The principle of forward error correction

Paul Nicholson, December 2016.

Introduction

The aim of this article is to demonstrate and try to explain the fundamental principle behind forward error correction codes. I'm not going to discuss any particular coding scheme (Convolutional, Reed-Solomon, LDPC, and so on), rather, my intent is to home in on the underlying principle that all forward error correction schemes exploit. I'm going to do so without using any mathematics at all, well perhaps just a little simple arithmetic.

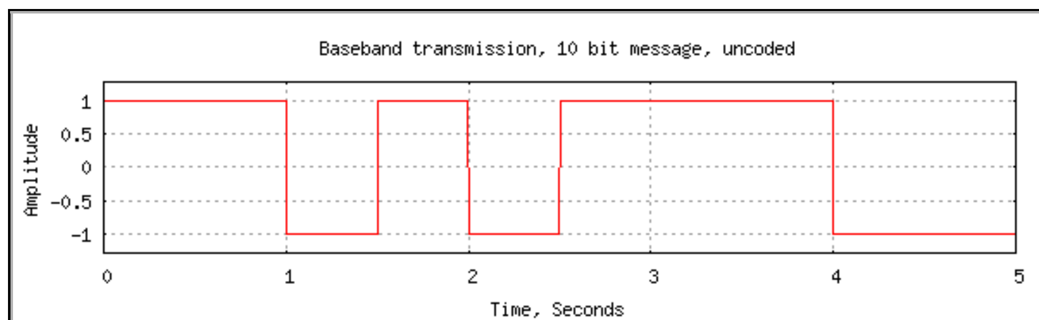
Uncoded transmission

First, the problem statement: we want to convey some quantity of information (measured in units of bits), with some degree of reliability, from one place to another across a noisy communication link, in some fixed amount of time using a fixed transmitter power. I will assume that the available bandwidth of the communication channel is unlimited.

To demonstrate with a concrete example, I will try to send 10 bits of information in five seconds using simple bipolar signaling over a pair of wires. For simplicity this is base band signaling, there is no radio frequency or modulation involved. It isn't needed to demonstrate the principles.

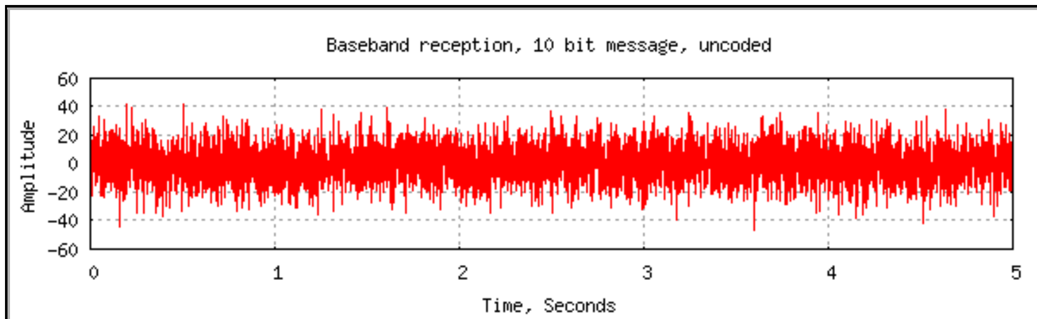
The signaling represents the 1s and 0s using voltages of +1V and -1V respectively. It doesn't get any simpler than this.

The 10 bit information message I will use is: 1101011100. Each bit is sent for 0.5 seconds with either +1V or -1V and the waveform of the outgoing signal looks like:



Now I will add noise to this signal to represent the noise picked up by the receiver at the far end of a long pair of wires. In reality the long wires, or a radio channel, will not only add noise but will also introduce various kinds of distortion which alter the shape of the signaling bits so that they are no longer rectangular at the receiver. For this exercise though, we will just use a noisy but distortion-less channel to keep things simple. Also we will let the line have no attenuation so that the received message bits have their original +1V and -1V amplitudes.

