**Michelle Thompson, W5NYV**
5379 Carmel Knolls, San Diego, CA 92130 | **w5nyv@arrl.net**

# Opulent Voice

## Error correction and interleaving together can reverse some of the damage done by noise and interference.

Opulent Voice is a modern open source amateur radio voice and data protocol suitable for 222 MHz and above. Transmitted voice is of excellent quality, with bitrates starting at a default of 16 kbps and can be configured up to a maximum available bit rate of 500 kbps. Data is transmitted without having to use a separate packet mode.

Opulent Voice uses modern digital communications techniques including randomization through scrambling, error correction enhancement through interleaving, and more. The transmitted signal is divided up into frames. There is a Preamble, Frame Headers, Sync Words, and Payloads. There are two types of Forward Error Correction.

A Preamble is a pre-defined burst of signal that allows an Opulent Voice transmission to be quickly recognized by a receiver. It's sent once, at the beginning of a transmission. Frame Headers contain vital information about the link such as transmitter identification and authentication values. Sync Words are pre-defined bursts of signal that help keep frame timing. They help ensure that we are looking at the correct part of the signal at all times. Forward Error Correction inserts additional bits in the transmission that make it possible to correct errors at the receiver.

At the transmitter, we do the following in order. We encode our data with additional error correcting bits, apply interleaving, scramble the interleaved bits to make our data appear more random, insert a Sync Word to mark frame boundaries, and finally modulate and transmit it over the air. **Figure 1** is a drawing that summarizes the process of creating a Payload Frame and inserting a Sync Word at the transmitter.

Each frame in Opulent Voice is 40 ms long and was formed using minimum frequency shift keying modulation with four tones. Think of this as four distinct frequency tones being sent one after another. It's like sitting at a piano and playing four particular notes. Each note is heard across the room by someone with perfect pitch, who then writes down the notes on a piece of paper. Since there are four tones, we have four different values of information sent per piano note. Four values translates into two bits of information per tone. 40 ms frames of Opulent Voice sent at 18,700 FSK tones per second means we are getting 37,400 bits per second. Minimum Frequency Shift Keying means that we set the frequency difference between the tones to half the symbol rate. The waveforms that represent 00, 01, 10, and 11 differ by 9350 Hz. This makes our signal spectrally efficient.

As the signal travels between transmitter and receiver, it picks up noise and interference of many types. Some noise comes from the circuits we're using. Some noise comes from the natural environment. Some noise comes from other radio signals. The distance that the signal has to travel affects the received power. All of these deleterious effects combine. The result is that our received signal will be noisy and damaged. What can we do at the receiver to take advantage of all the work the transmitter did? Error correction gives digital communications a critical advantage over analog communications. Error correction can reverse damage done by noise and interference. How does an Opulent Voice receiver use the extra bits that were sent to detect and correct errors?

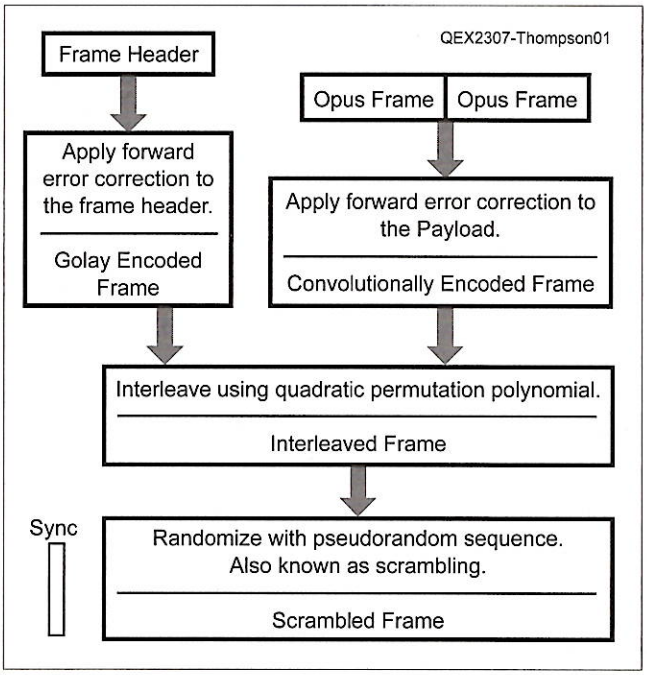


**Figure 1 — Creating a Payload Frame and inserting a Sync Word at the transmitter.**

There are some measurements and some calculations that we need to make in order to use and understand error correction. The math is simple, but the concepts are quite powerful.

