IMPLEMENTATION OF COMPRESSIVE SAMPLING FOR WIRELESS SENSOR NETWORK APPLICATIONS

Nathan A. Ruprecht

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

May 2018

APPROVED:

ACKNOWLEDGEMENTS

Of course, this one page cannot possibly show the gratitude I have towards the people that have helped me get to this point and tolerated my levels of stress.

I will forever be grateful to the UNT Department of Electrical Engineering for not only giving me the skills and foundation in my education, but finding ways to make my life just a tad more convenient or bearable. Since I started, the faculty and staff have treated me much better than I could hope for. And with a limited time at the university, they helped me take advantage of every opportunity I could to make the most of it. I thought I would be leaving here with just a bachelor's degree, but you have helped the nerd in me find my passion in forever learning and striving for the next level. Thank you Dr. Li for pushing me beyond my limits. No matter how much sleep I would get at night, I have been exhausted this past year trying to show my worth. I am fortunate to have worked under you, and hopefully we will stay in touch as I go forward.

I would be lost without my family and wife, Jazmin. There were days I forgot to eat because I was too "in the zone" or trying to produce enough results for my next checkpoint. Or the days where I was mad at the slow progress or failed tests that I was itching to break something. The small significant successes does not even compare to the failures and frustrations. You saw all the bad times and helped me get through it – whether it was a glass of wine and dinner or a 3am gym session with banging weights and loud music.

I could not have done any of this program without everyone in my life helping me along. I am proud of the work done, and excited for the next chapter.

TABLE OF CONTENTS

LIST OF TABLES AND FIGURES

1 INTRODUCTION

1.1 Digital Signal Processing Background

The first challenge of processing a signal is acquiring one. We have analog-to-digital converters (ADCs) that sample an analog signal, quantize it to levels, and convert it to a binary equivalent. Once digitized begins the works of digital signal processing (DSP) for a number of reasons and applications. How well the continuous, analog signal is represented as a discrete, digital signal all depends on the ADC. Think of it in terms of a sinusoid represented on an x and y axis coordinate plane. We see a smooth line as time carries on, but a computer needs set points to represent that line. The more accurate we can capture the y value depends on the number of bits used in the ADC, or its resolution. The x value corresponding (along the time axis) depends on the speed of the system or its sampling frequency (F_s). Every $1/F_s$ seconds, the ADC will sample and start conversion. The resolution of the ADC is a hardware specification that is set when it is manufactured, but the sample frequency depends on the user that is using the chip.

The Nyquist-Shannon Theorem originally came about in 1928 by work of Harry Nyquist and proved true by Claude Shannon in 1948 to help those users choose a sampling frequency. Or more so to tell them the lowest sampling frequency that can be used. The Nyquist Rate (or Nyquist Frequency) is a lower bound saying the slowest one can sample an incoming signal and be able to accurately reconstruct or represent it is at two times the maximum frequency component within the signal. Any slower and we risk aliasing. Since a signal follows the structure of $x(t) = sin(2\pi f t)$, then a signal such as

$x(t) = sin(3000\pi t)$ would have a frequency of 1,500 hertz (Hz) and require the ADC to sample at 3,000 Hz to accurately capture the signal.

To rewind and break it down, take a signal with a frequency of 2 Hz, so 2 sample per second. Nyquist Rate says the ADC needs to sample at a rate of at least 4 Hz. So the ADC starts sampling a time $t = 0$ and will again sample at $t = 0.25, 0.5, 0.75$ and so on. If you pull out a calculator and take the $sin(2\pi ft)$ with $f = 2$ and $t = 0, 0.25, 0.5, 0.75$ you will get zero as the results for all of them. Now go back and do the same process, same time periods for sampling, but at 4 Hz – we get the same string of zeros. Same thing for 6 Hz, and infinitely many other frequencies. So how do we know which frequency is represented when all the system is given is zeros? Better yet, how can we represent a sin wave with a flat line of zeros? It turns out that the reconstruction of x from a sequence like that above is a complicated process. It involves the weighted some of a interpolation function, or in other words the weighted sum of infinitely many sinc functions. We bring that up to be aware and reassured that reconstruction is taken care of, but not to go down that hole itself.

The term before of aliasing happens when we think we are looking at a signal of a certain frequency, when we are actually seeing the results of another, or the bleed through of a higher frequency component. For our above example, 4 Hz and 6 Hz is an alias of our original 2 Hz signal that was being sampled. The deciding factor on what to believe is the sampling frequency. Since the sampling frequency was set at 4 Hz, the maximum frequency component that can be reconstructed is the 2 Hz signal. And for the past 70+ years, this theorem has been the guideline for those doing DSP. But after 70 years of incredible innovation in the world of technology, is that really the best we can do? The radios in our cars operate at megahertz ($1 * 10^6\ Hz$) so in order for me to listen to the 103.7 radio station, there is a receiver having to sample once every 0.00000000482 seconds! Machines like radars operate in the 300 MHz to 1 GHz range known as the ultra-high frequency band (UHF). Surely today's engineers can outdo 70 year old theories. And there is such a theory to challenge Nyquist, but still young with hardly any application work.

1.2 Motivation and Breakdown of Thesis

Compressive sensing (also referred to as compressed sampling, sparse sampling, or CS) is a theory to challenge Nyquist Shannon Theorem. The idea came about in the 1970s when seismologists were able to produce images of layers of Earth they were studying based on data that was not satisfying the Nyquist – Shannon criteria. The idea took shape and developed since then. In 2004, Emmanuel Candes and David Donoho better defined CS and proved that knowledge of the signal's sparsity will allow fewer samples to be taken and still reconstruct the original signal.

Since then, there have been a considerable amount of development in the field. Most notably in fields with imaging such as medical MRI, radar, and simplifying every day digital cameras. There has been development of using CS in astronomy, biology, and image and video processing. In communications, there has been an example of using CS in cognitive radio systems. With modern applications of DSP, the performance of ADCs are pushed to their physical limits. CS implements a concept of analog-to-information conversion (AIC) as a way to reduce redundancies in usually oversampled signals.

An easy example of applications not developing because of ADC constraints is ultra-wide band (UWB) signals that range from 3.1 to 10.6 GHz. Needing an ADC take a sample every 94 ps or faster is a heavy demand. In the network layer, one can use CS to reduce redundancy and nodal constraints in a wireless sensor network (WSN) used for data gathering. Or at the application layer, we can again use CS to help characterize network performance. This paper aims to implement CS in a WSN at an easy to understand, and easy to afford use.

To break down the problem, the objectives are to first understand the theory and foundation for CS systems. We do not want to get too far into the mathematics that have already been proven, but know their performance and how to use them in either simulation or hardware implementations. After a solid foundation in theory, we move to know the pieces that fit into the system, how to measure/rate their performance, and how to choose different pieces depending on the scenario for system design. Knowing the theory and characteristics of a system, we move to simulating a system designed in Matlab, then to move onto hardware implementation in both a microcontroller unit (MCU) and a single board computer (SBC). The format of this thesis follows the order of our objectives: foundation of CS Theory in Chapter 2, “pieces” of a CS system and optimization is Chapters 3 and 4, implementation on our three platforms in Chapter 5, overall system results in Chapter 6, followed by a wrap up Chapter 7 on how this thesis performed.

2 COMPRESSIVE SAMPLING THEORY

2.1 Compression

The overarching goal of a CS system is to compress a signal vector to a size that would be sub-Nyquist, then being able to reconstruct it. For this paper, we still need to sample at Nyquist Rate to have the original signal to compress and compare to. There has been some literature on how to randomly sample observations of a signal, but only theory was discussed and it would still require an ADC capable of sampling at Nyquist Rate. The two principles CS relies on is (1) sparsity, which is a characteristic of the original signal that helps with compression, and (2) incoherence, which is a characteristic of the system's measurement matrix that helps with reconstruction [5]. To compress a signal in our case, it is just a matter of matrix multiplication. Say that we have our original signal, $x \in \mathbb{R}^{N\times 1}$ which is a 1-Dimensional column vector (i.e. audio signal) that has N samples. If we multiply x by $A \in \mathbb{R}^{M\times N}$, which is a 2-Dimension matrix with M amount of rows and N amount of columns where $M \ll N$, we get a resulting column vector $y \in \mathbb{R}^{M\times 1}$ that is a representation of x, just smaller. Compression then to y observations is simple matrix multiplication by solving:

$$y = Ax. \tag{1}$$

Equation 1 is better illustrated by:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,N} \\ A_{2,1} & \ddots & A_{2,N-1} & \vdots \\ \vdots & A_{M-1,2} & \ddots & \vdots \\ A_{M,1} & \cdots & \cdots & A_{M,N} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}. \tag{2}$$

For terminology sake, we call x our original signal, y is the observation matrix, and A is the measurement matrix. The measurement matrix can actually be broken down into two parts – Φ and Ψ. So we can better describe the above equations as:

$$y = \Phi\Psi x, \tag{3}$$

where Ψ is a transformation matrix such as the Discrete Cosine Transform (DCT) or the Fast Fourier Transform (FFT), and Φ is a distribution matrix that will randomize x using a Gaussian/Normal or Uniform Distribution. We will discuss and examine characteristics of Φ and Ψ later but we need to introduce them now as a foundation for CS Theory.

With the compression knowledge of CS Theory and terminology used, we can better show an example in Matlab. Let x be a function of time consisting of the sum of two sinusoids at frequencies of 750 Hz and 1750 Hz, we have an example signal being:

$$x(t) = \sin(\,1500\pi t\,) + \sin(\,3500\pi t\,). \tag{4}$$

Nyquist Rate dictates that our system samples at a minimum of 3500 Hz. Real-world systems over sample in which the system uses a sample frequency, Fs, much larger than the required Nyquist Rate. For our example, let us use $Fs\ =\ 8\ kHz$. To paint a better picture, that means we should have 8,000 samples if we run this system for 1 second since Hertz is a unit of samples per second. Figure 1 shows the first 32nd of a second

(0.03125 of a second) with x being shown in the time domain on the left and frequency domain on the right.

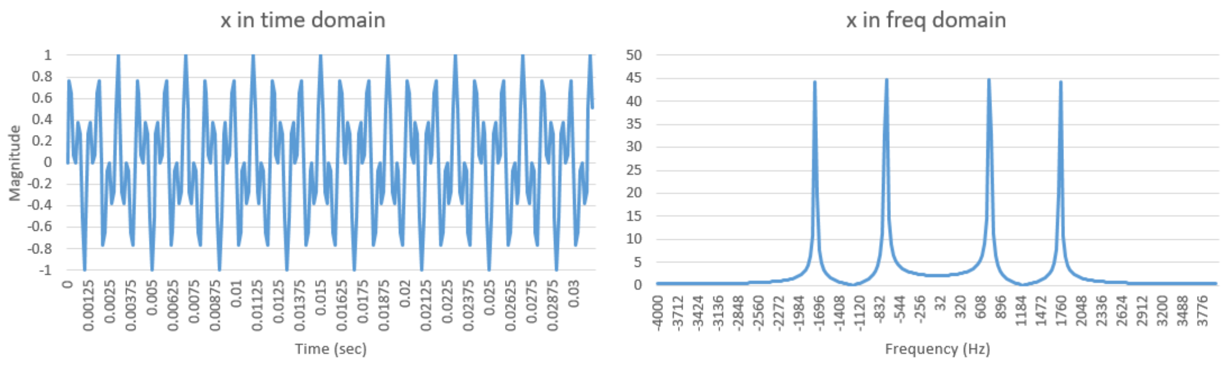


Figure 2.1: Example Original Signal (x)

With x being a column vector of size 8,000 being compressed to sub-Nyquist (let’s say Nyquist Rate – 1 for proof of concept), the measurement matrix would be $3499\ x\ 8000$. That is computationally huge and would take a long time to reconstruct the original signal. What we do instead is break it up into frames. Taking every 1000 samples of x instead would mean A is resized to $437\ x\ 1000$ which is much more manageable for reconstruction. This idea of using frames to break up a large signal is how we address all signals in this paper.

It is also worth noting how we show Nyquist Rate. For the above example, Nyquist Theorem dictates we have 3500 samples to represent each second of x ($Fs = 3500\ Hz$). Since we broke up the original signal into 8 frames, each frame would need $3500/8 = 437.5$ samples to represent x. Since we cannot have a fraction of a sample, we round $1749/8$ down to 437. Rounding like this happens often in code and causes error that adds up. How we measure error is the Root Mean Square (RMS) of the original signal versus the reconstructed, shown by:

$$RMS = \sqrt{\frac{\sum_{i=0}^{L=\,length(x)-1}[\,x(i)-\hat{x}(i)\,]^2}{L}}. \tag{5}$$

In the examples shown later in this paper, the signals are normalized (divided by the maximum value held in that signal) in order to better represent our data. For example, if we test an original signal whose amplitude components range from 0 to ± 5, the potential difference of the reconstructed signal can be different than if the original signal ranged from 0 to ± 2. With all signals being normalized, all reconstructed signals can be approached with the same level of criticism. Since we normalize x and $\hat{x}$ to compare to each other, the amplitude of the respective signals will range from -1 to 1. If the reconstruction process does not make any effort to guess what the original signal would be, it would return a string of zeros for values. With the scenario being a DC signal at 1, and the reconstructed signal being 0, the RMS would be 1. Worst case would be $\hat{x} = -x$ which would yield a RMS of $\sqrt{2}$. But this is unlikely since $\hat{x}$ would return zeros as default as stated, so our range of error should fall in $0 \leq RMS \leq 1$.

2.2 Reconstruction

Different reconstructions techniques coupled with rounding errors will produce different results, but all with some degree of error compared to the original signal. Figure 2 shows a reconstruction of our above example also zoomed in to the first 32nd second.

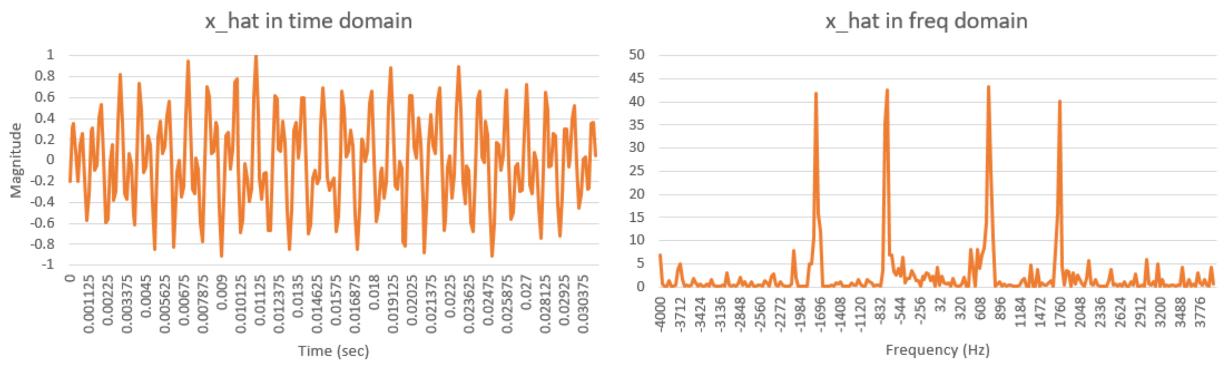


Figure 2.2: Example Reconstructed Signal ($\hat{x}$)

Note that the time domain of $\hat{x}$ (reconstructed) versus x (original) from Figure 1 are very similar, but we more noticeably see the error talked about when comparing $\hat{x}$ and x in the frequency domain shown by seemingly random spikes or noise.

The challenge in trying to reconstruct the original signal is trying to solve and undetermined system of equations (SOE). After compressing x into the observation matrix y, we have more unknowns than equations available to go backwards to reconstruct x *from* y. Think of the reconstruction process as a Sudoku problem – the complete puzzle is x but we are given just a few samples, or clues, of x to use (the observation matrix y). The more we compress, the less samples we have, and the harder the puzzle to complete.

With an infinite number of solutions, choosing the right one involves vector normalization. Most people are familiar with Euclidean distance norm, l_2, or even the “Manhattan” norm, l_1. The most correct solution is an iterative process of minimizing $l_p(x)$ normalization [2]. For example in our system, solving the l_2 norm of a system is equivalent to the inverse of A in order to solve for $\hat{x}$ as our reconstructed signal, such as:

$$\hat{x} = A^{-1}y. \tag{6}$$

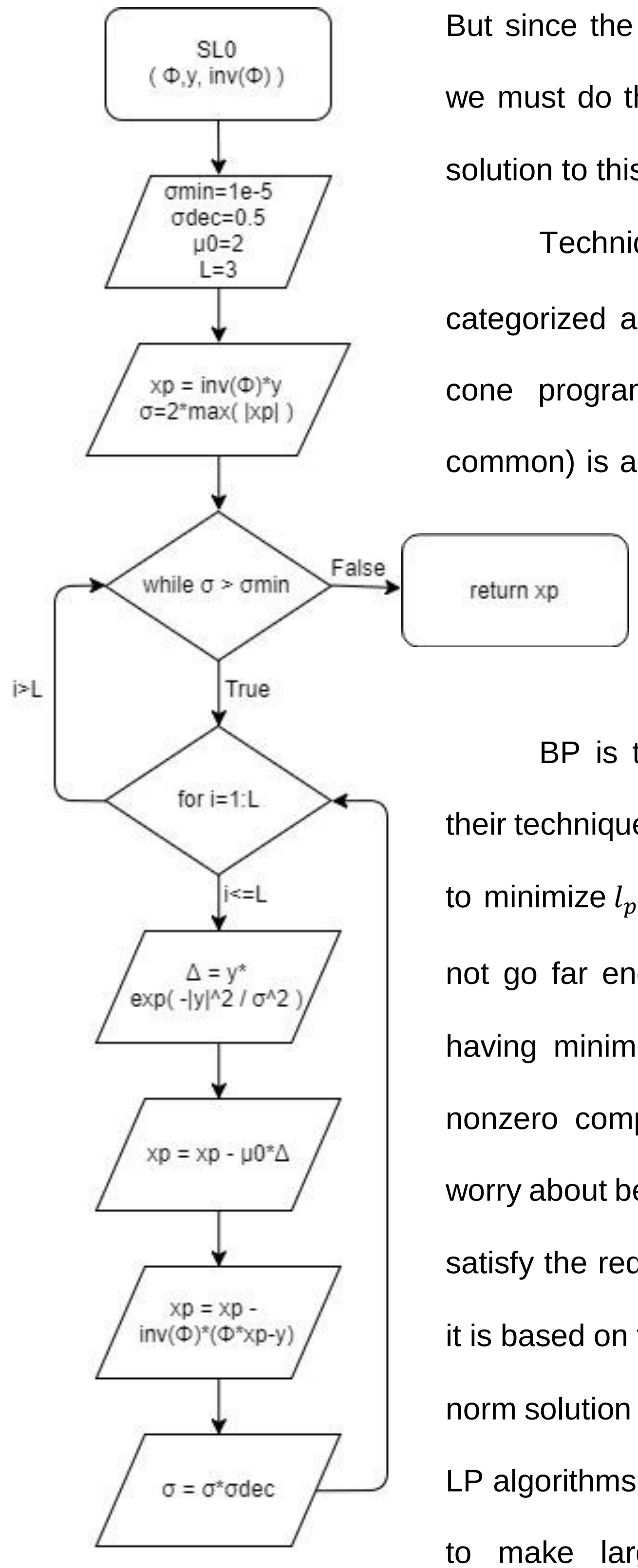


Figure 2.3: SL0 Algorithm Flowchart

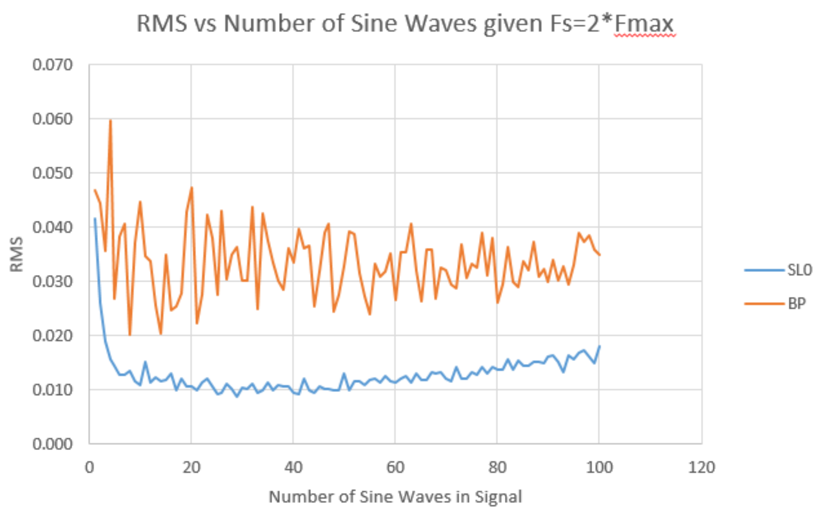
But since the measurement matrix will be rectangular, we must do the pseudo-inverse of the matrix to find a solution to this l_2 norm.