To make a good demonstration, I'm going to add quite a lot of noise, enough to completely bury the signal on the plot of the received signal:

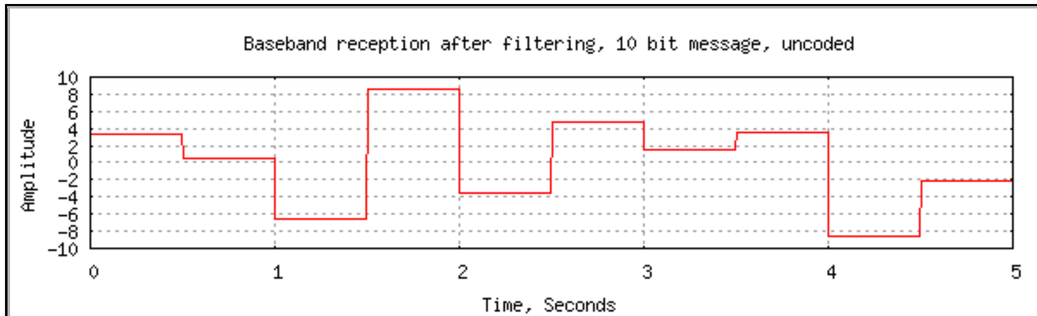


Note the change of amplitude scale on the left. The noise is a random signal of about 12V RMS amplitude. The +/- 1V message is still there but is completely invisible under the much larger noise.

Receive filtering

How does the line receiver go about detecting the message bits? Well the first thing it must do is to throw away most of the noise. The bandwidth of the plots that I'm using is 500Hz but my message signal is only changing state at most twice per second and therefore has a bandwidth of 1Hz. We can allow the line receiver to filter out any frequency components above 1Hz without losing any of the wanted signal. Another way to describe the same thing is to think of the filtering as averaging the noise over the half second period of each message bit.

The filtering improves things considerably. The received waveform after filtering looks like:



The filtering (averaging) means that the signal is now constant during each 0.5 second bit period because any faster variations have been removed. The result is that our receiver sees the noisy signal as a set of ten voltage samples, one for each bit.

Uncoded detection

Even though we have removed most of the noise, the signal still has to compete with the noise that was within the bandwidth of the signal and which therefore can't be removed by filtering. The bit amplitudes are no longer +/- 1V but now have quite a range of values, sometimes exceeding 8 volts, so the remaining noise is still having a severe effect.

From these noisy filtered samples the receiver has to guess what the original message was. The simplest way it can do this (actually, the only way) is to decide that if the voltage is greater than zero, a 1 bit is detected, and if less than zero, a 0 bit is detected.

Inspection of the filtered signal shows that the receiver does in fact detect the original message correctly in this case. All the transmitted 1 bits are received with a voltage greater than zero so are detected as a 1, and likewise, the 0 bits are all received with voltage less than zero.

We were lucky with this transmission. If the noise within a bit period averages to more than 1 volt and happens to have the opposite polarity to the transmitted bit, it will be enough to cause the sample amplitude to cross the zero and the receiver will make an error: a 'bit error'. This very nearly happened with the 2nd bit of the message. If there are one or more bit errors then the message as a whole will be decoded incorrectly: a 'message error'.

To see how often the message is received incorrectly, all I have to do is repeat the message many times, say several thousand times, and count how often the message is detected correctly or not.

In fact with this level of noise, the message is correctly received, on average, in 65.5% of attempts. In other words we have a message success rate of 65.5% and a failure rate of 34.5%.

There is no way to improve on that success rate with this level of noise so long as we are sending the message bits 'uncoded'. Of course we could do better by increasing the transmitted amplitude, or by sending each bit for longer so that the receiver can use a narrower filter to remove more of the noise. But for this exercise I'm going to keep the same +/- 1V amplitude and the same 5 second message duration and demonstrate what can be done by applying some forward error correction.