First, we need to know our Signal to Noise Ratio (SNR). SNR is the signal power divided by the noise power. We use SNR to measure how tough it is out there for our signal compared to the noise. We usually see SNR expressed in dB, but in this calculation we are going to keep SNR as a linear ratio. Second, we need to know our bandwidth in Hz. Third, we calculate our Channel Capacity:

$$Capacity = Bandwidth \times \log_2(1 + \mathrm{SNR}) \text{ in bits per second.}$$

What does this calculated value of Channel Capacity tell us? At any bit rate below our Channel Capacity, we know for sure that an error control code can be designed that can bring our error rate down to something arbitrarily small. For any particular SNR and bandwidth, we know for a fact we can control our error rate if we stay below the Channel Capacity.

There's a catch. This theorem doesn't tell us how to design this error control code. It just tells us that one exists. It's up to us to figure out how to construct a particular error correcting code. Figuring out better error correcting codes is where a lot of energy is spent in digital communications engineering. The results over the past few decades have been nothing short of amazing, with a variety of error correcting codes that operate very near the Channel Capacity and are also relatively easy to implement.

Opulent Voice transmissions can be thought of as something like a freight train. There's an engine up front pulling a lot of train cars. Each train car, or frame of data, is full of bits. We take the information that we want to send, which might be digitally encoded voice, and pack these bits into the train cars. When we pack them in for their journey, we prepare the bits in two ways. First, we take out any unnecessary redundancy in the digitized voice. This is called source coding. Think of this as re-packing the boxes we want to send as freight to eliminate empty space in order to make more efficient use of the train car. Opulent Voice uses an open source voice coding and decoding system called Opus. With Opus, our voice frames are dense and valuable cargo.

Once we've gotten rid of any extra bits that don't help voice quality, by using Opus, we then pack the bits into the train cars by adding carefully chosen additional bits for error correction, by using our channel codes. Golay Codes and Convolutional Codes are used for channel coding.

Source coding and channel coding work together. The goal is to first get rid of bits that aren't strictly necessary to express our information. We may also optimize the bits we have left, with techniques like equalization or filtering. We then add in bits that are going to help make our signal more resilient for the journey over the air. These extra bits are like having extra players on the field in a ball game, who are all working together to get the ball down the field without it being lost to the other team. Forming an effective team and picking the right strategy to get past the opposing players is a very similar process to what happens in forward error correction.

The default version of Opulent Voice, with a 16 kbps version of the Opus voice codec, operates at 18,700 symbols per second (37,400 bits per second), and sends 1496 bits in every 40 ms frame.

Golay Code is used for the Frame Headers, and a Convolutional Code is used for the Payload frames. The Payload frames contain our data. If it's voice, then we are sending Opus frames. If it's data, then we use Consistent Overhead Byte Stuffing (COBS) to frame arbitrary data for our Payload. Regardless of the cargo, whether Opus or other digital data, our train cars all end up being the same length as they roll down the track.

We are now at the receiver, waiting for an Opulent Voice transmission. How do we find our signal? We are looking for the engine pulling the train of frames.

The first frame produced by a transmitted Opulent Voice signal is called the Preamble. It functions like the bright light on the front of a train engine, cutting through the darkness and warning us a train is on the way. The Opulent Voice Preamble is a series of the highest and lowest tones from the set we're using, alternating back and forth. This produces a very recognizable signal in the frequency domain. Once we have identified and received the Preamble, we discard it. Our Preamble has no error correction.

The rest of the frames are sent using combinations of all four tones. Through coding and other techniques, the four tones create a spectrum that looks much more random than the Preamble.

