Implementation of compressive sampling for wireless sensor network applications

Nathan A. Ruprecht

Department of Electrical Engineering,
University of North Dakota,
Grand Forks, ND, 58202, USA
Email: nathan.ruprecht@und.edu

Xinrong Li*

Department of Electrical Engineering,
University of North Texas,
Denton, TX, 76203, USA
Email: xinrong.li@unt.edu
*Corresponding author

Abstract: Since mid-20th century, Nyquist-Shannon Sampling Theorem is accepted as we need to sample a signal at twice the max frequency component in order to reconstruct it. Compressive sampling (CS) offers a possible solution of sampling sub-Nyquist and reconstructing using convex programming techniques. There has been significant advancements in CS research and development (more notably since mid-2000s in theory and proofs), but still nothing to the advantage of everyday use. There has been little work on hardware in finding realistic constraints of a working CS system used for digital signal processing (DSP) applications. Parameters used in a system are usually assumed based on stochastic models, but not optimised towards a specific application. This paper aims to address a minimal viable platform to implement compressive sensing if applied to a wireless sensor network (WSN), as well as addressing key parameters of CS algorithms to be determined depending on application requirements and constraints.

Keywords: compressive sensing; compressive sampling; WSN; wireless sensor network.

Reference to this paper should be made as follows: Ruprecht, N.A. and Li, X. (xxxx) 'Implementation of compressive sampling for wireless sensor network applications', *Int. J. Sensor Networks*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Nathan A. Ruprecht received his BS and MS in Electrical Engineering in 2017 and 2018 respectively from the University of North Texas (UNT). He is currently pursuing his PhD in EE at the University of North Dakota (UND) while concurrently serving on active duty as an officer in the U.S. military.

Xinrong Li received his BE from the University of Science and Technology of China, Hefei, China, in 1995, ME from the National University of Singapore in 1999, and PhD degree from Worcester Polytechnic Institute (WPI), Worcester, MA, in 2003, all in Electrical Engineering. From 2003 to 2004, he was a Post-doc Research Fellow at the Center for Wireless Information Network Studies, WPI. He joined the Department of Electrical Engineering, University of North Texas, Denton, Texas, as an Assistant Professor in 2004, and then he was tenured and promoted to Associate Professor in 2010. His recent research has been focused on statistical signal processing, machine learning, geolocation, and wireless sensor network systems.

1 Introduction

Because of the Nyquist-Shannon Theorem dictating that one needs to sample a signal at least twice the maximum frequency component, engineers have hit a sort of plateau in utilising high frequencies for practical applications. It seems the need for higher frequency use is outrunning the advancement in ADC hardware. A possible solution then is to sample at a lower rate. The idea of sampling sub-Nyquist through Compressive Sampling came about in the 1970s when seismologists

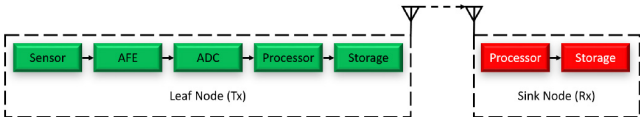
able to produce geographical images from data that were sampled at less than the Nyquist rate. Slow improvements were made over the years, but the most notable revitalisation of the topic came in 2006 with research done by Candès et al. (2006) and Donoho (2006). Since CS takes some strain off of the ADC and instead is more computationally heavy, the more recent improvements in processing power have made CS more viable.

More updated research finds useful applications of CS in medical imaging (Lustig et al., 2008) or communication systems (Zhang et al., 2009). Not far after is its need in wireless sensor networks (WSN). Narrowing in on WSNs, there are uses for CS in occupancy in a room for tracking or navigation (Au et al., 2013), surveillance or security (Mahalanobis and Muise, 2009), or environmental monitoring systems (Wu et al., 2016). This paper looks to implement CS in WSN systems in order to expose and study challenging issues in practical applications.

A basic WSN may follow a star topology in that a number of leaf nodes transmit sensor data to a singular base station (BS) node or sink node. There are network configurations, sensor parameters, or software optimisation that can all help WSN performance (Acevedo, 2015), but keep in mind that the typical WSN encounters similar issues including: power, transmission speeds, connectivity, etc. CS would help with power consumption and transmission speeds since it's reducing the amount of data that needs to be sampled and transmitted, but one would have to worry about the signal characteristics since theory is based on the assumption that the signal of interest is sparse.

Looking into applying CS to a WSN, this paper takes a specific interest in the interaction between a single leaf (i.e., transmitter or Tx) node and the sink node (i.e., receiver or Rx). A block diagram of the two nodes is shown in Figure 1 where the ADC of the leaf node samples at the Nyquist rate.

Figure 1 Typical architectures of Leaf Node and Sink Node in a wireless sensor network (see online version for colours)



The difference when implementing CS is in adjusting the sampling rate of the ADC or even to randomly take samples. This brings up the idea of analogue-to-information conversion (AIC) in that only important aspects that represent the information are captured. In this, the signal seems to be processed as it is being sampled. Such a slight difference in component relationships is how CS can create a tradeoff between taking workload off of the ADC and the need for more computationally heavy processing.

Implementing CS comes with its own set of hurdles. How much compression can be achieved? What levels of error are to be expected? How slow can a platform be and still able to perform the math functions for real-time processing? Compressive Sampling is based on exploiting sparse signals; what if the signal is not sparse as in most practical signal processing applications? The basic theory of CS says the processor still needs the original signal sampled at Nyquist, so is not CS just another compression technique? How does it compare to compression standards already widely used such as MP3 or zip? These will be the main questions focused on in this paper.

More specifically, the objectives of this paper are to:

- show a practical implementation of CS for WSN applications
- explore and expose the issues that need to be addressed when implementing a CS system
- investigate the issues in applying CS more broadly to the signals that are not perfectly sparse.

In this paper, without loss of generality, we employ audio signals of varying length, source, and sounds for both simulation and implementation. The three music files used for simulation and analysis in this paper are 'Clarity' by Zedd, 'Whatever It Takes' by Imagine Dragons, and 'Nuvole Bianche' by Ludovico Einaudi; all three music files are saved using the Waveform Audio File format (i.e., .wav extension). Although we look into implementing CS explicitly to audio signals, there are still a number of applications in audio-surveillance systems (Valenzise et al., 2007), audio-habitat monitoring (Wang et al., 2003) and other fields. As solutions get more complex for occupancy detection (Lam et al., 2009) or surveillance analysis (Cucchiara, 2005), one runs into the same WSN challenges mentioned, especially data collection and storage (Shah et al., 2003) (Luo et al., 2007). Compressive Sampling has a usefullness in application that carries into multiple areas of sensor networks, we narrow in on audio to prove so. Since audio monitoring will generate a significant amount of data, it's highly desirable to use CS techniques to reduce the amount of data that needs to be transferred through sensor networks.