A crazy idea

What I want to demonstrate here is the principle of coding which allows us to use the same transmit power and the same overall message duration, but achieve much better than a 65.5% success rate.

To start with, I'm going to do something quite remarkable and rather non-intuitive. I'm going to replace the 10 bit message with a 100 bit message!

Specifically, instead of transmitting 1101011100, I'm going to transmit the 100 bit 'codeword':

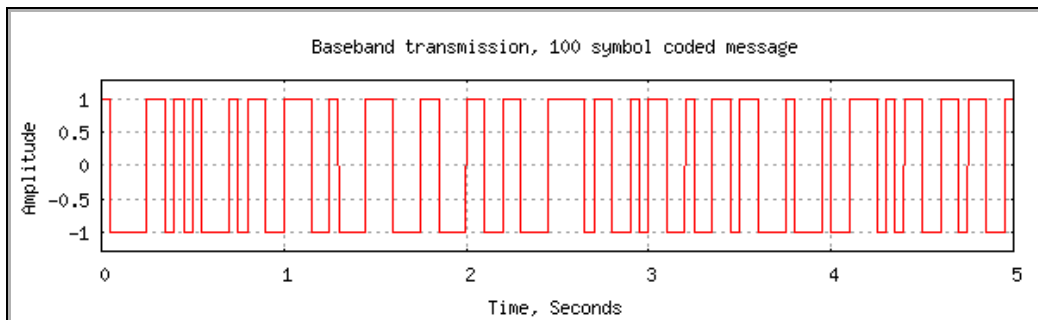
```
11011110100001010100001111110011010100001111011101101110100001011101011001100011000110011000101101001
```

At first glance this seems a crazy thing to do. I'm proposing to send this 100 bit codeword in the same 5 seconds as the original uncoded 10 bits, so now each bit is only sent for 0.05 seconds and the bandwidth the receiver must use is now 10Hz instead of 1Hz which means that the noise after filtering is going to be a lot higher than before. How can the receiver have a better chance of a correct message detection when it now has to detect 10 times as many bits and is faced with 10 times the noise power? It is certainly going to suffer many more 'bit flips' due to the extra noise in the signal bandwidth. This looks crazy at 2nd and 3rd glance too.

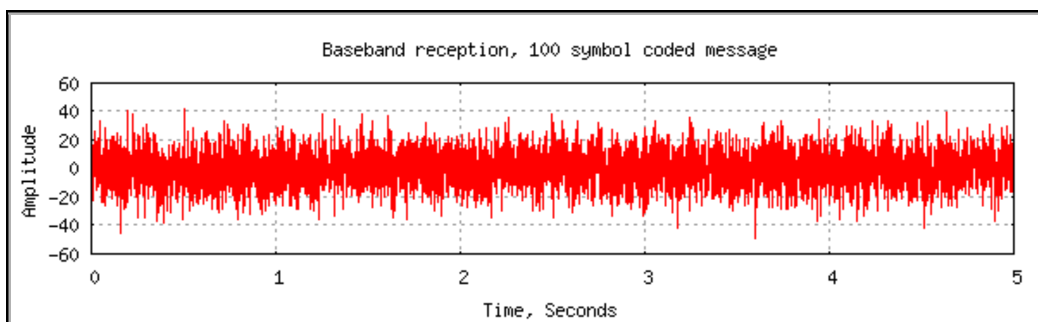
Before I proceed to test the crazy idea, I must carefully introduce some terminology here to avoid confusion, especially with words like 'bit'. We have a 10 bit message to send. I'll refer to those bits as info bits. We are sending a 100 bit codeword and I'm going to call those bits 'symbols' or 'code symbols'. Now there are 1024 possible 10 bit messages (2 to the power 10) so to be able to transmit any 10 bit message I'm going to need a total of 1024 distinct codewords, each of 100 symbols. The collection of 1024 codewords is called a 'codebook'. To send a 10 info bit message, the transmitter must look up the message in the codebook to find the corresponding 100 symbol codeword and send that instead.