Techniques that find the minimum l_p norm are categorized as linear programs (LPs) or second-order cone programs (SOCPs). The most notable (and common) is a LP method known as Basic Pursuit (BP) that finds the vector with the smallest l_1 norm [3]. Basic Pursuit solves for $\hat{x}$ by:

$$\min\|\hat{x}\|_1 \text{ subject to } A * \hat{x} = y. \qquad (7)$$

BP is the standard that programmers compare their technique to in order to test viability. Since we want to minimize $l_p(x)$ norm, one could argue that BP does not go far enough by just solving l_1 norm. A solution having minimal l_0 norm is minimizing the number of nonzero components. But then the system needs to worry about being too sensitive to noise that could easily satisfy the requirements of l_0 norm. Fortunately for BP, it is based on the observation that for large SOEs, the l_1 norm solution will equal l_0 norm solution [11]. By utilizing LP algorithms, specifically interior-point LP, we are able to make large scale problems tractable. Another common approach is Matching Pursuit (MP) which is very fast compared to BP but does not necessarily provide a better solution in regards to error. There are a number of techniques that exist for CS reconstruction such as Orthogonal Matching Pursuit (OMP), Regularized OMP (ROMP), and Compressive Sampling Matching Pursuit (CoSaMP) to name a few.

The reconstruction technique we look more into is called Smoothed l_0 norm (SL0) which has Matlab code (and CS example) available online [11]. As mentioned before that a problem with using l_0 norm is that it is too sensible to noise, this comes from the l_0 norm being a discontinuous function. SL0 attacks the l_0 norm by approximating an equivalent continuous (hence the naming "smooth") function paired with a minimizing algorithm suited specifically for continuous functions [11]. The resulting algorithm (flow chart shown in Figure 2.3) is based on interior-point LP solvers to approximate the l_0 norm, followed by minimization using steepest descent method, looping on itself as an iterative process to better estimate the target function.



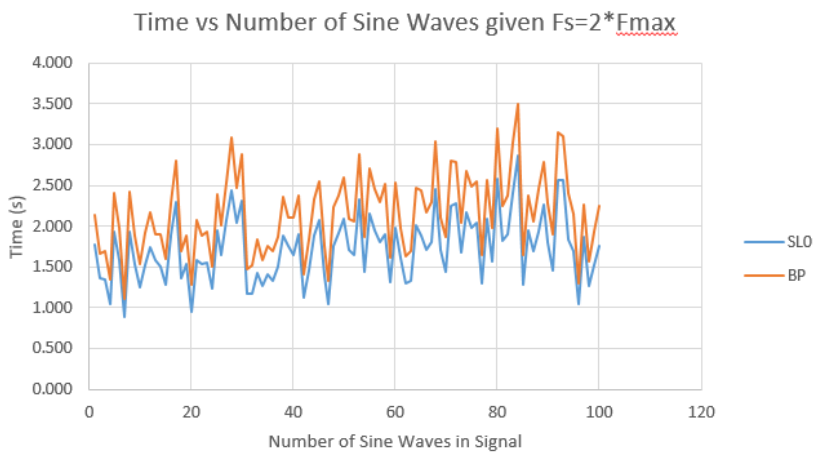
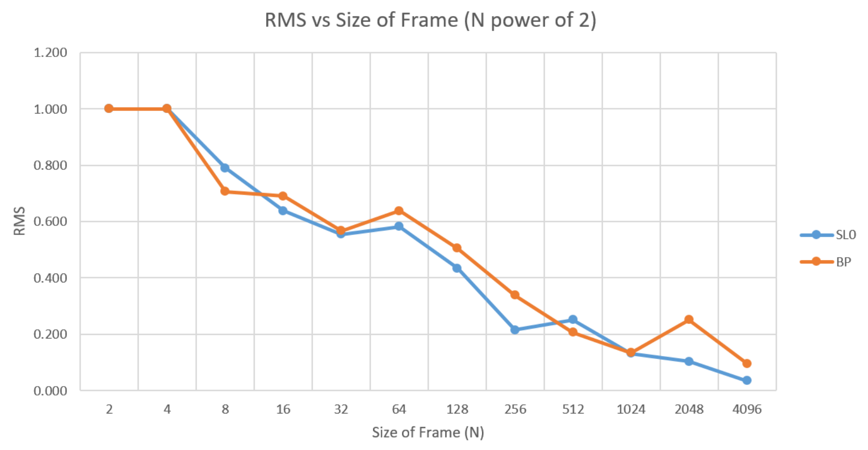


Figure 2.4: BP vs SL0 with Increasing Complex Signals

Then the question is how SL0 compares to BP? Time and error are both a critical factor in today systems and a matter of balance between the two. But also a matter of how to test the two techniques? Let's first show how BP and SL0 deal with an increasingly complex signal. If x is simple and only has one frequency component, we could argue that x can be compressed to just one sample – the value of that frequency. But as we added sinusoid waves at different frequencies, x is more complex and harder to represent at a compressed state. The most complex signal is one whose entire spectrum is full with infinitely many frequency components from 0 Hz to $F_s/2$ Hz. The top graph of Figure 2.4

shows the error of the two techniques versus the number of sine waves comprising x. In this case, SL0 performs better than BP with regards to error. But looking at the lower graph which shows the time it takes for the algorithms to compress and reconstruct, BP and SL0 are very close. They follow the same trend with SL0 just slightly better for speed. So with increasing signal complexity, SL0 is much better with reducing error, and just slightly faster than BP.



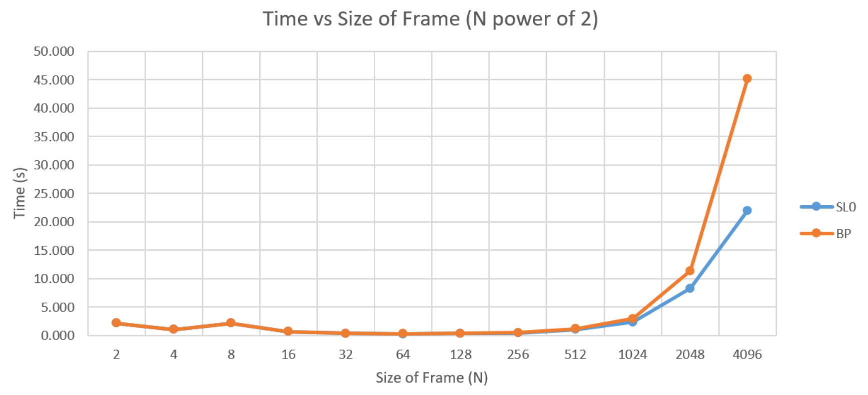


Figure 2.5: BP vs SL0 with Varying N sized Frames

Another concern goes in line with how we approach processing a signal. As mentioned before, we process x in frames by using every N samples. So if the entire length of our signal is 8000, and we can represent (compress) those 8000 into 1000 ($A = 1000x8000$ matrix), we introduce rounding error when splitting into frames. Using 9

frames for example, each frame would have $M = 111.\overline{1}$ ($A = 111.\overline{1}x888.\overline{8}$). Now we're representing x with a rounded number of samples into y. So how do we choose a size N, or number of frames, that finds a balance between speeding up computations, while reducing error due to rounding? Figure 2.5 is a similar format as before with error of BP and SL0 shown above with the time of execution compared on the lower graph. A key point is that we icremented N to be a power of 2. Reason being for hardware implementation on a MCU such as the MSP432, transform functions can only be performed on certain sizes of N, all of which much be a power of 2. With the above comparison, we have a better idea of how the two techniques would compare should they be implemented. Again, the two are very similar in results and trend, but SL0 is slightly less erroneous and faster.

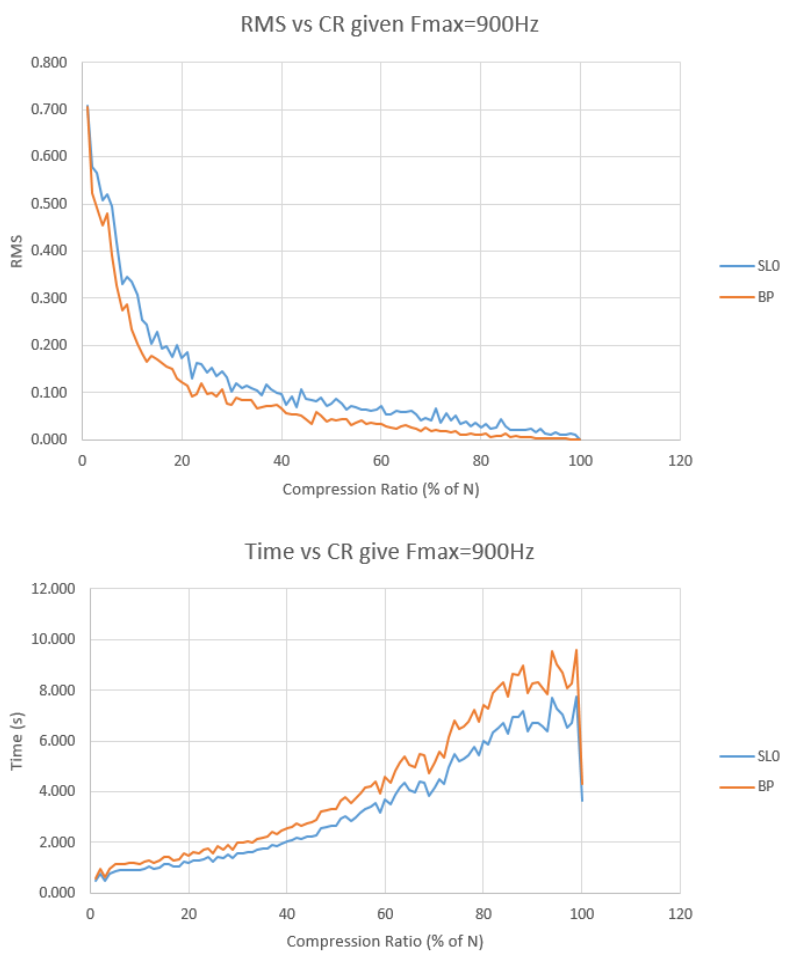


Figure 2.6: BP vs SL0 with Varying Compression Ratio (CR)

We compared BP and SL0 by varying the signal complexity, and size of N, now if we were to vary M (the size of y which represents all of x). Figure 2.6 again shows the comparison of the two as we did for the previous figures mentioned. We changed M as a percentage of N. This ratio will be referred to often in this paper as our compression ratio given by:

$$CR = 100\% * \frac{M}{N}. \tag{8}$$

Holding the signal to be the same with a single frequency component at 900 Hz, we sample at Nyquist Rate to create x, then loop the program to decrease M as a percentage of N (varying the Compression Ratio). Understandably, both algorithms reach the upper bound of RMS when M is 0 (100% CR) of 1, and lower bound of RMS when $M = N$ (0% CR) of 0 error. But the in between values still define the performance of the two. With similar values and trends, BP has a slightly lower RMS than SL0, yet slightly slower time of execution.

Based on varying the CR alone, we could not determine a better algorithm. But from Figures 2.4 and 2.5, we know that SL0 overall performs better than BP and beats the standard set by it. For the rest of this paper, results are based solely on SL0 reconstruction given by a reference [11]. Instead of digging into an algorithm already created that is one piece of the compressed sensing puzzle (reconstruction), this paper will go more in depth to CS properties and how we can objectively measure the performance of a CS system. And knowing the characteristics of all the pieces beforehand will let us design a better performing system without blinding guessing coefficients and parts. The next two chapters describe how the measurement matrix, A, is so key to the system's performance and how Φ and Ψ come to work together.

3 SIGNAL SPARSITY AND MATRIX COHERENCE

3.1 Need for Sparsity and How to Measure It

Compressive Sensing exploits that many signals are sparse or compressible. Remember back when comparing SL0 to BP depending on the complexity of the signal – one value is easier to represent than a thousand. Sparsity then is defined by how many non-zero values are in the signal. The original signal needs to be sparse in *some* domain. Rarely (if ever) x is sparse in the time domain so we must transform it. For a given $x(t)$, f is the sparse representation in a basis domain Ψ shown with:

$$f = \Psi x, \tag{9}$$

where x and Ψ are still our original signal and transform functions respectively that we have discussed thus far.

We show the Matlab implementation of 5 transform matrices so f does not necessarily have to be the FFT or DCT of our original signal. Because of that, we have to keep Ψ in mind when representing/analyzing the spectrum as the length or characteristics of the signal change drastically after multiplying with the transform matrix.

In a sparse domain, the majority of a given signal can be represented by the K largest coefficients [1] [2]. For example in an ideal audio signal with zero noise, the number of spikes in the frequency domain *is* the K sparsity. In real-world implementation, those signals are anything but ideal so there will be some level of noise within the signal. Since sparsity then is not as simple as the non-zero elements in the transform domain, we need to somehow measure the noise in the system, then go back through to measure the sparsity depending on a threshold to account for the noise. We implemented two different functions to do just that.

Figure 3.1 shows the first portion of finding the sparsity of the signal. Depending on the transformation matrix used (FFT and DCT for example) we shows how different parameters are paired with different Ψ. We want to find out the noise floor of a signal, or the value of pure noise. Since the lower frequencies (or indices if talking about DCT) contain the majority of the information, we do not consider measuring them. To avoid those lower indices of the

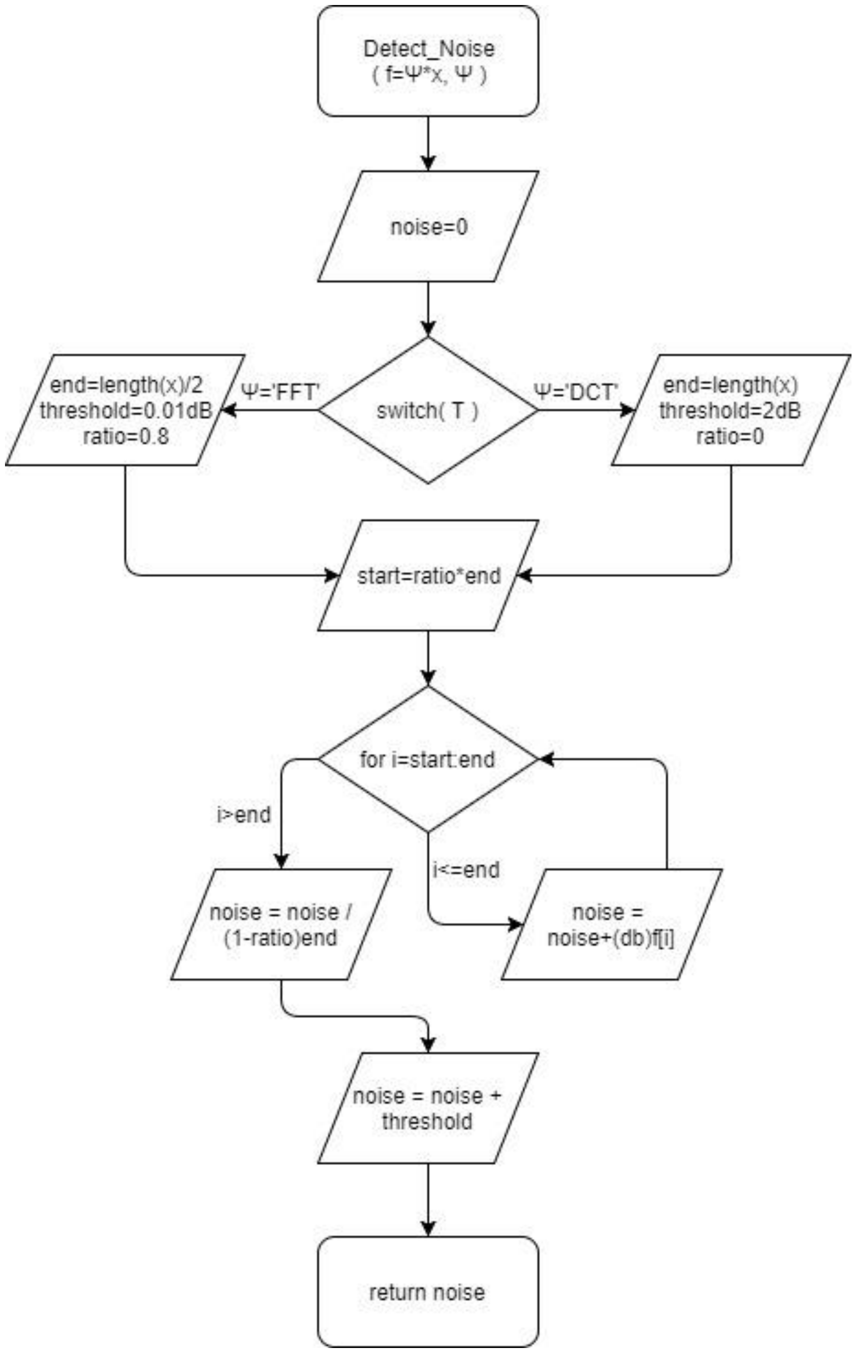


Figure 3.1: Detect Noise Function Flowchart

signal, we set a ratio of what percentage of the signal the function will measure. Going through the last small fraction of the signal, we are more likely to see values only containing noise. We average the energy of that portion (in decibels) and average what we measure. To be sure of any outliers, we add an additional buffer (threshold) to the noise floor we just measured. The noise can vary depending on the signal we expect to measure so this function could be used for calibration of the system before using it in the field.

With a value for noise of our expected signal, we can compare all energy values of the signal versus the noise level. Figure 8 shows the flowchart of our function that detects K sparsity of a signal. This function (unlike noise) goes through the entire spectrum of the signal, and simply counts the number of values that are above our noise threshold. We could zero out any values that are below the noise level so that we can fit our definition of sparsity being the number of nonzero spike or values of a signal in a sparse

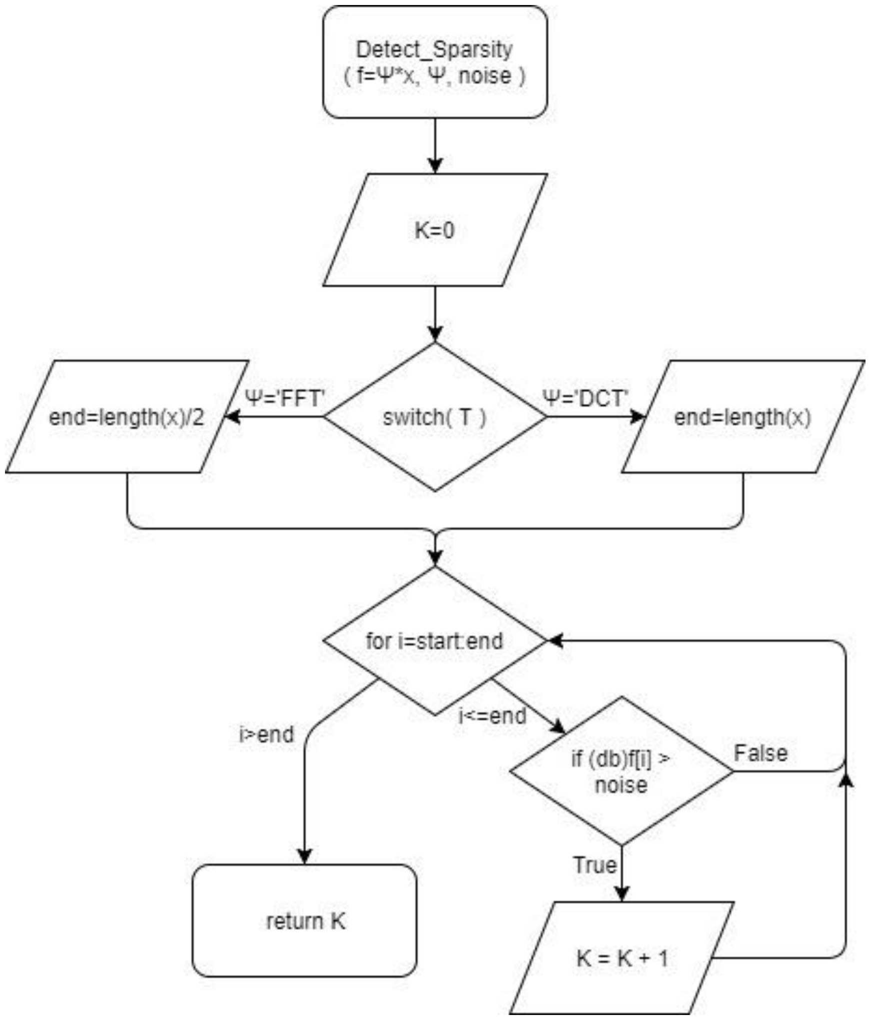


Figure 3.2: Detect Sparsity Function Flowchart

domain. These two function come together to detect a signals sparsity, but the accuracy depends on the very subjective threshold and ratio of signal spectrum to measure that is set when detecting the noise floor. A ride range of values are tested with figures showing the comparison later in the results section.

