Bitwise Transmission Using an LED and Photodiode

N. Pun[1]

[1] *Department of Physics and Astronomy, University of British Columbia, Vancouver, British Columbia, V6T1Z4, Canada*

(Dated: January 10, 2020)

Three methods of encoding and transmitting a sequence of bits are analyzed in this paper, all of which pursue the same goal of transmitting the most possible bits in a given time frame of 30 seconds. The three methods are compared with one another and one is determined to be theoretically the fastest and most efficient. This paper examines an experiment to test the theoretically most efficient method and the flaws in implementing such a method in reality. Although the tests show that this method can accurately transmit data, they also reveal issues specific to this method. Due to uncovering of these issues, arguments are made to propose which method should then be the new theoretically most efficient method.

The ability to remotely send and receive messages and data has long been an essential tool in the field of telecommunications, and while data rates are becoming increasingly fast, they are still limited by both the physical method by which they are transferred and the implementation of the software that encodes and decodes them. The advantage of using light to transmit data is that there is now minimal latency due to physical limitations, and so the majority of the data rate is then determined by the implementation of the software. Such is the case for our experimental procedure.

Our method of experimentation involves using an LED to transmit an encoded array of bits which can be detected by a photodiode. The photodiode converts the optical signal into an electrical signal, which can be decoded back into the original array of bits. Since we are using light to transmit the data, we have effectively minimized the time the data takes to travel from the transmitting part of the setup to the receiving part of the setup. The exact schematic diagram is as shown in FIG. 1a.

We use an Arduino Leonardo to encode a sequence of bits such that it can be transmitted using the LED. While

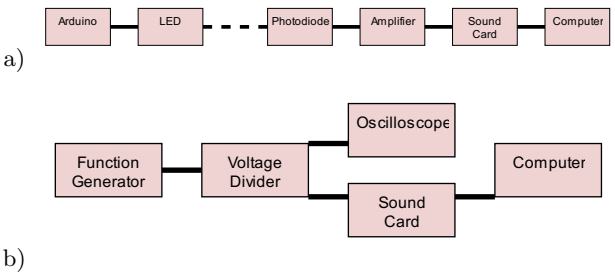


FIG. 1. a) An Arduino uses an LED to emit an optical signal that can be detected and converted into an electrical signal by a photodiode. The signal is then enhanced by an amplifier and converted to a digital signal by a sound card, which is then analyzed by a computer. b) The circuit used to calibrate the scaling factor of the data in order to scale the axes into physically realizable units. This is how the sound card readings are turned into real voltages.

there are many possible encoding methods, we explore three in our experiments. At first, we assumed that it would be logical to represent a 1 as an LED state of ON while a 0 corresponded to OFF. However, the issue with this is that at distances larger than a meter the intensity of the ambient light exceeds that of the LED, which means that there exists no discernible difference between a 1 and a 0. Instead, all three of our methods involve turning the LED on and off a certain frequencies to represent certain numbers, as our analysis of the code will be able to extract frequencies even when the noise is larger than the signal.

Before exploring the three methods, there are important parameters that needed to be established. Firstly, the LED must pulse on every period as opposed to changing states on every period. If the LED were to behave as per the latter option, the frequency calculated by Python's *signal.periodogram* would be half of what was originally sent. Of course, another way around this is to simply double the original frequency. *signal.periodogram* is Python's function for applying a Fourier transform on a set of data. We applied the first method when blinking our LED and found that the shortest pulse we could use while still maintaining reasonable accuracy and consistency in the pulse timing and height is 0.5 ms.

Another issue that required resolving is the synchronization of the transmitter and receiver. Our solution was to have the receiver already collecting data when turning on the transmitter. During data processing we can determine when the signal starts in one of two ways. If the signal is in fact distinguishable from the noise, we can manually remove the data points that come before and after the signal data. Otherwise, we can send a "start message" before the actual signal to signify and synchronize the start of the data. We can now move on to the three methods.

Method 1, which I will refer to as M1, follows closely from the original idea discussed earlier; a certain frequency $f0$ represents a 0 while another frequency $f1$ represents a 1. Then for every 0 in the sequence of bits the LED blinks at frequency $f0$ for a certain period of time t, and similarly for every 1 it blinks at frequency $f1$.