(You'll notice I have avoided mentioning how I chose 1024 codewords for this experiment, all will be revealed later.)

Now the transmitted signal with 100 symbols looks like:



Using exactly the same noise as we did with the uncoded message, the received signal looks like:



It actually looks the same because the coded message is buried deep under the noise - all we can really see on the plot is the noise.

Now the question is, what does the receiver do with this? Well, the receiver must do the best it can to detect the 100 code symbols and then examine the codebook to find the codeword which is nearest (in some sense) to the noisy 100 symbol word it has received. Then it can report the 10 information bits that are associated with that codeword in the codebook.

Before going into that in more detail, let's put this coding scheme to the test. Once again I send several thousand messages and count how many of these were correctly received. The results are: 82.0% of the messages are 'decoded' correctly, only 18.0% are decoded incorrectly! That's a significant improvement over the uncoded message - the error rate is almost halved!

In fact we shall see later that the receiver can do a great deal better than that, without even changing the codebook or the signal in any way. But we'll come to that later. For now we want to understand how we got from a 65.5% success rate to a 82.0 % success rate by the counter-intuitive step of making the receiver deal with a ten times wider bandwidth and therefore a signal/noise ratio which is 10dB worse than the uncoded message. It is certainly extremely odd that reception improves by reducing the signal/noise ratio and if you had made this demonstration prior to 1948, you would probably have been declared a charlatan and escorted from the building.

Decoding

Now a look in more detail at what the receiver is doing. The 100 symbol message is filtered to the 10Hz bandwidth of a code symbol to average the noise in each 0.05 second symbol period. This produces a set of 100 amplitude samples, one for each symbol. The detector then decides whether

each symbol is a 1 or a 0 according to whether the sample voltage is greater or less than zero.

The result of the detection is a 100 symbol word which is almost certainly not one of the codewords in the codebook because many of the symbols will have been flipped by the noise. The receiver must compare the received word with each of the 1024 codewords of the codebook. For each codeword the comparison involves counting how many symbols differ between the codeword and the received word. This is known as the 'Hamming distance'. After comparing with each codeword in the codebook, the receiver chooses the codeword which has the smallest Hamming distance and outputs the associated 10 info bits.

We begin to home in now on the crux of this apparent miracle of coding. It must be something to do with this business of choosing from the codebook the 'nearest' codeword to the received word. It must be this, because that's the only thing that the receiver is doing differently when it is working with these coded messages instead of the original info bits.

Consider some numbers: The received word of 100 symbols has been scrambled by the noise so that many of the symbols have been flipped from a 1 to a 0 or vice versa. There are 2 to the power of 100 possible received words which the noise can produce. That's a huge number - to be exact there are 1267650600228229401496703205376 possible different received words that my transmitted codeword could have been turned into by the noise. That's a bigger number than the size of the observable universe measured in millimetres! But only 1024 of these are listed in the codebook. That means that the actual codewords are very sparse indeed in the set of all possible received words. It is something to do with the codewords being so well separated that allows the receiver to not only overcome the extra noise of the wider bandwidth but also to perform better than the uncoded message detection.

Random codes

At this point I will reveal that my codebook of 1024 codewords was generated randomly, subject to the single constraint that no two 100 symbol codewords differ from each other in less than 38 symbol positions. In other words the Hamming distance between any two codewords is at least 38. This is known as the 'minimum distance' of the code. I chose to use a random code for this demonstration to show that the magic of forward error correction isn't a consequence of using any particularly special set of codewords. In fact, randomly generated codebooks perform about as well as any coding scheme possibly can. The reason they are not used in practice is the monumental difficulty of decoding messages of any significant length. With my 10 bits of information there are only 1024 codewords in the codebook and it takes the decoder just a fraction of a second to compare the received word with the entire codebook. If I were sending 30 bit messages I would need a codebook with 1073741824 codewords, which apart from taking up a great deal of memory, would need a lot of computer time to compare each one with the received word to determine the decoder output. A realistic message might be a few hundred info bits and it becomes quite impossible to even construct a big enough codebook (there are not enough electrons in the universe), let alone get a receiver to search it for the closest match.