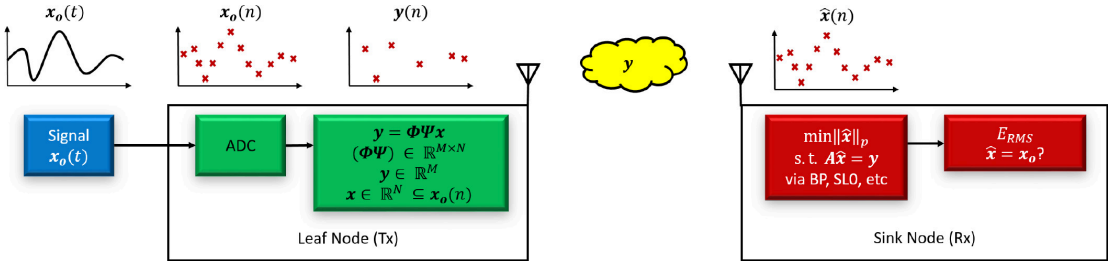
To address the objectives outlined above, the rest of the paper is organised as follows: the next section will cover the basic theory of CS along with the terminology used in order to stay consistent. Section 3 shows how to tailor parameters of a CS system to the user's needs. Section 4 covers the issues that should be kept in mind for further implementation, then Section 5 shows successful implementation in Matlab and the Raspberry Pi 3, while Section 6 is comparing expectation to reality. Section 7 discusses results and future work. Finally, Section 8 wraps up research and will touch back on how well this paper completes its objectives.

2 Theory

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. The two principles that CS relies on is

- sparsity, which is a characteristic of the original signal that helps with compression, and
- incoherence, which is a characteristic of the system's measurement matrix that helps with reconstruction (Candes and Wakin, 2008).

Figure 2 Illustration of CS and reconstruction in WSN systems (see online version for colours)



To compress a signal in this case, it is just a matter of matrix multiplication. Let there be a signal $\mathbf{x} \in \mathbb{R}^N$ being a one-dimensional column vector (e.g., audio signal) that has N samples. If $\mathbf{x}$ is multiplied by $\mathbf{A} \in \mathbb{R}^{M\times N}$ where $M \ll N$, the result is a $\mathbf{y} \in \mathbb{R}^M$ a representation of $\mathbf{x}$ given by (Moler, 2010):

$$\mathbf{y} = \mathbf{A}\mathbf{x}. \tag{1}$$

For terminology, the complete, original signal is referred to as $\mathbf{x_o}$, $\mathbf{x}$ is an N-sized frame of $\mathbf{x_o}$, $\mathbf{y}$ an M-sized observation matrix of $\mathbf{x}$, the measurement matrix $\mathbf{A}$ consists of Φ and Ψ, and Root Mean Squared error (RMS) as the system's performance metric. With Φ as a distribution matrix and Ψ a transform matrix, equation (1) is broken down to:

$$\mathbf{y} = \Phi\Psi\mathbf{x}. \tag{2}$$

As mentioned earlier, $\mathbf{x_o}$ is the original signal of unknown length, whether it be a music file or recording from a microphone. In order to process $\mathbf{x_o}$, it is broken into N-sized snippets or frames shown as $\mathbf{x}$ and then compressed using equation (1). After compressing to $\mathbf{y}$, the reconstruction process is to find $\hat{\mathbf{x}}$ comparable to $\mathbf{x}$, then concatenated together to be the reconstruction of $\mathbf{x_o}$. The amplitude can vary depending on $\mathbf{x_o}$ and therefore vary $\hat{\mathbf{x}}$. To view all results with the same level of scrutiny (consistent RMS calculations), $\mathbf{x_o}$ and $\hat{\mathbf{x}}$ are normalised preprocessing. Being normalised, the maximum error would occur with $\mathbf{x_o} = \mathbf{1}$ and $\hat{\mathbf{x}}$ guessing a default value of $\mathbf{0}$. Error is then bounded by $0 \leq E_{RMS} \leq 1$ and defined as:

$$E_{RMS} = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(\mathbf{x} - \hat{\mathbf{x}})^2}. \tag{3}$$

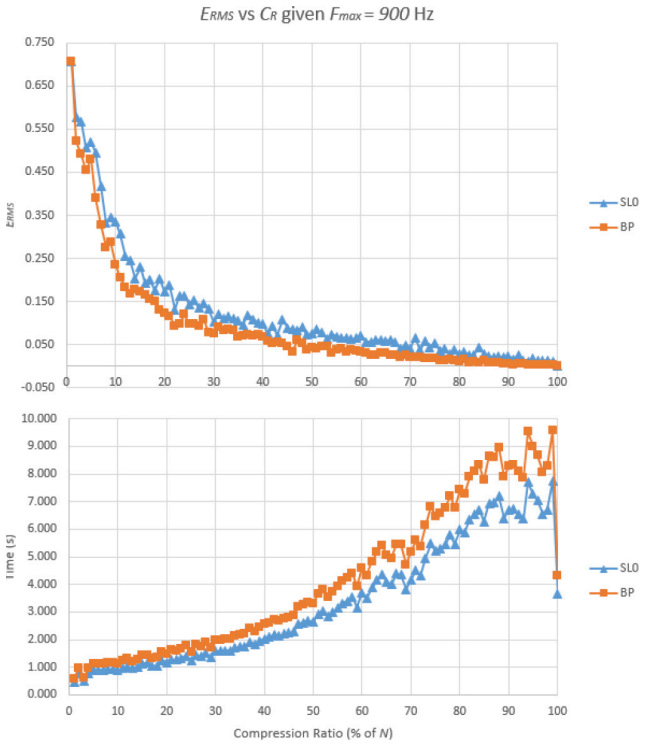
Since there is compression, the reconstruction is a problem of solving an underdetermined system of equations. With an infinite number of solutions to reconstruct $\mathbf{x}$, finding the right one, $\hat{\mathbf{x}}$, involves constrained minimisation of the $l_p(\hat{x})$ norm (Candes and Romberg, 2005):

$$\begin{aligned} \hat{\mathbf{x}} &= \underset{x}{\text{argmin}} \|\mathbf{x}\|_p \\ &\text{s.t. } \mathbf{A}\mathbf{x} = \mathbf{y}. \end{aligned} \tag{4}$$

The perfect solution would be l_0, but it is too sensitive to noise since the solution is just a matter of finding all nonzero elements (easily satisfied by noise). The most notable linear programming technique that others are compared to is Basic Pursuit (BP) which solves for the l_1 norm. With high probability, using BP will find l_1 norm is equivalent to l_0 norm (Candès and Romberg, 2007). Other techniques include orthogonal matching pursuit (OMP) (Tropp and Gilbert, 2007), regularised OMP (ROMP) (Needell and Vershynin, 2010), and compressive sampling matching pursuit (CoSaMP) (Needell and Tropp, 2009) to name a few, but this paper looks more closely at Smoothed-L_0 (SL0) (Mohimani et al., 2007). The process of CS and reconstruction is illustrated in Figure 2.

SL0 is on par with BP in performance with varying compression ratio of a signal shown by Figure 3. Whereas, when it comes to increasing number of frequencies in the signal, or with a varying N-sized $\mathbf{x}$, SL0 is both faster, and less error prone than BP shown by Figure 4.

