Free Space Optics

Matt Kubas, *Member, IEEE* and Ethan Morton

***Abstract*—Transmitting data over distance is of paramount importance to today's society. To this end, we researched and developed ways to transmit information using visual light broadcast through a computer display and received through a camera. We tested how the camera worked, and how it could be used as a receiver for this type of transmission. We developed a set of algorithms to encode and decode information into a transmittable format. Finally, we tested the effectiveness of our systems and found it could transmit at high speeds (500 kbps) with low error ($< 1\%$). Furthermore, an analytical analysis of two error detection and correction methods was done: Hamming and convolution encoding.**

I. Background

EFFICIENT and effective data transmission is a crucial aspect of nearly every technology on the planet. As a result, many different methods for transmitting data have been developed over the years. Arguably, the first of these even dates back to before electricity was used, when people would flash lights over long distances in some predefined code. Then, the use of wires and electricity took over. In tandem, and soon after, wireless methods of communication were developed, such as radio, cellular networks, wifi, etc. As data transmission has become more necessary, the methods for this transmission have become more and more sophisticated, allowing for higher bandwidths and lower latencies.

One such method is called *Free Space Optics (FSO)*. This method takes notes from the first wireless transmissions, where lights were flashed over long distances, over *free space*. FSO works by transmitting data via directed light waves arranged in a pattern specific to the receiver-transmitter system, making it extremely dependent on the hardware in use. FSO is also a new field, allowing more freedom in its initial design, but having the drawback of no standardized protocols and hardware. As a result, the specific encoding/decoding scheme varies per implementation.

The specific attributes of each component in the system varies greatly depending on the physical system. For example, a transmission between satellites in LEO (Low Earth Orbit) will have very little environmental noise compared to a transmission between two similarly distant objects on the Earth's surface. A camera directly in front of a computer will have almost no environmental noise or spacial cross-talk due to the proximity. A transmission that travels kilometers or more will have much more significant beam divergence and, thus, much higher crosstalk. Using specialized transmitters and receivers, many people have also created new multiplexing and error correction techniques, which take advantage of quantum properties of light, such as polarization. This specialized hardware also allows for taking advantage of many different oscillatory modes of light, which can be configured to increase bandwidth or resiliency.

II. Methods

For the purposes of this module, we are using a significantly more simple setup than the companies and labs which are producing gigabytes per second transfer speeds. We have the Teledyne FLIR Blackfly® S USB 3.1 Color Camera (BFS-U3-16S2C- CS) camera. It can achieve frame capture rates of 800 fps (dependent on the resolution), and has two color depths it can record, 8-bit or 12-bit (MATLAB allows for the selection of 16-bits). Throughout this project, we mainly used the 8-bit color space, with the full resolution (1440x1080), which resulted in around 226 fps when unlimited. To precisely control the frame rate, we adjusted the camera's exposure time, which would force it to extend the time the sensor was exposed and, thus, the time per frame. For example, if we had a target frame rate of 120 fps, then:

$$T_{exposure} = \frac{1}{fps} = \frac{1}{120} \approx 0.008\bar{3}s = 8333\mu s \tag{1}$$

So, we would set the exposure time to 8333 μs.

Since the encoding/decoding schemas are not standardized, this allows for open investigation into different encoding schemes and camera correction algorithms in an FSO system. In order to develop parts of this system, we created several setups to determine camera characteristics and test transmission systems. Our main goal with these setups is to learn more about the camera and system and to aid our efforts in transmitting data.

A. Camera Characteristics

During this module, we experimented with several different aspects of the camera, determining what characteristics we could change to increase our control over it and optimize it for our use case.

Hot Pixels: We want to test to see if any of the camera sensors were "hot." This would mean there was some hardware defect in the camera, causing the photodiode to not correctly activate/deactivate or the electrical system to produce false counts. To accomplish this, we set up the camera to use a very high gain (45dB), left the lens cover on, and put the camera facing a wall. This would prevent most of the light from hitting the sensor, meaning that whatever counts we received would be due entirely to electrical noise in the system. To detect these pixels, we first perform the experiment mentioned above; then we calculate the mean and standard deviation for each pixel across all frames, as well as *all* pixels across all frames. We label any pixel whose mean is greater than the overall mean plus five standard deviations as a hot pixel, as shown in the equation below:

$$\mu_{pixel} > \mu_{all} + 5 \cdot \sigma_{all} \tag{2}$$

Color Ranges: We also want to test if the camera could detect one of the three colors better than the others. If certain colors are detected more easily and/or more frequently, then those colors will have different thresholds when attempting to detect binary data. To test this, we first generated figures in MATLAB, which were pure red/green/blue. We then used one of the lab computers in Urbauer 320E to display that image and took videos from the camera at different gains. This allows us to see a gain when the camera readings become saturated. By comparing the points where each color becomes saturated, we can tell how sensitive the camera is to each color.