To avoid trying to use an astronomically large codebook, mathematicians have come up with various ingenious ways to generate a codeword from a message in such a way that the receiver can search for the closest codeword by much more efficient methods than the brute force comparisons necessary for a random code. The whole vast subject of forward error correction is concerned with finding ways to implement a codebook without actually having to generate the codebook or make a brute force search of it at the receiver. The aim is to invent codes which approach as closely as possible the desirable 'distance' properties of random codes, but which contain enough mathematical structure to allow the receiver to very quickly determine the nearest codeword to a given received word.

So, all the codewords in my codebook differ from each other in at least 38 symbols. Most pairs differ by more than this but the worst case is 38, the minimum distance of my code. That means that when the receiver comes to search the codebook, it is guaranteed to select the correct nearest codeword if 18 or fewer symbols have been flipped by the noise. We would say that this code is guaranteed to 'error correct' up to 18 symbol errors. With most received words it will do better than this, but 18 is the worst case.

Shannon's theorem

So the crazy idea works! It seems that although we have made the chance of symbol errors much higher by transmitting more of them in the same time, the error correcting strength of the code is able to overcome those errors and actually do better than just sending the uncoded 10 info bits.

How far can we go with this? Claude Shannon proved his remarkable 'Noisy channel coding theorem' in 1948 using the properties of random codes of the kind I am using here. Broadly speaking, the theorem proves that we can achieve higher and higher success rates by using longer and longer codewords and by encoding more info bits into each coded message. As we increase the number of symbols in the codeword, the probability of each symbol being received incorrectly increases, but so does the error correcting strength of the code because the minimum distance of the longer codes will be higher: codewords become more widely separated in the set of all possible received words.

Shannon proved that, subject to certain conditions, the error correcting strength of a random code increases faster than the error creating effect of having a shorter symbols in a wider bandwidth. In fact we can actually make the success rate approach as close as we like to 100% by using increasingly longer coded blocks of information, so long as the signal to noise ratio exceeds a certain critical value. I'll discuss this in more detail further on and I'll also demonstrate this with some longer codes.

But first, a short but important digression.

Soft decisions

I mentioned several paragraphs back that we can actually do rather better than the 82% message success rate of this random code, without altering the codebook or the way the symbols are sent. In fact we can boost the success rate to an amazing 97.5% using the same code, just by making a neat and simple improvement to the receiver!

The change involves using a different definition of 'distance' when we are searching the codebook for the 'closest' codeword. The receiver is comparing the received word with all the possible codewords by looking at the Hamming distance - the number of symbols which differ. By analogy, we can imagine describing the distance between two buildings in New York. You may say that to go from A to B, you travel 3 blocks east and 4 blocks north (or by some zig-zaggy combination of easts and norths). You travel a total of 7 blocks, and that's like your Hamming distance between A and B. But as the crow flies, B is a actually a distance of 5 blocks from A (Pythagoras: sqrt(3*3 + 4*4) = 5). This distance is called the 'Euclidean distance'. By changing the receiver to use Euclidean distance instead of Hamming distance when scanning the codebook for the closest match to a received word, we obtain the above mentioned dramatic further improvement.

I'll explain how this change is done: My decoder has been assigning a 1 or a 0 to each received symbol according to whether its amplitude is positive or negative. It takes no notice of whether the amplitude is just a little bit positive, or very positive. The receiver has been making what is called a 'hard decision' about each symbol. But by doing so, it is throwing away potentially useful information contained in the noisy symbol amplitudes. It is throwing away valuable information about the likelihood

of the 1/0 decision being correct. A very positive symbol amplitude is much more likely to be a 1 than an amplitude which is only just positive.