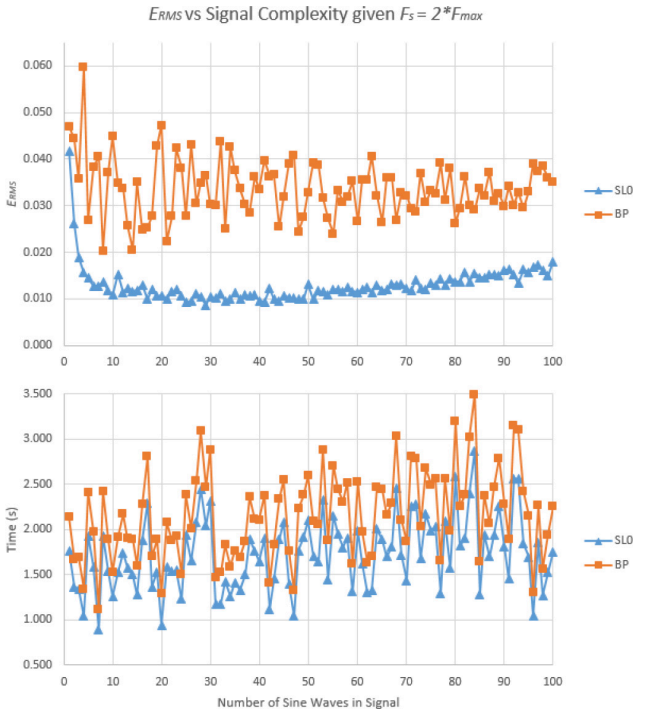
Figure 3 BP vs. SL0 with varying compression ratio (C_R) (see online version for colours)



A single tone signal with a frequency component at 900 Hz was used to generate Figure 3. With that same signal, running CS over and over again with an increasing compression ratio for both BP and SL0, then plotting both error and time of execution for the two algorithms. Figure 4 was generated in the similar way, except each loop added an additional frequency component. The maximum frequency component stayed at 900 Hz in order to easily satisfy the Nyquist rate for sampling

purposes. So a frequency component of 15 Hz less was added to the signal each time as to increase the signal complexity. The idea is that a compression algorithm takes advantage of a sparse domain, but what happens when the frequency spectrum is filling up? The purpose of Figure 4 was to show what to expect as the signal becomes less sparse. The aforementioned system implementing BP and SL0 is further explained in Section 5.

Figure 4 BP vs. SL0 with increasing complex signals (see online version for colours)



With these two figures, it can be assumed that the system implemented using SL0 will perform as well as, if not better than, BP. In any case, both algorithms are available for implementation should the user compare on their own or choose a different outcome. After setting the foundation and choosing a reconstruction technique, next is setting parameters for a specific application.

Shown in Figure 2, the relationship between the leaf and sink node is better shown with the leaf using equation (2), then transmitting the $\mathbf{y}$ observations to the sink node. The transform matrix, Ψ, is not shown because it is shared between the leaf and sink node, and not transferred during run time. Although Figure 2 shows $\mathbf{y}$ as a function of n as random observations of $\mathbf{x_o}$, this is just for illustration purposes. Because of the matrix multiplication, $\mathbf{y}$ is a weighted sum of N elements of $\mathbf{x_o}$.

3 Parameter optimisation

Using Matlab to simulate different situations, there are pieces to a CS system that can be further explored and optimised. The performance of a system depends on the measurement matrix, $\mathbf{A}$, seeing as it is used both to compress and decompress the data. Choosing the distribution and transformation matrices along with a process to choose the 2D size of $\mathbf{A}$ are all within the user's control and are preset before use.

This section shows the experimental results of

- choosing any combination of Φ and Ψ,
- how to use an aggressive equation to calculate the number of rows in $\mathbf{A}$ given by M
- the limitation of choosing a value for N.

3.1 *Measurement matrix* $\mathbf{A} = \Phi\Psi$

CS exploits the fact that in some applications, signals are sparse or compressible in some domain – i.e., capturing the few spikes in the frequency domain instead of sampling at a uniform rate in the time domain. A Ψ is needed to transform $\mathbf{x}$ into some domain that can represent the signal by the K largest coefficients. The distribution matrix is used to randomise the transformed signal. Almost like encoding a signal before transmitting, there would exist a unique solution when decoding. With infinite number of solutions, a perfectly random Φ would have only one correct solution.

In 2004, when Donoho and Candes were bringing life to further research and development of CS, they gave the idea of the random isometry property (RIP) that is a pass/fail test on if the measurement matrix chosen would work in a system (Moreira, 2014). Fortunately, stochastic matrices would pass the test with high probability, so it was more for those looking to implement a deterministic $\mathbf{A}$. To measure the expected performance of $\mathbf{A}$, calculate the mutual coherence coefficient, μ. Since a random matrix is wanted, the best $\mathbf{A}$ is perfectly incoherent and the lower bound of $1/\sqrt{N} \leq \mu \leq 1$ found by (Nguyen and Shin, 2013):

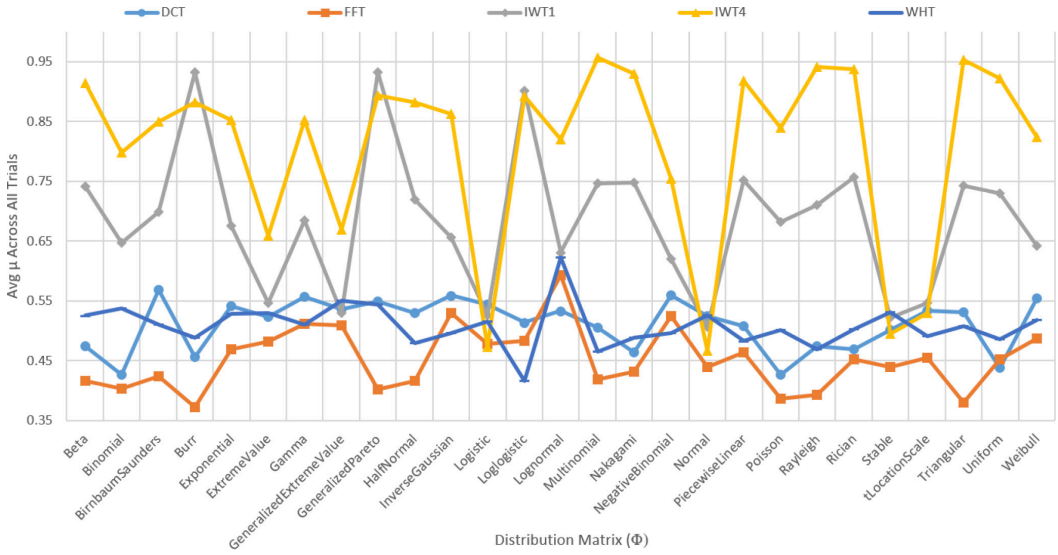
$$\mu(\Phi, \Psi) = \max_{1 \leq i,j \leq N} \frac{|\langle A_i, A_j \rangle|}{\|A_i\| \|A_j\|}. \tag{5}$$

With this equation, let's compare different transform vs. distribution matrices. Tested below are 27 different distributions against 5 different transforms. Not all combinations are implementable in hardware, but were still simulated in Matlab. Besides the DCT and FFT, also tested Ψ are the integer wavelets (IWT) with a db1 (Haar) and db4 filter along with the fast Walsh-Hadamard transform (WHT). The coherence coefficient was calculated for all 135 combinations of $\mathbf{A}$ at $N = 2^1$ through 2^{10}. The average μ for every combination was then ranked from smallest to largest since a lack of coherence is desired. All results are shown in Figure 5.

Narrowing in on a simpler comprehension, Table 1 shows every transform with the top performing distributions along with the Normal (i.e., Gaussian) distribution if not already included. Reasoning for including the Normal distribution for each transform is because most references assume Normal being the best distribution since a Normal distribution will have the highest entropy of any random variable with the same variance (Wang et al., 2001).

In order to test the validity of this assumption, each combination of $\mathbf{A}$ in Table 1 was then put into a CS system to measure its performance. The Matlab implementation of

Figure 5 Coherence Coefficient of all combinations of **A** (see online version for colours)



such a CS system is explained in Section 5 in detail. All combinations were tested on the same signal, multiple times, and averaged results of both E_{RMS} and time of execution (compression and reconstruction) in seconds. The results on the top performing Φ (along with Normal if it's not the best) are shown in Table 2.