Method 2 (M2) stems from the idea that in M1 only one bit is being sent for every period of time t. We can instead send multiple bits at the same time t, which prompts two new definitions. I will refer to a t-bin as a period of time during which one or more bits are being sent simultaneously, and a b-bin as the section of the total sequence of bits that is being sent during that t-bin. To achieve the simultaneous transmission of bits, we interpret a b-bin as a binary number and convert it to its decimal representation. We then blink the LED at a certain frequency based on the decimal representation and as calculated by EQN. 1,

$$f = f_{min} + \frac{2x+1}{2} \cdot w \tag{1}$$

where f is the frequency that will be sent, f_{min} is the minimum frequency that we will use, x is the decimal number calculated from the b-bin, and w is the width in hertz of what I will refer to as an f-bin. I define an f-bin to be the range of frequencies that correspond to a certain number, which is determined by the position of the f-bin on the frequency domain. If, for example, we set $f_{min} = 500$ and $w = 10$, all f-bins then have a width of 10 Hz and the frequency range 500 - 510 Hz corresponds to f-bin 0, 510 - 520 Hz corresponds to fbin 1, etc.

The f-bin width w is used in order to account for uncertainties throughout the experiment. For example, if our desire is to send a frequency of 515 Hz corresponding to a 2 but the Fourier transform resulted in a spike at 518 Hz, this would still be within f-bin 1. We can then vary w such that it is at least twice the uncertainty in the frequency measurement.

As an example of M2, if the 10th b-bin contained the bits 1101, the Arduino would convert this to 13 and the LED would blink at frequency $500 + 13 \times \frac{3(10)}{2} = 635$ Hz throughout the duration of the 10th t-bin. 635 Hz is within f-bin 12 who's range is 630 - 640 Hz. Since this is the 13th f-bin, we then convert 13 into binary to retrieve the original b-bin 1101.

Method 3 (M3) further increases the number of bits that can be sent per t-bin and arises from a subtle yet critical problem found in M2. Suppose we have n number of bits per b-bin. We would then need 2^n number of f-bins for the 2^n possible decimal numbers generated from the b-bin. While this is acceptable when n is small, the number of f-bins grow exponentially as n increases. This both slows down the compile time of our code and forces us to use very large frequencies if the number in a given b-bin is large. This is a problem as frequencies larger than 2000 Hz will cause the pulses of the LED to overlap due to the pulses being 0.5 ms long. As such, the LED will be in a constant state of ON.

To eliminate the need for 2^n f-bins, M3 maps each individual bit in a b-bin to its own f-bin and sends a linear combination of frequencies. For example, suppose there are 4 bits per b-bin, we can set 100-110 Hz as f-bin 0, 110-120 Hz as f-bin 1, 120-130 Hz as f-bin 2, and 130-140 Hz as f-bin 3. Now if a b-bin contains the sequence 1101, M3 would simultaneously blink at frequencies 105, 115, and 135 Hz such that a Fourier transform of the received data would generate a spike in f-bin 0, 1, and 3. Since f-bin 0, 1, and 3 contain spikes and 2 doesn't, we can then use this information to say that the original b-bin contained 1101. In theory, using M3's implementation, we can send n bits using n f-bins for any given t-bin.

An issue that exists in both M2 and M3 is the existence of what I refer to as pseudo spikes in the Fourier transform. Pseudo spikes are the the spikes that occur at integer multiples of the input frequency and are artifacts of the imperfections of equipment, setup, and data processing. Pseudo spikes can cause a single input frequency to generate spikes in multiple f-bins. For M2, the solution to this problem is to only use the first f-bin that contains a spike, as this will be the primary spike for all subsequent pseudo spikes. For M3, the solution is to introduce a new variable $f_{max} = 2 \times f_{min}$. We then ensure that all f-bins are contained within the range $[f_{min}, f_{max})$. We do so based on the knowledge that no pseudo spikes can exist in such a range, as the first possible pseudo spike can only exist at f_{max} which is not included in the range $[f_{min}, f_{max})$. This gives us an upper bound of 1000 Hz for f_{min} as f_{max} would then be 2000 Hz, the upper limit for any frequency as discussed earlier. Of course, if we do not use the full range $[f_{min}, f_{max})$, then f_{min} can be larger than 1000 Hz, but this would simply decrease the number of possible f-bins that can be fit within the range.

The final issue that must be accounted for in the implementation of M1, M2, and M3 is the electromagnetic field generated by the lab equipment. This field causes interference in the data in a periodic pattern at 60 Hz, and multiples of 60 Hz as well due to its pseudo spikes. Fortunately, pseudo spikes decrease in amplitude as their frequencies increase and so we can simply set f_{min} to be larger than the largest visible 60 Hz pseudo spike.