Instead of making a hard decision the receiver can make a 'soft decision' about each symbol. We do that by using the symbol amplitudes themselves without detection to a 1 or 0. To use the undetected amplitudes we make the receiver perform a 'cross correlation' between the received word and each codeword of the codebook. Cross correlation is a very simple way to measure the similarity between two signals. It produces a score and the score is higher the smaller the Euclidean distance between the received word and the codeword. The receiver is modified to pick the codeword with the highest cross-correlation score which is therefore the codeword with the minimum Euclidean distance to the received word.

It turns out that using the symbol amplitudes to find the codeword with minimum Euclidean distance, is the best possible method. It identifies the 'maximum likelihood' codeword from the codebook. There is no better way.

Longer messages

Usually we want to send messages of more than 10 bits. We could split a longer message into blocks of 10 info bits and send them across the noisy link. Using the soft decision decoder with its 97.5% success rate we would get about 5 bad blocks in every 200 sent. However, Shannon's theorem tells us that coding performance improves if we put more info bits into a message block. For example, coding 50 bits into 500 symbol codewords would keep the same signal bandwidth and signaling rate, but would have a higher success rate for each block of 50 info bits. The optimum performance would be obtained if we were able to encode the entire long message into one giant codeword. I'll discuss this in more detail later.

Longer codes

Now let's put one aspect of Shannon's theorem to a quick test. Encoding 10 info bits into 100 symbol random codes gives a 97.5% success rate with my chosen noise level. Will encoding 10 info bits into a longer random code perform even better?

I have a codebook of 500 symbol random codes (minimum distance 224) and I modify my demonstration to use this. It is now transmitting 100 symbols per second, the noise power is 50 times higher than with the uncoded messages and 5 times higher than the already quite successful 100 symbol codes. When I put this through several thousand trials, the success rate increases to 98.1%. An even longer code using codewords of 5000 symbols produces 98.5% success. As you can see, using longer and longer codes reduces the error rate but in diminishing steps. In principle though, we can make the success rate approach as close to 100% as we like, even with this extremely noisy demonstration signal, so long as we are able to use ever longer codes with their associated wider bandwidth.

The crux

The mystery which puzzled us is that performance increases even though the S/N of the individual symbols is going down. Actually we can see now that the symbol S/N doesn't matter at all - the optimum decoder never looks at a symbol, it is comparing whole messages (words).

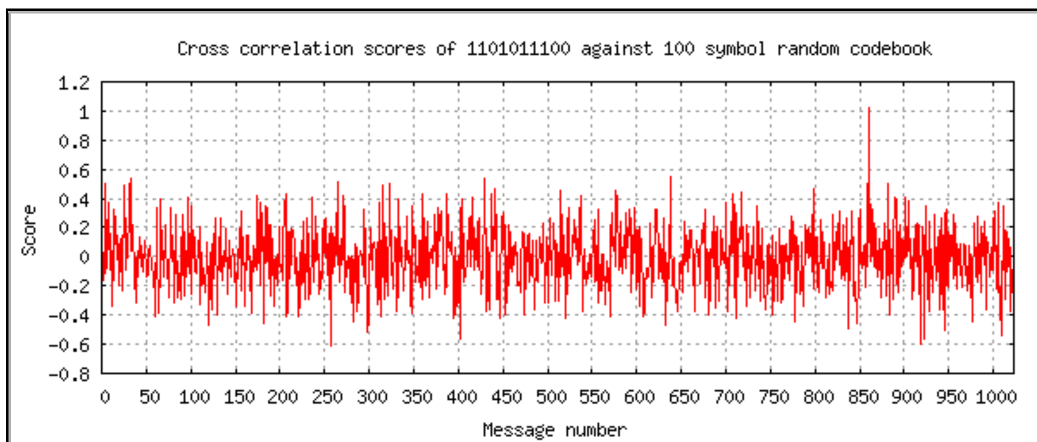
The decoder is looking for whole codewords using cross correlation over the entire message and *the noise bandwidth that's relevant for this is the inverse of the whole message duration*. In other words, the cross correlator is averaging the noise over the full message duration. This noise bandwidth is

constant regardless of the coding used or indeed whether we send coded or send uncoded. For my demonstrations, the relevant noise bandwidth is 1/(5 seconds) = 0.2Hz.

