ADC Final Project Write Up: USRP "Airdrop"

Lauren Anfenson, Vivien Chen, Samantha Young, Anusha Datar

December 13, 2019

1 Summary

In this final project we send images wirelessly between two USRP B210 software defined radios. We first use Lempel-Ziv-Welch (LZW) compression, a popular source coding technique, to compress the image file. We then modulate the compressed signal using Quadrature Phase Shift Keying (QPSK), or 4-bit QAM. We transmit this information with one radio, receive it with another radio, and then correct for errors and decode the values in the received signal. This combination of compressing the image and using QPSK increases the data rate and reduces the time needed to send and receive the signal. The code associated with our system is hosted at `https://github.com/anushadatar/USRP-airdrop`.

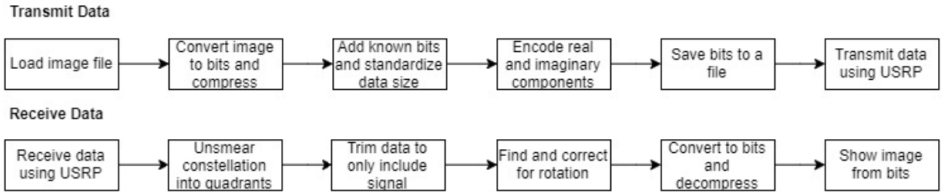

Figure 1: Overall Block Diagram

2 System Parameters

Our system contains two B210 software-defined radios that are approximately 25 inches apart. The system can accommodate longer distances however 25 inches works reliably with no error. We use a carrier frequency of 2.478 GHz and a sampling rate of 2 MHz. We use QPSK to transmit our message signal, so our symbol rate is 2.

We convolve each of our bits with a pulse of width 500. Our timing synchronization strategy involves using a correlation function to determine the location of the the known bits associated with the pulse. We also use a Costas loop to correct for frequency errors.

The data rate of our system is equal to the product of the sampling rate (in samples per second) and the bits per sample. Because we convolve each point (which is made up of 2 bits) by a pulse of width 500, our total data rate is equal to $2 * 10^6 * 2/500 = .8x10^6$, or 800 kilobits per second. We have a 0 percent error rate for each of the signal bits sent once we line up the synchronous points - because we use a compression algorithm on our image file, we must maintain a very low error rate to ensure that we can produce the original image.

3 USRP Data Transmission

3.1 Quadrature Amplitude Modulation (QAM)

In order to increase our data rate we chose to implement quadrature amplitude modulation (QAM). QAM uses amplitude and phase components to encode data. By utilizing both the real and imaginary domain we can double the throughput of our data transmission.

QAM signals are generated from 2 carrier signals that are modulated and combined. One signal is multiplied by a sine and the other by a cosine. This results in a 90 degree phase offset between the two signals.

This was implemented within the USRP hardware.

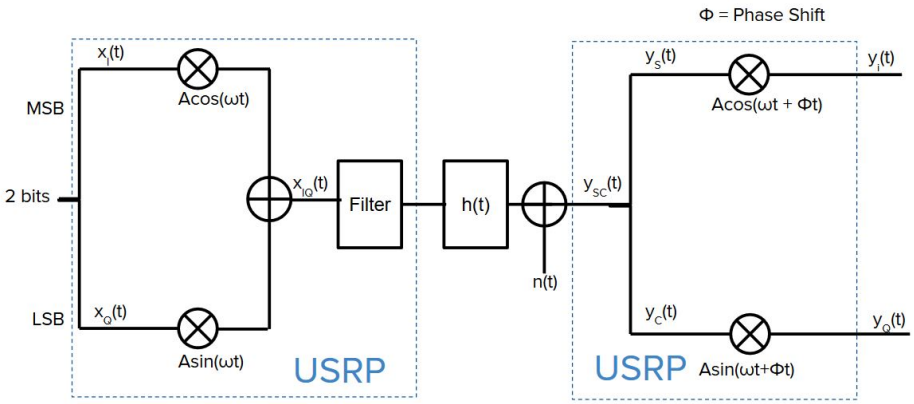

Figure 2: QAM Block Diagram

3.2 Quadrature Phase Shift Keying (QPSK)

To further increase our data rate, we implemented quadrature phase shift keying (QPSK). In QPSK, the carrier frequency undergoes four shifts in phase, so each point represents two bits of data. Thus, QPSK has a symbol rate of 2.