Light Leakage: A central concern for FSO transmission is the crosstalk between transmission lanes. Over long distances, this crosstalk can heavily influence the resulting signal. We decided to determine the effect of this crosstalk on our system. To do this, we generated a figure in MATLAB which has one white side and one black side. If the crosstalk is a significant factor, then the edge of the white and black will be smooth, and there will be a discernible runoff on the black side from the leaking white light. If there is not much crosstalk, then both sides will be relatively homogeneous throughout, with no discernible dropoff.

B. Transmission System

There are two main sections to the transmission system: the physical setup, including the display and the camera position; the algorithmic analysis of the received data from the camera, and processing it into data. The signal processing will be discussed later in section III.

Physical System: To effectively transmit, and compare transmissions, a consistent environment is needed. To achieve this, we tried to maintain a consistent physical setup across multiple variations of data collection, including keeping the camera the same distance from the screen, using the same resolution, and keeping outside noise to a minimum (ie. having the message take up 90% of the screen). Despite that, there was still an assortment of physical setups were used in the process of this module. In general, the process for setting up the system consisted of:

- Finding an appropriate screen to display the video/image on.
- Pointing the camera at the display, and load up the preview in $saveCameraVideo$.
- Lining the camera up and adjusting focus, gain, etc to get a good picture quality.

The specific physical system pairs included combinations of displays. These consisted of either Ethan's laptop, Matt's laptop, or a display in Urbauer 208, 215, 216, or 320E. The camera was sometimes set slightly farther away from the screen, but in general, it was attempted to keep the camera roughly 0.25-0.3 meters from the screen. This would allow a slightly wider viewing angle and reduce the fisheye effects on the fringes of the lens.

III. Signal Processing

The basic idea of our signal processing approach is to divide the broadcasting display into discrete sectors and then assign a bit (or set of bits) to each sector. The sector size relative to the display size can be adjusted to allow for different bitrates. The framerate of the video also can be adjusted. To calculate the theoretical bitrate for any given configuration, we use this formula:

$$\begin{aligned} bpse &= bits\_per\_sector \\ spf &= sectors\_per\_frame \\ fps &= frames\_per\_second \\ R_{bps} &= bpse \cdot spf \cdot fps \end{aligned} \tag{3}$$

So for example, if we are running a perfect system where every pixel is a sector, at 60 fps, with 3 bits per sector, we get:

$$R_{bps} = (3)\cdot(1440\cdot1080)\cdot(60) \approx 280Mbps = 35MB/s \tag{4}$$

Throughout the course of the module, multiple processing configurations were attempted, with each progressively smaller sector size bringing new challenges (ie. higher density of sectors per frame). For testing purposes, we stayed at 30 fps throughout. Higher framerate tests are discussed in section IV. The low frame rate tests are discussed below.

1-sector Display: The simplest technique was simply flashing the screen white or black. This allows for the overall intensity of the screen to be taken as the single bit on the screen. However, we want to be sure that the average we are taking is over only the signal portion of the video. So, we force the first few frames of every video to be completely white and use a simple difference-based bounds-checking algorithm to find exactly where the edges of the display being recorded are and take the average inside those bounds.

We defined white as "1" and black as "0". However, even this simple case requires the design of some sort of handshake or synchronization logic. Without the "go ahead" signal from a handshake to start processing, the system would have no idea whether there is a signal being broadcast or not. To implement this handshake, we simply defined a handshake signal as "1010101010", which translated in the physical system to the screen flashing white and black.

After that, the screen would flash white or black corresponding to the bit it is showing. With this approach, if there are multiple ones or zeros in a row, it would be unclear how many bits were in that period. By finding the frame-time between each screen flash during the handshake, we were able to determine how long each bit was, and from there, determine how many bits were in a given stretch of constant color.

At this stage, the message length is predefined for simplicities sake, so we did not transmit any metadata along with the message.

The theoretical max bitrate for this system would be:

$$\begin{aligned} R_{bps} &= bpse \cdot spf \cdot fps \\ R_{bps} &= (1)\cdot(1)\cdot(fps) \\ R_{bps} &= fps \end{aligned} \tag{5}$$

We see that the bitrate is exactly equal to the frame rate. This should be pretty obvious, since we are using the entire

display to send a single bit. It is also obvious that this is a highly inefficient way of solving this problem. To make this faster, we add dimensions and improve the detector to be as fine-grain as possible.