Table 1 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

The takeaway note from this comparison study is that the Normal distribution should not be accepted as the best choice of Φ without concern for a specific application. As we can see with both Ψ = DCT or FFT, Normal is slower and more error prone. Only with integer wavelets is Normal the best option. But even then, E_{RMS} is much higher than any other combination shown in that table. Even more, this test was done on an audio signal where FFT has an innate advantage, but DCT was faster than FFT with only a slight tradeoff in E_{RMS}. This table is critical when deciding Φ and Ψ to optimise a CS system for both speed and memory space. A DCT paired with Binomial will only deal with real values, and a simple 0 or 1 for the distribution matrix. Whereas FFT paired with Normal will require twice as much memory in order to save a real and imaginary part, along with floating point representation for the distribution matrix. Yet it seems FFT performs better than DCT in aims to reduce E_{RMS}. This tradeoff of memory vs. error or time becomes a problem when implementing in hardware platforms and discussed more in Section 4.

Table 2 Best Φ performance in a CS example

Ψ	Φ	*RMS*	*Time* (s)
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

3.2 Choosing compression ratio

With an idea of how the completed system will perform by optimising the measurement matrix on compression, and using SL0 to reconstruct at both a faster rate and less error prone than BP, the question then becomes how small of a size can the original signal be compressed and still maintain its integrity? In other words, how to minimise compression ratio defined by:

$$C_R = \frac{M}{N}, \tag{6}$$

where M and N are still the size of $\mathbf{y}$ and $\mathbf{x}$ respectively. N can be chosen based on computational power needed by the platform in performing N-point transformations and matrix multiplication. M is then left to either be chosen setting C_R to a set value, or calculated depending on the signal itself. Shown later are the results of defining a set C_R, but let's first look into

the effects of calculating M. With a very broad overview, M is calculated by (Do et al., 2012):

$$M = \mathcal{O}(K \log N), \tag{7}$$

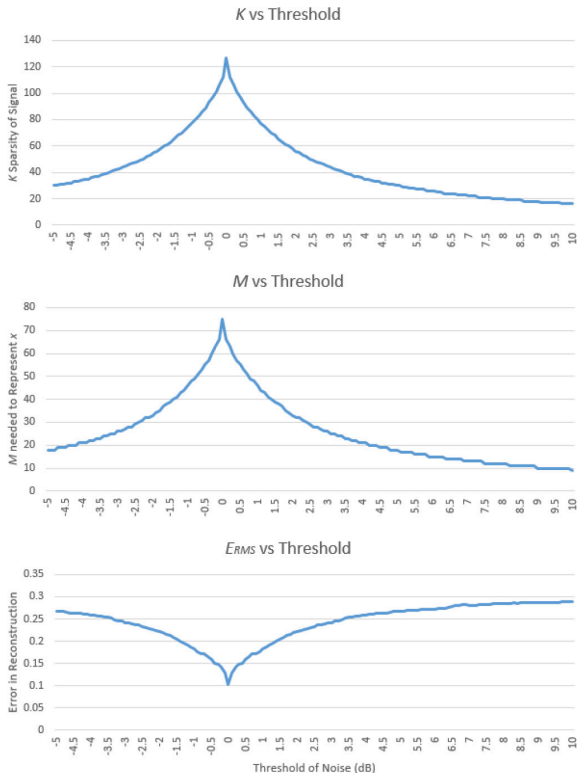
such that M is on the order of multiplying the sparsity of the signal, K, times the logarithmic size of the frame, $\mathbf{x}$. This equation is impractical since it is used where $N \to \infty$. Honing in on a more suitable equation specifies equation 7 and explanation on M with (Huang et al., 2013):

$$M = \mu^2 K \ln N, \tag{8}$$

where μ is the coherence coefficient discussed earlier, K and N are still the sparsity and length of the frame, $\mathbf{x}$. This equation is still not exact (more aggressive in compression) nor universal (DCT vs. FFT), but is a good starting point and able to show proof of concept within the CS system created in this paper.

The next variable to define is how to measure K, the sparsity of signal $\mathbf{x}$. In an ideal signal with no noise, the sparsity K can be determined with a simple for-loop to count the number of nonzero elements within the signal data frame. Instead, in more practical application scenarios, a threshold needs to be set at a level above the noise floor, then counting the number of elements above that threshold. Coincidentally for this scenario and the samples used, the threshold to minimise E_{RMS} is at 0 dB shown in Figure 6.

Figure 6 K, M, and E_{RMS} shown vs. the noise threshold (see online version for colours)

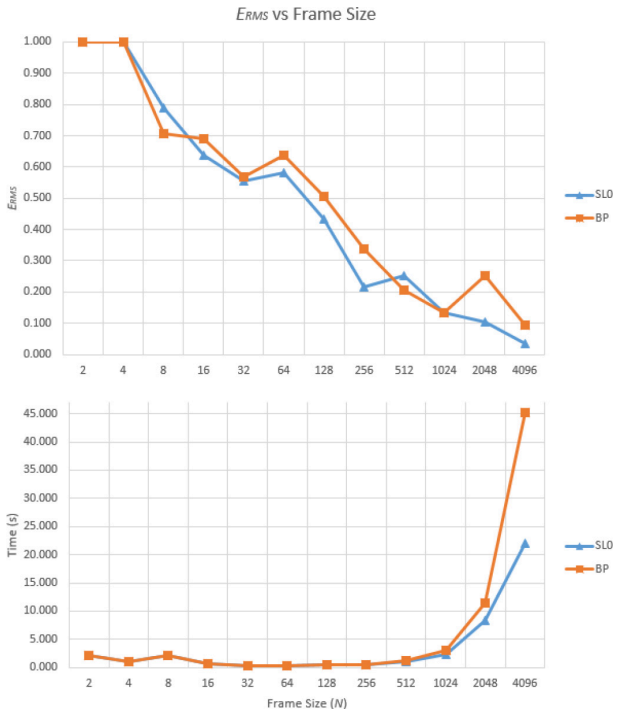


See that as K increases, the resulting calculation of M is higher, and a smaller C_R means less error in reconstruction. It just happens that the threshold corresponding to such results is at 0 dB.

3.3 *Limitation of choosing N*

Since $\mathbf{x_o}$ is broken into N-sized frames, $\mathbf{x}$, there will be some level of error due to rounding and N-point transform representation. Referring back to SL0 vs. BP, one of the metrics was to vary N and compare the two techniques in regards to E_{RMS} and time to execute. Not only does Figure 7 show that SL0 performs better in both error and time, but more importantly that a larger N corresponds to less error. This helps confirm the earlier assumption on equation (7) working for $N \to \infty$.

Figure 7 BP vs. SL0 with varying N sized frames (see online version for colours)



4 Issues in practical implementation

Usually small and easily fixed, there are issues found that will restrict the user on the platform of choice. In order to find such errors and find the minimum viable platform, a familiar microcontroller (MCU) was used to quickly implement a CS system - the MSP432P Launchpad developed by Texas Instruments.

Two factors that played into originally choosing this board is the CMSIS library that is ARM software developed for Cortex-M processors to simplify complex DSP functions, as well as the EDU booster pack from T.I. (EDUMKII) that has a microphone for testing the system. Using code composer studio (CCS) as the coding platform, and CMSIS library for transformation and matrix algebra, CS was implemented using a triple buffer with ISRs running on a PWM timer at $F_s = 8$ kHz (Welch et al., 2012).