This is an important difference. The Preamble is chosen to be very distinctive, with very little uncertainty. It has very little information in it. These characteristics make it easy for the receiver to find. The rest of the frames have to pack in a lot of information. They have a much richer spectrum, and look more like a lump of noise. We use the Preamble to find the start of the transmission.

Once we've identified the start of the transmission, we wait for the end of the Preamble and start looking for the start of the first full frame. We know we have 40 ms frames, but where do those frames start if the rest of the transmission just looks like an undifferentiated bunch of noise? The Preamble gave us some indication, but it's not the only thing the transmitter did to help us at the receiver. The start of every frame coming after the Preamble begins with a distinctive Sync Word. Think of this as a smaller light or reflective patch on the front of each train car. The transmitter constructed each frame to begin with a fixed pattern of 16 bits. We know that pattern, so when we see it we know exactly where the data frame starts, every time. When we receive a frame, we confirm the first 16 bits match our expected pattern. We can then set those bits aside and examine the rest of the frame with confidence that we know when and where we are in the transmission.

After removing the 16 bits of Sync pulse from the 1496 total bits in the frame, we are looking at 1480 bits of data. These 1480 bits were scrambled up in a particular way to make the signal look more random. This is an important characteristic for digital communications. A random signal doesn't have long stretches of zeros or ones, but our structured voice and data signals certainly can have long stretches of zeros or ones. We don't want long stretches of zeros or ones. More transitions in the data pattern helps our receiver synchronize to the proper bit timing. Long stretches of zeros and ones don't have transitions on the bit boundaries. So, we use scrambling to change the statistics of the signal to something we want without destroying or changing any of the underlying information. We do this by using a pseudorandom scrambling sequence to transform the bits in our frame so that we're sure we have lots of bit transitions to work with.

Pseudorandom sequences have a starting point and the sequence repeats after some number of bits. They aren't random, but they have statistically similar characteristics to random numbers. We exploit the characteristics of pseudorandom sequences to get large benefits for a very small cost.

Specifically, in any pseudorandom sequence, 1/2 of the runs (of zeros and ones) have length 1, 1/4 have length 2, 1/8 have length 3, 1/16 have length 4, and so on. The number of runs of zeros is equal

to the number of runs of ones. And, a pseudorandom sequence has an equal number of zeros and ones. It might be off by one if the sequence has an odd number of digits.

We need to unscramble our received bits in order to get any further. We know the exact pattern the transmitter used, so we reverse that pattern and restore the original bit values. How did the transmitter scramble up the data to make it look random? The transmitter will XOR (exclusive OR operation) the entire frame with a fixed known sequence of pseudorandom bits the same length as the frame. **Table 1** shows some of the XOR operation done at the transmitter. **Table 2** shows the XOR operation done at the receiver.

Applying XOR operation twice, using the same pseudorandom sequence, gives us our original data bit back. So, at the receiver, we take a frame, XOR it with our pseudorandom sequence, and we get back what the transmitter had before it scrambled the data. So, now we've got unscrambled data. The next layer down is interleaved data.

Interleaving re-orders the bits. It's different from scrambling, where the bits were flipped in place to make them appear more random. Interleaving moves bits around in a particular way. We want bits to spread out as far apart from each other as possible. Why interleave? Well, it has to do with being clever about sudden crashes of noise blasting our signal. A steady level of noise will cause some amount of damage to our transmission. But, every so often, we will get a large amount of noise that might appear for a long enough period of time to completely wipe out a sequence of bits. The rest of the time, the signal is in decent shape with an occasional single-bit error, but with these larger bursts we miss out on a chunk of data, and many bits might be missing in a row.

Why is this a problem? We can only correct so many bits at a time. The background error rate from typical channel conditions may be well within our ability to correct, but these sudden bursts of noise are a problem because our error correction algorithm will fail. Wouldn't it be a great thing to be able to distribute these burst errors throughout the frame? Can we somehow turn the burst errors into a slightly higher average error rate? We can and this is called interleaving. And, this is something our Opulent Voice transmitter did for us. That means it is something we have to un-do at the receiver before we can get to the layer with the error correction.

Here's an example of how interleaving works with a famous poem called Ozymandias by Percy Bysshe Shelley. Here's the data we sent:

"My name is Ozymandias, king of kings:
Look on my works, ye Mighty, and despair!"