Black and White Grid: The next simplest technique is dividing the screen space into sectors, each representing a single bit. To encode this, it is very easy. We simply designate a hyperparameter called "grid_size" which represents the width and height. For this stage of development, we made every frame a square (1080x1080) to make things simpler (though we would come to regret this). One such grid can be seen in fig. 1.

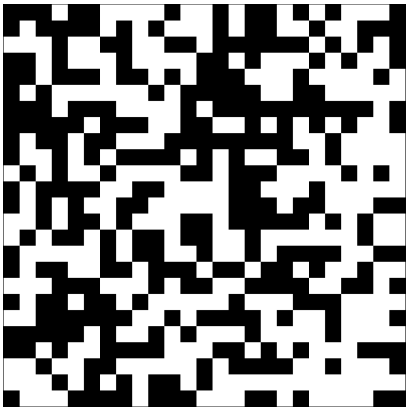

Fig. 1: Black and white grid, at 25x25 grid size

To decode this, it becomes more complicated. We can no longer simply average the entire screen (within the detected bounds) to get the bit value. Now, we must construct the grid that we know was encoded into the other display. We simplified things by keeping the grid size constant and known across both the encoder and decoder so that we can use it as a hyperparameter later. We used a grid size of 25 by default (meaning the grid is 25x25, with 625 distinct sectors). Again, the fact it is square will come back to haunt us...

Throughout this process, we assumed no skew from the camera to the display, meaning the camera would never be at an angle to the display. This was to simplify the math. If we assume no skew angle, we can simply calculate the edges of the display being recorded, and then cut that space into our 25x25 grid. We then just take the average intensity of each sector to get the bit value (white="1",black="0").

This iteration used the same handshake system as the simple one. The screen is held white, then flashed a few times, and then the message starts. This time, the message frames contained the binary grid rather than the single color as previously.

The maximum theoretical bitrate is calculated as:

$$\begin{aligned} R_{bps} &= bpse \cdot spf \cdot fps \\ R_{bps} &= (1) \cdot (grid\_size^2) \cdot (fps) \\ R_{bps} &= grid\_size^2 \cdot fps \end{aligned} \tag{6}$$

For our test case of 30 fps and a grid size of 25, this gives us:

$$\begin{aligned} R_{bps} &= (25)^2 \cdot (30) \\ R_{bps} &= 18750 \end{aligned} \tag{7}$$

So our max bitrate would be nearly 19 kbps.

A New Hope: Color: The next easiest improvement to make was to have each color channel transmit its bits separately. This is called multiplexing, in our case, each sector represented three consecutive bits. By introducing color channels as a usable dimension in the signal space, we again can multiplicatively increase our bits per frame.

The most basic addition of color just treats each color channel like we already were treating the white/black binary choice. We use the same binary decision (on/off) but for each color channel. Each sector then becomes a superposition of the individual color states inside of it. For example "101" would be a pink/purple color. These colors can be seen in fig. 2.

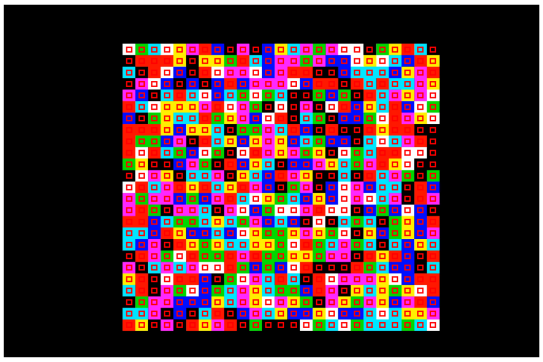

Fig. 2: Bit grid including color, 25x25

Other additions were also made to the handshake system and to the general frame processing. Instead of providing the handshake and then just sending the message to the wind, we want to add synchronization and some amount of real communication. To do this, we do a few things:

- add a frame that contains metadata such as message length, so we can send messages of arbitrary length without changing code on either side
- use a white/black checkerboard prefix to the message so that the decoder can detect the edges of the grid
- add clocks to each frame to allow synchronization between the encoder and decoder; and allow for easier detection of frame swaps

With these features added, the message transmission becomes significantly more consistent. Synchronization allows the decoder to detect when a frame is displayed more consistently than before. The clocks were placed in the top right and bottom left corners. These clocks never show any bits, but instead flip between black and white. The clocks were placed in opposite corners to allow for the detection of fully formed frames, ie. both clock frames must fully transition from black to white, or vice versa, for the system to decode the frame.