This was also implemented by the USRP hardware.

4 Lempel-Ziv-Welch (LZW) Compression

In an effort to create a robust and efficient way to send images wirelessly, we studied the popular data compression algorithm Lempel-Ziv-Welch (LZW) compression. LZW is a dynamic, lossless compression algorithm commonly used in .zip compression and .gif compression. It achieves up to a 5:1 compression ratio, working best when the data contains frequent repetition. It was discovered in 1977 by Abraham Lempel and Jacob Ziv and further refined in 1984 by Terry Welch.

The LZW algorithm utilizes a code table sequence for its compression. It starts with default entries that are single values. In the standard cases, this would be single bytes from 0 to 255, but the algorithm can be modified to use single bits, numbers, letters, or other appropriate symbols. It continues generating the code table as it compresses, incrementally adding new sequences to the code table. In the standard case, the algorithm adds up to 4096 entries to the code table so that sequences are 12-bit binaries. Because the code table is dynamically created based only on sequences it has seen as it loops through the intended data transmission, the receiver can recreate this code table from the compressed data. The receiver only needs to know what the default single value entries are.

This compression algorithm steps through each character present in the data stream. At each character, it checks whether or not the character has a corresponding code in the code table. If the character sequence does not have a code it then adds the code to the code table and starts a new sequence. If it does have an already existing corresponding code in the code table, the compression algorithm outputs the already existing code.

The below flowchart illustrates the data path of how compression of a data stream works.

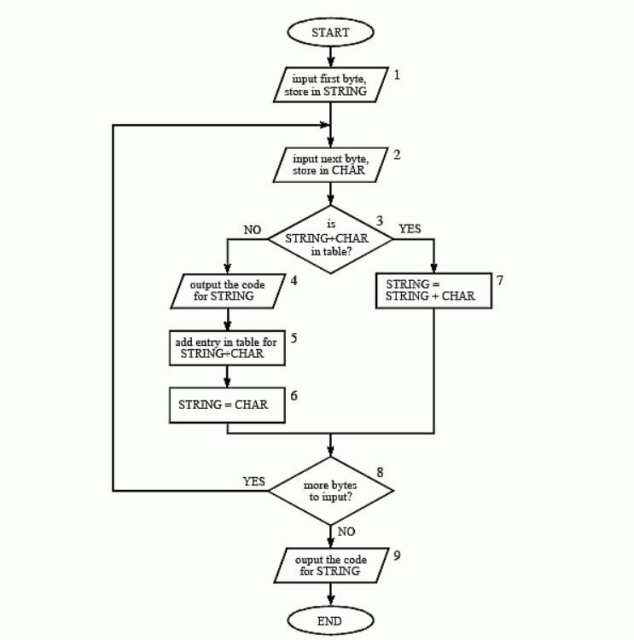

FIGURE 27-7
LZW compression flowchart. The variable, *CHAR*, is a single byte. The variable, *STRING*, is a variable length sequence of bytes. Data are read from the input file (box 1 & 2) as single bytes, and written to the compressed file (box 4) as 12 bit codes. Table 27-3 shows an example of this algorithm.

Figure 3: LZW compression flowchart

In the following example we want to compress the string "banana". With Lempel Ziv compression we can cut the length of the transmitted data in half.