By encoding my 10 bit messages we are making all the 1024 possible messages more distinct from one another, increasing their mutual distance, when viewed against the noise in the narrow bandwidth of the inverse message duration.

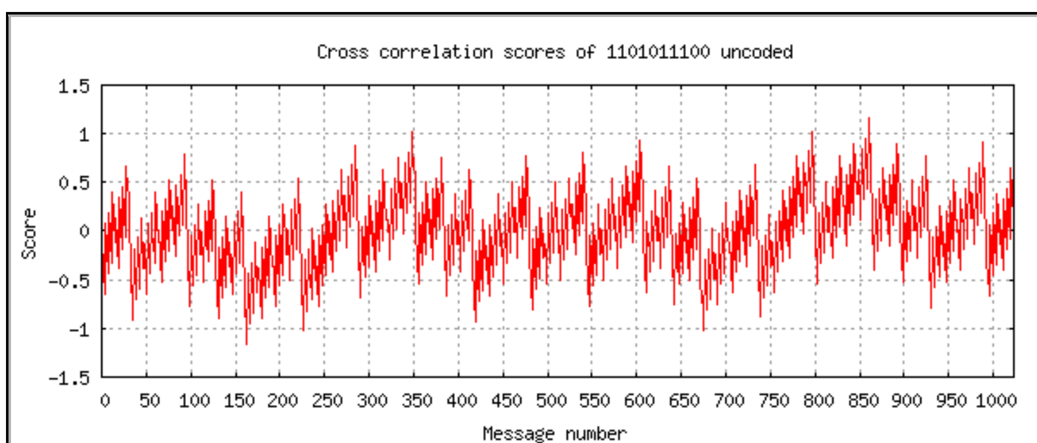
I can demonstrate this most clearly as follows. I will present a noisy received message to the decoder and plot on a graph the cross correlation with each of the 1024 codewords in the codebook. I'll number the codewords 0 to 1023 for the horizontal axis and my transmitted message 1101011100 is entry number 860 in the codebook.

First I will send it with my 100 symbol random code:



The wanted message, number 860, stands out clearly and none of the other codewords come anywhere close to a similar score.

Next, I repeat the message but this time it is sent uncoded and I make the decoder test the cross correlation with all 1024 uncoded bit patterns:



The patterning that's apparent here occurs because all of the 10 bit messages significantly overlap their bit patterns with other messages. For example for each message there are 10 others that differ by only one bit and 45 that differ by two bits. Message 860 still wins in this particular run but only just and it doesn't take much noise to displace it which is why 34.5% of uncoded messages fail.

I hope you can see clearly that it actually doesn't matter how much bandwidth the coded transmission takes up because the noise which the decoder is faced with is the noise in the fixed narrow bandwidth of the inverse message duration. We are free to use as much bandwidth as we like to make a set of codewords which differ from each other as much as possible.

We have also seen that the strength of the code increases as we encode more bits at a time. We can encode 10 info bits into 100 symbols for 5 second transmission, or 20 info bits into 200 symbols for 10 second transmission, and so on. In each case the bandwidth is the same but with every increase of codeword length, the noise bandwidth in the decoder's cross correlation decreases in inverse proportion to the increase of codeword length.

Summary

The longer the code the more symbol errors we get, but so long as the signal exceeds a certain critical signal to noise ratio, the greater error correcting strength of the longer code overcomes the extra errors to achieve a net gain in performance. The strength of the code arises because the codewords are chosen to be as different as possible from each other and the decoder simply looks for the closest one. The longer the codewords, the more different they can be from each other, and the more noise is needed to make the decoder choose the wrong one.