Nothing beside remains. Round the decay
Of that colossal wreck, boundless and bare
The lone and level sands stretch far away."

Here's our transmission with a significant burst error marked with **X**.

"My name is Ozymandias, king of kings:
**XXXXXXXXXXXXX**ye Mighty, and despair!"

Nothing beside remains. Round the decay
Of that colossal wreck, boundless and bare
The lone and level sands stretch far away.

Well, we aren't going to be able to tell what was under those errors. What if we interleave the poem before sending it? We know we're going to get this type of burst damage, so we mix up the letters in a predictable way and then send it over the air.

soyba hh y a bk oii a h skn !.l bpu" sia nhynri
nn wcia y,tc, rsegnrynyf"e aeso d,rioamodoe
Ms,gameaan .fwMs ah oeskvtacnNdokL
or:hndsdkcmieaOa nsl odadusl yetef elTnlelmtn
dtogatir rRiseewet dnd gesOzes

We get a burst error. Errors are marked with **X**.

soyba hh y a bk oii a h skn !.l bpu" sia nhynri
nn wcia y,tc, rsegnrynyf"e aeso d,rioamodoe
**XXXXXXXXXXXXX**wMs ah oeskvtacnNdokL
or:hndsdkcmieaOa nsl odadusl yetef elTnlelmtn
dtogatir rRiseewet dnd gesOzes

We deinterleave the poem at the receiver.

"My name i**X** Ozymandias**X** king of kings:
Look on my works, ye **X**ighty, and despair!"

Nothin**X** beside re**X**ains. Round the d**X**cay
Of th**X**t colossal wreck, boundless**X**and bare
The lone a**X**d level s**X**nds stretch **XX**r away**X**

When we receive the poem, we can see it has some missing letters. But, we now have a much higher probability of being able to sort out all the damage because the errors are distributed throughout the poem instead of all in a row. We can recover most if not all of the damaged words if the words are missing a letter or maybe two. We can't recover an entire missing phrase, which is what would have happened if the original poem was sent over the air with all the letters in order.

This is how interleaving and error correcting codes work together. The error correcting codes are like your brain, fixing the

**Table 1 – Input 1 XOR with Input 2 = Result for transmission**

| Input 1: Data bit | Input 2: Pseudorandom bit | Result: Transmitted bit |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Table 2 – Receiving bits**

| Input 1: Received bit | Input 2: Pseudorandom bit | Result: Data bit |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |

words you read that have occasional errors. The interleaving makes sure that those errors are scattered around rather than concentrated in one place. Interleaving is a common technique in digital signal processing and is required to get all of the functionality out of certain types of error correcting codes. Since we are using one of those types of codes, a convolutional code, we need to interleave.

We do have to know exactly where each original bit came from. For Opulent Voice, we have a formula that told us how to interleave, and this same formula will tell us how to get everyone back in line in the correct position. Without knowing this, nothing else will work. Once we re-order things, then we have all the bits in the right sequential order and can use our error correcting code.

There are several ways to interleave, but Opulent Voice uses a quadratic permutation polynomial interleaver. It is defined by a simple equation. $x$ is the original position of the bit. $y$ is the interleaved position:

$$177x+130x^2 = y$$

We are interleaving 1480 bits of data, so if the result is larger than 1480 or a multiple of 1480, we take the remainder. In other words, the equation is done with modulo 1480.

Let's look at the first few results in **Table 3**. The bit at position 0 stays where it is. The bit at position 1 is moved to position 547. The bit at position 2 is moved to position 354. The bit at position 3 is moved to position 901. The bit at position 4 is moved to position 708. The bit at position 5 is moved to position 1255.

Let's say we get a burst of noise that wiped out 6 bits in a row of interleaved data. At the receiver, we reversed the interleaving. Those 6 bits in a row of damaged interleaved data are all sent back to their original positions. They are, by mathematical design, as far away from each other as is possible. The quadratic permutation polynomial is selected specifically for how far apart it scatters contiguous bits.