1. Code table contains default entries "b"=0, "a"=1, "n"=2
2. Take in first byte "b"
3. Input next byte "ba"
4. Is "ba" in current code table? → No.
5. Output code for "b" Output = 0
6. Add "ba"=3 to table
7. Make "a" your string
8. Input next byte 'n" string = "an"
9. Is "an" in current code table? → No.
10. Output code for "a" Output = 1
11. Add "an"=4 to table
12. Make "n" your string
13. Input next byte "a" string = "na"
14. Is "na" in current code table? → No.
15. Output code for "n" Output = 2

16. Add “na”=5 to table
17. Make “a” your string
18. Input next byte “n” string = “an”
19. Is “an” in current code table? → Yes.
20. String = “an”
21. Input next byte “a” string = “ana”
22. Is “an” in current code table? → No.
23. Output code for “an” Output = 4
24. Add “ana”=6 to table
25. Make “a” your string
26. Output code for “a” Output = 1

    Final Output = “01241”

At the receiver, we again start with the default entries. The first code has to be one of the default entries, so we translate and output that value first. For each subsequent code, we add unseen sequences into a table like in the compression algorithm. Figure 4 illustrates the data flow of how decompression of the data stream works.

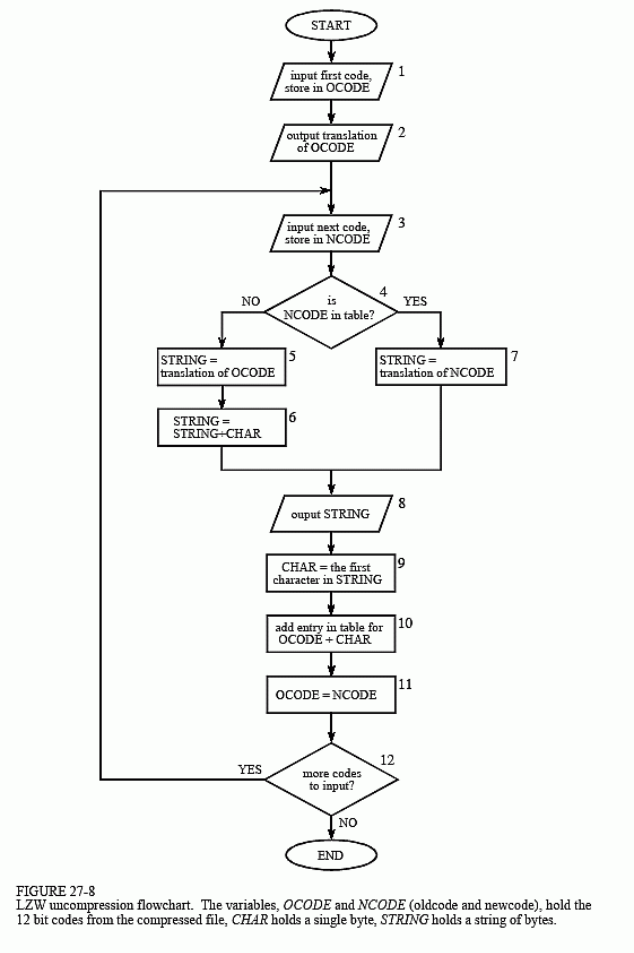

Figure 4: LZW decompression flowchart

In the following example we want to decompress the string "01241".

1. Code table contains default entries "b"=0, "a"=1, "n"=2
2. First code = 0
3. Output "b"
4. Next code = 1
5. Is next code in code table? → Yes.
6. Temporary string = key of 1 = "a"
7. Output "a"
8. Temporary character = "a"
9. Add "ba"=3 to code table
10. Old code = 1

11. New code = 2
12. Is new code in code table? → Yes.
13. Temporary string = key of 2 = “n”
14. Output “n”
15. Temporary character = “n”
16. Add “an”=4 to code table
17. Old code = 2
18. New code = 4
19. Is new code in code table? → Yes.
20. Temporary string = key of 4 = “an”
21. Output “an”
22. Temporary character = “a”
23. Add “na”=5 to code table
24. Old code = 4
25. New code = 1
26. Is new code in code table? → Yes.
27. Temporary string = key of 1 = “a”
28. Output “a”
29. Temporary character = “a”
30. Add “ana”=6 to code table
31. Old code = 1