A realistic constraint to be aware of is which Ψ are actually usable. Five different transformations were implemented in Matlab, three of which were capable of any sized N. Whereas

the CMSIS library can perform the FFT and DCT with just a couple options for size (i.e., DCT can only be operated on sized $N = 128, 512, 2048$). Beyond available N-point transformations limitations, problems came about on this Launchpad because of 2 factors that are issues CS is used to overcome in practical WSN use: memory and speed.

4.1 Memory

Let $N = 128$ and $\Phi =$ Normal distribution needing floating point representation, an expected 65.5 KB of memory is used to store a complex $\mathbf{A}$ when the MSP432 only has 256 KB of Flash Main Memory. Setting $N = 128$ and $\Phi =$ Bernoulli would reduce memory usage to 16.3 KB, but is using the lowest possible N therefore increasing E_{RMS}. Even then, that combination of $\mathbf{A}$ is too large to be stored on the MSP432.

A possible solution to move forward with is saving a set amount of rows of $\mathbf{A}$ into memory according to a level of error allowed by the user. Upon testing, there seemed to be no compromise in that E_{RMS} would be acceptable so it's a matter of finding a capable platform with significantly more storage.

4.2 Speed

Another hurdle would be how fast the platform can perform the necessary computations. Since CS would take some load off the ADC in full implementation, that work is then put on the processor. There must be some lower bound to a minimum viable platform.

In the implemented CS systems, minimum speed capability is shown by the presence/absence of a buffer overrun. In order to do real-time processing, a triple buffer needs to be able to process the data before its input buffer is filled. If not, there's a backup of data needing to be processed and no longer real-time. A "buffer overrun" is when the input buffer is filled yet the previous input buffer is not done being processed; meaning the time it takes to process is larger than the time it takes to fill the input buffer. The solution is to either run at a faster clock speed or make the input buffer larger in order to give more time to processing. Table 4 shows the Launchpad set to different clock speeds and the resulting time to process a single N-sized frame (using $\Psi =$ DCT).

Table 3 Sampling period in seconds corresponding to varying frame size N

N	*128*	*512*	*2048*
Sampling period	0.016	0.064	0.256

Table 4 Processing time in seconds for varying clock frequencies F_{clock} and frame size N

F_{clock} \ N	*128*	*512*	*2048*
48 MHz	0.4983	0.6398	0.7838
24 MHz	0.5224	0.5447	2.9056
12 MHz	0.538	0.5795	4.8783

With Sampling Period simply the number of data samples (N) divided by F_s as shown in Table 3, it's observed that every value for its clock frequency would still result in a buffer overrun. There is another T.I Launchpad that has a faster clock than the MSP432 that could prove a viable MCU, the Tiva C Launchpad series. Depending on the platform, there are boards that have a clock speed upwards of 80 MHz along with an ARM processor in order to use the CMSIS library.

Early application of CS on this Launchpad quickened the process of identifying early issues to keep in mind and helped in optimising parameters for the final platform.

5 Implementation

The basic goal is to implement a system for practical use comparable to those that typically sample at $F_s = 8$ kHz, but now with CS for ease of transmission within a WSN. Looking for the minimum specifications to do so involves finding the least capable platform that could still handle the computational intensity in order to successfully run a CS system. Everything discussed up to this point is now taken into Matlab to prove its use, then into hardware platforms to see how well reality matches expectations set by simulations.

5.1 Matlab Simulation

Using Matlab Figure 8 mimics the process of implementing an AFE, detecting the noise floor and thereby K-sparsity, then running through the CS system to find both time and error of a file or audio recording. Functions used by main are:

- *Down sample*: Sample at $F_s = 48$ kHz when the target is 8 kHz, so down sample $\mathbf{x_o}$ before processing.
- *Detect noise*: Need to know the noise floor to help find the sparsity. To do so, sum the energy of entire signal and use 95% BW plus a small additional threshold from Figure 6 (Proakis and Manolakis, 2006).
- *Detect K-sparsity*: Go through the frame and count how many spikes are above the noise floor from before.
- *Calculate* μ: Calculate the coherence coefficient for the given Φ and Ψ.
- *SL0*: The reconstruction function that takes in a number of parameters, most importantly Φ and $\mathbf{y}$.

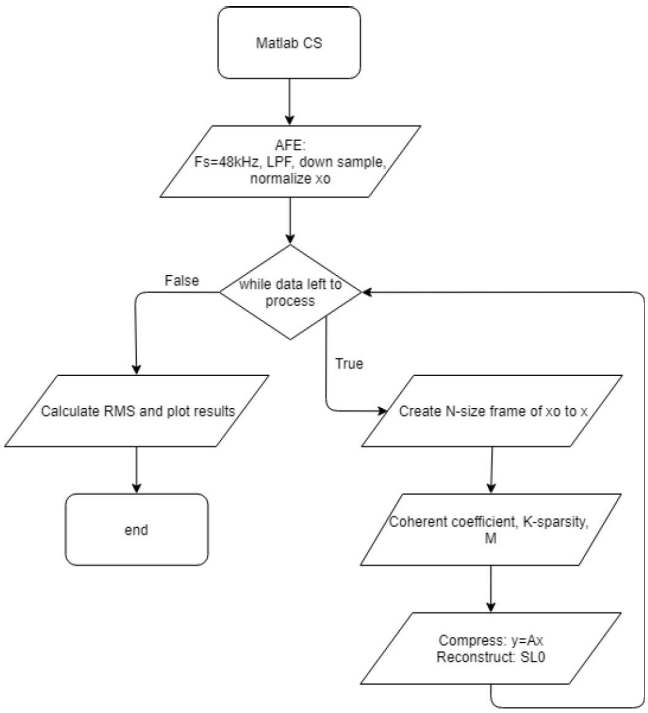
Table 5 Average μ over 100 tests

Ψ	Φ	*Average* μ
DCT	Normal	0.3589
	Bernoulli	0.3488
FFT	Normal	0.2776
	Bernoulli	0.2728

A useful finding is that the coherence coefficient tends to stay relatively constant across multiple tests. With Table 5 in mind, let's set $\mu = 0.35$ for the rest of implementation to

save computation time and cut down on varying factors. Also created is a SL0 function in Python 3.6 which is a close match to what was originally implemented, but with a few changes for efficiency (Lutz, 2013). This code is available online with a GitHub URL in the Conclusion.

Figure 8 Flowchart of Matlab algorithm

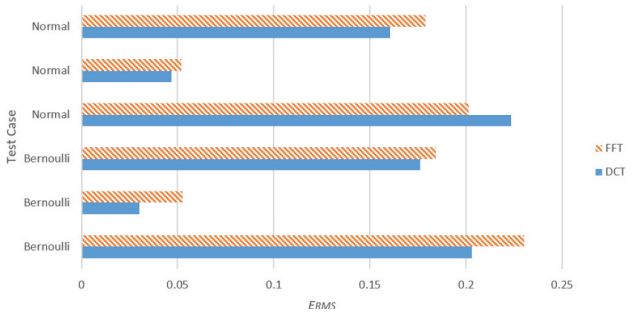


Let's go back to the topic of DCT vs. FFT and using a Binomial (specifically Bernoulli) vs. Normal distribution. DCT is better for memory space since each element is just a real value, whereas the same M-sized $\mathbf{y}$ represented after FFT would take twice as much space in order to save both the real and imaginary parts. Even more with a Normal distribution, there would need to be space for each floating point value vs. just 8-bit word when using a Binomial distribution. With this, one could expect to choose DCT coupled with a Binomial distribution. With an assumption that the system would perform with the results from Table 1, let's show 3 more test cases. Given three music files of varying length and genre, let's run them through a CS system with the combinations of FFT, DCT, Normal, and Bernoulli distribution. Looking at Figure 9, see that DCT usually has a lower error compared to FFT, and Bernoulli tends to be better than a Normal distribution. Confirming that it is safe to use $\Phi =$ Bernoulli and $\Psi =$ DCT in order to save memory space with a manageable amount of error.