The checkerboard allows us to use MATLAB's *detectCheckerboardPoints* method to detect the grid points, which we can interpolate to generate our grid of sectors. However, this introduced a slowdown to our code. Where previously we had been taking simple differences on the

checkerboard and white frames to see where our bounds were, we were doing complicated edge detection, which is not a linear time operation. As a result, the initial calibration step is very long at high grid sizes such as 75, 89, or 101.

This MATLAB function also introduced a problem with the way the grid points were detected. The function required a checkerboard with one even and one odd side to work. It could work reasonably well even with odd size sides, and square shapes, but it would return the grid points of the checkerboard in a seemingly random order. As a result, we had to modify the code to accommodate the one odd, one even, grid size. From that point on our grid dimensions were *grid_size* by *grid_size+1*.

The equation for the max theoretical bitrate will now also change, due to the change in grid dimensions and the use of two sectors as clocks. The max theoretical bitrate, including the color dimensions and protocol improvements, is:

$$\begin{aligned} R_{bps} &= bpse \cdot [(grid\_size \cdot (grid\_size + 1)) - 2] \cdot fps \\ R_{bps} &= (3) \cdot ((25) \cdot (25 + 1) - 2) \cdot 30 \\ R_{bps} &= 58320 \end{aligned} \tag{8}$$

The theoretical max is roughly 58 kbps. So by including color in our 25x25 grid, we can achieve the goal of 56 kbps at a frame rate of just 30 fps.

The Light Side: Another key aspect of processing the signal is dealing with the decreasing brightness farther from the center. To combat this, we can scale the image up to an analogous brightness, or scale down our thresholds to take into account the reduced dynamic range of the fringe pixels. We implemented a brightening scheme to brighten the pixels to a reasonably uniform brightness. Then, we use the regular constant thresholds across the entire frame rather than implementing a radial threshold function. This adds another hyperparameter, unfortunately, because the relative brightness scaling requires some scalar value to use as the scaling multiplier.

IV. RESULTS

To condense results and not include information needlessly, such as bitrates and tests at lower or older versions of our system, we will only show information pertinent to the camera characteristics and the best version of the encoder/decoder combo.

A. Camera Characteristics

Hot Pixels: Following the procedure described in section II, we first tested the camera for "hot pixels," or pixels that produced high counts even in an environment with little to no light. The results of this test can be seen in figs. 3 and 4. In fig. 3, we see that this specific pixel had a significantly higher mean than the pixels around it. In fig. 4, we see why. The red channel of the pixel reads 255 counts (which is the max for an 8-bit color depth) for every single frame. The other color channels also showed elevated readings despite the environment being as sterile as possible. From this, we gather that there are pixels in the camera's sensor which are unusable for calculations. For our camera, which we named *Jerry*, we found that there were 56, 38, and 43, hot pixels for R G and B respectively.

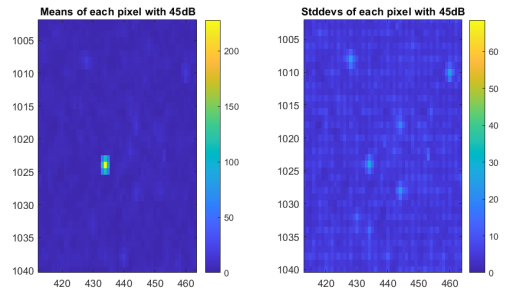

Fig. 3: A zoomed-in graph of the mean and standard deviation of a set of pixels over time (100 frames at 226 fps).

Light Leakage: After following the processes outlined in section II, we found no significant difference in error due to light leakage. Looking at figs. 5 and 6, we can see that although there is some minor difference, this is more due to the fact that it is much easier to saturate the camera completely than to desaturate it completely, meaning darker colors such as black will always have a higher error rate. The large line of errors in the middle is the portion of the figure in which MATLAB smoothed the transition between the two colors, making it gray instead of black or white. As a result, this portion of the figure is as wrong as wrong can be.

B. Signal Processing

Using the checkerboard point finder, pinpointing each square, and offsetting that point such that we were not gathering any data from crosstalk between pixels, we were able to get error rates below 1%.