Now we've got deinterleaved data. Our frame starts to show some structure. There are two sub-frames. The first is the Frame Header, and the second and larger of these two sub-frames is our Payload data. In most cases, the Payload will be Opus voice.

The Frame Header is stuck on to the front of each Payload. The information in the Header allows a listener to join a transmission in progress. For a narrowband signal, this can be a lot of overhead. There are techniques to mitigate the overhead, like splitting up the information in the Frame Header and distributing it round-robin style. For higher bit rate modes like Opulent Voice, the relatively small number of bits in the Header are not a burden, so they are sent every time in full.

How do we decode these frames? Well, we read the Opulent Voice specification document and we know we have to deal with 8 sections of 24 encoded bits for 192 bits total. We used a 12 to 24 bit Golay Encoder at the transmitter. We took our twelve 8-bit bytes of Frame Header data, organized it into eight 12-bit groups of bits, and multiplied each of these 12-bit sections by the Golay Code generator matrix. Each of these multiplications resulted in 12 parity bits. For each 12-bit portion of the original data, we attach the corresponding 12-bit parity result to it, and this creates eight 24-bit codewords. These eight 24-bit codewords are what we then interleaved, scrambled, and then sent out over the air. Codes that send the original data plus some parity bits are called systematic codes.

**Table 3 – Interleaving**

| x | y |
|---|---|
| 0 | 0 |
| 1 | 547 |
| 2 | 354 |
| 3 | 901 |
| 4 | 708 |
| 5 | 1255 |

When we receive this 24 bit codeword, we unscramble and deinterleave. Then, we correct errors by multiplying the received codeword with another special matrix. This one is called the parity check matrix. It's the partner of the generator matrix that created the parity bits at the transmitter. The result from this multiplication is called the syndrome. If the syndrome is zero, then there were no errors in the codeword. We drop the parity bits and what is left is the original data. If the syndrome has ones in it, then we take that result and go to a lookup table that we've made based on the protocol specification. This table maps the result of the multiplication to a list of which bit positions in the codeword have errors. Whatever bits have errors, we flip them. Now we have a corrected codeword. We drop the parity bits and what is left is corrected original data. This process is called syndrome decoding.

We can correct damage wherever it occurs across the 24-bit codeword, within reason. The limit for this particular type of code is that it can correct 3 errors. It can detect up to 7. If we have a situation where we've detected 1, 2, or 3 errors, then we can correct them on our own based solely on the results of our "secret decoder ring" matrix math we use at the receiver. If we have 4, 5, 6, or 7 errors, we know we have them, but we cannot correct them on our own. 8 or more errors and some interesting things might happen. We will be pulled off course so far that our codeword resolves to something completely different than what was sent. The resulting data stream will be very damaged. There's not much we can do about this, but the level of noise or interference that would cause half or more of lost bits in a codeword are extremely high.

Now that we have these very helpful bits of information about our link decoded, we can turn our attention to the payload itself, which contains Opus voice frames. The payload is convolutionally encoded data. We know that the deinterleaved bits were convolutionally encoded at rate one-half.

When we talk about the rate of a code, we are talking about a ratio between how many bits go into an encoder (the number on the top) and how many bits come out of an encoder (the number on the bottom). A rate 1/2 means that one bit of our original information was turned into two bits sent over the air. For a rate 1/2 encoder, every bit of source data is converted into two parity bits. Notice that the Golay Code is also a rate 1/2 code. 12 bits went in and 24 came out. Not all error correction codes are rate 1/2. There are a variety. Rates won't be less than zero or greater than one.

With a convolutional code, what we send over the air is the parity bits. No original data is sent. At the receiver, we figure out what the original data was by analyzing the stream of coded parity bits. Each sequence of coded parity bits stands for a unique original data stream. We infer the original data from the path the parity bits trace through a particular graph.

For a convolutional encoder, it's all about this path through a graph. The graph is called a trellis diagram, and it's used for both encoding and decoding.