3.2 Magic Behind A = ΦΨ

Referring back to the reconstruction process as a Sudoku problem – at first glance, there are a large number of solutions. But with an iterative process (SL0), a unique answer can be zoned in on. Now the puzzles can range from easy to hard of course. The difficulty can depend on the number of values that are given to us (M) or maybe the actual value given is a hint to eliminate possible solutions. In a CS system, the measurement matrix is the key reducing the difficulty level of our problem by providing those hints to the machine doing reconstruction.

As we mentioned, A is actually a combination of a transform matrix (Ψ) and a distribution matrix (Φ). The transformation matrix helps with are need for sparsity that we just discussed, but what about Φ? This goes back to our original problem of why reconstruction is so difficult in the first place – and undetermined SOE. There are too many generic equations that satisfy reconstructing a sparse signal. To find that very specific answer, we need a very specific sequence, almost like encoding our signal. Multiplying our now sparse signal with a distribution matrix encodes that data we send. It appears like jumbled noise, but with the key can be simply decoded to our original signal. So the secret to a perfect CS system is a transform matrix that makes the signal perfectly sparse, and a distribution matrix that makes the signal perfectly random. This rough definition on if a matrix can be used for compressive sampling is what we call the Random Isometry Property (RIP).

In the introduction, we mentioned David Donoho and Emmanuel Candes front running the development of CS. They published a series of papers that attacked all fronts of the issue, including RIP. RIP is a pass fail equation of:

$$(1-\delta)\|x\| \leq \|\Phi x\| \leq (1+\delta)\|x\|. \tag{10}$$

Following same syntax as what we defined with $M \ll N$, $K\ sparsity\ < M$ saying the Φ is (K, δ)-RIP for every K-sparse vector $x \in \mathbb{R}^N$. It is the generic standard (much like BP for reconstruction) for confirming a distribution matrix as being 'good' for compressed sampling [19]. But using a Φ with some random distribution has a very high probability of passing the RIP test, so this is more useful to those trying to make a CS system with a deterministic matrix [8]. To test Φ alongside Ψ in how well the duo perform as A, we have matrix coherence.

We can first define a matrix coherence coefficient when comparing the columns of Φ against the rows of Ψ. For a perfectly random matrix, we actually want the lower bound of coherence coefficient, or to say to maximize the incoherence of the two matrices. With the coefficient set to μ below:

$$\mu(\Phi, \Psi) = \max_{1 \leq i,j \leq N} \left|\langle \Phi_i , \Psi_j \rangle\right|, \tag{11}$$

we have bounds of $1/\sqrt{N} \leq \mu \leq 1$ [14].

Multiplying a random matrix by any other matrix *should* yield another random matrix by some measure. The problem here is we do not just want the results of $\Phi * \Psi$ to be random, but the result of $\Phi * \Psi * x$ to be random since it is this we are transmitting. We then find the *mutual* coherence of A as a measurement matrix. We will also find that the mutual coherence coefficient is higher than just comparing $\mu(\Phi, \Psi)$ since it is not a random matrix projected onto another matrix, but the properties of the matrix itself. We also need to verify that A is orthonormal so our new equation for the coherence coefficient is:

$$\mu(A) = \max_{1 \leq i,j \leq N} \frac{|\langle A_i, A_j \rangle|}{\|A_i\| \|A_j\|}, \tag{12}$$

and implemented in Matlab with:

```
for i=1:N
     for j=1:N
          temp = abs( dot(A(i,:),A(:,j)) );
          temp = temp./[norm( A(i,:) ).*norm( A(:,j) )];
          mu = max( mu, temp );
     end
end
```

where i and j are still the rows and columns respectively, and the bounds are still$1/\sqrt{N} \leq \mu \leq 1$. What we aimed to show is how different transformation matrices are characterized when paired with different distribution matrices. One of the first problems we encounter is a familiar one that we have mentioned before: error due to rounding. When calculating the mutual coherence of A, we need a square matrix of size N so there are equal number of rows to compare to columns. The above equation is ideal as $N \rightarrow \infty$. Much like in the Introduction when comparing BP to SL0, we will show a varying N as being a power of two to better show performance when we switch to actual implementation. Table 3.1 shows the 5 transforms and 27 distributions implemented in Matlab to test against each other in regards to mutual matrix coherence. We already defined DCT and FFT, but there is also the Integer Wavelet Transform with a db1 (or Haar) filter (IWT1), the Integer Wavelet Transform with a db4 filter (IWT4), and the Walsh Hadamard Transform.

Table 3.1: Transforms and Distribution Tested for μ

Ψ	DCT	FFT
	IWT1	IWT4
	WHT	
Φ	Beta	Binomial
	Birnbaum Saunders	Burr
	Exponential	Extreme Value
	Gamma	Generalized Extreme Value
	Generalized Pareto	Half Normal
	Inverse Gaussian	Logistic
	Log Logistic	Log Normal
	Multinomial	Nakagami
	Negative Binomial	Normal
	Piecewise Linear	Poisson
	Rayleigh	Rician
	Stable	t Location Scale
	Triangular	Uniform
	Weibull	

The full results of the data is not shown, but the Figure 3.3 shows the average $\mu(A)$ across all trials for each given pair of distribution and transform. But it is a complete picture look at all the possible combinations. We want the best performing. To get a better picture of the forerunners, we take the 3 best distributions for each transform, along with the Normal distribution.

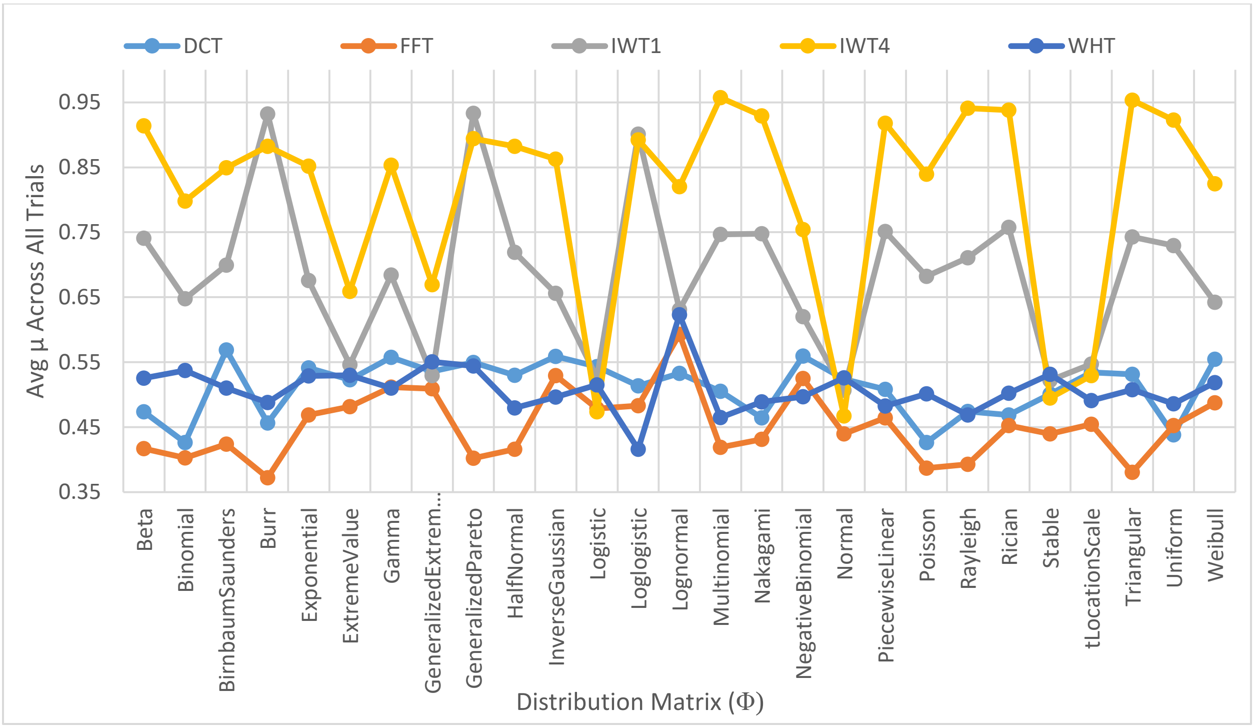


Figure 3.3: Coherence Coefficient of All Combinations of A

The reason we also compare the Normal distribution regardless of its performance is because almost all references found on CS defaults to using a Φ which is random with a Gaussian distribution. It is hardly explained, unless we look at it from an entropy point of view. A brief note on entropy is that is it a measure of information in a signal. It provides bounds on the minimal transmission rates using lossless and lossy coding. The more information in a signal, the more uncertainty between values, the more variation, the higher the entropy. A Gaussian variable has the greatest entropy compared to all random variables with the same variance [20]. This note, along with realistic constraints of implementing certain distributions in hardware, it is an understandable assumption for CS system designers to use a Normal distribution matrix.

Table 3.2: Top Performing Distributions for Each Transform

Ψ	Φ	Overall Rank (Out of 135)
DCT	Binomial	12
	Poisson	13
	Uniform	15
	Normal	61
FFT	Burr	1
	Triangular	2
	Poisson	3
	Normal	17
IWT1	Normal	49
	Stable	59
	Logistic	64
IWT4	Normal	25
	Logistic	29
	Stable	42
WHT	Log Logistic	8
	Multinomial	24
	Rayleigh	27
	Normal	65

To confirm the performance rankings of Table 3.2, we tested each combination of $\Phi * \Psi$ in a CS example using a music file for higher complexity and SL0 reconstruction of x. This example was ran 100 times to better represent the data's true average time of execution and RMS error. The size of each frame (N) was held constant at 128 (this is again set because of realistic constraints on what we have to look forward to for hardware implementation), and M set to 64 so a CR of 50%.

Table 3.3: Best Φ Performance in a CS Example

Ψ	Φ	RMS	Time
DCT	Binomial	0.007	3.126
	Normal	0.008	3.837
FFT	Burr	0.004	5.637
	Normal	0.005	6.394
IWT1	Normal	0.235	3.138
IWT4	Normal	0.233	2.548
WHT	Log Logistic	0.031	1.987
	Normal	0.030	4.240

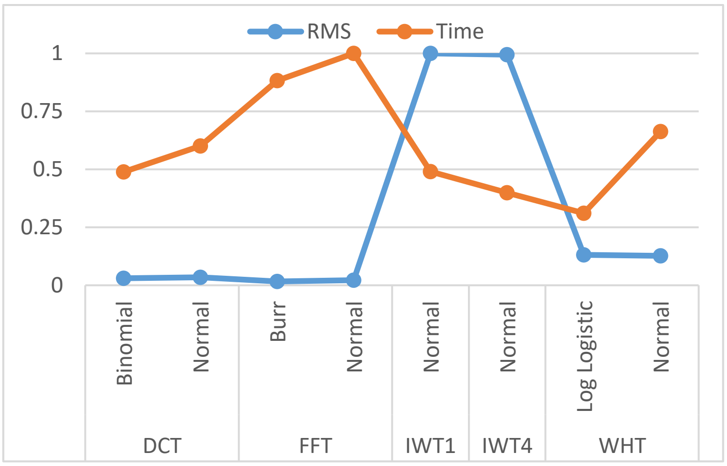


Figure 3.4: Normalized Results of Table 3.3

The best distribution for each transform (along with the Normal distribution if not already included) was ran against the same example, with the same CR, 100 times, and the results of the average error and average total time of execution are shown in Table 3.3. To the right of that in Figure 3.4 is those same results, but normalized to better show how the combinations fair against one another. Looking at the distributions paired with the FFT, we see it took the longest time to execute, but also the smallest error. But the reciprocal does not hold true. The Normal distribution paired with IWT1 and IWT4 had the highest error, but not necessarily the fastest to execute. Choosing a pair of Φ and Ψ for an actual system is about finding a balance between error and time. Finding the margin of what is allowed and how one factor is more important than the other.

In any sense, the results of Table 3.3 coincide with our assumptions made after looking at Table 3.2. Interestingly enough, when applying a DCT paired with a Normal distribution to an actual CS system, it performs on pair to using the FFT. Overall, we now have objective results that one could use for their argument of choosing any combination of A. Instead of blindly accepting Gaussian as the best distribution matrix, one could argue to use another depending on the signal, and therefore depending on the type of transform matrix best suited for the situation.

There is still room for error in our methodology. The coherence coefficient is tested with A being a square $N \times N$ matrix, but a CS system will employ a $M \times N$ measuring matrix. Performance did slightly change comparing Table 3.2 results against Table 3.3/Figure 3.4, but the top distributions were, for the most part, in an unchanged order. Since the performance of A depends on the mutual coherence, a good distribution is not enough, but how it works when paired with a transform matrix. That and as performance changes going from theory to implementation, different properties can push combinations for A up in those performance rankings.

The result of $\mu(A)$ actually help in more ways than one. Not only giving us direct results on expectations of the performance of A, but we also use the coherence coefficient as a variable when calculating how much we can compress x. In other words, the minimum M to represent N.

4 MINIMUM SAMPLES TO REPRESENT SIGNAL

The big question is how much our system can compress a signal, and how it stacks up to what is currently being implemented in the engineering industry. Optimizing A is a huge step in the right direction seeing as that one variable is half the challenge (compression). Then the choice in reconstruction technique completes our system and there by answering the performance. But what is that smallest compression ratio (CR) that a system can achieve?

Again, we start with a very broad and general equation on how much a signal can be compressed to: [12] [14]

$$M = O(K * logN), \tag{13}$$

where M and N are the variables we have been using thus far, and O is called "Big O notation." This equation is saying that the minimum M that can be used to represent N is "on the order of" multiplying the sparsity of the signal into a logarithmic scale of N. But this is another situation where the equation is generic in the sense that is would characterize a $N \to \infty$. This and the vague sense that M is on the order of a mix of coefficients and variables. From a combinations of ideas from multiple references, along with testing against examples for subjective measuring, we define a more specific formula for calculating M [14]:

$$M = \mu^2 K \ln N, \tag{14}$$

where μ is the coherence coefficient discussed in the previous chapter, K is the K-sparsity of our signal x, and N is the length of each frame that we split the signal into for better processing efficiency. From here, we see how all the previous chapters come together to form this equation. How much we can compress the signal depends on the complexity of the signal itself (K), the length of each segment we are processing and the possible error due to rounding (N), and the matrix properties that we are using to compress x in the first place (μ).

This equation still is not exact or universal in use. Since all the N we have tested up to this point have been relatively small and much less than infinity, we could use the equation as a starting point to narrow down to what we now have. As shown in Figure 4.1, more closely looking at the reconstructed $\hat{x}$ in the frequency domain, we see an improved accuracy for reconstruction with higher N sized frames. How we made these observations is by taking the same original signal x, and displaying it in both the time domain (upper left of each graph) and the frequency domain (upper right of each graph). Setting N to a different size each time, we run our CS system to entirety (compressing with MxN measurement matrix then reconstruction with SL0) and plot those results alongside the original in the same manner.

With $N = 128$ (Figure 4.3a), the reconstructed signal matches the shape of the original in the time domain, but looks to have a large amount of noise at all levels in the frequency domain. As we increase N (Figures 4.3b, c, and d), we see a more defined signal being reconstructed comparing the frequency domains of $\hat{x}$ versus x.

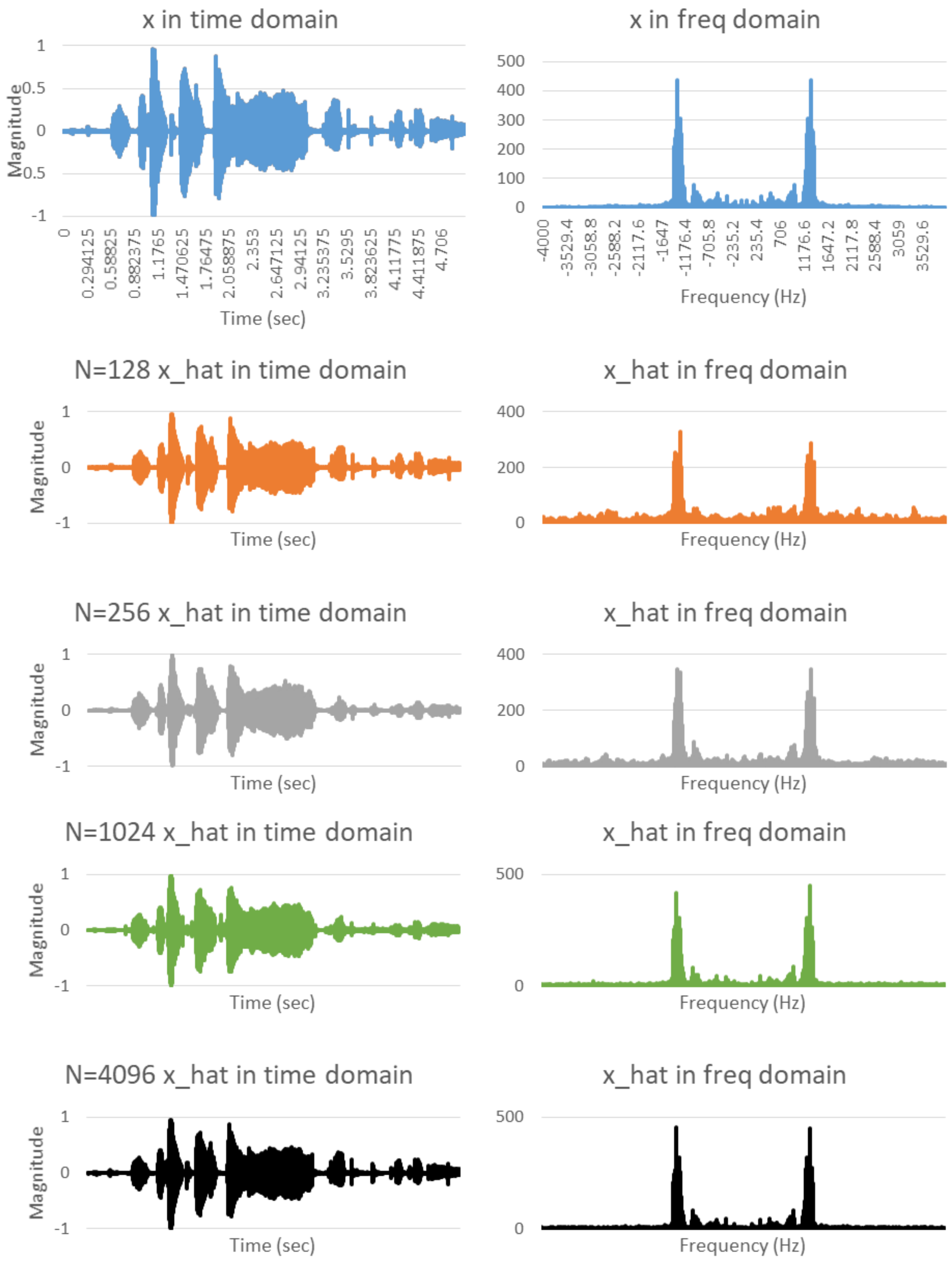


Figure 4.1: CS System Using Different Sized Frames (N)

There is still some wiggle room for a use to adjust parameters since M can also be an iterative process to calculate (going along with Big O Notation). But the most subjective part of our equation that is up for debate is calculating K. As discussed, in order to find K, we need to set some threshold for the noise to account for and then measure all spikes above that noise floor. The threshold can vary wildly with change in signal source or even type of transform matrix used. We need to see more on how changing the threshold effects the value of M, and therefore the error during reconstruction.