5.2 SBC – Raspberry Pi 3

Moving up from the MSP432 is a more powerful platform whose 1.2 GHz ARM Cortex processor is certainly up for the task: Raspberry Pi 3. For the sake of continuity between leaf (Raspberry Pi) and sink (PC) nodes, Anaconda was installed on both machines in order to run Python 3 through Spyder. Libraries include time, math, numpy as np, pandas as pd, pyAudio, scipy fftpack, threading, and msvcrt (Windows only, not usable on the Raspberry Pi).

Figure 9 FFT vs. DCT coupled with Gaussian vs. Bernoulli (see online version for colours)



Since the Raspberry Pi has a Linux OS taking priority, real-time processing cannot be achieved with a traditional triple buffer using ISRs. Instead, multithreading is used to mimic the input, processing, and output buffers. Still running into a buffer overrun, but now just creating more processing threads that will be completed in less time than the sampling period.

Expanding on Algorithm 1, pyAudio is used to sample at F_s filling the input thread (callback function of threading) for a set number of $SAMPLES$, making $NUM\_DB = SAMPLES/N$ to determine the number of processing threads needed. Also created is a thread to stop recording based on user input (msvcrt) or timer. With this, the main while-loop is simply checking if threads are active and re-initialising them to continue processing until the stop thread is triggered. After which, a text file will be created with all of $\mathbf{y}$ and corresponding M values to be transmitted. The below snippet is how the leaf node is set to run.

Equation (2) is used inside the thread tasked with processing the data shown by Algorithm 2. In it is the components of the CS system touched before, where the noise level from the main thread sets a threshold for the function to find K-sparsity of a frame of $\mathbf{x_o}$. Since this system is using a number of data buffers, the resulting observations are saved to a data variable inside a class defined as dataBuffer.

With the idea of sampling at a target $F_s = 8$ kHz, a down sampling function was made similar to its Matlab counterpart. Mixing lessons learned from the MCU and Matlab, the Raspberry Pi "triple buffer" are the input, process, and data threads mentioned above. The user only needs to choose N and F_s and the system will adjust the number of samples the device will take at a time ($SAMPLES$) and number of data buffers needed for processing (NUM_DB) in order to avoid a buffer overrun. A downside to a larger N would mean more samples taken for each frame. Meaning, if one of these frames has a lot of distortion when reconstructing $\hat{\mathbf{x}}$, that could be an entire second of recording with more noise as opposed to an eighth of a second when using a smaller N.

After the leaf node transmits the text file containing $\mathbf{y}$, Algorithm 3 comes into play on how reconstruction is shown. After reading $\mathbf{y}$, it's a simple while-loop so long as there is data to process in order to reconstruct $\hat{\mathbf{x}}$, then take the inverse transform to return it to the time domain.

Algorithm 1 Leaf node – main thread

$\Phi \leftarrow$ PhiN.txt
for $i = 1$ to NUM_DB **do**
  $db \leftarrow$ append with another data buffer vector
  $t \leftarrow$ append with another "processData" thread
end for
$tout \leftarrow$ initialize "outputData" thread
$tstop \leftarrow$ initialize "stopData" thread
$\Psi \leftarrow$ transform of N-sized identity matrix
$calib \leftarrow$ PyAudio stream for a few seconds
$noise \leftarrow detect\_Noise(calib)$
$inputData \leftarrow$ PyAudio continuos stream
for $i = 1$ to NUM_DB **do**
  start processData thread, t[i]
end for
start outputData thread, tout
start stopData thread, tstop
$stop \leftarrow 0$
while not $stop$ **do**
  for $i = 1$ to NUM_DB **do**
    if $t[i]$ is not alive **then**
      $t[i] \leftarrow$ reinitialize t[i] with db[i] vector and started
    end if
  end for
  if $tout$ is not alive **then**
    $tout \leftarrow$ reinitialized and started
  end if
end while
Save y and x to text files

Algorithm 2 Leaf node – processData thread

if (inputBuffer is full) & (threads alive $< NUM\_DB$) **then**
  $f \leftarrow \Psi * input\_buffer$
  $K \leftarrow detect\_K(f)$
  $M \leftarrow \mu^2 * K * \log(N)$
end if
$temp\Phi \leftarrow M$ rows of Φ
$dataBuffer \leftarrow temp\Phi * f$

The code for the sink node is straight forward and will not change. But for Results, the leaf node parameters were set so that the system maximised N while recording less than half a second worth of samples, user sets F_s.

6 Results

TeamViewer is a virtual machine access programs used in order to show 'transmission' from a leaf to sink node. The leaf (Raspberry Pi) would store Φ from the sink (PC) by loading it from TeamViewer. The Pi runs the above code for however long the user needs, and saves the text file containing $\mathbf{y}$ via TeamViewer for the sink to retrieve and reconstruct.

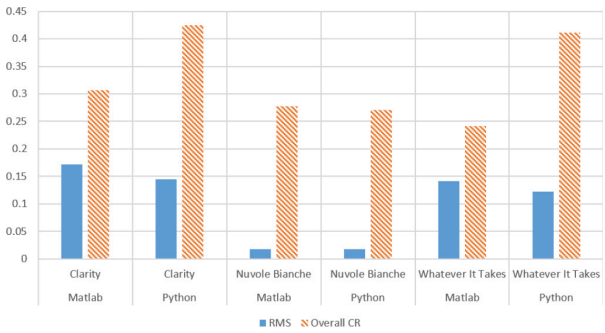
Anytime Matlab is mentioned hereafter, it's compressing and reconstruction done on the same machine (being a PC in this case), and Python will correspond to compression done on the Raspberry Pi and reconstruction done on the same PC as used for Matlab for continuity in machine performance.

Algorithm 3 Sink node – reconstruction

$y \leftarrow$ vector of y vectors from received text file
while have y to process **do**
  $tempY \leftarrow y[i]$ through $M[j]$
  $i \leftarrow i + M[j]$
  $j \leftarrow j + 1$
  $temp\Phi \leftarrow$ M rows of Φ
  $\hat{x} \leftarrow$ appended with resulting $SL0(temp\Phi, tempY)$
end while
$\hat{x} \leftarrow$ inverse transform of $\hat{x}$
$E_{RMS}(x_o, \hat{x})$

To start, let's see how Python compared to Matlab with regards to C_R and E_{RMS} on the 3 music files used before. Figure 10 shows the averaged results of running this CS system for $N = 128, 256, 512,$ and 1024:

Figure 10 CS system comparisons of E_{RMS} and C_R (see online version for colours)

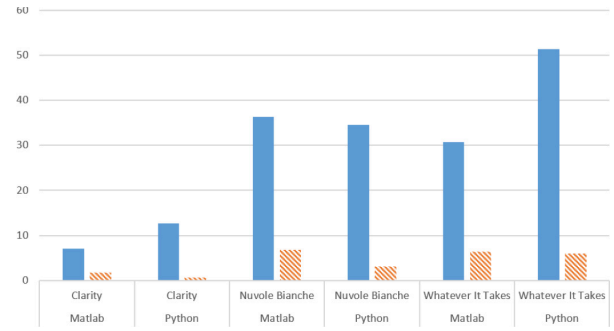


It is observed that Python has similar, if not less, error than Matlab, yet with a larger compression ratio for two of the three files. At least for Python, those two instances were of pop music so a much higher K-sparsity was measured as opposed to the third case, which is classical and more likely to have pauses or clear spikes of notes. A possible difference in why Matlab did not show similar trends in compression ratio is how the two techniques detect the noise floor of the frame. Since $\ln(0) = -\infty$, this is avoided in Python by setting $\ln(0) = 0$. Taking the natural log of all those small values would ideally bring the average value into the negatives, as opposed to what's happening in Python where the average is being brought 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 it full circle to a higher K results in a larger M and less compression taking place.