Final Output: “banana”

4.1 Our Implementation

In our implementation of LZW compression algorithm, we started the code table with only the single bits 0 and 1 instead of the single bytes 0 to 255 to simplify the system and reflect that our digits are binary. From there, we build up the code table as we feed in the bit array of the image. To store our table, we used the MATLAB implementation of a map container, which takes in a string key and a numerical value (or a numerical key and a string value). As a result, we could not directly use binary array, so we instead iterate through integer values in the container and convert the integer value to an n-bit binary value prior to encoding.

For the purposes of our image of a chicken, we used a 13-bit binary to encode each sequence of bits. We then append these sequences of bits to the end of the current encoded output to build up the encoding as the algorithm iterates through each bit of the image.

Even with our slight variation in the LZW algorithm, we were still able to achieve a 2.6:1 compression ratio with our chicken image. Given the constraints of MATLAB functions, much of our code was dedicated to converting between strings, integers, and binary arrays that the functions can handle. These conversions, even when optimized, took about ten seconds for each image.

On the decoding side, we again started the code table with only the single bits 0 and 1 to match the encoder. We read the input of the decoder as 13-bit chunks to reflect the way in which the we encoded the data. We then converted these values from binary values to integers before being looking them up in the map container to match them with the appropriate bit or bit sequence. As it reads and translates each binary chunk, the algorithm updates the code table with sequences that it sees. It then appends the translation to the end of the current decoded output to build up the decoding as the algorithm iterates through the encoded array.

5 Carrier Frequency and Phase Correction

When we transmit data from the USRP through the channel, the received data signal, $y[k]$, will have a frequency offset and phase offset due to clock frequencies, realistically, not being synchronized between the radios.

$$y[k] = he^{j(f\Delta k+\phi)}x[k] + n[k]$$

$f\Delta$ is some frequency offset and ϕ is some phase offset. This terms create a characteristic "smearing" when we receive our 4-QAM encoded data, rather than the neat clustering in each quadrant that we would expect. In order to correct for this smearing in a long data transmission, we used a Costas Loop implementation of a proportional-integral controller to minimize error between our estimate offset and the actual offset.

We use the symbol $\psi[k]$ to represent the phase offsets resulting from both the frequency discrepancies between the USRPs and the angle of the channel over which we're transmitting our signal.

$$\psi[k] = \phi[k] + \angle h$$

The error in the $\psi[k]$ of a signal encoded using QPSK can be computed using the real and imaginary components of each point according to the equation below.

$$e[k] = sign\Re(x[k]) \times \Im(x[k]) + sign\Im(x[k]) \times \Re(x[k])$$

We then feed the error term into a proportional-integral controller and use it to determine the update term, $d[k]$ for our estimate of ψ.

$$d[k] = \beta e[k] + \alpha \sum_{l=0}^{k} e[k]$$

$$\psi[k+1] = \psi[k] + d[k]$$

The updated estimate of psi is then used to correct the next point in our signal, $x[k+1]$, with the equation below.

$$\hat{x}[k] = e^{j\psi[k]}$$

As a block diagram, this set of equations looks something like this.

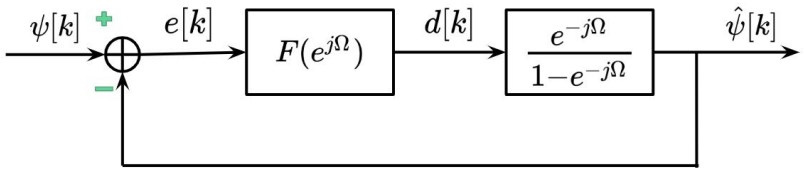

Figure 5: Block diagram of Costas Loop for center frequency offset correction.