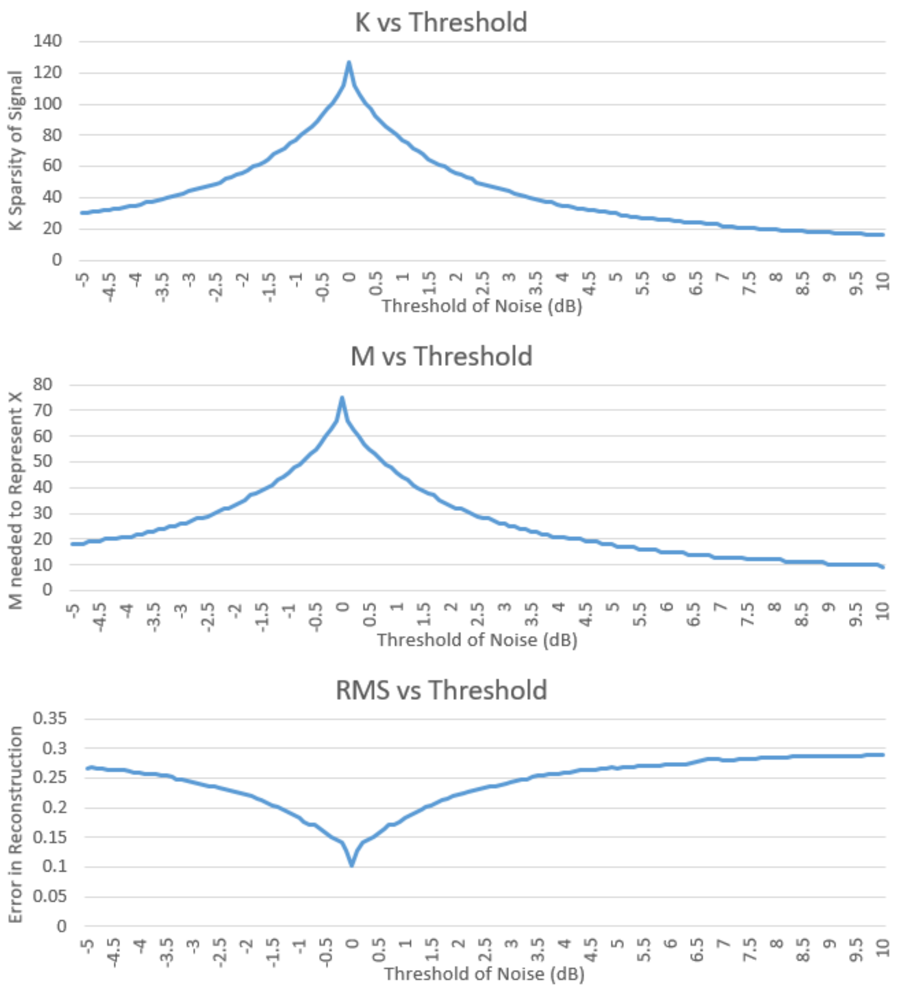


Figure 4.2: K, M, and RMS versus Noise Threshold

Using Matlab, and using the same original signal (a music file), we ran 100 loops of varying threshold. So on each loop, the function for detecting noise would be passed in a value going from $-5\ dB \leq Threshold \leq 10\ dB$. The sparsity of that signal was recorded for each threshold, along with the resulting M which depends on the value K. The system would continue to compress with that M, reconstruct to get $\hat{x}$, then get an RMS to characterize it all. So looking at Figure 4.2, we can see how setting the threshold of noise trickles down and affects the performance of the system. It happens that in our case (whether it is because of our system or because of the sample x used), the best threshold to use and reduce error is $Threshold = 0dB$.

It stands to note the equation we use for calculating M, along with all parameters now defined and ready for implementation, that our CS system does not compress to what would be below Nyquist Rate. In which case one could argue that it would be better to sample at Nyquist and transmit that data to be reconstructed instead, which is a valid procedure. But although our system is worse than Nyquist in size, it can be better in regards to RMS of the reconstructed signal. With more data points, and depending on the reconstruction algorithm, there stands to be less error. Either way, this system stands to be an easily implemented compression technique that is able to go below Nyquist and maintain error expectations.

5 CS SYSTEM DESIGN

Almost everything up to this point has been more theory than application. We have shown some basic examples/results to prove our point but not the actual system itself. All of what has been said is still vital to understanding how the system works as it builds on itself, but now we have laid down the foundation for designing and implementing a compressive sampling system.

5.1 Matlab Implementation

So let us start with the Matlab implementation as it has produced all the simulated results that we hold as the standard. We had Matlab go through and compress x frame by frame, and reconstruct $\hat{x}$ at right after to measure both time of execution and error. The first part of the Matlab code is actually to over sample. The baseline we are shooting for is to make a system comparable to those being implemented at a $\mathrm{F}_s = 8\,\mathrm{kHz}$.

Usually in those systems, there are analog front end (AFE) circuits that include an op amp and low pass filter set at $\mathrm{F}_c = 4\,\mathrm{kHz}$. What is basically happening is the ADC sample in the hardware system is sampling at $8\,\mathrm{kHz}$, but the samples being processed could be an alias of a much higher frequency. So in order to prevent aliasing, the signal is run through a LPF at $4\,\mathrm{kHz}\ (\mathrm{F}_s/2)$ to set all higher frequency components to zero. Now with a filtered signal, the ADC can convert the signal without the user worrying about aliasing. That in an actual system, not Matlab. So there first thing we do is over sample (say $48\,\mathrm{kHz}$ or $44.1\,\mathrm{kHz}$ for music files) to capture all frequency components that we can, and then run it through a LPF to mimic the response of the actual system we just described.

```
N      = 250;    % Order
Fpass  = 3250;   % Passband Frequency
Fstop  = 4250;   % Stopband Frequency
Wpass  = 1;      % Passband Weight
Wstop  = 1;      % Stopband Weight
Dens   = 20;     % Density Factor
% Calculate the coefficients using the FIRPM function.
b = firpm(N, [0 Fpass Fstop Fo/2]/(Fo/2), [1 1 0 0], [Wpass Wstop], {dens});
x = filter(b, 1, x);
```

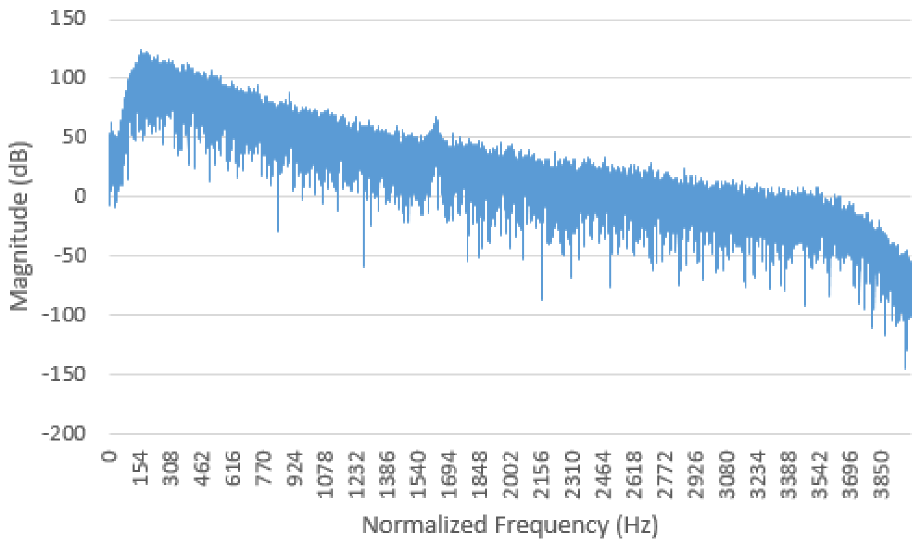


Figure 5.1: LPF Design using Matlab

Using the filter design function available through Matlab, Figure 5.1 shows our coded filter with an example frequency response. The main thing we want to see is how the magnitude of the signal shows a rounded dip. Ideally, the LPF is a square function with an undefined slope at $\mathrm{F}_c = \mathrm{F}_s/2$. But in reality, we have a band where the filter slowing deteriates the signal as it rises in frequency (that rounded off slope). For us, we set that band where the filter goes from completely off to completely on from $3250\ \mathrm{Hz}$ to

$4250\ \mathrm{Hz}$. It actually helps our system to slightly clip of the edges. When running the detect noise function, see the same magnitude the entire spectrum makes the noise floor close to the same as the signal. Whereas the sloped off edges help bring down the average value of noise across the entire spectrum.

With a filtered signal, we can down sample the signal which will keep the frequency components, but change the Fs from our set value of 48 or $44.1\ \mathrm{kHz}$ down to $8\ \mathrm{kHz}$. Using the code below, we set an integer value called $downratio$ to see how much we need to reduce the signal. If $\mathrm{F_s}$ is 48 kHz, downratio is a simple 6 down to $8\ \mathrm{kHz}$. But if we use audio files, in which most $\mathrm{F_s}$ is then $44.1\ \mathrm{kHz}$, we have a fractional $downratio$. That is why we round down to the nearest whole number. But now we have a mismatch in length between our original signal and the signal we want at $8\ \mathrm{kHz}$. So we “trim” our signal. We create a temporary holder of the trimmed length holding the values of x from the first index up to the new length. With the temporary x being a clipped version of x, we go back and reassign x to a down sampled version of the temporary variable. Our process of down sampling can be seen in Matlab by:

```
downratio = floor( Fs/8e3 );

trimmer = mod( length(xo), downratio );

len = length(x) – Trimmer;

temp = zeros( len, 1 );

for i=1:length(temp)

     temp(i,1) = x(i,1);

end



x = zeros( length(temp)/downratio, 1 );

for i=1:length(temp)

     x(i,1) = temp(i*downratio, 1);

end
```

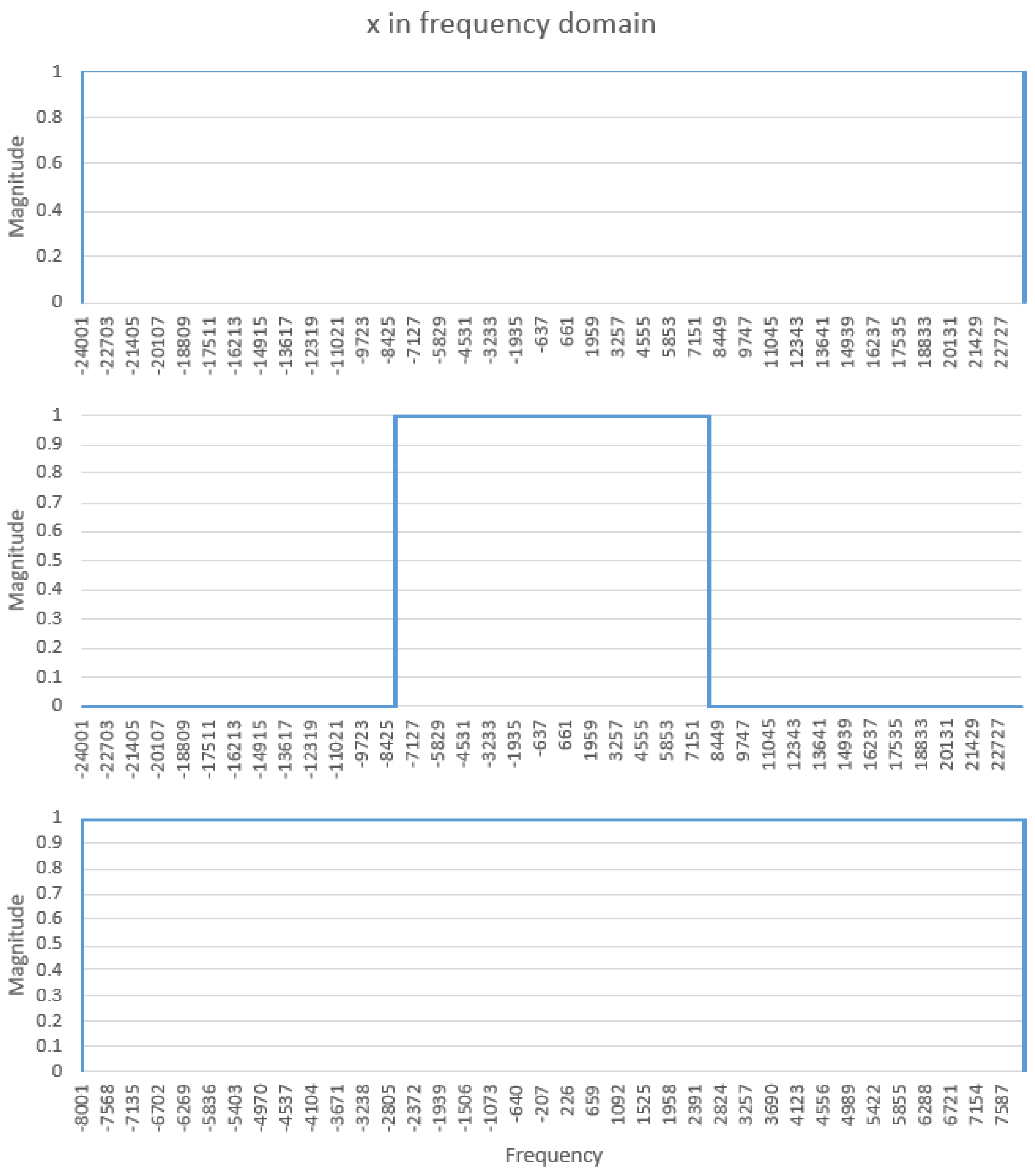


Figure 5.3: Ideal, Full Matlab Signal Going through a LPF and Down Sampling

In an ideal case with a completely full sqectrum from 0 to $\mathrm{F_s}/2$ ($\mathrm{F_s} = 48\ \mathrm{kHz}$), then running that complete signal through a LPF and downsampling, Matlab would produce graphs similar to those shown in Figure 5.3 above. Now that we have “simulated” an AFE by having Matlab pass the signal through a LPF and down sampled to our target $\mathrm{F}_s = 8$ kHz, we move on to our actual CS system.

For comparison, our Matlab code detects the maximum frequency component in the signal and determines what would be Nyquist Rate. This way, at the end of our program, we can see if our system is going below Nyquist or not along with the error and time of execution at that CR.

Figure 5.4 shows a flowchart for implementing a function to detect the maximum frequency component of a signal input. No matter the assigned Ψ, we look at the FFT of x so it is fed into the function as an input. Since the FFT will have a reflection of the signal around $\mathrm{F_s}/2$, we set the end index to the half of the length in order not to include that reflection's energy. The function goes through one half of the signal and add up to have a total energy level. We set the cutoff for detecting the fmax to be 95% bandwidth so 0.05 of

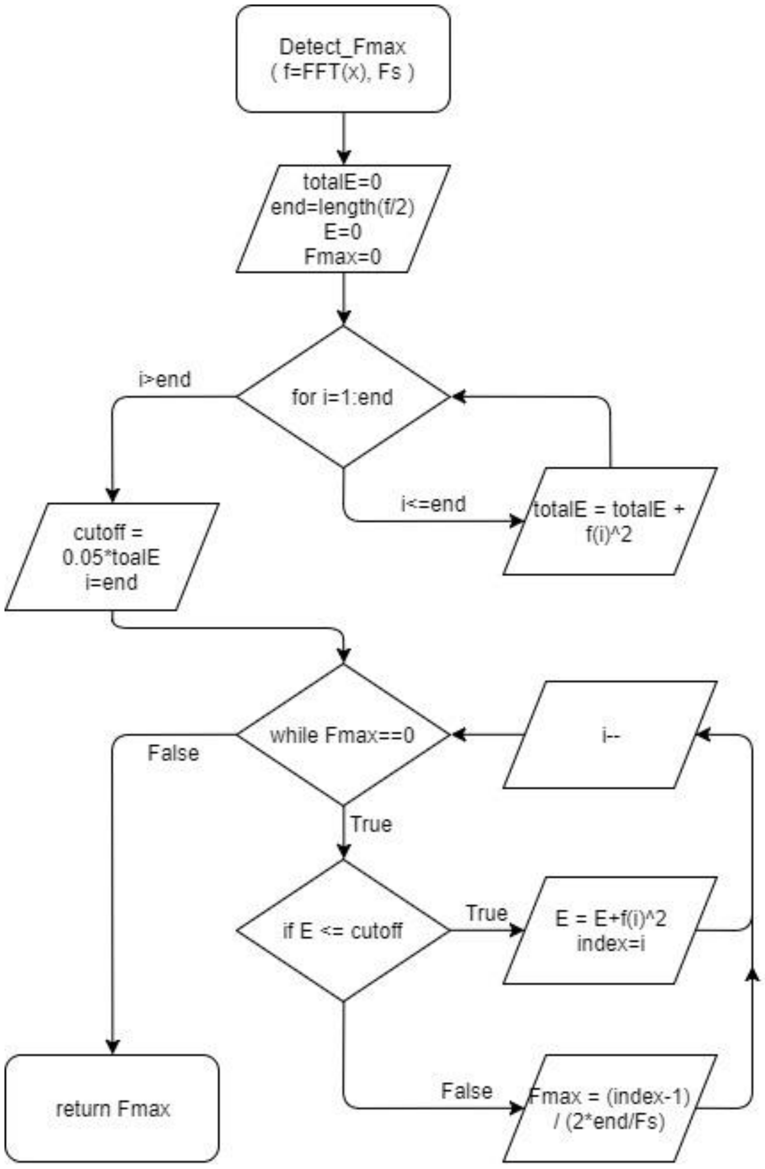


Figure 5.4: Detect True Fmax Function

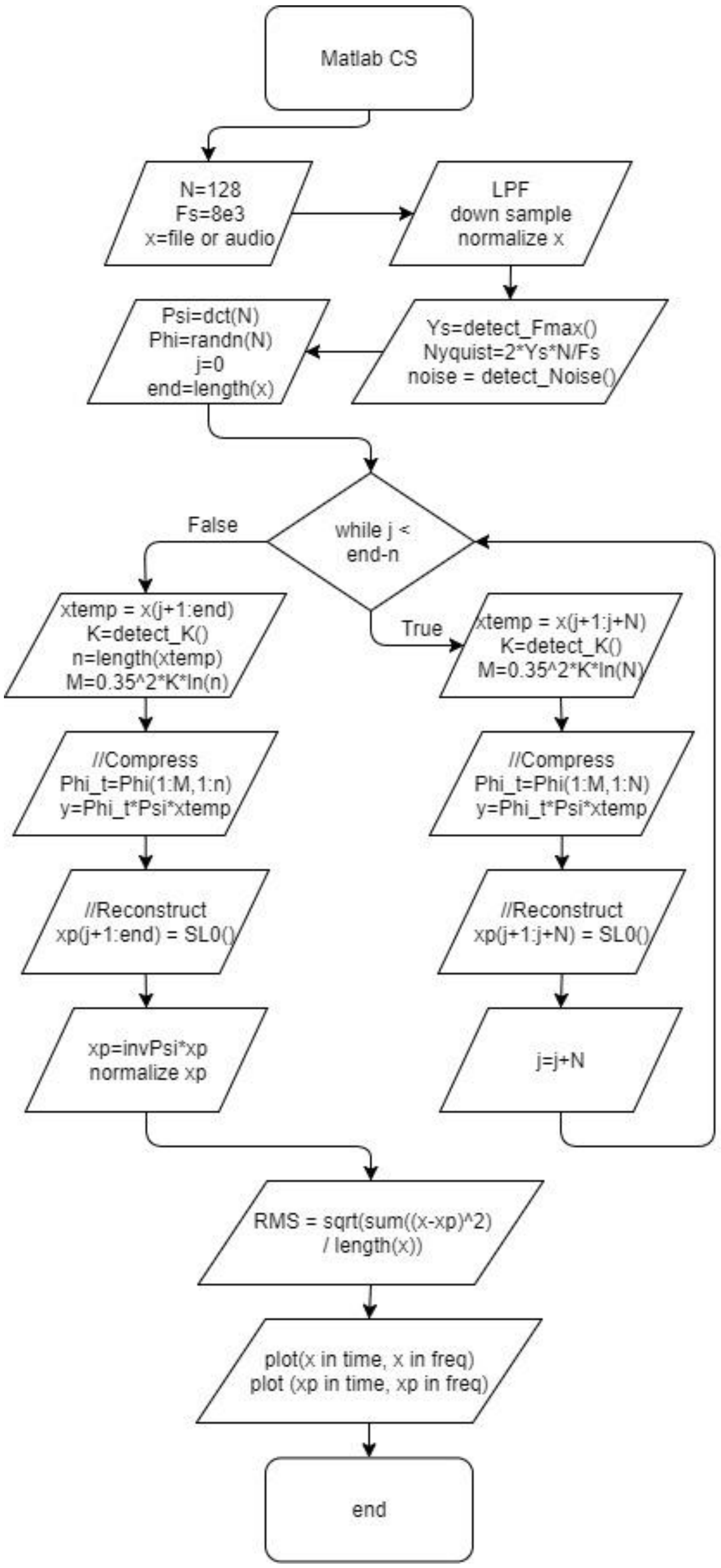


Figure 5.5: CS System Implementation in Matlab

the total energy. Then going back through half of the signal to find the index where the energy level breaks that threshold. The corresponding frequency to that index is the true max frequency in that signal.