Since M3 has the potential to send the most data in a given amount of time, we use M3 in our experiment. It was important that we didn't set f_{min} too high or else many LED pulses would overlap from the linear combination of multiple frequencies. This could cause error in the Fourier transform. Upon implementation of M3, it became apparent the f-bins can be spaced out to both use low frequencies and avoid the 60 Hz interference. The centers of the f-bins were calculated according to EQN. 2,

$$f = f_{min} \cdot (1 + \frac{n+0.5}{N}) \tag{2}$$

where f is the frequency that will be sent as well as the center of the f-bin, f_{min} is the minimum frequency that we will use, n corresponds to the n'th f-bin as well as the n'th bit in a b-bin, and N is the total number of f-bins as well as the size of the b-bin.

Upon receiving the data, the processing code in Python must calibrate the data and scale both the vertical and horizontal axis in order for us to use real units. This is because the sound card and Audacity does not record the data using SI units. To find the scaling factors, we use the circuit in FIG. 1b. To vertically scale, we divide the amplitude measured by the oscilloscope by that which is measured by Audacity and result in a scale of of 1/300000, or 3.33×10^{-6} to turn the vertical axis into units of volts. To horizontally scale, we divide the total time of a sample of data as measured from Audacity bu the total number of data points and result in a horizontal scale of 8.055/355227, or 2.27×10^{-5} to turn the horizontal axis into units of seconds.

The processing code in Python must also account for both uncertainty and noise when extracting the original bit sequence. To do so, we introduce two new parameters. u is the uncertainty in the frequency location of the spikes and consequently half the width of our f-bins. We must ensure that u is large enough such that the whole width of each spike is contained within their respective f-bins. VT is the voltage threshold on above which a spike is considered to be a 1 and below as a 0. Of course, this requires voltages to be extracted from each f-bin. These two parameters can be calibrated/varied during the analysis of the data.

Using our implementation of M3, we begin by sending 60 bits. We limit ourselves to 30 s total to send the data. We use 15 t-bins each of length 2 s and each sending a b-bin that is of size 4 bits. We set our $f_{min} = 40$ Hz, which automatically sets our frequency range to be 40 - 80 Hz and our four frequencies to be sent as 45, 55, 65, and 75 Hz. For this test the LED was positioned approximately 0.1 m from the photodiode such that the recording of voltage over time can visibly show the start and end of the data transmission, which allows us to manually remove the data points before and after the desired section. The recording of voltage on the time domain after clipping the ends is as shown in FIG. 2a. In it there appears a large noise anomaly, however the effects on the data will be minimized after taking the Fourier transform and adjusting u and VT accordingly.

The data processing algorithm then divides the data into the 15 t-bins of 2 s each and applies a Fourier transform to each individual t-bin. The voltage recording of t-bin 0 is shown in FIG. 2b. The b-bin being sent in t-bin 0 contained the bit sequence 1001 and thus the frequencies being linearly combined are 45 Hz and 75 Hz. Upon enlarging the graph, it becomes clear that these frequencies are low enough such that there is minimal overlap in the pulses.

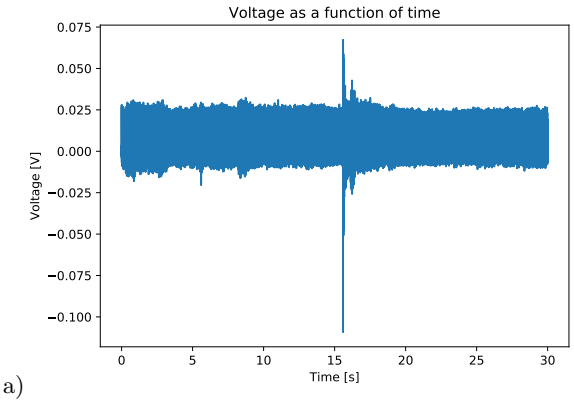
The Fourier transform of t-bin 0 produces the power spectrum shown in FIG. 3. There is a large spike at 60 Hz due to the electromagnetic interference discussed earlier. By setting $u = 1.7$ Hz, this spike is not contained within any of the four f-bins:

f-bin 0: 43.3 - 46.7 Hz
f-bin 1: 53.3 - 56.7 Hz
f-bin 2: 63.3 - 66.7 Hz
f-bin 3: 73.3 - 76.7 Hz