We can see the estimated ψ being fed back into the error term of the loop, allowing us to lock on to the corrected phase offset and track it as it drifts over the transmission of our entire signal.

This correction is done for every point in the received signal, resulting in the figure below. We can see the corrected constellation, shown in red, has largely grouped itself into the four quadrants that we expect.

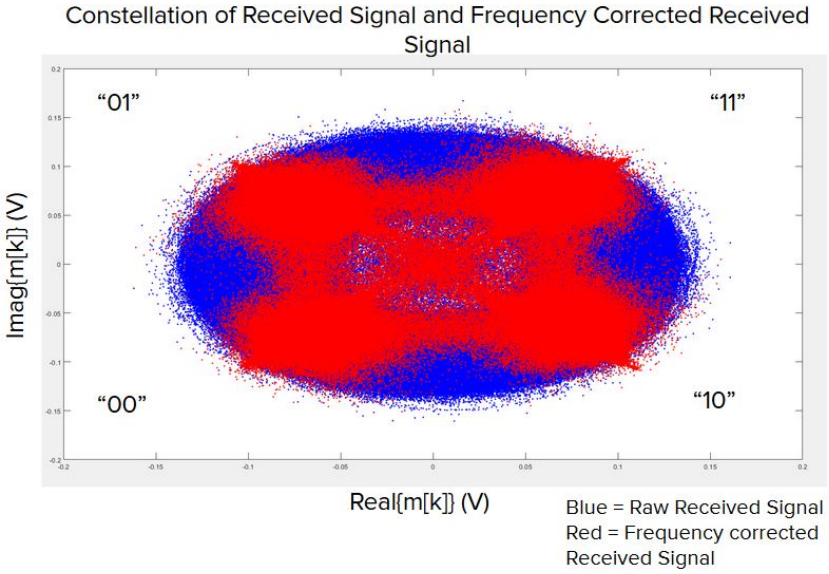

Figure 6: Received constellation is shown in blue, constellation corrected for center frequency offset is shown in red.

There are a large number of bits that still lay on transitions between quadrants, or close to the center of the constellation. This is because the data shown here still includes the samples that are transitions between pulses, as we have not sampled our data yet.

5.1 Other Phase Locking Approaches

Although our primary method of carrier frequency synchronization was using Costas Loop, we also implemented an FFT-based phase lock. We used this method for shorter bitstreams and it successfully carried out carrier frequency synchronization. However, in longer data transmissions, the phase continued to shift and the estimated phase shift and frequency offset were no longer accurate enough to the actual values to properly synchronize the data.

In order to carry out this approach we analyzed the received signal, $y[k]$. We can denote the received signal as:

$$y[k] = he^{j(f\Delta k+\phi)}x[k] + n[k]$$

$f\Delta$ is some frequency offset and ϕ is some phase offset that are dependent on differences between the clocks of the transmitter and of the receiver. The phase

is dependent on k, so as k increase the phase offset increases as well. Using an FFT-based approach, we can estimate this angle and correct for it.

In order to compensate for the phase offset, we take the normalized signal of $y[k]$, $\bar{y}[k]$

$$\bar{y}[k] = \frac{y[k]}{rms\{abs\{y[k]\}\}}]$$

assuming noise is negligible

$$\bar{y}[k] = he^{j(f\Delta k+\phi)}x[k]$$

We can estimate the phase shift and frequency offset by raising $\bar{y}[k]$ to the fourth power and then taking the discrete time Fourier transform. $x[k]^4$ is either -1 or +1.