Putting everything together, we have, for the first time in this paper, a flowchart of an entire system! Figure 18 shows a working Matlab implementation of CS from start to finish. Looking through it, we see the familiar functions built in that we have covered up to this point. Of course the first thing to do is initialize global variables. This is also where one could change the parameters of SL0 to pass them into the function. Once we have x as an audio file or recording using a microphone, we pass it through the LPF and down sample it as previously discussed. Then running functions to compare Nyquist Rate as well as detecting the noise floor for finding the sparsity of x. For hardware implementation of a system, we would want to detect the noise floor as a kind of calibration of the system to know what the noise level is of the environment before running the system. Then we define Ψ as the DCT paired with a normal distribution for Φ. Now to go into the actual compression and reconstruction portion of the system is a while loop. It is a while loop of “while there is still data, do this.” Since we split x into frames, we set a temporary value to hold a portion of x, detect the sparsity of the frame just snipped, and the resulting M needed to represent this temporary x. Choosing M rows of Φ, Matlab does matrix multiplication to get y. It does this while there is still an N sized frame left to process.

But splitting x into frames of N length, we need to account for x not being equally divisible by N, or a fraction of N left over. That is why there is one last stage of compression and reconstruction after exiting the while loop. We set a temporary n to the length of what is left in x, and run the same procedure to compress the remainder of x and reconstruct into $\hat{x}$ all while keeping indices organized. As a final note of comparison, we find the RMS and plot x and $\hat{x}$ both in the time and frequency domain for visible representation.

With “tic” and “toc” functions in Matlab, it is easy to inject timer components into the code and time any combination of parameters to see the performance of the system. When we talk about “time of execution” for a system, we talk about the time it takes to compress (the matrix multiplication command) along with the time it takes to reconstruct $\hat{x}$ (running the SL0 function and transforming into time domain). It was not explicitly noted before, but when SL0 reconstructs the signal, $\hat{x}$ is in the sparse domain. SL0 finds the unique solution to our randomized key, but we need to do the inverse transform to put $\hat{x}$ in the time domain before either saving it into a file or comparing it to x.

Table 5.1: Average Mu Over 100 Tests

		Average Mu
DCT	Normal	0.3589
	Bernoulli	0.3488
FFT	Normal	0.2776
	Bernoulli	0.2728

We noticed after running so many tests, that μ stays relatively the same. Of course it is going to change each time we re-initialize a distribution matrix, but it stayed around 0.35. To see about reducing the amount of unknowns and computation power used for CS, we ran 100 tests of making a measurement matrix with DCT and FFT paired with a Normal and Bernoulli distribution. We saved each μ and averaged them at the end of the loop. Table 4 shows those results. If using DCT as the transform matrix, it could be a safe assumption to assign $\mu = 0.35$ in order to cut down on computation power and time spent towards calculating μ, along will one less unknown to worry about in a system.

We have mentioned multiple times in this paper that we use DCT paired with a normal distribution, even showing it Figure 5.1 of our complete implementation, yet not completely showing why. We have the results from Chapter 3 on objectively measuring the matrix coherence coefficient as a better estimate of performance, but we want to continuously narrow down our options to the best fit solution for our system. So we need to compare FFT versus DCT using a normal distribution across multiple example signals.

The first thing to note is that M used for DCT is $M/2$ when using the FFT. Since FFT has a real and imaginary component, we need 2 memory spaces allocated to save FFT. Such that and M-point FFT requires $2 * M$ space; whereas DCT is only real components so an M-point DCT requires M space. We tested 3 difference music files: the first 15 seconds of “Clarity” by Zedd is a relatively short length of x but with a more full spectrum of frequency components, the first 60 seconds of “Whatever It Takes” by Imagine Dragons is a longer length of x with a quiet beginning then going into the more full spectrum, and the first 60 seconds of “Nuvole Bianche” by Ludovico Einaudi which is a classical song on the piano that provides more crisp notes at defined frequencies. We also had the idea of trying a deterministic measurement matrix. We still measure a Gaussian distribution with DCT and FFT, but also trying a Bernoulli distribution with DCT and FFT. Trying all 4 combinations of a measurement matrix against the three samples mentioned above is what we read in Figure 5.1.

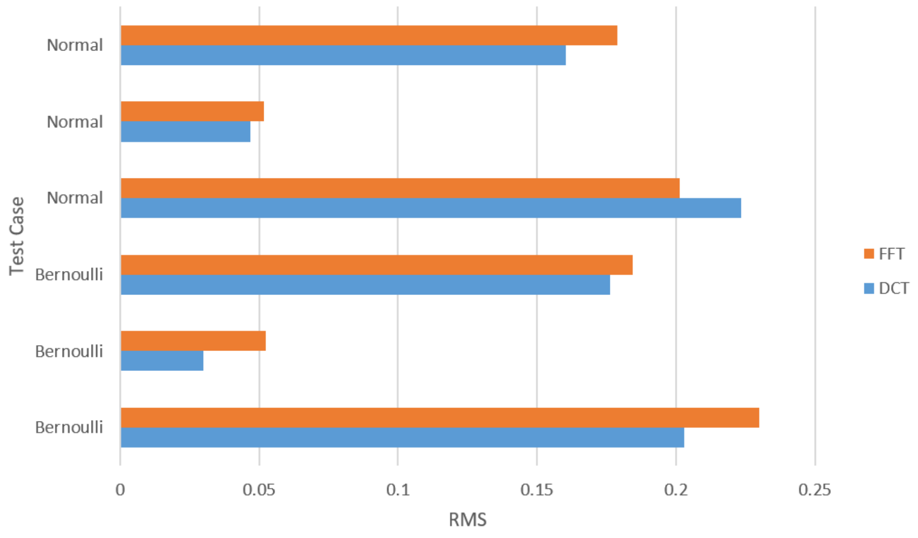


Figure 5.1: FFT vs DCT, Gaussian vs Bernoulli

The main thing to note is that DCT is superior to FFT in 5 of 6 of the tests. Although the best test results occur from pairing the DCT with Bernoulli distribution, it looks that the DCT and Normal distribution has an overall better performance (lower error) than using the Bernoulli distribution.

On the other hand, one could still make an argument for choosing a Bernoulli distribution instead. A Normal distribution is going to take more memory space in order to save such a large matrix of floating point values, whereas an $N \times N$ matrix following a Bernoulli distribution is a matter of saving 8-bit integers (chars). This will be an interesting note to comeback to in the actual hardware implementation of this CS system.

5.2 MCU Implementation

For implementation onto a MCU, we used the MSP432 Launchpad from Texas Instruments (more specifically the MSP432P401R). There were two main factors that played into this decision: the CMSIS library and EDU booster pack.

Cortex Microcontroller Software Interface Standard (CMSIS) is ARM software developed for Cortex-M processors that is a library that significantly simplifies (or makes possible what was not before) complex DSP functions. What we need it for in our system is doing matrix multiplication and its transform functions (DCT and FFT available). This is why we have been biased towards the DCT and FFT throughout the paper, all in aims to satisfying hardware constraints. Explanations on functions use, along with a few examples, are available from multiple sources online (we used www.keil.com/pack/doc/CMSIS/DSP). TI launchpads are used as a teaching tool for DSP classes, so with the Cortex-M4 CPU on the MSP432, this Launchpad was almost a no brainer to test our CS system.

Along with the MSP432, we use the Educational Booster Pack MKII (EDUMKII) from TI. It has a wide range of peripherals on board, but we need it for its microphone. A lot of source code is available for almost any of the functions for this booster pack and can all be accessed online (dev.ti.com).

Code Composer Studio (CCS) is the programming platform used with the MSP432. Along with new project setup, there are plenty of tutorials around for incorporating the CMSIS library into the system. To the best of our availability, compressive sampling will not work on this microcontroller. We were able to test it extensively and documented all results to present and prove our conclusion on this product.

Let us first start with an overview on how the program was setup. The MSP432 would include all necessary library and define global variables before entering the main function.

```
const uint16_t period = FREQ_CLK/Fs;
Timer_A_PWMConfig pwmConfig = {
      TIMER_A_CLOCKSOURCE_SMCLK, //clockSource
      TIMER_A_CLOCKSOURCE_DIVIDER_1, //clockSourceDivider
      period, //timerPeriod
      TIMER_A_CAPTURECOMPARE_REGISTER_1, //CCR1
      TIMER_A_OUTPUTMODE_SET_RESET, //compareOutputMode
      period/2 //dutyCycle };
uint16_t K, M;
arm_dct4_instance_f32 S;
arm_rfft_instance_f32 S_RFFT;
arm_cfft_radix4_instance_f32 S_CFFT;
float32_t normalize=0.125, pState[2*N], y_f32[rows]={0};
arm_dct4_init_f32(&S, &S_RFFT, &S_CFFT, N, N/2, normalize);
arm_matrix_instance_f32 y={rows, 1, 0}, Phi={rows, N, 0},
      x={N, 1, 0};
```

It would enable its functionalities such as the processing clock speed, timers for sampling, ADC for the conversion, and a universal asynchronous receiver-transmitter (UART) used for serial communication between the Launchpad the computer running the code (we used Putty as the median for displaying messages). With a pulse width modulating (PWM) timer triggering at $Fs = 8\ kHz$, the program would enter an interrupt service routine (ISR) where it would trigger an ADC conversion and put the resulting value into a buffer. There were a number of parameters that were critical to this system, and must have been defined exactly or risk the program not working regardless of “errors” shown. Most could be figured out from online forum posts and internet searches, but a lot of time was spent on making sure parameters were at least initialized with the correct size, structure, and data.

We used a triple buffer concept for collecting data, processing it, and outputting the results with the code in the ISR below:

```
MAP_ADC14_toggleConversionTrigger();
while(!ADC15_isBusy()){}
triple_buffer[input_index][sample_index] =
ADC14_getResult(ADC_MEM0);


//Update buffer indices and sample index
if(++sample_index >= N) {
     sample_index = 0;
     if(data_ready){buffer_overrun = true;}


     tmp_ui32 = input_index;
     input_index = data_index;
     data_index = output_index;
     output_index = tmp_ui32;
     data_ready = true;
}
```

Continuing the concept of splitting the incoming signal into N sized frames, we set the buffer length to a globally defined N. Again looking forward to realistic constraints on our system, the DCT on the MSP432 can only be operated on sizes $N = 128, 512, and\ 2048$, so are previous test results presented are using one of those lengths. All incoming data was stored in “input_buffer.” Once full, the pointer was changed to “data_buffer” so all processing would occur on this data as the timer ISR would add new data to the now new input buffer. This would continue for however long we wanted, always checking for a buffer overrun (explained in the next few paragraphs) and switching the buffers to keep taking in data from the ADC and compressing it for transmission to a sink node.

```
while(1) {

     MAP_PCM_gotoLPM0();

     if( buffer_overrun ) {

          UARTprintf( EUSCI_A0_BASE, “Error: buffer overrun\r\n”);

          buffer_overrun = false;

     }

     if( data_ready ) {

          x.pData = triple_buffer[data_index];

          arm_dct4_f32( &S, (float32_t *)pState, x.pData);

          K = detect_K(x.pData);

          M = pow(0.35, 2)*log(N)*K;

          for( i=0; i<=M/rows; i++ ) {

               if( i==0 ) {

                    Phi.pData = A0;

               } else if( i==rows ) {

                    Phi.pData = A1;

               } else if( i==n*rows ) {

                    Phi.pData = An;

               }

               arm_mat_mult_f32( &Phi, &x, &y );

               y_f32[0] = *y.pData

}}}
```

The above code snippet is responsible for processing the data, or doing the matrix multiplication and transmitting y. It is the infinite while loop inside the main function of our program. It remains in low power mode (LPM) except when the time interrupt is triggered for sampling the ADC. As explained, the timer ISR is only responsible for adding new data to the input buffer and switching the buffers for processing. If it is ready for processing, the variable “data_ready” is set to true and the program would enter the if-statement shown. Matrices are already made of set length for $x \in \mathbb{R}^{N \times 1}$ and $\Phi \in \mathbb{R}^{rows \times N}$. As we mentioned, we split the distribution matrix into rows in order to save portions of it into memory sectors. The data from the input buffer is put into the data structure of the matrix x, and we make it sparse by doing the DCT of x. We find the K sparsity of the frame and the corresponding M to represent the frame, just as planned. But then, since we are u sing only “rows” number of rows from Φ at a time, we enter a for-loop to process that many rows at a time. The figure shows that the if statement would continue to assign Phi.pData to an Nth element of rows used from A. Then it is a simple matrix multiplication function to find $y = Ax$ and transmit y (not shown).

If it took more time to process the data buffer than it did to fill the input buffer with data, we run into the issue of a buffer overrun (referring back to Figure 21). So it is critical that our processing speed of the MCU is capable of completing all instructions within the amount of time it takes to fill the data buffer. And this is our main problem and why this Launchpad is unsuitable for a CS system.

Even running the MSP432 at its fastest CPU setting of 48 MHz, it takes too long to complete the compression instructions. Figure 23 shows the processing time of multiple clock settings compared to how long it takes to fill the data buffer (which is simply N/F_s).

Table 5.2: Processing Time of Varying Clock Frequencies

	N	128	512	2048
Clock Frequency (s)	Sampling Period	0.0160	0.0640	0.2560
48MHz	Processing Time	0.4983	0.6398	0.7838
24MHz	Processing Time	0.5224	0.5447	2.9056
12MHz	Processing Time	0.5380	0.5795	4.8783

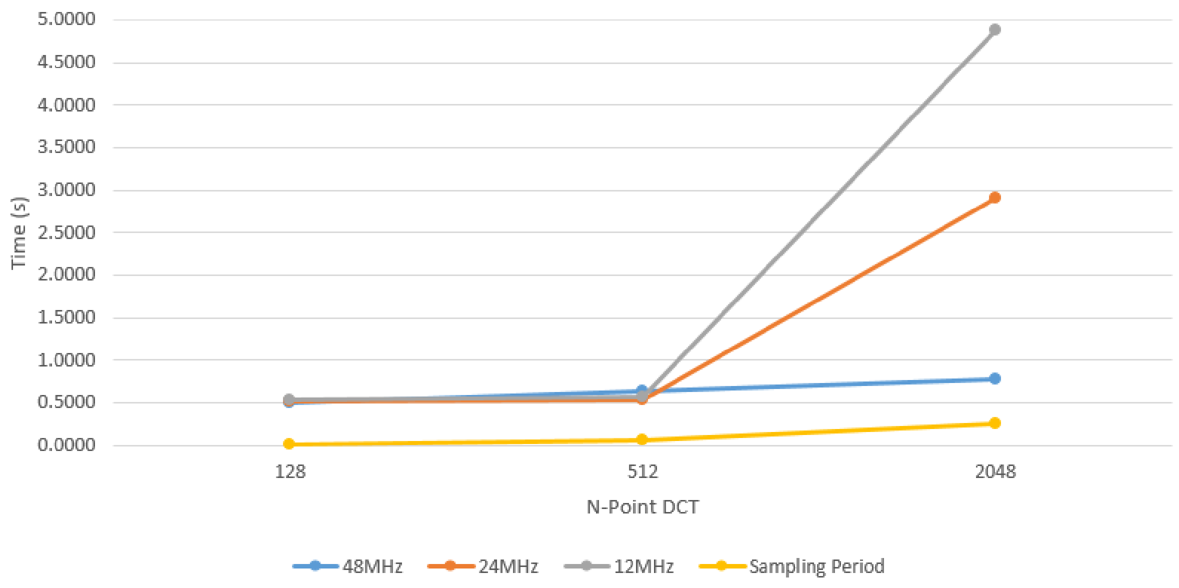


Figure 5.2: Comparing Processing Time to Sampling Period

Usually the answer to a buffer overrun error is to increase the length of the buffer. A longer buffer means more time to sample and fill the buffer. Unfortunately for us, it is a double edged sword in that a larger N also means more computation time for processing the data: longer to compute the DCT of information, comparing all indices for finding the K sparsity, and matrix multiplication of a larger array of data. So as we increase N to try and provide more time of processing, we take even more time to do the actual computations. In all cases of possible N, compared to multiple clock speeds, there is no solution for processing time being less than sampling time, making the primary reason for the MSP432's failure in CS implementation.

Another problem faced early on in writing the code is memory space available. This model of the MSP432 has 256 KB of Flash Main Memory organized into two banks split into 4 KB sectors. There is also the 16 KB of Flash Information Memory and 64 KB of static random access memory (SRAM), but they are either too small or unable to use. What happens if the signal contains a full spectrum of frequency components, another way to say it is that it has a maximum K sparsity, such that compression is not possible because $M = N$. In this specific case we would do no data processing and transmit the data as is. But there could be a situation where we need a distribution matrix as large as $(N-1) \times N$ to even compress by one data point. Or if we still want the security factor of transmitting "encrypted" data, we would still want to process the data using an $N \times N$ matrix. With the N sizes available to us, this is a *huge* amount of potential matrix indices that need to be saved on the Launchpad.

For example, using the smallest size of $N = 128$ and using a normal distribution (so float data structure needed), we are looking at 65.5 KB of memory just for an $N \times N$ distribution matrix. In the case of using a Bernoulli distribution where each element can instead be an 8 bit character, we still need 16.3 KB of memory.

But that is worst case where we can hardly compress. We can argue we would rarely (if ever) need that many rows of a distribution matrix. So what if we sacrifice the potential undershoot of rows, but save memory space? Even then, with the memory storage limitations, we are able to save a distribution matrix of size $30x128$ before having too much to fit on the board. Increasing N means decreasing M.

Using Bernoulli over Normal would save space but pose converting issues when using CMSIS functions which will take even more processing time (where we are already failing). It is faster (but still too slow and too large of memory) to use floats with a Normal distribution matrix, but less memory (and slower than we already have shown) to use 8 bit char (q15 data structure) with a Bernoulli matrix.