How can we think of this? Is there something in everyday life that might be similar? Yes, there is. **Figure 2** shows a peg game. We drop a disc or ball in at the top, and it hits pegs on the way down. At each peg, it goes either left or right. We can think of this as going in either the 0 or 1 direction. At the bottom, we get a score or payout depending on where it fell. Each path that our game token took from top to bottom is unique. When we play the game over and over, we can see the patterns that we take

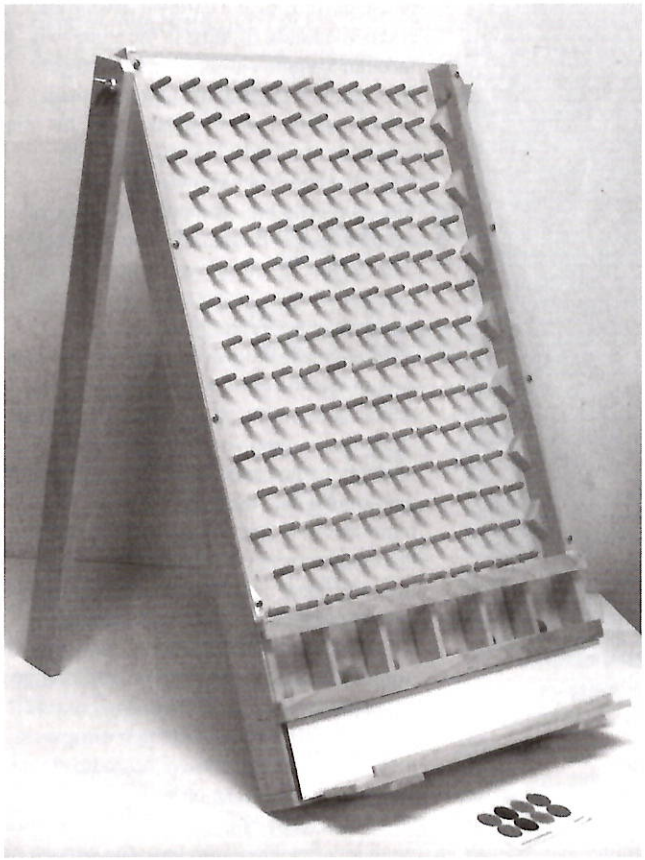

Figure 2 — A peg game.

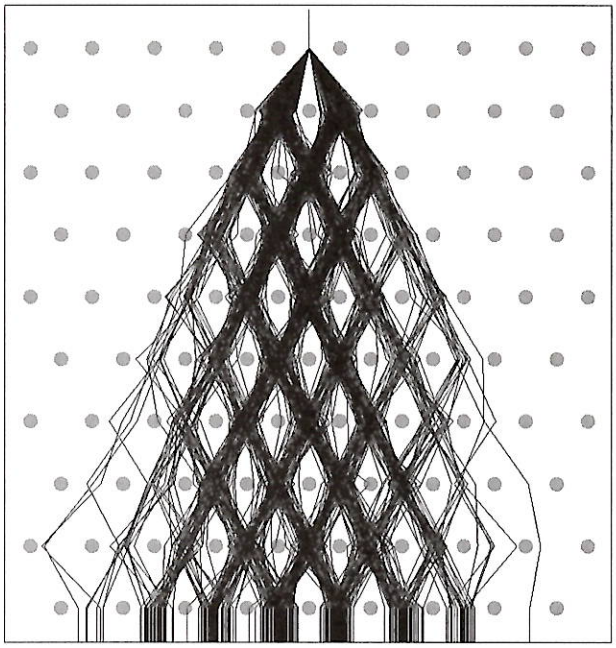

Figure 3 — Paths make something that looks a lot like a trellis.

falling through the pegs. When we do this over and over, and record them, the paths make something that looks a lot like a trellis (**Figure 3**). We can think of gravity the way we think of time. Gravity pulls our token down to the ground through the field of pegs. Time pulls our received parity bits through the states of a trellis diagram (**Figure 4**).

An enduring truth of engineering is that you don't get something for nothing. If you want to magically correct errors, lessen the impact of large bursts of errors by distributing them over time, and then deliver clean and clear results at the receiver as if nothing had happened, then there is a price to be paid. That price is complexity.