$$\bar{y}[k]^4 = he^{4j(f\Delta k+\phi)}x[k]^4$$

$$\bar{y}[k]^4 = he^{j(4f\Delta k+4\phi)}$$

$$DTFT\{\bar{y}[k]^4\} = e^{j(4\phi)}\delta(\Omega - 4f\Delta)$$

where the angle of the impulse 4ϕ can be used to estimate ϕ and the location of the impulse $4f\Delta$ can be used to estimate $f\Delta$. In our case, when we raise x[k] to the fourth power we lose some phase information and the above estimated phase offset is not exactly correct. To compensate for this lost data we can say add a phase shift of π to $\bar{y[k]}$.

$$\bar{y}[k]^4 = he^{j(4f\Delta k+4\phi)+\pi}$$

6 Cleaning Data (Trimming, Sampling, and Rotating)

To decode the signal, we need to reliably detect the location of the beginning of the intended transmission. To find the beginning of the transmission in our received signal, we implemented cross correlation. By cross-correlating our received signal with a known signal that we place at the beginning of our transmission, we can find the timestamp with the highest correlation between these two signals. This timestamp corresponds to the time at which the known signal was most closely found in the received data, which contains the signal along with the noise associated with the channel. We trimmed the noisy part of the signal before the intended transmission and after the transmitted data to minimize computation time.

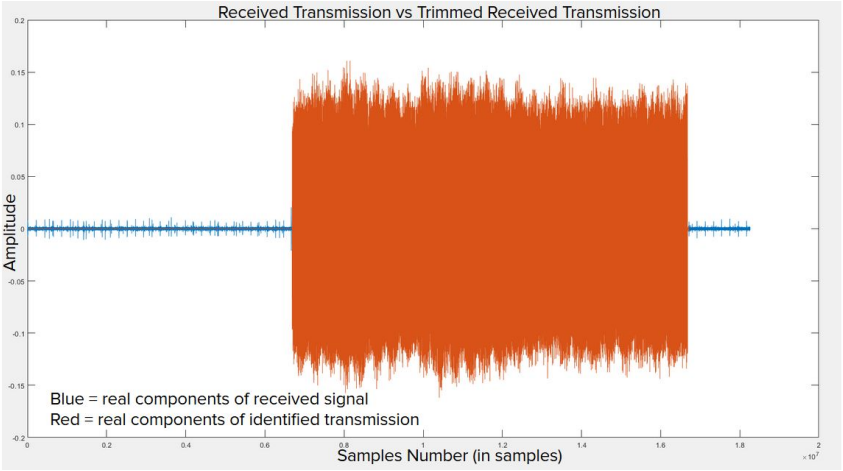

Figure 7: Received Data vs Trimmed data

We chose to use 400 positive bits as our known signal, which translates to 200 symbols with the value 1+1i. Because this data is positive in both the real and imaginary domain, we expect it to be in the first quadrant of the data constellation. However, transmitting data across the channel results in an ambiguous phase shift, so the QPSK data can rotate, causing our known bits to appear in an unexpected quadrant. To correct for this offset, we identify where the known data bits were actually received and then calculate its angle offset from its expected location. We then apply this same phase shift to the rest of the data so that all of the points are in the quadrants that align with the actual value of the bits.

The graph below illustrates the known signal 1+1j as it was received with no phase correction. It falls in quadrant II of the constellation instead of the expected quadrant I. After we apply an offset of $\frac{\pi}{2}$ to the known data, we see it falls in the expected quadrant, quadrant I.