5.3 SBC Implementation

Implementing CS on a single board computer platform was significantly more manageable, understandably so since it is a more capable platform. We used the Raspberry Pi 3 which is a Broadcom SBC using a Linux based OS who's CPU is a 1.2GHz quad-core ARM Cortex A53. Comparing this to the 48MHz capabilities of the MSP432, we can see the upgrade in processing power alone.

We can install and use most programming languages, but most used is Python with Python 2.7 included. We use Python 3.6. A nice program to use to get a head start on installing all necessary libraries is called Anaconda. It is a free program online that will install Python 3.6 (using a platform called Spyder for the actual programming) and will install more libraries than one would need for this project. Installing Anaconda on both the Raspberry Pi 3 and the personal computer made sure there was no continuity or compatibility error when switching between the leaf to sink node.

Most important libraries to import into Python are numpy and scipy. Numpy offers most math functions that we need to use and especially matrix functions. By default, we can use n-dimension arrays (ndarray) with python and use functions that would simulate matrix algebra. Using Python matrix functions offer an advantage for syntax similar to Matlab as well as speed for some computations. Scipy is what we use for transform functions such as DCT and inverse DCT. We also import a time library so we can measure how much time is used to compress the data and a matlab plot library (pyplot) so we can plot graphs as we would in Matlab.

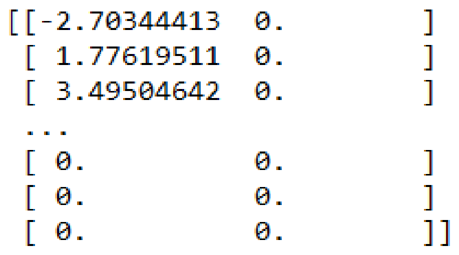
We split our code into two separate python files thinking about our goal application for a WSN – one for a transmitter (Tx) and code for a sink node receiver (Rx). With 1GB of RAM, we have no problem saving a $N \times N$ distribution matrix in memory. To go with our Matlab simulations, we create text files of a Bernoulli distribution with $N = 128, 256, 512, \text{and } 1024$. The Tx file will open and read the data into a local variable to be used for the matrix multiplication. Much like we have shown with Matlab, we are able to generate a sinusoid at multiple frequencies, read in audio from a given file, or use a microphone to record for a set time interval. This is not like the MSP432 where we are doing real time DSP. On the Launchpad, we used a triple buffer to record N sized bits of information and then transmit it. But for the Raspberry Pi, we record audio for a set time interval and then process it N elements at a time, just as we did in Matlab.

Like Matlab, our Python Tx acquires the signal x and need to down sample. In the case of an audio file, the signal frequency is most likely 44.1 kHz. We find the down ratio and trim x in order to reduce error due to rounding. Once we trim x, we down sample it so the sample frequency is now at our target frequency for sampling – 8 kHz. Then we normalize x and move onto the CS initializations. With DCT being our transformation matrix, and a Bernoulli distribution matrix, we can use multiplication like in Matlab to compress to y.

But we need to keep y and the M used for compression. The sink node needs to know how many indices of y are used to represent each frame since we are not doing real time processing. We create a matrix y with two columns. The first column will hold the actual observations or results from the compressed matrix multiplication, while the second holds the values of M.

```
y = np.matrix( np.zeros( (len(x),2,) ))

M = 0

indy = 0

while loop < (length-N):

      xtemp = np.matrix( x[loop:loop+N] )

      K = detect_K( noise, dct(xtemp) )

      M = int(round((0.35**2)*K*np.log(N)))

      if M>N: M=N

      Phitemp = np.matrix( Phi[:M,:N] )

      y[indy:indy+M,0] = np.matrix( Phitemp*Psi*xtemp )

      y[int(loop/N),1] = M

      loop = loop+N

      indy = indy+M

xtemp = np.matrix( x[loop:length] )

K = detect_K( noise, dct(xtemp) )

M = int(round((0.35**2)*K*np.log(len(xtemp))))

if M>N: M=N

Phitemp = np.matrix( Phi[:M,:(length-loop)] )

Psi = np.matrix( dct(np.identity(length-loop)) )

y[indy:indy+M,0] = np.matrix( Phitemp*Psi*xtemp )

y[int(loop/N),1] = M
```

With a simple while loop shown above, Python continues to process x while there is data available. Once it has compressed all of x, the code will write both columns of y to a text file in order to transmit it back to the sink node. Figure 5.3 shows an example print out of the *y* matrix. In this case the first 3 values of M needed to represent an N size frame were zero (the 4th and on were non zero). This is written to a text file used by the Rx file to reconstruct and create $\hat{x}$.



```
[[-2.70344413  0.        ]
 [ 1.77619511  0.        ]
 [ 3.49504642  0.        ]
 ...
 [ 0.          0.        ]
 [ 0.          0.        ]
 [ 0.          0.        ]]
```

Figure 5.3: Python Example y Matrix

We show later in the results, but Python will give slightly different results than the Matlab equivalent. Specifically in regards to the functions used for detecting noise and finding the K sparsity of the frame being processed shown below:

```
noise = detect_Noise( dct(x) )


def detect_noise(f):

    length = len(f)

    noise = np.zeros( (length,1,) )

    for i in range(int(length)):

        noise[i] = 20*np.log( np.fabs( f[i,0] ))

        if noise[i] == float(‘-inf’):

            noise[i] = 0.0


return( np.mean(noise) )
```

We see this when converting the energy signal to decibels for detecting the noise floor and taking the natural log of zero. Python will return negative infinity for any elements of a transformed x being zero (which happens often). So finding the average noise will be negative infinity if there is even one zero element. To counter it, we pre-screen the signal. For any elements that would return negative infinity, we set it to zero. The problem here is we are changing the average value of the energy signal by setting all those large negative values to zero.

So before where we showed that the best threshold value was no threshold, we have now biased the noise level towards zero that we need a threshold to offset it. Although we tested different values for this threshold, it varies with the type of signal we would process (such as classical versus pop music). We would need to do more testing with a blanket scan of threshold values similar to what we did with Matlab. The only concern is if we are still trying to be sub-Nyquist sampling. Shooting for a threshold that minimizes the error is going to require a higher M which risks being above the Nyquist sampling rate.

Exchanging data between the leaf node (Tx – Raspberry Pi 3) and the sink node (Rx – PC machine) was using a third party remote access program called TeamViewer. TeamViewer allows the user to sink multiple machines to an account. Through this account, one could log on to their computer, and pull up a virtual machine of another platform so long as it is logged into the account. So the Raspberry Pi is setup so that it will power up and immediately connect to a Wi-Fi network and log into TeamViewer, giving access for me to log on and see what is happening. TeamViewer has a file transfer feature so once the Tx code is run, we are able to simply transfer the needed files to the Rx machine (Phi.txt and y.txt).

On the side for the sink node, the Python code again followed its Matlab equivalent nicely. A great part about it is only global variables needed for this code are those needed for SL0 reconstruction and the text file with Φ and y. From "Phi.txt" we are able to read in all its data and tell N from the square root of how much data was just read. We already know Ψ is the DCT. Now the Rx file is just a while-loop for processing data for all available M in the 2nd column of "y.txt" such as:

```
F = open( “y.txt”, “r” )

data = f.readlines()

f.close

data = [y.strip() for y in data]

data = [y.split() for y in data]

length = len(data)

y = np.zeros((length,1,))

M = np.zeros((length,1,))

for i in range( length ):

     y[i] = data[i][0]

     M[i] = data[i][1]

y = np.matrix( y )

M = np.matrix( M )
```

We need to account for the possibility that some frames of x on the Tx side were of no sound and needed no representation (M=0). So when reading in the next value of M, we need to distinguish between a zero meaning frame size, and a zero showing the end of data. It may be a bit of overkill, but a summation of a range of M is a simple solution for detecting if there are any more M sized data sets to be processed. Below is the Rx while-loop for reconstructing the signal:

```
indx = 0

indy = 0

j = 0

do = 1

while do:

      if( indx+N ) > length:

            N = length-indx

            invPsi = np.matrix( idct(np.identity(N)) )

      ytemp = y[indy:indy+int(M[j])]

      indy = indy+int(M[j])

      Phitemp = Phi[:int(M[j]),:N]

      x_hat[indx:indx+N] = SL0( Phitemp, ytemp, sigma_min,
sigma_decrease_factor, mu_0, L )

      x_hat [indx:indx+N] = invPsi* x_hat [indx:indx+N]

      indx = indx+N

      j=j+1

      if( np.sum(M[j+1:j+3000]) == 0.0 ): do=0

x_hat = np.divide(x_hat, max(np.fabs(x_hat)) )

RMS = np.sqrt( np.mean( np.square( x- x_hat)))
```

For cases like pop music (the “Clarity” and “Whatever It Takes” examples), there would be small gaps of M=0, if any. Much like we saw for the example of y, M was zero at the beginning and end. But for examples like classical music (our “Nuvole Bianche”), there are many time where M=0 and for varying lengths.

Since we are using numpy matrices instead of ndarrays, we are better able to recreate the SL0 function in Python from what we use in Matlab by:

```
def SL0( A, x, sigma_min, sigma_decrease_factor, mu_0, L ):

    A_pinv = np.linalg.pinv( A )

    s = np.matrix( A_pinv*x )

    sigma = 2*max( np.fabs(s) )

    while sigma > sigma_min:

        for i in range( L ):

            delta = np.matrix( OurDelta(s, sigma) )

            s = s-mu_0*delta

            s = s-A_pinv*(A*s-x)

        sigma = sigma*sigma_decrease_factor

    return s


def OurDelta( s, sigma ):

    delta = np.multiply( s, np.exp(-np.square( np.fabs(s) ) /
sigma**2))

    return delta
```

A main difference that one would see in implementation between matrices and ndarrays is in the “OurDelta” function that is called within SL0. With an ndarray, we would called “np.dot” which is the dot product and usually regarded as an equivalent to matrix multiplication of two 1D arrays. In this case, “s” and “sigma” are both $N \times 1$ vector arrays which would results in returning a $N \times N$ answer. But what the function is designed to do is do element wise multiplication so the ith element in s is multiplied by a variation of the ith element in sigma. The actual output of the math is different as shown in the code snapshot, but that is the underlying goal. Using numpy dot product of ndarrays will not suffice. Whereas using the numpy multiply function that is used for matrix structures does the trick. This is one small example where the results depend heavily on the fine print of matrices versus arrays, but using matrices brings our Python code closest to reflecting the Matlab source code we use.

After all the data in y.txt is processed and $\hat{x}$ is reconstructed, we do our familiar RMS test to the original x to see how the system performed and how it compared to the Matlab simulations. We already know there will be differences from rounding and dealing with the natural log of zero, but just how different are the results? One of the largest question is how much time played a factor in it. We sprinkled the code with timing functions (both Tx and Rx) to see how it stacks up. On the receiver side in the sink node, we should see similar values for time to reconstruct since we used the same PC machine for Matlab and Python reconstructions. But we need scrutinize the Python code more when looking at the compression time. That code is run on the Raspberry Pi 3 and is a better look at how our system would perform in field as a WSN.

The biggest questions is if the Raspberry Pi is able to handle multithreading in order to simulate real time DSP. We saw that a triple buffer implemented in the MSP432 would fail because it is too slow at doing computations on the data buffer – is the Raspberry Pi any better? Implementing a triple buffer (using three threads other than main), the short answer is no. But going through the process we find a way to make it work.

We can start with our three buffers as functions in Python. Instead of an input buffer, data buffer, and output buffer, we have an input data function recording real time audio given by:

```
def inputData(in_data, frame_count, time_info, flag):

      global xo, input_buffer, data_ready

      if data_ready: print("Error: Buffer Overrun")

      input_buffer = np.matrix(np.frombuffer(in_data,

dtype=np.float32))

      xo = np.append(xo,input_buffer)

      with threading.Lock():

            data_ready=1

      return (input_buffer, pyaudio.paContinue)
```

Which uses the pyaudio library using callback recording to record a certain number of samples into the input buffer, append the variable xo that will hold the entire original signal, then triggering the processing function by setting data ready to 1. Inside this function, we first check if the sampling rate of the microphone (F_o) is the same as the reduced sampling frequency of 8 kHz. If not, we need to down sample. After which we see the same process we have been doing with detecting the K-sparsity of that frame, the minimum M needed to represent it, and the actual matrix multiplication. We use data done as a variable similar to data ready to tell the output data function that processing is done. We will see it change later, but below is the first take of a function to process the data in a simple triple buffer:

```
def processData(data):
      global data_done, data_ready
      if data_ready:
            with threading.Lock():
                  data_ready=0
            if Fo != reducedFs: downsample()
            f=Psi*np.transpose(input_buffer)
            K = detect_K(f)
            M = int(round(0.35**2*K*np.log(N)))
            if M>N: M=N
            Phitemp=np.matrix(Phi[:M,:N])
            y = np.asarray(np.matrix(Phitemp*f))
            with threading.Lock():
                  data_done = 1
      return
```

After the data done flag is toggled, the output data function below is run on each frame:

```
def outputData():

      global output_buffer, Ms, data_done

      output_buffer=np.append(output_buffer, y)

      Ms=np.append(Ms, M)

      with threading.Lock():

            data_done=0

      return
```

We append an output buffer and temporary M variable to save for transmission. After we are done recording, the program will package the output buffer into a data frame with Ms and save it to a text file to send through TeamViewer to the sink node. This function will also change slightly.

The above three functions are the foundation for a triple buffer in Python. It was implemented using multithreading so that we can process or output data while the microphone was taking in more samples. But as we mentioned, there will be a buffer overrun where it takes more time to run process data than it take to fill the input buffer. This will happen for any size frames where we tested $N = 128, 256, 512, 1024$. What we can do is break up the sample into even smaller frames. For example, processing $N = 1024$ is slower than taking 1024 samples, so what about processing $N = 512$ twice while the microphone takes 1024 samples? Or processing $N = 128$ four times for the same amount of sampling. Or any combinations for sampling and splitting the processing into a factor of that size. What we are talking about is multithreading multiple processing functions. We still have an input and output buffer, but now our data buffer is split into multiple, smaller buffers to avoid a buffer overrun. It is worth noting that we need to take into account that even smaller frames will result in more error as we have talked about. But we show this concept working at all sizes available. We will also show all functions of Python for the complete picture. Let us start with key initializations given by:

```
pa = pyaudio.PyAudio()

FORMAT = pyaudio.paFloat32

WIDTH = 2

CHANNELS = 1

Fo = 44100

TIME_TO_CALIBRATE=3

reducedFs=8e3

downratio=math.floor(Fo/reducedFs)

N = 1024

SAMPLES = N*2*downratio

NUM_DB = int(SAMPLES/N)

temp_m=np.log(N)*(0.35**2)
```

For the entire program, libraries imported are time, math, numpy as np, pandas as pd, pyAudio, scipy fftpack, threading, and msvcrt (windows only, not on the Pi). In the above snippet, the first 5 lines will be used for initialization of a pyAudio stream to record from the microphone. In this case, we are going to sample at 44.1 kHz and need to down sample to the reduced Fs variable which is set at 8 kHz. To make sure we down sample to a predetermined size (N), we go ahead and calculate the down sampling ratio and initialize N which will be the size of our data processing buffers. The number of samples the microphone will take in at a time (or the length of the input buffer) is set to SAMPLES.

We found that to avoid buffer overrun, of any size N given the down ratio is 1, is to take in twice as many samples as size N. NUM_DB is simply the number of data buffers so no matter if the user changes the size of N, the rest of the system will adjust accordingly. And finally the variable temp_m is to hold part of the calculations of finding M as to further reduce the time spent in the processing data functions.

Since we are looking at making multiple data buffers, let us look at the class we define as:

```
class data_buffer:

     def __init__(self, buff, M, start, end):

          self.buff = buff

          self.M = M

          self.start = start

          self.end = end
```

Where each data buffer will have a start and end index of the input buffer, and hold the y observations in buff and M integer in M. We create all the needed buffers in the first few lines of the main function along with creating the necessary threads for our program:

```
t=[]

db=[]

for i in range(NUM_DB):

      db=np.append(db, data_buffer([], M, i*N, (i+1)*N))

      t=np.append(t, threading.Thread(target=processData,

      args=(db[i],), daemon=True))

tout = threading.Thread(target=outputData, daemon=True)

tstop = threading.Thread(target=stopData, daemon=True)
```

where t signifies threads, and db are the data buffers. The input thread is created when initializing the pyAudio stream variable, tout is our output thread, and tstop is made so we can record until the user tells the system to stop via a keypress.

The main thread will initialize variables needed for compressive sampling, the above thread, then into calibrating the system. Calibration would be in a quiet setting to find the noise level. It will record for a certain number of seconds, then run a detect noise function like we have explained before. We will skip showing the calibration code because it is very similar to making the input data thread shown below:

```
stream = pa.open(format=FORMAT, channels=CHANNELS, rate=Fo,
input=True, frames_per_buffer = SAMPLES, stream_callback=inputData)
stream.start_stream()
for i in range(NUM_DB):
     t[i].start()
tout.start()
tstop.start()
```

We wait for user input to start recording, then create and start the input data thread in the form of a stream. We start all other threads but they will not execute because of the data ready flag. So we make sure they continue to try to execute for as long as the user does not tell the system to stop by:

```
while not stop:
     for i in range(NUM_DB):
          if not t[i].isAlive():
               t[i] = threading.Thread(target=processData,
               args=(db[i],), daemon=True)
               t[i].start()
          if not tout.isAlive():
               tout = threading.Thread(target=outputData,
               daemon=True)
               tout.start()
```

In the above code, we check if the thread is alive (the purpose of the daemon variable set to True). If it is not alive, the thread has run and has since been terminated, so we re-initialize that thread and start it again.

Once the user has told the system to stop, we have the needed code after the above while loop to close the stream, terminate it, and join all threads as a means to clean up the variables. This is the point to package the data and transmit it since we are done receiving data from the environment. We do so by:

```
start=len(Ms)

end=len(output_buffer)

if end>start:

      for i in range(end-start):

            Ms=np.append(Ms,0)

else:

      for i in range(start-end):

            output_buffer=np.append(output_buffer,0)

y=pd.DataFrame(data={'Y': output_buffer, 'M': Ms})

x=pd.DataFrame(data={'X': xo})

np.savetxt(r'y.txt', y.values, fmt='%d')

np.savetxt(r'x.txt', x.values, fmt='%1.4e')
```

We first check to see if we have more indices of M or y, we need to zero pad the other so that it is the same length as the longer array. Once they are the same length, we can create a data frame that holds both y and M to be saved to a text file. We also save the original signal, xo, to be compared to the reconstructed for error comparison on the receiver side. Also as a note, the stop thread is simply waiting for user input by:

```
def stopData():

	global stop

	input("Press ENTER in console to STOP")

	msvcrt.getch()

	stop=1
```

Now that we explained the entire process, let us back track just a bit to show what has changed. As mentioned, the code for a simple triple buffer was not enough, so we had to adjust for multiple threads in the data processing and output threads. With multiple data buffers, processing is done by:

```
def processData(data):
      global data_done
      if( (data_ready) & (data_done<NUM_DB) ):
            if Fo != reducedFs: downsample(data)
            f=Psi*np.transpose(input_buffer[:,data.start:data.end])
            K = detect_K(f)
            M = int(round(temp_m*K))
            if M>N: M = N
            data.M = M
            Phitemp = np.matrix(Phi[:M,:N])
            data.buff = np.asarray(np.matrix(Phitemp*f))
            with threading.Lock():
                  data_done = data_done + 1
      return
```

with down sampling done by:

```
def downsample(data):
      data.buff=np.transpose(input_buffer)
      trimmer = len(data.buff) % downratio
      length = len(data.buff)-trimmer
      x = []
      for i in range(length):
            x = np.append(x, data.buff[i])
      data.buff=[]
      for i in range(int(length/downratio)):
            data.buff = np.append(data.buff, x[i*downratio])
      data.buff= np.divide(data.buff, max(np.fabs(data.buff)))
      return
```

After splitting up the incoming data, the output data thread has the job of putting y in order it is received. Since data buffers are assigned a start and end index, the output is easily done by:

```
def outputData():

      global output_buffer, Ms, data_done, data_ready

      if data_done>=NUM_DB:

            with threading.Lock():

                  data_ready=0

      for i in range(NUM_DB):

            output_buffer = np.append(output_buffer, db[i].buff)

            Ms = np.append(Ms, db[i].M)

      with threading.Lock():

            data_done=0

      return
```

With multithreading using the above code, we will always avoid a buffer overrun error. The user would need to be aware that the larger N used will results in more samples – meaning more time for each recording frame. This will come into play on reconstruction where if one frame has a lot of distortion, that could be an entire second of recording wasted instead of an eighth of a second which would hardly be noticeable. We tried to minimize error while staying less than half a second of recording samples. Fortunately, both factors are made to depend on N.