For our grid that was 26×25, the decoder mistakenly decoded 291 bits of 50,000 bits that were sent, approximately 0.582% error rate. The 51×50 grid size performed similarly; this time, the decoder mistakenly guessed 3,359 bits of 500,000 bits that were sent, approximately 0.672%. Next, the grid size was 76×75 with 3,969 incorrectly decoded bits out of 500,000 bits sent, an error of 0.7938%. We then attempted grid sizes of 89×88 and 101×102 with error rates of roughly $35\% - 45\%$, and thus we did not use these grid sizes due to the error rate.

We attempted all of the above at 60 frames per second, but the displayed did not have a fast enough pixel drop time, which means that the white pixels were unable to drop fully to black before the next frame started displaying (see fig. 7). This means that many times the clock bits would not trigger, not recording a frame, and thus would create error rates greater than 65%.

V. FUTURE CONSIDERATIONS/EXTENSIONS

A key improvement to our process would have been working on a working prototype first, and then characterizing the hardware. Since the goal of this module was always to build a working data transmission system, it would've been better to

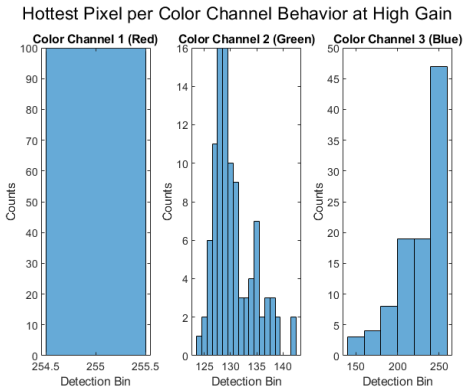

Fig. 4: A histogram of the counts of a single hot pixel over time.

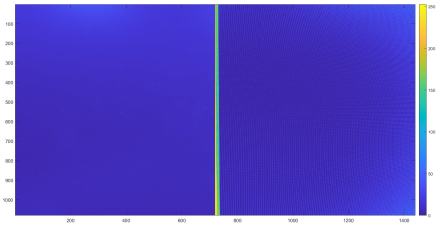

Fig. 5: A graph of the error recorded by the camera facing a black and white screen as described in section II

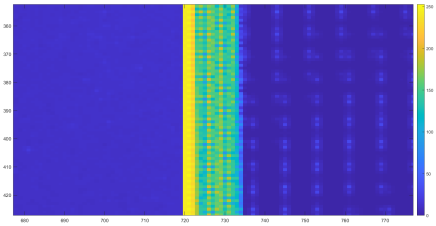

Fig. 6: A zoomed-in version of the graph of the error recorded by the camera facing a black and white screen as described in section II

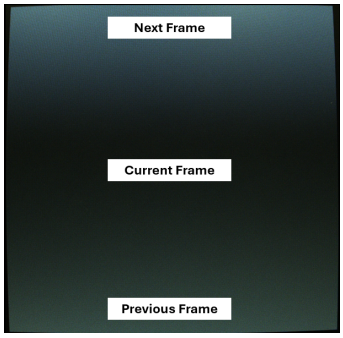

Fig. 7: Example of the issue of poor pixel response times of the monitor that was used. At the top is a new frame, the middle is the current frame, and at the bottom is the old frame.

spend the first few weeks simply building out the infrastructure necessary to transmit in the simplest case, before complicating things with camera specifications, hardware quirks, etc.

Another aspect that we wished we had time for was exploring a continuous approach to data transmission. As we saw during the presentations, every group landed on the same strategy, consisting of a grid with flashing clocks to indicate new frames. We began work too late on the idea of transmitting data in the frequency domain rather than in the physical domain. Instead of breaking the display into spatial sectors, we would break the 2-dimensional frequency spectrum into sectors, and assign each a bit value. Just like other forms of communication that rely on frequency domain transmission, we would choose some window to alter the shape and smooth out the edges, and then compute a 2-dimensional inverse Fourier transform to obtain the spatial frame we wish to send. This would still involve sectioning off portions of the screen as a clock to define when frames were produced, but instead of transmitting a grid, we would transmit whatever shape the inverse Fourier transform produced. This way, every bit is transferred using the entire screen space, rather than only a small section, which could lead to an increase in robustness.

One consideration to consider when using Fourier analysis is that we can only ever transmit a *real* signal. Therefore, we must structure our frequency space such that the complex function is even, implying that the real part must be even and the imaginary part must be odd. This means that the available space in our frequency spectrum will decrease by 4 times, leaving us only one-quarter of the display size to work with to ensure we obtain a real signal. However, we believe this would be offset by the ability to stack many more bits into the frequency dimensions than in the spatial dimensions. By choosing windows effectively, we can overlap different modulations and increase reliability, even in challenging environments.