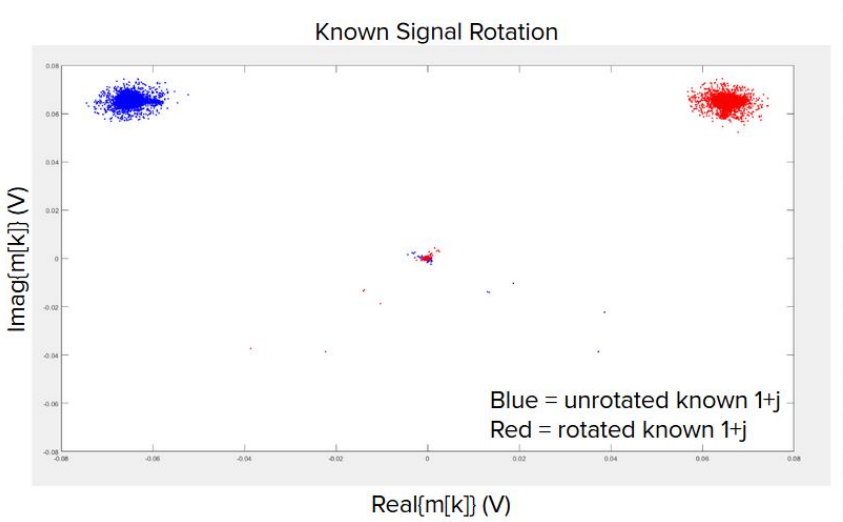

Figure 8: Known Bits being rotated

Once the data has been corrected for the carrier frequency offset and for phase, we sample it. The sampled data falls into four clusters on the real and imaginary axes. Each quadrant of the real and imaginary plane corresponds to two bits.

Complex Transmission	Decoded Data
1+1j	'11'
1-1j	'10'
-1+1j	'01'
-1-1j	'00'

We sampled our received signal at approximately the middle of each pulse. We decoded our sampled signal by examining the sign of each data point. Below is a plot of the received data after correction for both phase shift and frequency offset.

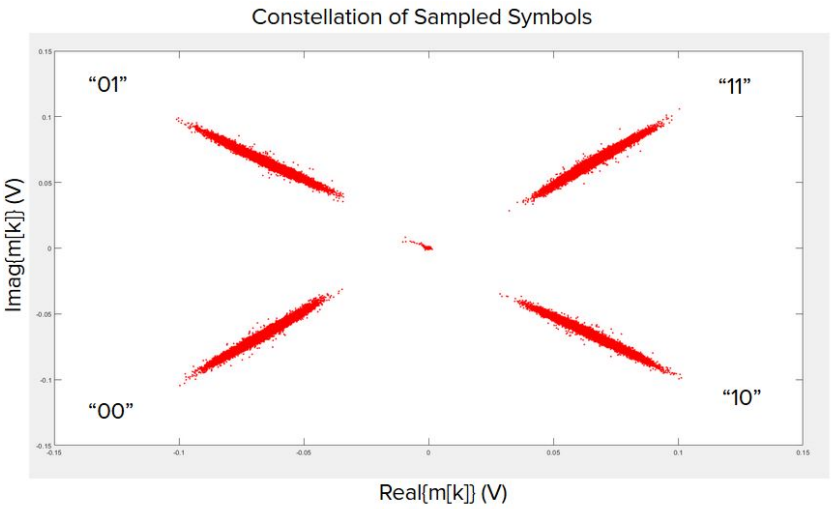

Figure 9: Rotated and sampled data

We then decode these complex symbols into their corresponding pairs of bits, and we have our corrected, sampled, and decoded bitstream ready to be decompressed and displayed as an image!

7 Versatility and Image Size

Every transmission was comprised of 400 known bits, 20 housekeeping bits, the compressed image, and padding zeros to ensure every transmission was 100,000 bits long. The 20 housekeeping bits contain information about the size of the image being sent. The maximum size the housekeeping bits can contain are images 1023x1023 or 2^10x$2x^10$ bits. By having the option to pad smaller image transmissions with zeros the we increased the versatility of the program by allowing for various image sizes while still keeping a standardized transmission length. This standardized transmission length was useful when trimming and data and reducing processing time.