When we use convolutional error correction, which does the function that our brain does in correcting the missing letters in the poem *Ozymandias*, then there are things that we have to give up. We are giving up simplicity and flexibility for the ability to repair the data that we received. The benefit that we are getting is that we can repair this data on our own, without requiring it to be re-sent. At least, up to a point. There are limits to how many bits per received codeword we can correct, but with error correction, we can clean up badly damaged signals. An equivalent signal sent without error correction would be useless.

A convolutional encoder is matched to its convolutional decoder. The signal we've created in Opulent Voice can only be undone by the matching Opulent Voice decoder at the receiver. Going back to the comparison to a peg game, our receiver recreates the path the ball took at the transmitter in the peg game. We can do this because the receiver knows what particular transmitted values each peg "stood for" as that peg was hit on the way down. The original information sequence that we sent was defined as "which direction do we choose go at each peg." The path labels between pegs represents the unique parity code. These parity codes are what was sent over the air. Those two bits sent over the air represent the one bit that got us to that path. So, it is the path itself through the field of pegs that stands for a particular sequence of data. The way that we construct this path allows us to lose some bits here and there over the air and still be able to see the most likely path. The path is like a sentence in English. If we lose a letter here and there, we can fill it in. There's only so many missing letters that make sense. We can only paper over so many missing segments of the path through the trellis.

Another metaphor for this process is if we mapped the paths a robotic vacuum cleaner takes while cleaning a room. Let's say we record the paths with time-lapse photography. Even if the robot goes under a coffee table, and we lose part of the path, we can successfully guess which path it took by looking at the trails it made and extrapolating which paths connect together under the table.

The Opulent Voice specification tells us exactly what we need to know to do convolutional decoding in a compact and standard way. We express the particular encoder we are using in the form of a polynomial. A diagram for the trellis defined by the polynomial in the specification can be found at **https://www.openresearch.institute/wp-content/uploads/2022/09/opv-trellis-diagram-segment.png**.

Here is how we encoded the data, with reference to the image at the URL. We start at dark blue node 0 and we take a bit of our input data. We take the green path if the input data was a 1, or a red path if the data input was a 0. Each node has two pairs of numbers separated by a slash. These tell us what parity bits we transmit. We write down the stream of parity bits we create by moving through the trellis with our input data guiding the way. Look at node 15.

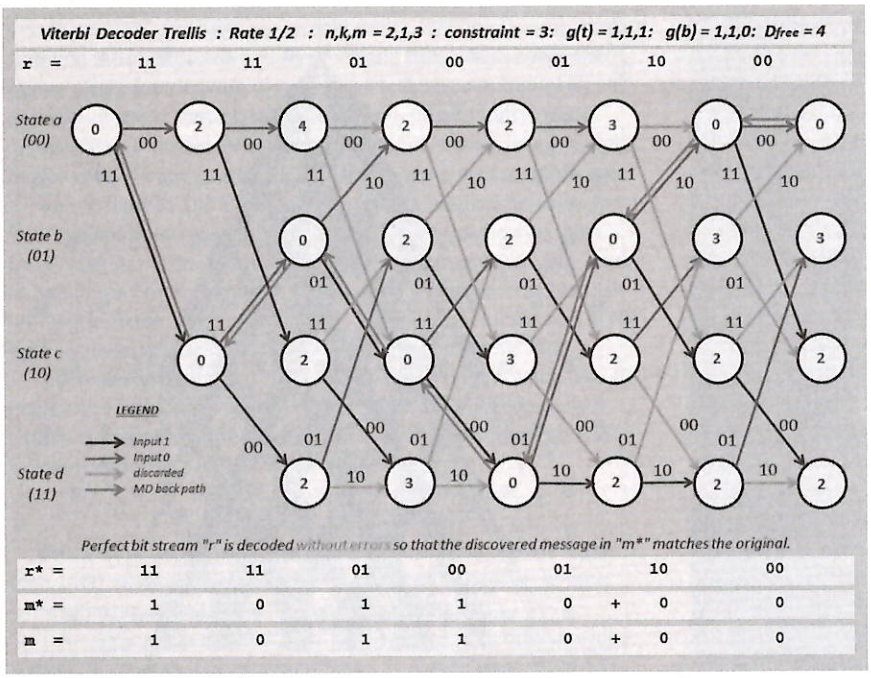


**Figure 4** — Time pulls our received parity bits through the states of a trellis diagram.

The top pair of numbers is 0/3. This tells us that if we are at this node, and the input data was 0, we emit 11 (3 in binary). We then take the red path to 15 and select the next data bit. Let's say that data bit is 1. We emit 00 (0 in binary) and then take the green path, which moves us to node 7.

