           maximum absolute amplitude in the benchmark signal.
%   Outputs:    noisy   -   noisy is the noisy version of the benchmark signal
%                           that was input.  It will have some noise added in
%                           and be offset in time.
%               offset  -   this is the number of samples in time that the
%                           signal was offset by.  The number is random, but is
%                           limited to MaxOffsetPercent% of the signal's length.
%               noise   -   this is the noise that was added to the offset
%                           signal.  The noise array will have a length equal to
%                           (benchmark's length + offset).
%   Description: This function takes a non-distorted signal (a signal that is
%               roughly crystal clear), adds random Gaussian noise to the signal
%               (the noise has a standard deviation of NoiseStdDev), and offsets
%               the signal by a random amount of time samples.  The amount of
%               time samples that the signal is offset by is returned as the
%               "offset" output.  The added noise is returned as the "noise"
%               output.  The final distorted version of the benchmark signal is
%               returned as "noisy".  This function returns the noise and offset
%               so that the original signal can be reconstructed from the three
%               outputs: subtract "noise" from "noisy", and then remove "offset"
%               samples from the beginning of the ("noisy"-"noise") array.
%   Assumptions: -The entire benchmark signal is included in the received signal
%                -The benchmark and received signal are the same sampling
%                   frequency

%%%%  Error checking - making sure the input   %%%%
%%%%  vector is either a column or row vector  %%%%
[rows, cols] = size(benchmark) ;
if rows==1
    benchmark_is_row_vector = true ;
elseif cols==1
    benchmark_is_row_vector = false ;
else
    disp('ERROR: input array is not a column or row vector') ;
    return ;
end

%%%%  Handle different numbers of inputs  %%%%
if nargin < 4, NoiseStdDev = 0.1*max(abs(benchmark)); end; % the default
                % standard deviation is 10% of the max amplitude of the signal.
                % notice that I took the absolute value of "benchmark" before
                % applying max.  This is to address a situation where the
                % maximum amplitude actually occurs at a negative magnitude.
                % max() ignores any negative value in lieu of a positive one
                % (since positive numbers are greater than negative numbers),
                % unless you apply abs() to the array before max().
if nargin < 3, MaxOffsetPercent = 30; end; % max offset value is 30% of the
                                           % signal's length
if nargin < 2, disp('ERROR: sampling frequency and/or benchmark are not
specified'); return; end;
            % if fs or benchmark are not specified, display an error and return

%%%%   Now, I know that benchmark is either a column or row vector, that   %%%%
```

```
%%%%  benchmark and fs are specified, and that default values are set for  %%%%
%%%%                    other arguments not specified                      %%%%
%%  Offset the benchmark signal  %%
range_of_offset_values = (MaxOffsetPercent/100)*length(benchmark); % I want an
    % offset value between 0% and MaxOffsetPercent% of benchmark's length.
    % Thus, the window of possible offset values has a width equal to
    % MaxOffsetPercent% of benchmark's length.  Note that range_of_offset_values
    % will rarely be an integer.  That's okay at this point because I'll round
    % the result in the next step.
offset = round( range_of_offset_values/2 + range_of_offset_values*rand  );
if benchmark_is_row_vector noisy = [zeros(1, offset) benchmark] ;
else noisy = [zeros(offset, 1); benchmark] ;
end;% if benchmark is a row vector,
    % add a row of zeros to the left of the "benchmark" array and store it to
    % noisy.  if its not a row vector, then it must be a column vector, so add
    % a column of zeros to the top of the "benchmark" array.  Recall that the
    % "benchmark" array MUST be either a row or column vector at this point
    % because of the error checking at the beginning of the function.

%%  Generate normally (Gaussian) distributed noise  %%
noise = NoiseStdDev*randn(size(noisy)) ; % randn() yields normally distributed
% random noise with mean 0 and variance 1.  Recall from statistics that when a
% random variable is multiplied by a scalar, A, its variance is multiplied by
% A^2, so its standard deviation (which is the square root of variance) is
% multiplied by sqrt(A^2), or simply A.