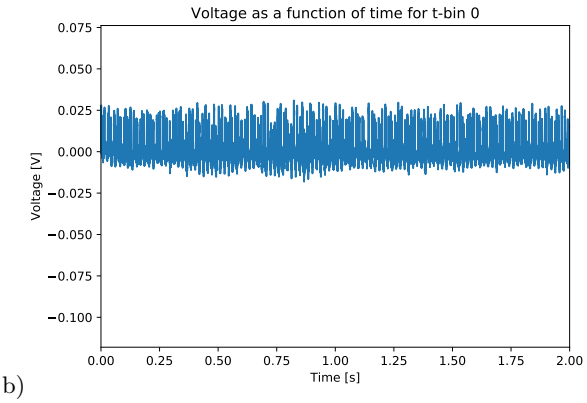
Then by integrating over each f-bin using a Reimann Sum method, we retrieve the root mean square of the voltage for each f-bin, which we can use to calculate the voltages measured from the photodiode using EQN 3,

$$V = \frac{\sqrt{V_{rms}}}{A \cdot \Delta f} \tag{3}$$

where V is the voltage measured from the photodiode, V_{rms} is the root mean square of the voltage calculated from the integration of the power spectrum, A is the scaling factor of the amplifier, and Δf is the frequency bandwidth over which the integral was taken. In our experiment we used an amplifier with a known scaling factor of $A = 2208$ [1].



a)



b)

FIG. 2. a) Voltage measurements as taken by the receiver. This is the data that will be processed and decoded back into the original bit sequence. b) The same as graph (a) except enlarged to fill the graph with only t-bin 0. A single Fourier transform will be applied on this set of data.

For our test we found that $VT = 4.578 \times 10^{-7}$ V allows us to correctly distinguish between every 1 and 0. For t-bin 0, the following voltages were calculated:

f-bin 0: 5.412×10^{-7} V
f-bin 1: 1.187×10^{-7} V
f-bin 2: 9.129×10^{-8} V
f-bin 3: 5.473×10^{-7} V

From the voltage data, it is clear that f-bins 0 and 3 are greater than VT while f-bins 1 and 2 are not. This tells the processing algorithm that b-bin 0 contained 1001.

This process is repeated for all 15 t-bins, and the results of using $u = 1.7$ Hz and $VT = 4.578 \times 10^{-7}$ V were 60/60 correctly identified bits.

While our tests prove that M3 is a possible way to encode data, there exists a subtle issue that limits the amount of data that can be sent. This issue resides in the capabilities of Python's *signal.periodogram* function. If there does not exist enough input data points for *signal.periodogram*, the resulting power spectrum will have very few data points. When we increase the number of bits per b-bin and decrease the length of each t-bin, the total number of data points per t-bin decreases and thereby decreasing the number of data points that is used when applying *signal.periodogram*. This lowers the resolution of the power spectrum and increases the uncertainty in the calculation of V_{rms}. When we use 10 bits per b-bin and 1 s per t-bin, *signal.periodogram* resulted in only 3 data points in the power spectrum on the range 40-80 Hz. This cannot be used to accurately calculate the ten V_{rms} for each of the ten required f-bins as a single spike will overlap multiple f-bins.

Due to this issue, we now believe that M2 is our most efficient method of sending the data and should be explored in future experiments. This is because M2 only analyzes one spike in the power spectrum. When the power spectrum can then theoretically reduce its resolution to a single data point, the location of which can then be used to determine which f-bin it is in, which of course can then be used to calculate the original binary bit sequence.

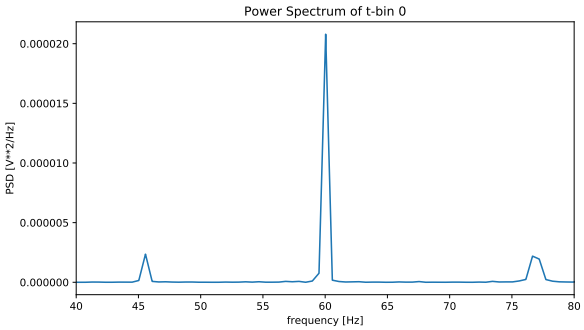


FIG. 3. The power spectrum resulting from taking the Fourier transform of t-bin 0. The four f-bins are at 43.3 - 46.7 Hz, 53.3 - 56.7 Hz, 63.3 - 66.7 Hz, and 73.3 - 76.7 Hz.

The author acknowledges helpful discussions with J. Folk and A. de St Croix. Experiments at UBC were supported by PHYS 309.

[1] N. Pun, Johnson Noise and Its Existence in Resistors , 1 (2020).