Another benefit to this transmission technique is the increase in the multiplexing multiplier. Assuming a perfect transmission, information could be sent through each color channel, as shown by many of the groups, *as well as* through both the real and imaginary parts of the frequency space. As long as the real part is even and the imaginary part is odd, we should be able to send a real signal and recover the real and imaginary parts effectively, giving us six times the bits per frame rather than three, as with the bare color channel approach.

Unfortunately, due to time restraints and our own plans for what to do and when to do it, we were not able to test this approach. If it seems worthwhile and achievable, this may be an option as a final extension to the class.

VI. Error Detection and Correction (Addendum)

For the final extension, we chose to test two Forward Error Correction (FEC) techniques, Hamming Codes and Convolutional Codes, that could be used with our FSO system. These were benchmarked with two specific types of errors in mind: burst errors and random noise errors. In an FSO system, burst errors can arise when portions of the screen are physically blocked, creating gaps in the received bit string. Random noise errors result from the environmental noise of the medium(s)

that the light passes through. We aimed to determine effective ways to implement these FEC techniques in the system and simulate their performance with synthetic benchmarks. This was accomplished by changing the physical location of the bits written to the frame to reflect which bits were more valuable to the decoder.

A. Error Types

In order to allow for greater freedom of the group to work asynchronously, we developed synthetic benchmarks to simulate the real environment. This system's two main types of error are burst errors due to a physical object blocking the light's path and random noise errors due to changing environmental conditions and the asynchronous nature of the transmitter-receiver system. In particular, the primary source of errors was the random noise injected into the system from the display refreshing mid-capture. This results in unusable frames and lower overall reliability.

Burst errors were not an issue in the original experiment due to the controlled nature of the system, but are still of concern. In an actual transmission system, there will undoubtedly be times when an object blocks information between stations. As a result, it is still of interest to study these errors and integrate methods for correcting them.

B. Simulations

To maintain relevancy to the FSO system, the simulations must target certain conditions that *actually appear* in the physical system. As a result, the simulations were created with the specific error scenarios from the previous section in mind. The general process for both simulations involved the following steps:

1) Generate a random bitstring
2) Generate a frame from the bitstring using the encoder pipeline
3) Inject error into the system
4) Generate the output bitstring using the decoder pipeline
5) Compare the two bitstrings for accuracy

Error was injected into the system through two main methods: flipping the bits of a specific section of the frame and randomly flipping bits across the frame. By flipping bits randomly, random noise errors are simulated. To better represent the physical system, the probability of a bit flip was set to increase radially from the center of the frame, capturing the intensity falloff in the frame and accounting for errors due to fisheye. The simulation iterated through an increasing number of bit flips, up to a final total of 50% loss. Sections of the frame were flipped to simulate a foreign object entering the field of view. This was done with increasingly large sections to generate a graph of accuracy versus message corruption, again up to 50% loss.

C. Hamming Codes

A simple implementation of a *block code* FEC is the Hamming encoding. This encoding creates *parity* bits, thus this adds more bits to the transmitted message, decreasing the true message bitrate. The most straightforward Hamming encoding is the [7,4] encoding. This means that there are 7 total bits per block (B), 4 of which are the message (M) and 3 of which are the parity bits (P). To calculate these values, we must first choose the number of parity bits we will be using and can find the number of message bits and block size from eq. (9).

$$\begin{aligned} B &= 2^P - 1 \\ M &= 2^P - P - 1 \end{aligned} \tag{9}$$

A single block can take the form below, but these vary based on the encoding process.

m_3	m_2	m_1	m_0	p_2	p_1	p_0

The parity bits (labeled with p) hold information on whether the sum of message bits is even or odd. The bits included in each parity bit can vary between encoding strategies, but in this case, the parity bits are encoded as seen in eq. (10). This encoding strategy can detect up to two errors happening simultaneously in the same block but can only correct the error if a single error occurs.

$$\begin{aligned} p_2 &= m_3 \oplus m_2 \oplus m_1 \\ p_1 &= m_3 \oplus m_2 \oplus m_0 \\ p_0 &= m_3 \oplus m_1 \oplus m_0 \end{aligned} \tag{10}$$

In MATLAB, this encoding was implemented using matrix transformations, where the message was encoded using a generator matrix (G) and decoded using a parity check matrix (H). To learn more about these, visit Wikipedia and University of New Brunswick resources.