Although E_{RMS} and C_R of Python are on par with Matlab, timing is a whole other issue. Figure 11 shows the time it takes for Python and Matlab to compress (striped orange) and reconstruct (solid blue) one of the sample audio files. The main reason Python is slower is because it took much longer for the Raspberry Pi to process these music files than the computer. Figure 11 is again showing averages of varying N. With increasing N, the longer it takes for the Raspberry Pi to

perform N-point transformation and matrix multiplication for compression. It also stands to point out that NUM_DB $= 1$ for all these to show single frame processing as in Matlab. Take a grain of salt with Figure 11 since compression is done on different platforms, yet still able to get at least an idea of similarities and differences on performance.

Figure 11 CS system comparisons of compression vs. reconstruction time (see online version for colours)



Since the main objective is to still produce a working CS system that can be used for practical WSNs, CS should be shown in comparison to a popular compression technique: zip. Zip being lossless and CS lossy, the error is in favour of zip. Regardless, let's show how time, C_R, and error (both objectively with E_{RMS} and subjectively) the two match up, and if it validates CS. Two figures help each other paint a picture for time and C_R.

Figure 12 shows the average compression time of CS and zip on the Raspberry Pi for $N = 128, 256, 512,$ and 1024 for the same three audio files used during this process. Meanwhile, Figure 13 shows the Raspberry Pi utilise CS and zip on the three files along with microphone recording in a different way than before. Figure 13 shows the compression time using a bar graph with the left side of the y-axis overlapped with C_R using a line graph with the right side of the y-axis, all done at an optimal $N = 1024$. Figure 12 shows a generalised look of averaging different values for N while Figure 13 is a specific look at a one-run test.

Another clarification is how E_{RMS} correlates to actual quality when the user listens to it - how objective error met subjective. Let's set $C_R = 10\%, 25\%,$ and 50% of the three audio files and running them through the same system. With the calculated E_{RMS}, a few colleagues listened to the reconstructed signal and gave their opinion on how well they compared to the original. When comparing, they were trying to gauge how the quality should be. If the original had poor quality at some point, the reconstructed should match it. It was found that people would prefer the samples that had a $E_{RMS} \approx 0.02$. This is considered the minimum; where the signal is distorted but the user can at least make out the message.

7 Discussion and future work

One problem the user would face with this system is an increasing F_s. As mentioned, there is the option of down sampling function at the beginning of the processing thread to bring the signal down to the target $F_s = 8$ kHz (so long as there is no worry of aliasing), but there will be a point where the sampling frequency is just too fast for the processor to avoid a buffer overrun. This limit was not explicitly looked into. What brought this on is that audio files are at 44.1 kHz, the implemented system cannot do real-time recording at that frequency without hitting a buffer overrun.

Figure 12 Comparing CS to zip on Raspberry Pi 3 (see online version for colours)

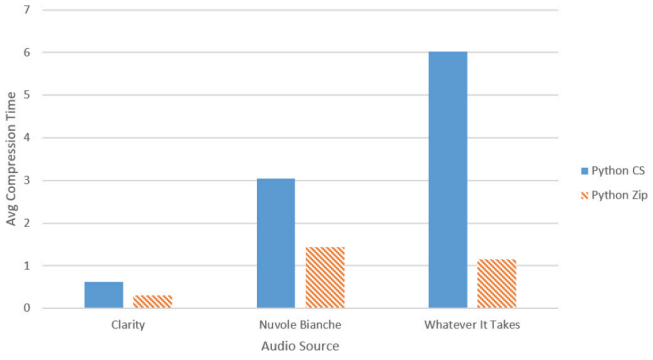
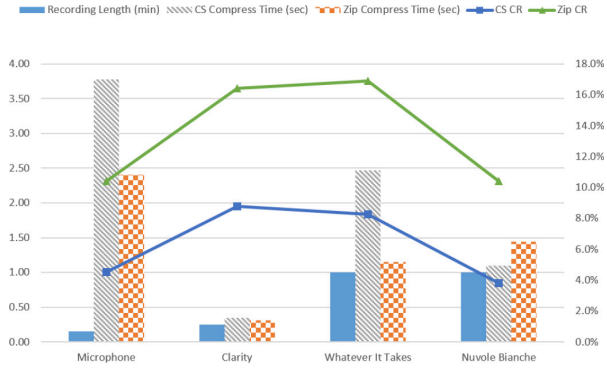


Figure 13 CS and zip comparison of speed and compression ratio (see online version for colours)



When arguing CS vs. zip, it was shown that the user would choose CS to compress more, saving storage space and transmitting time. Although, one could choose zip in order to save time and less error. A factor that his paper did not explore is random sampling in order to actually sample at sub-Nyquist.

As mentioned in the beginning, this is all based on having the original signal sampled at Nyquist rate, and is therefore currently a lossy compression technique. There has been research into what was shown here, but using truly sub-Nyquist signals which are randomly sampled using the same distribution, Φ (Laska et al., 2006). Random sampling would be the next step to advance on research done in this paper. But even then, there is still the slight probability that two consecutive samples are needed to be taken at the Nyquist rate. This brings us back to the original problem that the ADC needs to at least be capable of sampling at Nyquist rate in case of this scenario. The paper mentioned has possible theories on overcoming such scenarios.

The final thing to touch base on is the objective error measurement. Achieving $E_{RMS} = 0.02$ is possible, just at

higher C_R than this system normally uses. Consider this current system a bit too aggressive when compressing the original signal. Since how much it compresses goes back to how M is calculated in equation (8), more research is needed to refine it.

Talking about a minimum viable platform to implement CS in a WSN, could again go back to the idea of using TI's Tiva C series. The Raspberry Pi 3 showed promising capabilities for what is needed, but less could be possible. It depends on the user's needs and balancing price, size, power, and specific application.

8 Conclusion

From start to finish in making a CS system, start with the Nyquist-Shannon theorem of sampling at twice the maximum frequency component of a signal and the issues concerned with applications involving such high frequencies. Compressive sampling is a proposed solution in that it compresses the number of samples needed to represent the signal to be below Nyquist rate. But as of now, conventional sampling is still needed.

The performance of the system depends on the measurement matrix ($\mathbf{A}$) consisted of the distribution matrix (Φ) and transformation matrix (Ψ) that make the original signal sparse, then randomises 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), $\mathbf{x_o}$ (split into N sized frames) can be represented with M amount of observations that result from equation (2). Then, reconstruction requires l_p norm minimisation with a number of techniques available, all with their own pros and cons.

This paper simulated and tested 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 matrices. The MSP432 provided early knowledge of issues to be aware of for scenario specific applications. The Raspberry Pi 3 may run into a problem with processing time depending on the design constraints, but shows much better potential for implementing a CS system for a practical 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 favour of CS.