6 RESULTS

For this chapter, we plan to compare the results of a simulated CS system using Matlab on a personal computer to how an actual CS system would perform as a WSN using a Raspberry Pi 3 as the leaf node (transmitter – Tx) and a personal computer as the sink node (receiver – Rx). For the sake of terminology, any time during this chapter that we say “Matlab,” we are referring to using Matlab on a PC for both the sensing and reconstruction process for a CS example. Any time we say “Python,” we are referring to the dual work of the Raspberry Pi sensing and the PC doing the reconstruction.

We need to clarify that for the sake of compression and reconstruction time. The reconstruction times of simulation versus implemented are similar because they are done on the same machine, but the compression time is drastically different (because of realistic constraints on the Raspberry Pi) yet shown in the same results table as all work done on the PC.

For an overview for flow of data, the Raspberry Pi is retrieving the data used for Φ from the TeamViewer shared files. It will go through the entire source signal x and compress it into a 2D matrix test file containing y observations and values of M. The sink will retrieve this 2D matrix along with Φ used from TeamViewer and start its reconstruction process. For the sake of showing error, the sink node will also grab the x used in order to compare to $\hat{x}$.

We ran both the Python and Matlab CS systems on our three audio files containing different types of music of varying length. Figure 6.2 shows the time (given in seconds) it took to compress as well as reconstruct the test signal, while Figure 6.1 shows the RMS error between x and $\hat{x}$ along with the compression ratio (as a fraction, not percentage).

We ran these tests at $N = 128, 256, 512, and\ 1024$, then took the average of all results to plot in the below figures. These two figures contain the majority of information to draw conclusion on how well our Python system measures up to the simulations of Matlab.

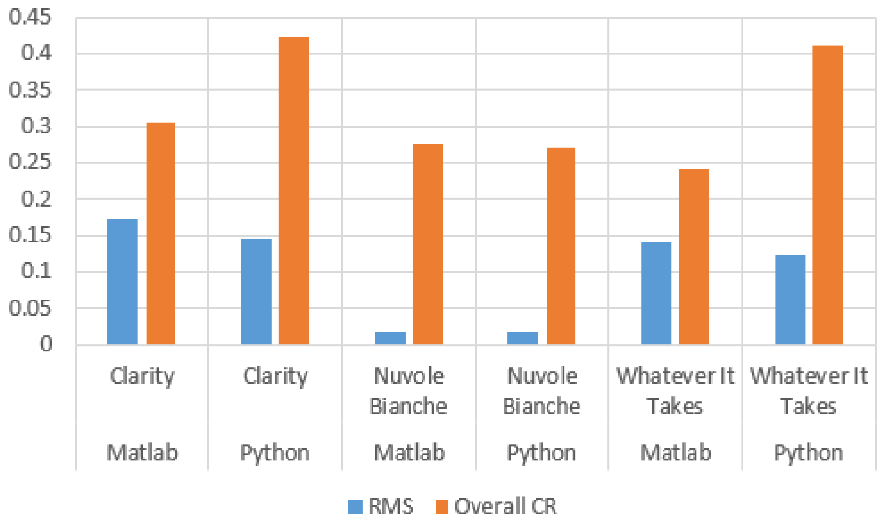


Figure 6.1: CS System Comparisons of RMS and Compression Ratio

Looking at the first figure of results, Python looks to measure up to our expectations and perform very similarly to its Matlab counterpart. For the test signals “Clarity” and “Whatever It Takes,” we see a much larger compression ratio (meaning more data kept and less actual compression) between Python and Matlab as opposed to “Nuvole Bianche” even though they were different lengths. We can assume this is from the spectrum being fuller and resulting in a higher K sparsity. Higher K resulting in more M needed to represent the signal. What is curious though is that Matlab was able to keep a consistent compression ratio for the three examples while Python needed more M to represent the two examples mentioned.

The most likely answer to the differences goes back to detecting the noise floor of the signal. In Python, taking the natural log of zero returns negative infinity, so we override it and say that $ln(0) = 0$. In the case of pop music like “Clarity” and “Whatever It Takes,” there are less zero elements, but still very small values. Taking the natural log of all those small values are going to bring the average value into the negatives, as opposed to the average being closer to zero when there a large amount of zero elements. With a smaller noise threshold, it is very easy for the energy of an element to be higher than that threshold, therefore driving up the value of K. Which brings us full circle to a higher K results in a larger M and less actual compression taking place.

With regards to error and compression ratio, the Python system both performs as expected and on par with our simulated results. Unfortunately the timing of our system is a matter of concern.

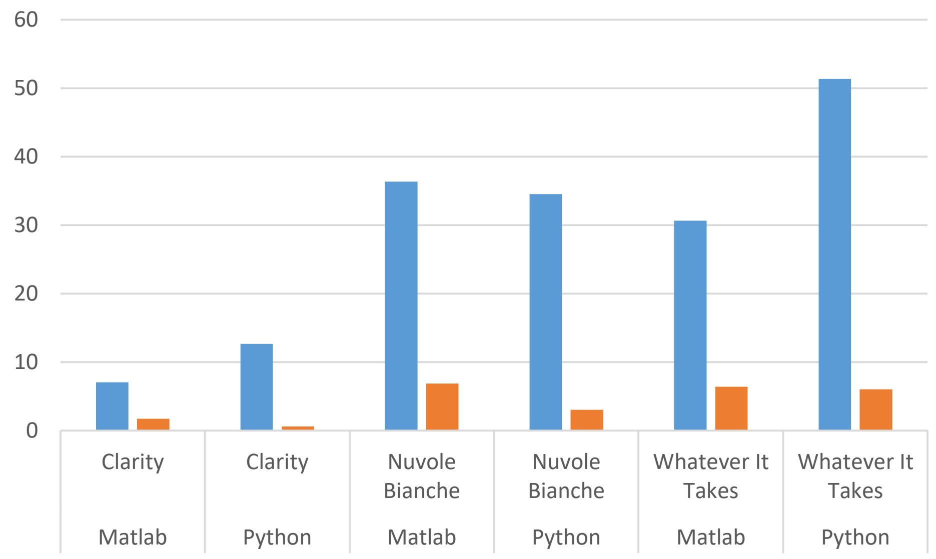


Figure 6.2: CS System Comparisons of Compression and Reconstruction Time

The time it took for Python to reconstruct (blue) was almost twice as long as Matlab (orange) in some cases. The main point to draw attention to is the large amount of time it took to compress the original signal in Python. Of course, the time grew almost exponentially as we tested larger N, but even the time at $N = 128$ took almost 40 times as long compared to compressing time in Matlab. A large margin of error is due to the system, as we expected since the Raspberry Pi system specifications are significantly lower than a desktop machine, but we could not visualize just how much slower until we look at the above figure.

With the implementation described in the previous chapter, the Raspberry Pi is able to implement a triple buffer with multiple data processing buffers. The results shown thus far of compression are those of the multithreaded Python code. The main question at this point is how it compares to what is already easily available in industry – namely zip. Zip being lossless and CS lossy, the error is in favor of zip regardless. We want to see how time compares though. We create two figures that help illustrate our conclusions. The first is cycling through all size of $N = 128, 256, 512, 1024$ and running CS and zip. We average the total times of each N size to produce the figure below:

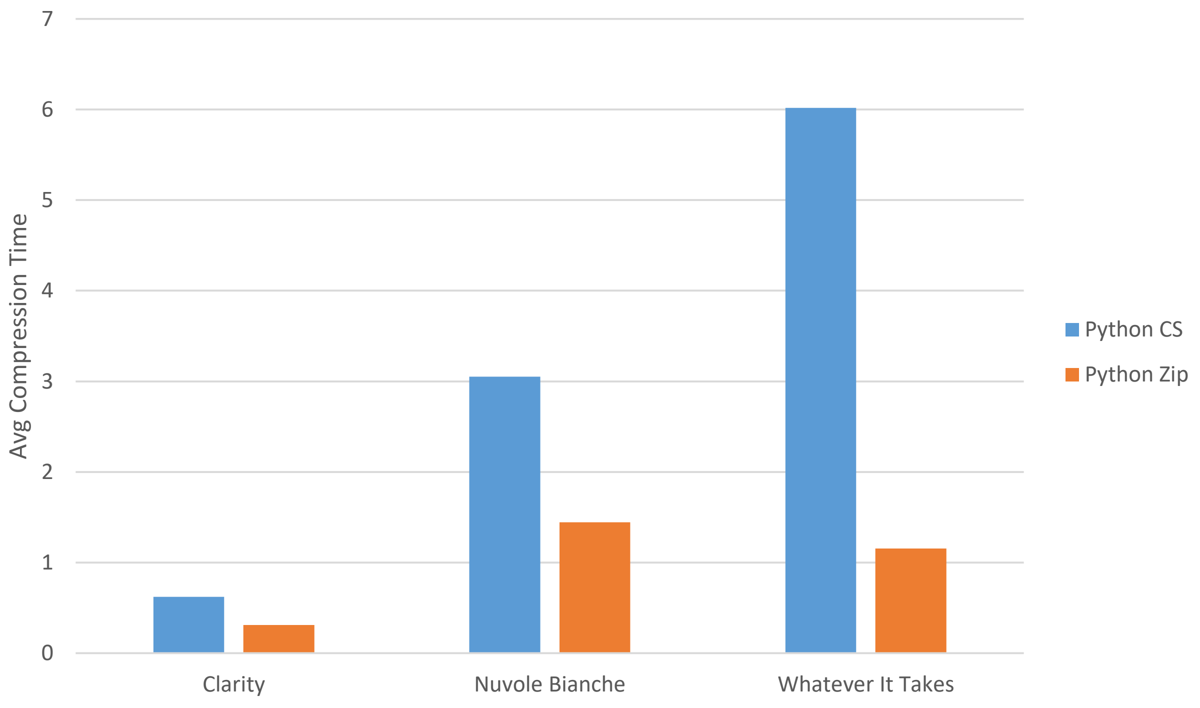


Figure 6.3: Compression Time between Matlab and Python CS and Zip

From here, we can see that our triple buffer in the Raspberry Pi is faster than our Matlab sequential implementation on a desktop machine, but still much slower than just zipping the data.

The next figure we look out looks at the best case. For CS, we use N=1024 since it was much faster than the others. And based on previous chapters' findings, a larger N will also yield lower error. Not only do we look at the music files we have been using, but also recording from a microphone. With the various types of audio, and the optimal values for our CS system, we compare the time it takes to compress along with the compression ratio given by:

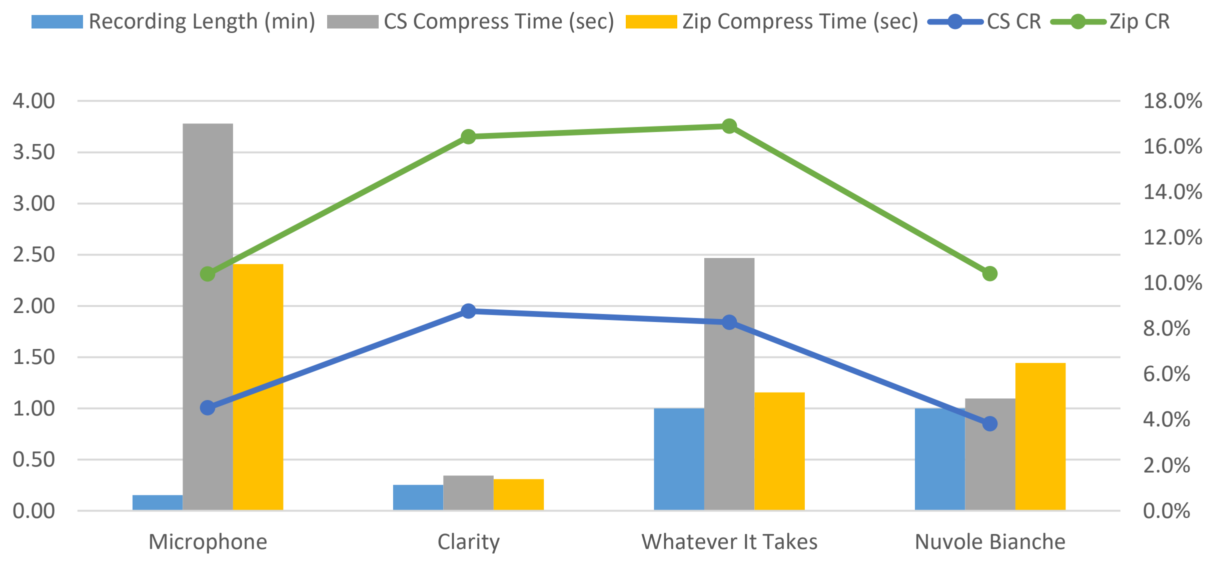


Figure 6.4: CS and Zip Comparison of Speed and Compression Ratio

This gives a little ore hope in arguing for compressive sampling. We saw that zipping the data would be faster on average, but the above shows that CS is still comparable to each data type, and better in terms of compression ratio. With this zoomed in view, we see CS is close to the performance of zip on "Clarity" and actually faster on

“Nuvole Bianche.” Although CS is almost twice as slow as zip with a microphone recording or “Whatever It Takes,” it is still pretty fast and avoids buffer overrun. The results showing that CS can take up almost half as much memory for all the data sources presented is remarkable and much more advantageous if size is significant factor in a WSN.

Even then there is still a question on the quality of sound once reconstructed. We have been using RMS error to quantize our results, but how does it relate to how well it sounds to a user? The subjective different between $RMS = 0.1$ and 0.01. Is it linear in the sense that the difference between $RMS = 0.1$ and 0.01 is the same as RMS=0.01 and 0.001? What are people’s opinions on what RMS level is “good enough” to know what the signal is, and being able to recognize the sounds or words from it?

Our answer is an $RMS = 0.02$ is *usually* a good baseline for “good enough” reconstruction. What we did was start with three audio samples: Whatever It Takes (pop), Nuvole Bianche (classical), and voice plus noise recording (everyday sounds). All of which were sampled or compressed to a certain ratio. We sampled the same data at Nyquist Rate as a standard, so if our CS system sounded better to the user than Nyquist Rate, we would consider it a success. Along with Nyquist Rate, we set the compression ratio of our system to 10%, 25%, and 50% of the original data length. We got a few people to listen to each example at the 4 different qualities, and give their opinion on how they sounded, or how well the reconstruction was done. Of course there would be some distortion in the reconstruction sounding like static, but could the user make out all the sounds or words in the signal. If the original signal had poor quality versus crisp sounds, we wanted to know if the reconstructed $\hat{x}$ had poor quality versus crisp sounds. As we compressed the data even more, the RMS would go up, so we could assume a correlation between the user’s opinion of the quality and the RMS of the reconstructed signal.

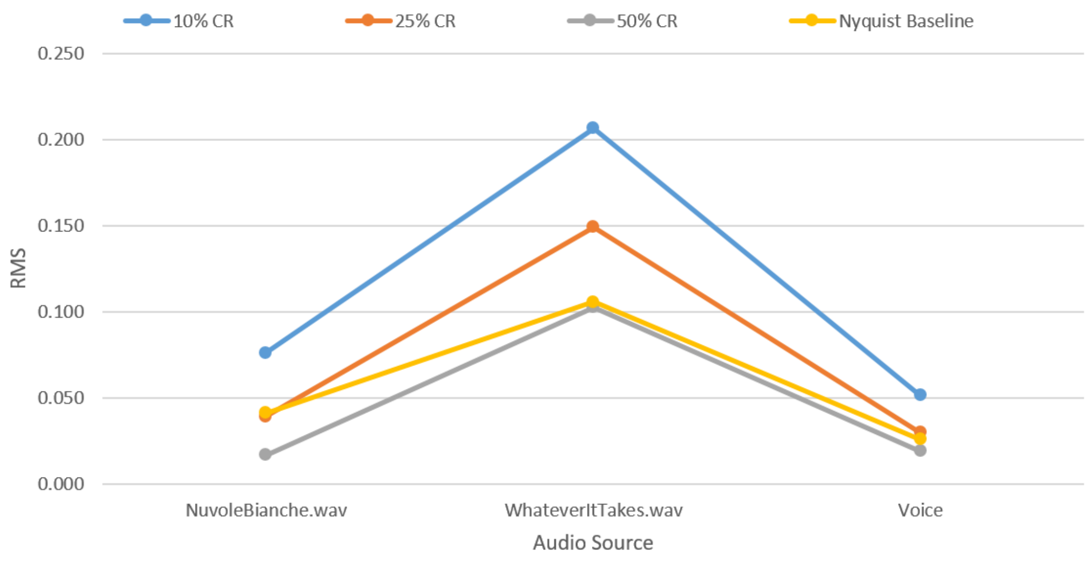


Figure 6.5: Relating Subjective Opinion on Audio Quality to Objective RMS Measurement

What we found was that people would prefer the samples that had an RMS of roughly 0.02. That low of an RMS was most likely achievable when setting the compression ratio to 50%. People would prefer samples at that compression ratio over the baseline Nyquist Rate sampling. The quality was subjectively better with a lower RMS, but it also meant that the M used was usually above the equivalent observations of a Nyquist system. Users can still make out the data of lower compression ratio, just at the cost of quality.

The main take away is that the RMS can be related or used as an expectation to how well the quality will be of the reconstructed signal. We also go back to the idea that a CS system *may* get below what would be sub-Nyquist, but sometimes it does not, and that is okay. We are still compressing a signal, removing the redundancies we can, all in order to make a more efficient WSN. There is still work to be done to develop an operations network, but we have shown that it is simulated to be possible, and the Raspberry Pi is currently the best option to go forward.

Another problem we did not fully explore is running our CS system at different sampling frequencies with the microphone. There are no issues with 8 kHz or less, but as we increase the sampling frequency, we are more likely to run into buffer overrun – there is a point where the sampling period is just so small that the Raspberry Pi cannot complete all the data processing buffers in time. To the point that at 44.1 kHz sampling frequency and $N = 1024$, we see a few flags tripped for a buffer overrun. The set limitation of the Pi and its calculation time was not thoroughly mapped since the number of data buffers currently depends on the number of samples taken by the microphone and its sampling frequency.

When arguing CS over zipping, our system would show that the user would most likely zip data to save some time and have a lower error, while CS to compress the data more and save space and time transmitting the data. A large factor that this thesis did not explore is random sampling in order to actually sample at sub-Nyquist [17]. Implementing a sampler along with our CS system would lessen the strain of the ADC, along with all the benefits we just showed. Since we would sample at sub-Nyquist randomly, zipping the data would not work since decompressing would produce seemingly random, under sampled data points.

7 CONCLUSION

There is a lot of material to recap on what we just talked about. We started with the Nyquist-Shannon theorem of sampling at twice the maximum frequency component in a signal. Compressive sampling is a technique where one could compress to a number of samples that would be less than Nyquist Rate, but as of now we still need the signal sample at the rate in order to compress it. The performance of such a system depends on the measurement matrix consisted of a distribution matrix (Φ) and a transform matrix (Ψ) that make the original signal sparse, then randomize it to create a unique solution to an otherwise underdetermined system of equations. With a measurement metric called the coherence coefficient (μ) and knowledge of the sparsity of the signal (K), we can represent x (split into N sized frames) with M amount of observations that result from matrix multiplication. Then reconstruction requires l_p norm minimization with a number of techniques available all with their own pros and cons. We simulated and test a system with many Φ and Ψ in Matlab, and implemented the design on a MCU (MSP432) and SBC (Raspberry Pi 3) using a Bernoulli distribution and DCT transform matrices. Although the MSP432 failed, we tested it extensively to prove so. The Raspberry Pi 3 may run into a problem with processing time depending on the design constraints, but show much better potential for implementing a CS system for a WSN application. Even though compression using zip is faster with less error, CS already has an advantage for compression ratio and therefore transmitting speed and power. More testing with sub-Nyquist sampling could tip the scales even further in favor of CS.