The Hamming encoding is good at recovering random bit flips that do not happen within the same block but is not resilient to burst errors. A workaround for the lack of burst error protection is to randomize the order of bits. For the hardware-based system, a per-frame scrambler was developed to generate a random pattern to shuffle the bits around (shared with the decoder), such that the parity bits are set in the center columns with the message bits using all other columns. This was done to mitigate errors for parity bits since it was noticed that the original system had many more errors in the corners than in the center. This was not tested in this case but can be easily implemented.

D. Convolutional Coding

Convolutional coding is a method that does not directly transmit the bitstring. Instead, it transmits a series of symbols representing the bitstring, and each symbol depends on multiple nearby bits in the bitstring. The basic idea is pictured in fig. 8.

The system acts as a state machine, and the encoded bits are generated by feeding the input bitstring to the state machine and calculating the symbol from the state at each bit along the way. This creates layered dependencies between bits that can help to correct strings of errors. The process to decode the received symbols involves tracing multiple paths backward and choosing the path with the least error and, thus, the highest probability of correctness. During the decoding process, "error" is measured by taking a symbol,

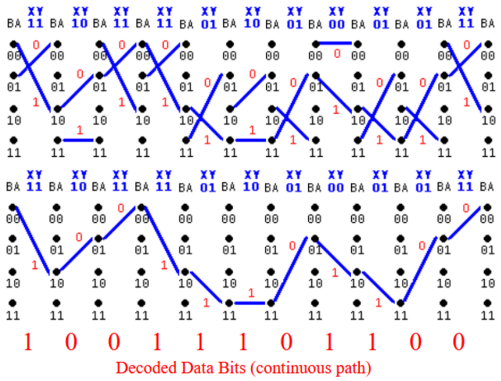

Fig. 8: A graphic created by the ECE department of the University of New Brunswick to visualize convolutional encoding. The resulting path (bottom) is the final output decoded from the received set of symbols (top).

calculating the expected previous possible symbols, and taking the difference. The path with the least error is the path with the least breaks. In fig. 8, the resulting path can be seen in the original received message with no breaks, making it the path with the least error. In practical settings, it is common that every path has breaks.

E. Interleaving

Interleaving is a method to increase robustness regardless of encoding scheme, by reorganizing the physical order of the bits to spread errors out and enforce the condition of Additive Gaussian White Noise (AWGN), which most statistical error models rely on as an assumption. The specific pattern of reorganization can differ between implementations. For the purposes of this report, two structures were used. For the Hamming codes, the parity bits were stacked in the middle of the frame, with the data bits left to the outside fringes. This increases the chance that all parity bits arrive safely at their destination, and any random errors resulting from fringe effects can be corrected. The convolutional codes used a different system that spread the bits across radial layers of the frame. This ensures that there is a few bits which make it through and can act as an anchor for the probabilistic decoding scheme by forcing fewer breaks on the correct path.

F. Results

1) Hamming: The Hamming encoder and decoder system were analyzed for error based on random bit flips since two consecutive bit flips would cause unrecoverable errors, thus burst error analysis is not the best factor for analysis. Four different numbers of parity bits were analyzed ($P = 3, 4, 5, 6$), and the error to varying numbers of bit flips is plotted in fig. 9. From the figure, it can be concluded that the smaller the block size ($P = 3$), the greater the accuracy since there is a higher ratio of message bits to parity bits, which means that there will be a lower error rate. The ratio of message bits to parity bits for each trace (Message/Parity) is 4/3, 11/4, 26/5, 57/6. This means fewer errors can occur in systems with more parity bits than with lower ones since there are fewer redundant bits. The traces are also mostly linear, ie, as more bit flips occur, more errors happen, but at a significantly lower rate than with no encoding. This means that the Hamming encoding can help eliminate much of our per-frame bit flips of the real system, allowing for

2) Convolutional: The accuracy level for varying levels of message corruption is displayed in figs. 10 to 13

We see that for the random noise, convolutional coding was able to keep a near-perfect accuracy when over 10% of the message was randomly flipped. This makes sense because the convolutional encoding and decoding process introduces dependencies between nearby bits that act as pseudo-parity bits, each giving some information about what the bit should've been. We also see that interleaving makes no difference whatsoever for random noise. This makes sense because interleaving is designed to force random noise, so it will not affect a system with error already dominated by random noise.

For burst errors, interleaving had a clear and significant impact. By introducing interleaving, the accuracy was greatly increased, going from a linear increase in error with message corruption to a more strange but significantly smaller increase. The shape of fig. 13 is irregular, and in our testing, we were unable to determine the exact cause of this irregularity. There may be a programming error we have missed that is causing this behavior. However, we still observe the behavior we would expect, with error increasing minimally as corruption increases.