This paper achieved its objectives in demonstrating a practical implementation of CS for WSN applications (Section 5), exploring the issues that need attention when implementing such a system (Sections 3 and 4), and applying CS to signals that are not perfectly sparse with manageable error (music files and Section 6). This paper was also able to disprove a common assumption of using a Normal/Gaussian distribution with a detailed look at the parameters of $\mathbf{A}$ using the coherence coefficient, μ. It was also successful in implementing CS on a popular and newer platform (Python 3 on the Raspberry Pi 3).

As mentioned, all code, reference documents, and raw data results are made available online for free use in hopes of continued research and development of Compressive Sampling (Ruprecht, 2018).

References

Acevedo, M.F. (2015) *Real-Time Environmental Monitoring: Sensors and Systems*, CRC Press, Boca Raton, FL, USA.

Au, A. W.S., Feng, C., Valaee, S., Reyes, S., Sorour, S., Markowitz, S.N., Gold, D., Gordon, K. and Eizenman, M. (2013) 'Indoor tracking and navigation using received signal strength and compressive sensing on a mobile device', *IEEE Transactions on Mobile Computing*, Vol. 12, No. 10, pp.2050–2062.

Candes, E. and Romberg, J. (2005) *l1-Magic: Recovery of Sparse Signals via Convex Programming*, www.acm.caltech.edu/l1magic/downloads/l1magic.pdf, Vol. 4, p.14.

Candès, E. and Romberg, J. (2007) 'Sparsity and incoherence in compressive sampling', *Inverse Problems*, Vol. 23, pp.969–985.

Candes, E.J. and Wakin, M.B. (2008) 'An introduction to compressive sampling', *IEEE Signal Processing Magazine*, Vol. 25, No. 2, pp.21–30.

Candès, E.J. (2006) 'Compressive sampling', *Proceedings of the International Congress of Mathematicians*, Vol. 3, Madrid, Spain, pp.1433–1452.

Cucchiara, R. (2005) 'Multimedia surveillance systems', *Proceedings of the Third ACM International Workshop on Video Surveillance and Sensor Networks*, ACM, pp.3–10.

Do, T.T., Gan, L., Nguyen, N.H. and Tran, T.D. (2012) 'Fast and efficient compressive sensing using structurally random matrices', *IEEE Transactions on Signal Processing*, Vol. 60, No. 1, pp.139–154.

Donoho, D.L. (2006) 'Compressed sensing', *IEEE Transactions on Information Theory*, Vol. 52, No. 4, pp.1289–1306.

Huang, H., Misra, S., Tang, W., Barani, H. and Al-Azzawi, H. (2013) *Applications of Compressed Sensing in Communications Networks*, arXiv preprint arXiv:1305.3002.

Lam, K.P., Höynck, M., Dong, B., Andrews, B., Chiou, Y.-S., Zhang, R., Benitez, D. and Choi, J. (2009) 'Occupancy detection through an extensive environmental sensor network in an open-plan office building', *IBPSA Building Simulation*, Vol. 145, pp.1452–1459.

Laska, J., Kirolos, S., Massoud, Y., Baraniuk, R., Gilbert, A., Iwen, M. and Strauss, M. (2006) 'Random sampling for analog-to-information conversion of wideband signals', *IEEE Dallas/CAS Workshop on Design, Applications, Integration and Software, 2006*, IEEE, pp.119–122.

Luo, L., Cao, Q., Huang, C., Abdelzaher, T., Stankovic, J.A. and Ward, M. (2007) 'Enviromic: Towards cooperative storage and retrieval in audio sensor networks', *27th International Conference on Distributed Computing Systems, 2007. ICDCS'07*, IEEE, pp.34–34.

Lustig, M., Donoho, D.L., Santos, J.M. and Pauly, J.M. (2008) 'Compressed sensing MRI', *IEEE Signal Processing Magazine*, Vol. 25, No. 2, pp.72–82.

Lutz, M. (2013) *Learning Python: Powerful Object-Oriented Programming*, O'Reilly Media, Inc.

Mahalanobis, A. and Muise, R. (2009) 'Object specific image reconstruction using a compressive sensing architecture for application in surveillance systems', *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 45, No. 3, pp.1167–1180.

Mohimani, G.H., Babaie-Zadeh, M. and Jutten, C. (2007) 'Fast sparse representation based on smoothed â„“0 norm', in Davies, M.E., James, C.J., Abdallah, S.A. and Plumbley, M.D. (Eds.): *Independent Component Analysis and Signal Separation*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp.389–396.

Moler, C. (2010) *Magic Reconstruction: Compressed Sensing*, Mathworks News and Notes, available online at: https://www.mathworks.com/company/newsletters/articles/magic-reconstruction-compressed-sensing.html

Moreira, J. (2014) *What is … a Rip Matrix*, Available online at: https://math.osu.edu/sites/math.osu.edu/files/RIP-matrix.pdf

Needell, D. and Tropp, J. (2009) 'Cosamp: Iterative signal recovery from incomplete and inaccurate samples', *Applied and Computational Harmonic Analysis*, Vol. 26, No. 3, pp.301–321.

Needell, D. and Vershynin, R. (2010) 'Signal recovery from incomplete and inaccurate measurements via regularized orthogonal matching pursuit', *IEEE Journal of Selected Topics in Signal Processing*, Vol. 4, No. 2, pp.310–316.

Nguyen, T.L. and Shin, Y. (2013) 'Deterministic sensing matrices in compressive sensing: a survey', *The Scientific World Journal*, Volume 2013, Article ID 192795, doi: http://dx.doi.org/10.1155/2013/192795

Proakis, J.G. and Manolakis, D.K. (2006) *Digital Signal Processing*, 4th ed., Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Ruprecht, N.A. (2018) https://github.com/NathanRuprecht/EENG5950_MasterThesis

Shah, R.C., Roy, S., Jain, S. and Brunette, W. (2003) 'Data mules: modeling and analysis of a three-tier architecture for sparse sensor networks', *Ad Hoc Networks*, Vol. 1, Nos. 2–3, pp.215–233.

Tropp, J.A. and Gilbert, A.C. (2007) 'Signal recovery from random measurements via orthogonal matching pursuit', *IEEE Transactions on Information Theory*, Vol. 53, No. 12, pp.4655–4666.

Valenzise, G., Gerosa, L., Tagliasacchi, M., Antonacci, F. and Sarti, A. (2007) 'Scream and gunshot detection and localization for audio-surveillance systems', *IEEE Conference on Advanced Video and Signal Based Surveillance, 2007. AVSS 2007*, IEEE, pp.21–26.

Wang, H., Elson, J., Girod, L., Estrin, D. and Yao, K. (2003) 'Target classification and localization in habitat monitoring', *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03)*, Vol.4, IEEE, pp.IV–844.

Wang, Y., Ostermann, J. and Zhang, Y-Q. (2001) *Digital Video Processing and Communications*, Prentice-Hall, New Jersy.

Welch, T., Wright, C. and Morrow, M. (2012) *Real-Time Digital Signal Processing from Matlab to C with the TMS320C6x DSPs*, 2nd ed., CRC Press, Boca Raton, FL, USA.

Wu, M., Tan, L. and Xiong, N. (2016) 'Data prediction, compression, and recovery in clustered wireless sensor networks for environmental monitoring applications', *Information Sciences*, Vol. 329, pp.800–818. Special issue on Discovery Science.

Zhang, P., Hu, Z., Qiu, R.C. and Sadler, B.M. (2009) 'A compressed sensing based ultra-wideband communication system', *2009 IEEE International Conference on Communications*, pp.1–5.