APPENDICES

Python Tx – Sensor Node

```
import AudioFile as af

import MicrophoneRecord as mr

import SL0_Tx_5multiproc as mr


def main():

    N=128

    source="file"

    filename =
"WhateverItTakes_1min.wav"

    timeInterval = 5

    Fs = 8e3

    if source=="file":

        af.main(filename, N)

    elif source=="record":

        mr.main(timeInterval, N, Fs)

    else:

        print("No selection")


#Window main parent and title

if __name__ == "__main__":

    main()
```

AudioFile()

```
import time

import math

import numpy as np

from scipy.io import wavfile

from scipy.fftpack import dct

import pandas as pd

import threading


reducedFs=8e3

M=1

N=1024

Phi=[]

Psi=[]

xo = [] #Original signal

n = [] #Calibration signal for noise

input_buffer = []

output_buffer = []

Ms=[]

data_ready=0
```

```
data_done=0
stop=0
timeC=[]

class data_buffer:
   def __init__(self, buff, M, start, end):
      self.buff = buff
      self.M = M
      self.start = start
      self.end = end

def processData(data):
   global data_done
   if data_done<NUM_DB:
      global start
      start=time.time()
      f=Psi*x[data.start:data.end, :]
      K = detect_K(f)
      M = int(round(m*K))
      if M>N: M=N
      data.M=M
      Phitemp=np.matrix(Phi[:data.M,:N])
      data.buff =
np.asarray(np.matrix(Phitemp*f))
      with threading.Lock():
         data_done = data_done + 1
   return

def outputData():
   global output_buffer, Ms, data_done,
data_ready, timeC, stop
   if data_done>=NUM_DB:
      with threading.Lock():
         data_ready=0
      for i in range(NUM_DB):
         output_buffer =
np.append(output_buffer, db[i].buff)
         Ms=np.append(Ms, db[i].M)
      timeC=np.append(timeC,
time.time()-start)
      with threading.Lock():
         data_done=0
         stop = 1
   return
```

```
def detect_Noise(f):
   length=len(f)
   noise=np.zeros((length,1,))
   for i in range(int(length)):
      noise[i]=20*np.log(np.fabs(f[i,0]))
      if noise[i]==float('-inf'):
         noise[i]=0.0
   return (np.mean(noise))

def detect_K(f):
   K=0
   length=len(f)
   for i in range(int(length)):
      if( 20*np.log(np.fabs(f[i,0])) > noise
):
         K=K+1
   return K

def main(filename, n):
   global Phi, Psi, stream, pa, noise, Ms,
output_buffer, t, db, x, NUM_DB, m, N
   N=n
   m=np.log(N)*(0.35**2)
   if N==128:
      phi="Phi128.txt"
   elif N==256:
      phi="Phi256.txt"
   elif N==512:
      phi="Phi512.txt"
   elif N==1024:
      phi="Phi1024.txt"
   [Fo, xo]=wavfile.read(filename)
   xo=np.transpose(np.matrix(xo[:,0]))
   print("File\t",filename)
   #Trim the signal so we can down
sample
   downratio=math.floor(Fo/reducedFs)
   trimmer= len(xo) % downratio
   length=len(xo)-trimmer
   x=np.zeros((length,1,))
   for i in range(len(x)):
      x[i] = xo[i]
   #Actual down sample
   Fs=Fo/downratio
   x1=np.zeros((int(len(x)/downratio),1))
   for i in range(len(x1)):
        x1[i] = x[i*downratio]
    x=x1
    x= np.divide( x, max(np.fabs(x)) )
    NUM_DB = int(len(x)/N)
    t=[]
    db=[]
    for i in range(NUM_DB):
        db=np.append(db, data_buffer([],
M, i*N, (i+1)*N))
        t=np.append(t,
threading.Thread(target=processData,
args=(db[i],), daemon=True))
    tout =
threading.Thread(target=outputData,
daemon=True)
    #CS variable Phi and Psi
    I = np.matrix(np.identity(N))
    Psi=np.matrix(dct(I))
    Phi=np.matrix(np.reshape(
np.loadtxt(phi),(N,N)))
    noise =
detect_Noise(dct(np.matrix(x)))
    print("Noise: ",noise,"dB")
    for i in range(NUM_DB):
        t[i].start()
    tout.start()
    while not stop:
        for i in range(NUM_DB):
            if not t[i].isAlive():
                t[i] =
threading.Thread(target=processData,
args=(db[i],), daemon=True)
                t[i].start()
        if not tout.isAlive():
            tout =
threading.Thread(target=outputData,
daemon=True)
            tout.start()
    for i in range(NUM_DB):
        t[i].join()
    tout.join()
    print("CS time:", np.sum(timeC))
    start=len(Ms)
    end=len(output_buffer)
    if end>start:
        for i in range(end-start):
            Ms=np.append(Ms,0)
    else:
        for i in range(start-end):
            output_buffer =
np.append(output_buffer,0)
    y=pd.DataFrame(data={'Y':
output_buffer, 'M': Ms})
    np.savetxt(r'y.txt', y.values, fmt='%d')
    np.savetxt(r'x.txt', xo, fmt='%1.4e')
    start=time.time()
    import gzip
    with gzip.open('x.gz', 'wb') as f:
        f.write(xo)
    end = time.time()-start
    print("Zip time:", end)
```

MicrophoneRecord()

```
import time
import math
import numpy as np
from scipy.fftpack import dct
import pyaudio
#import msvcrt #cannot use on Pi
import pandas as pd


import threading


pa = pyaudio.PyAudio()
FORMAT = pyaudio.paFloat32
WIDTH = 2
CHANNELS = 1
DEVICE=3
Fo = 8000
TIME_TO_CALIBRATE=3
reducedFs=8e3
Phi=[]
Psi=[]
xo = [] #Original signal
n = [] #Calibration signal for noise
input_buffer = []
output_buffer = []
Ms=[]
data_ready=0
data_done=0
stop=0
timeC=[]
```

```
class data_buffer:
   def __init__(self, buff, M, start, end):
      self.buff = buff
      self.M = M
      self.start = start
      self.end = end

def inputData(in_data, frame_count,
time_info, flag):
   global xo, input_buffer, data_ready
   if data_ready: print("Error: Buffer
Overrun")
   input_buffer =
np.matrix(np.frombuffer(in_data,
dtype=np.float32))
   xo = np.append(xo,input_buffer)
#saves filtered data in an array
   with threading.Lock():
      data_ready=1
   return (input_buffer,
pyaudio.paContinue)

def processData(data):
   global data_done
   if ((data_ready) &
(data_done<NUM_DB)):
      global start
      start=time.time()
      if Fo != reducedFs:
downsample(data)
      f =
Psi*np.transpose(input_buffer[:,data.star
t:data.end])
      K = detect_K(f)
      M = int(round(m*K))
      if M>N: M=N
      data.M=M
      Phitemp=np.matrix(Phi[:data.M,:N])
data.buff=np.asarray(np.matrix(Phitemp
*f))
      with threading.Lock():
         data_done = data_done + 1
   return

def outputData():
   global output_buffer, Ms, data_done,
data_ready, timeC
   if data_done>=NUM_DB:
      with threading.Lock():
         data_ready=0
      for i in range(NUM_DB):
         output_buffer =
np.append(output_buffer, db[i].buff)
         Ms=np.append(Ms, db[i].M)
      timeC=np.append(timeC,
time.time()-start)
      with threading.Lock():
         data_done=0
   return


def stopData():
   global stop
   time.sleep(timeInterval)
   #input("Press ENTER in console to
STOP")
   #msvcrt.getch()
   stop=1
```

```
def calibrate(in_data, frame_count,
time_info, flag):
   global n
   if flag:
      print("Calibration Error")
   ntemp = np.frombuffer(in_data,
dtype=np.float32)
   n = np.append(n,ntemp) #saves
filtered data in an array
   return (ntemp, pyaudio.paContinue)


def downsample(data):
   #Trim the signal so we can down
sample
   data.buff=np.transpose(input_buffer)
   trimmer= len(data.buff) % downratio
   length=len(data.buff)-trimmer
   x=[]
   for i in range(length):
      x = np.append(x, data.buff[i])
   #Actual down sample
   Fs=Fo/downratio
   data.buff=[]
```

```
    for i in range(int(length/downratio)):
        data.buff = np.append(data.buff,
x[i*downratio])
    data.buff= np.divide(data.buff,
max(np.fabs(data.buff)))
    return

def detect_Noise(f):
    length=len(f)
    noise=np.zeros((length,1,))
    for i in range(int(length)):
        noise[i]=20*np.log(np.fabs(f[i,0]))
        if noise[i]==float('-inf'):
            noise[i]=0.0
    return (np.mean(noise))

def detect_K(f):
    K=0
    length=len(f)
    for i in range(int(length)):
        if( 20*np.log(np.fabs(f[i,0])) > noise
):
            K=K+1
    return K


def main(sec, n, f):
    global Phi, Psi, stream, pa, noise, Ms,
output_buffer, t, db, timeInterval, N, Fo,
m, downratio, NUM_DB
    timeInterval=sec
    N=n
    Fo=int(f)
    downratio=math.floor(Fo/reducedFs)
    N = 1024
    M = 128
    SAMPLES = N*2*downratio
    NUM_DB =
int(SAMPLES/N/downratio)
    m=np.log(N)*(0.35**2)
    if N==128:
        phi="Phi128.txt"
    elif N==256:
        phi="Phi256.txt"
    elif N==512:
        phi="Phi512.txt"
    elif N==1024:
        phi="Phi1024.txt"
    t=[]
    db=[]
    for i in range(NUM_DB):
        db=np.append(db, data_buffer([],
M, i*N, (i+1)*N))
        t=np.append(t,
threading.Thread(target=processData,
args=(db[i],), daemon=True))
    tout =
threading.Thread(target=outputData,
daemon=True)
    tstop =
threading.Thread(target=stopData,
daemon=True)
    #CS variable Phi and Psi
    I = np.matrix(np.identity(N))
    Psi=np.matrix(dct(I))
    Phi=np.matrix(np.reshape(
np.loadtxt(phi),(N,N)))
    print("Calibrating - Stay quiet to
measure noise")
    calib = pa.open(format=FORMAT,
                    channels=CHANNELS,
                    rate=Fo,
                    input_device_index =
DEVICE,
                    output=False,
                    input=True,
                    frames_per_buffer =
SAMPLES,
                    stream_callback=calibrate)
    calib.start_stream()
    while calib.is_active():
        time.sleep(TIME_TO_CALIBRATE)
        calib.stop_stream()
    calib.close()
    pa.terminate()
    pa = pyaudio.PyAudio()
    noise =
detect_Noise(dct(np.matrix(n)))
    print("Noise: ",noise,"dB")
    #input("Press Enter key in console to
START")
    #msvcrt.getch()
    stream = pa.open(format=FORMAT,
            channels=CHANNELS,
            rate=Fo,
            input_device_index=DEVICE,
            output=False,
            input=True,
            frames_per_buffer =
SAMPLES,
            stream_callback=inputData)
    stream.start_stream()
    for i in range(NUM_DB):
        t[i].start()
    tout.start()
    tstop.start()
    while not stop:
        for i in range(NUM_DB):
            if not t[i].isAlive():
                t[i] =
threading.Thread(target=processData,
args=(db[i],), daemon=True)
                t[i].start()
        if not tout.isAlive():
            tout =
threading.Thread(target=outputData,
daemon=True)
            tout.start()
    stream.stop_stream()
    stream.close()
    pa.terminate()
    for i in range(NUM_DB):
        t[i].join()
    tout.join()
    tstop.join()
    print("N:", N, "\tNumber of
data_buffers used:", NUM_DB)
    print("Sampling period:",
SAMPLES/Fo)
    print("len(xo):",np.shape(xo))
    print("len(y) expected:", np.sum(Ms), "
vs Actual:",np.shape(output_buffer))
    print("Number of M indices:",
np.shape(Ms))
    print("CS time:", np.sum(timeC))
    start=len(Ms)
    end=len(output_buffer)
    if end>start:
        for i in range(end-start):
            Ms=np.append(Ms,0)
    else:
        for i in range(start-end):
            output_buffer =
np.append(output_buffer,0)
    y=pd.DataFrame(data={'Y':
output_buffer, 'M': Ms})
    x=pd.DataFrame(data={'X': xo})
    np.savetxt(r'y.txt', y.values, fmt='%d')
    np.savetxt(r'x.txt', x.values,
fmt='%1.4e')
    start=time.time()
    import gzip
    with gzip.open('x.gz', 'wb') as f:
        f.write(xo)
    end = time.time()-start
    print("Zip time:", end)

#Window main parent and title
if __name__ == "__main__":
    main()
```

Python Rx – Sink Node

```
import numpy as np
import time
from scipy.fftpack import dct, idct
from scipy.io import wavfile
from decimal import Decimal
import matplotlib.pyplot as plt

#Variables needed for SL0
L=3
sigma_min=1e-5
sigma_decrease_factor=0.5
mu_0=2
Fs=8e3
phi="Phi128.txt"

def main():
    #Import Phi from saved .txt file
    f=open(phi,"r")
    data=f.readlines()
    f.close
```

```
data=[x.strip() for x in data]
N=int(np.sqrt(len(data)))
Phi=np.zeros((N**2,1,))
for i in range(len(data)):
    Phi[i]=data[i]
Phi=np.matrix(np.reshape(Phi,(N,N)))
#Create inverse of Psi
invPsi=np.matrix(idct(np.identity(N)))
#Read y for observations and M used
f=open("y.txt", "r")
data=f.readlines()
f.close
data=[x.strip() for x in data]
data=[x.split() for x in data]
length=len(data)
y=np.zeros((length,1,))
M=np.zeros((length,1,))
for i in range(length):
    y[i]=data[i][1]
    M[i]=data[i][0]
y=np.matrix(y)
M=np.matrix(M)
#Read x for RMS calculations
    f=open("x.txt", "r")
    data=f.readlines()
    f.close
    data=[x.strip() for x in data]
    length=len(data)
    x=np.zeros((length,1,))
    xp=np.zeros((length,1,))
    for i in range(length):
        x[i]=data[i]
    x=np.matrix(x)
    length=len(x)
    dct_test=dct(x)
    #Initialize variables for while loop and
reconstruct xp
    indx=0
    indy=0
    j=0
    do=1
    p=np.zeros((length,1,))
    while do:
        done =
round(Decimal(indx/length*100),2)
        if (indx+N) > length:
            N = length-indx
            invPsi =
np.matrix(idct(np.identity(N)))
        start=time.time()
        ytemp = y[indy:indy+int(M[j])]
        indy = indy+int(M[j])

        Phitemp = Phi[:int(M[j]),:N]
        xp[indx:indx+N] = SL0(Phitemp,
ytemp, sigma_min,
sigma_decrease_factor, mu_0, L)
        xp[indx:indx+N] =
invPsi*xp[indx:indx+N]
        p[j]=time.time()-start
        indx = indx+N
        j=j+1
        if (np.sum(M[j+1:len(M)])==0.0):
do=0
    x=np.divide(x, max(np.fabs(x)))
    xp= np.divide( xp, max(np.fabs(xp)) )
    add=np.sum(p[:j])
    RMS = np.sqrt( np.mean(
np.square(x-xp)))
    print("CS RMS:\t:",
round(Decimal(RMS),3))
    print("Time",round(Decimal(add),3))
    wavfile.write('xp.wav',int(Fs),
np.asarray(xp, dtype=np.int16))

def SL0(A, x, sigma_min,
sigma_decrease_factor, mu_0, L):
    A_pinv=np.linalg.pinv(A)
    s=np.matrix(A_pinv*x)
    sigma = 2*max(np.fabs(s))
    while sigma>sigma_min:
        for i in range(L):
            delta =
np.matrix(OurDelta(s,sigma))
            s = s-mu_0*delta
            s = s- A_pinv*(A*s-x)
        sigma = sigma *
sigma_decrease_factor
    return s

def OurDelta(s, sigma):
    delta=np.multiply(s,np.exp(-
np.square(np.fabs(s)) / sigma**2))
    return delta


#Window main parent and title
if __name__ == "__main__":
    main()
```

BIBLIOGRAPHY

[1] D. L. Donoho, "Compressed Sensing," *IEEE Transactions on Information Theory,* vol. 52, no. 4, pp. 1289-1306, 2006.

[2] C. Moler, ""Magic" Reconstruction: Compressed Sensing," MathWorks, 2010. [Online]. Available: https://www.mathworks.com/company/newsletters/articles/magic-reconstruction-compressed-sensing.html. [Accessed 2017].

[3] E. Candes and J. Romberg, *L1 magic: Recovery of Sparse Signals via Convex Programming,* Caltech, 2005.

[4] S. Pinto, L. Menoza, H. Velandiz, V. Molina and a. L. Cervelon, "Compressive Sensing Hardware in 1-D Signals," *TECCIENCIA,* vol. 7, no. 19, pp. 5-10, 2015.

[5] E. Candes and M. Wakin, "An Introduction to Compressive Sampling," *IEEE Signal Processing Magazine,* pp. 21-30, 2008.

[6] Y. C. Eldar, "Compressed Sensing of Analog Signals in Shift-Invariant Spaces," Israel Science Foundation, 2009.

[7] H. Huang, S. Misra, W. tang, H. Barani and H. Al-Azzawi, "Applications of Compressed Sensing in Communications Networks," New Mexico State University, 2014.

[8] T. L. N. Nguyen and Y. Shin, "Deterministic Sensing Matrices in Compressive Sensing: A Survey," Soongsil University, 2013.

[9] "Complex Matrices; Fast Fourier Transform," MIT OCW, Cambridge, 2011.

[10] G. Mohimani, M. Babaie-Zadeh and C. Jutten, "Fast Sparse Representation based on Smoothed l0 Norm," Center for International Research, Tehran, 2006.

[11] E. Candes and J. Romberg, *Sparsity and Incoherence in Compressive Sampling,* Caltech, 2006.

[12] H. Xue, L. Sun and G. Ou, *Speech Reconstruction based on Compressed Sensing Theory using Smoothed L0 Algorithm,* Nanjing: IEEE, 2016.

[13] T. T. Do, L. Gan, N. H. Nguyen and T. D. Tran, "Fast and Efficient Compressive Sensing using Structurally Random Matrices," *IEEE Transactions on Signal Processing,* pp. 1-16, 2011.

[14] C. Luo, F. Wu, J. Sun and C. W. Chen, "Compressive Data Gathering for Large-Scale Wireless Sensor Networks," Microsoft Research Asia, Beijing.

[15] Z. Yu, S. Hoyos and B. M. Sadler, *Mixed-Signal Parallel Compressed Sensing and Reception for Cognitive Radio,* IEEE, 2008.

[16] F. Chen, A. P. Chandrakasan and V. Stojanovic, *A Signal-Agnostic Compressed Sensing Acquisition System for Wireless and Implantable Sensors,* Cambridge: IEEE, 2010.

[17] J. Laska, S. Kirolos, Y. Massoud, R. Baraniuk, A. Gilbert, M. Iwen and M. Strauss, *Random Sampling for Analog-to-Information Conversion of Wideband Signals,* DARPA, 2011.

[18] J. Moreira, *What is... A RIP Matrix?.*

[19] Y. Wnag, J. Ostermann and Y.-Q. Zhang, Video Processing and Communications, Upper Saddle River: Prentice-Hall, 2002.

[20] J. G. Proakis and D. G. Manolakis, Digital Signal Processing Principles, Algorithms, and Applications, Upper Saddle River: Pearson Education, 2007.