%%  Filter the noise:  %%
% When adding random noise to a signal, one should limit the noise to the
% bandwidth of the signal.  Most audio files tend to have a 3dB bandwidth of
% around 8kHz, so I'll low pass filter the noise with a digital filter, which is
% dependent on the sampling frequency, fs.

% [num, den] = butter(10, 8000/(fs/2)) ; % this returns a digital (butterworth)
%     % IIR filter.  The order is 10 (first parameter to butter()), and the cutoff
%     % frequency is 8kHz.  Note that the cutoff frequency must be passed to
%     % butter() relative to the Nyquist frequency, fs/2.


Amax = 3 ;          % maximum attenuation in passband
Amin = 8 ;         % minimum attenuation in stopband
f_passband = 8000 ; % stopband frequency
f_stopband = 9000 ; % passband frequency
% If I remember correctly from Linear Circuits, we warp the digital frequencies
% and treat them like analog filters, and then unwarp the results.
%   prewarp the frequencies:
w_passband = 2*pi*f_passband/fs;
w_stopband = 2*pi*f_stopband/fs;
w_passband = 2*tan( w_passband/2);
w_stopband = 2*tan( w_stopband/2);
%   treat like analog lowpass filter:
ratio = w_stopband/w_passband; % ratio of stopband to passband frequencies for
                               % calculating the filter's order
order = ceil(log((10^(Amin/10)-1)/(10^(Amax/10)-1))/(2*log(ratio))) ; % order of
                                                   % the lowpass Butterworth filter
s_poles = butpoles(order, Amax); % poles of the Butterworth Filter.
%   unwarp the poles:
s_poles = w_passband.*s_poles; % adjust for actual w_passband
z_poles = (2+s_poles)./(2-s_poles); % transform poles from s plane to z plane
%   construct coefficient arrays:
b_secondorder = [1 2 1]; % set the numerator coefficients, which are the same..
b_firstorder = [1 1];    % .. for any two sections of the filter with the same..
                         % .. order.
num = [1] ; den = [1] ; % start the numerator and denominator at 1
for i = 1:floor(order/2) % this for loop calculates the 2nd order sections'
                         % information and adds it to a cumulative denominator
                         % and numerator
   num = conv(b_secondorder, num) ; % update cumulative numerator
   a = [1 -2*(real(z_poles(i))) (abs(z_poles(i)))^2] ; % (no idea why this works)
   den = conv(a, den) ; % update cumulative denominator
end ;
if mod(order,2) == 1 % iff order is odd, calculate the first order info.
    num = conv(b_firstorder, num) ; % final update of cumulative numerator
```

```
    den = conv([1 -real(z_poles(ceil(order/2)))], den) ; % (no idea why this works)
end ;

H = freqz(num, den, 1024) ; % get frequency response vector to attain the
                             % currect DC gain
num = num/abs(H(1)) ; % the magnitude of the first value of the frequency
                      % response vector is the magnitude response at 0
                      % frequency, a.k.a. the DC Gain.  Divide the numerator
                      % coefficients by the DC Gain to make the DC Gain 1, or
                      % 0dB.

%% Filter the noise: %%
noise = filter(num, den, noise) ; % apply the digital filter to the noise.

%% Add the filtered noise: %%
noisy = noisy + noise ; % the previous 'noisy' vector was simply the offset
                        % benchmark.  After this command, it also contains the
                        % noise, making it the final noisy vector.


%% for debugging: %%
% figure(1)
% [H, w] = freqz(num, den, 1024) ;
% plot(fs*w/2/pi, 20*log10(abs(H)));
% title('freqz of the digital filter')
% xlabel('frequency (Hz)')
% ylabel('magnitude (dB)')
% figure(2)
% [H, w] = freqz(num/abs(H(1)), den, 1024) ;
% plot(fs*w/2/pi, 20*log10(abs(H)));
% title('freqz of the 0db dc gain digital filter')
% xlabel('frequency (Hz)')
% ylabel('magnitude (dB)')
% figure(3) ;
% psd(noise, 256, fs) ;
```