To receive, we use the same trellis diagram. We feel our way forward through the graph, comparing the received set of parity bits to the ones that the path said we'd see. If they match closely, then we'd be wise to pick that path. In the end, the path that most closely matches what we have to work with is picked as the winner. We use this path to define what we think was the most likely original data sequence.

Figures 4, 5, and 6 at the following URL: **https://en.wikibooks.org/wiki/A_Basic_Convolutional_Coding_Example** are examples of decoding by hand using a trellis diagram for a four-state system. Each vertical line of circles is a step in time. The green arrows are where we explore forward, comparing the received parity bits to what was generated by that path when the trellis diagram was used to encode. For each path forward, we mark down how many errors we've accumulated along the way. When we reach a stopping point, we then turn around and look at all the paths we explored, and pick the one that had the best metrics. Usually, this is the path with the lowest number of bit errors.

When we decode we must periodically stop and look back at our path and make a call on what the sequence was up to that point. The choice of how far to go, before looking back and making a decision about a sequence of data, is an important one. The longer we go, the more time it takes to process that section of the bit stream. Too short, and we might end up with not enough space around the coffee table to clearly see which Roomba path was the right one. The determination of how long to let each batch of convolutional decoding run is an important design decision for these types of codes.

This process of convolutional decoding is like going through a house of mirrors at a carnival. Everyone enters the house of mirrors the same way, through a single entry point. As you go forward, you explore the potential paths. Paths that don't work out are paths that you remember as "bad." You do eventually reach the end. Your knowledge of the directions you took through that house of mirrors can be thought of as the original data stream. To decode the next batch of received bits, you enter another house of mirrors with different glass walls and different mirror locations. The original data stream sets the glass and mirrors in a particular pattern. As long as there are not too many errors, then you will have a path through the house of mirrors. You will get the original data back. If you have too many errors in your received parity stream, then you may be completely cut off from the exit. There's no obvious path forward and you can't recreate the data stream.

Digital forward error correction, and digital signals in general, make for a different radio experience from analog communications. In analog, a relatively simple receiver circuit can recover an audio signal from a wide variety of analog transmitters. In order to be able to receive digital signals that have been damaged by noise, our transmitter and receiver are much less flexible. If we have the wrong decoder, then we may get nothing at all from the headphones. A different set of polynomial representations for a convolutional encoder is like having a completely different house of mirrors to work through.

We now have the Frame Header and the two Opus frames that we put into the data Payload. The next step is to decode these two Opus frames and recover the voice signals. We use the decoding functions in the Opus protocol standard, and we get audio waveforms.

That is the story of forward error correction in the Opulent Voice receiver. The system uses two types of forward error correction, a convolutional encoder and a Golay code, to protect our data from noise and interference it encounters over the air. This is a solid protocol done in a common-sense manner that uses modern error correction to achieve very high quality 222 MHz and above voice and data communications. Source code for a C++ implementation of Opulent Voice modulator and demodulator can be found at **https://github.com/phase4ground/opv-cxx-demod**.

If you would like to see more projects like Opulent Voice succeed in the amateur radio community, please join **https://openresearch.institute** at the "Getting Started" menu option. There is no cost. You do not have to be an expert to join, you just have to be willing to become more of one along the way. Everything ORI does is open source and education is a central part of the mission.

*Michelle D. Thompson, W5NYV, enjoys thinking and doing — not necessarily in that order! Book learning includes BSEET, BSCET, math minor, MSEE Information Theory. Actual doing includes engineering at Qualcomm, engineering at Optimized Tomfoolery, Amateur Extra-class license, AMSAT Phase 4 Ground Lead, DEFCON, IEEE, Burning Man, and community symphony.*