8 Reflection + Suggestions for Improvement

We built the software associated with our project in a systematic way. We wrote modular functions for each of the components in our block diagram, and we made sure that we knew what the intermediate data should resemble at each point in the

algorithm. Then, we could incrementally work on different steps of the overall process. Taking this approach allowed us to make sure that all of the parts seamlessly integrated, and it also allowed us to develop each portion of the project separately so that we could make sure that it functioned without additional variables or uncertainty. We also worked on the paper and presentation as we developed each part of the project so that we could document as we worked and maintain references for ourselves and each other. We could have improved our process by blocking out the entire system diagram earlier in the development of individual parts of the process and doing more full-group share-outs and tech talks about what each person was working on so that everyone got all of the content from all parts of the project ex. compression, data processing, carrier synchronization.

There are a few ways through which we could improve our project. One function we struggled with was trimming our data to isolate the timestamps that contain the transmitted signal. This was not always robust, and when there was a lot of noise, it would fail. By using a more sophisticated technique to determine exactly where our data begins, we could make the method by which we process the received data more robust. Furthermore, we could also improve the performance of our LZW compression algorithm by implementing it another programming language, like Python. Our MATLAB implementation is high-overhead as we cannot stop our code table at twelve bits and it requires multiple datatype conversions which have high time complexity.

We can extend our project by adding features or improving existing ones. We could support transmitting and receiving color images by processing entire bytes of data with the RGB values of each pixel instead of just using the grayscale image. Alternatively, we could have supported sending four-color images by assigning each symbol in our QPSK constellation to a single color when processing the image itself. We also could experiment with increasing our robustness to error either by adding error correction mechanisms or using a more flexible compression strategy - tolerating a larger degree of error would allow us to increase our data rate.

8.1 Work Distribution

Back when we were young and still thought that we needed to implement QPSK before giving our transmission to the USRPs, Lauren worked on implementing QPSK encoding. Once we got better, she mainly worked on implementing center frequency offset correction, using both the FFT and Costas Loop approaches. When that was

working reliably, she also helped to integrate that correction with our trimming, rotating, and sampling functions. She spent an inordinate amount of time debugging receiver-side MATLAB code and chanting "chicken.... chicken..." at her computer until an image appeared.

Anusha mainly focused on the code for preparing and transmitting the data and creating the transmit data packet, mapping/building/integrating the pieces of the larger software architecture of the entire system, debugging the overall system, convincing her computer to communicate with the USRP devices (and helping Vivien set hers up to at the last possible minute), and documentation in the code and in the paper. She also often forgot to close MATLAB when transmitting and receiving data and dealt with merge conflicts when no one else wanted to.

Samantha worked on the system architecture block diagram, helping identify the processes and functions needed to make a complete transmitter and receiver of compressed images. She worked on Lempel-Ziv compression research and algorithms with Vivien. She also wrote many of the data processing functions including the trimming algorithms (several), finding and implementing phase shift and rotations, and converting photos to bitstreams. She worked with Lauren on learning and understanding the QAM carrier frequency synchronization, both the DTFT approach and the Costas Loop approach. She also worked on keeping the documentation and the presentation up to date throughout the project for easy reference. The Fall 2019 ADC class witnessed Sam's reemergence into the acting sphere (since her school's 4th grade production of Annie where she starred as "Man 3") in the award winning skit, "Airdrop - but not Bluetooth".

Vivien worked mainly on the Lempel-Ziv compression algorithm and wrestling with MATLAB in that regard. She also worked on packing data for transmission and unpacking on the receiver end, as well as debugging and formatting code in various phases. At one point she attempted to install USRP to her computer for a demo between two computers, but unfortunately the transmission did not work. Afterwards, she spent quite some time making sick graphics for the slideshow and working on the report.

References

[1] Smith, Stephen W. "Chapter 27: Data Compression / LZW Compression." *The Scientist and Engineer's Guide to Digital Signal Processing*. Poway, CA: California Technical Publishing, 1997. *DSP Guide*. Web. 9 December Year 2019. http://www2.cs.duke.edu/csed/curious/compression/lzw.html.

[2] Bhat, Sooraj. "LZW Data Compression." Wired, Condé Nast, 19 March 2002, http://www2.cs.duke.edu/csed/curious/compression/lzw.html.