VII. Conclusion

Overall, we enjoyed using high-performance hardware and building out a system of our own. We wish we had more time to flush out other aspects of the goals and implement alternate solutions that provide a different set of challenges. Our system works well when tuned to the right parameters for the given environment it is used in, and successfully transmits relatively large amounts of data (50-500 kbps) with little error ($< 1\%$). This was achieved through a combination of brightening, edge/bounds detection, and transmission protocol design, which takes notes from modern-day transmission protocols. In the future, we will focus more on developing and building than obtaining metrics without proper motivation, allowing us to complete our goals more effectively.

The chance to add an addendum to this project allowed us to revisit some aspects of the original experiment, as well as extend to aspects that were unexplored. The error correction systems introduced produced significantly lower error rates and had higher accuracy overall than the previous systems we developed, and the simulation environment allowed for a more thorough examination of the effects of these systems.

If this project were to be further extended, a more comprehensive overview of methods, both original and new, would be produced, allowing for a deeper and more meaningful comparison between techniques and systems.

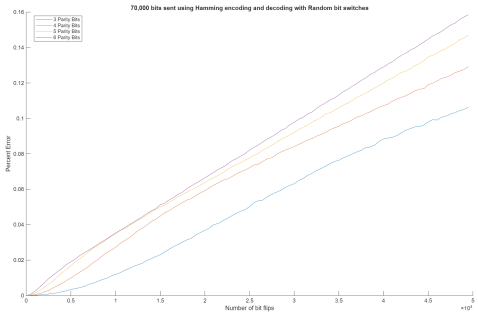

Fig. 9: Hamming encoding and decoding process with error correcting plot of Error rate (%) vs. number of bit flips

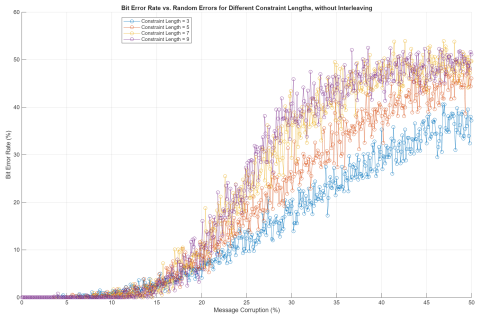

Fig. 10: Graph of the accuracy of convolutional coding with varying random noise from 0 to 50% frame corruption, without interleaving.

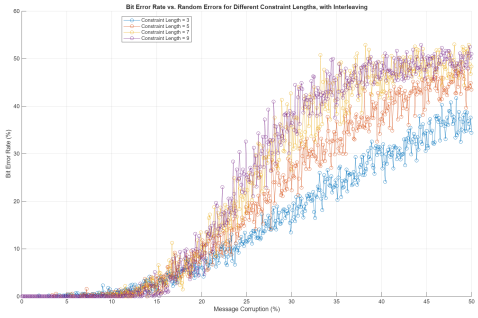

Fig. 11: Graph of the accuracy of convolutional coding with varying random noise from 0 to 50% frame corruption, with interleaving.

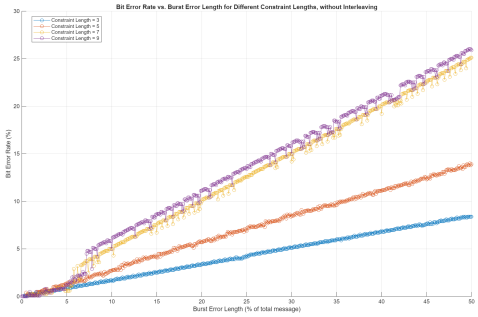

Fig. 12: Graph of the accuracy of convolutional coding with varying burst errors from 0 to 50% frame corruption, without interleaving.

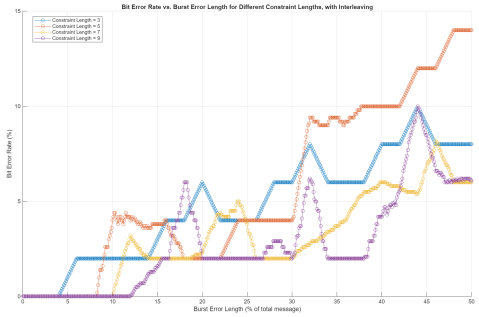

Fig. 13: Graph of the accuracy of convolutional coding with varying burst errors from 0 to 50% frame corruption, with interleaving.