I have demonstrated this with codes made of random numbers which are actually quite strong but practically impossible to use. Practical codes use mathematics so that the codeword can be calculated from the message bits and the decoder will in some way reverse that calculation in a manner that arrives at the closest guess. A well designed mathematical code will employ codewords which are as equally separated from one another as possible. With a random code that happens naturally as the codewords are made longer (a consequence of the law of large numbers).

The language of forward error correction often refers to correcting symbol errors, but really the decoder is correcting whole words. This is made clear in the case of the soft decision decoder which never makes a decision about the individual symbols.

That concludes my explanation of the fundamental principle of error correcting codes. I hope you can see that the basic idea is very simple. All the difficulty of this subject comes from the terrible mathematical complexity required to invent strong and practical codes.

Additional Notes

Some further reading which elaborates on the above discussion. I will introduce some terminology that you are sure to encounter if you delve further into this topic. A little bit of mathematics unavoidably creeps in here but nothing very hard.

Code rate

My 10 information bits were replaced with 100 code symbols, a ratio of 10/100 or 0.1 as a decimal fraction. This is the 'code rate': the number of information bits per code symbol and is usually given the symbol R. For binary signaling the code rate lies between zero and one, where one corresponds to uncoded signals. A lower rate means more symbols are transmitted per info bit and that means a wider bandwidth and generally a stronger code (meaning a higher success rate for a given signal to noise ratio and message duration). My 500 symbol random code has rate R = 0.02 and my 5000 symbol random code has R = 0.002.

Note that despite the name, code rate says nothing about how fast the symbols are being sent - that

would be the 'symbol rate': so many symbols per second. We also have the 'data rate' which is the number of info bits being encoded per second. The two are related by the code rate R,

data rate = R * symbol rate

Note that a smaller code rate actually means a faster symbol rate for a given information rate.

Block codes

Block codes are what I've been demonstrating in this article. The essence of a block code is that some number of info bits are encoded, sent, and decoded, as a self-contained unit without reference to previous blocks or following blocks. A long message might be split into convenient blocks before transmission, then reassembled at the far end. Blocks are usually a fixed size for a particular coding scheme. Sometimes the entire message is encoded into a single large block (which in fact is the most effective way to code a message).

A block code is usually denoted by (n, k, dmin) where k is the number of info bits going into the encoder and n is the number of encoded symbols coming out. dmin is the minimum distance of the code: the minimum number of symbols which differ between any two codewords in the codebook.

The code rate R is therefore equal to k/n. My 100 symbol random code would be notated as (100, 10, 38) and my 5000 symbol code as (5000, 10, 2423).

The number of possible different messages which can be sent by a (n, k, dmin) block code is 2^k, ie 2 to the power of k. That is the size of the codebook. The number of possible received words when using hard decision symbol detection is 2^n and the code is guaranteed to be able to fix up to floor(0.5 * (dmin - 1)) symbol errors.

The notation may often omit the minimum distance and then it is just written as (n, k).

You will sometimes see square brackets used instead. This indicates that the block code is a of a particular type known as a linear block code.

Practical codes

Random codes are very powerful but not used in practice because they are impossible to decode efficiently. Precisely because the codes are random, there is no systematic way for a decoder to choose the nearest codeword other than by a brute force comparison of the received word with the entire codebook.

The error correcting strength of a code depends on its minimum distance - the number of symbols that differ between the closest two codewords. Long random codes work well because they naturally tend to have a high minimum distance for a given length of codeword.

Since Shannon announced his theorem, mathematicians have spent several decades trying to find practical codes which approach the capability of an average long random codebook. It is fair to say that so far they haven't done very well. As far as I know only one method of coding, LDPC (low density parity check), comes anywhere close. However, there is a trick often employed: using a cascade of two codes. The idea is you take the encoded symbols produced by one code and present them as input to a second stage of coding. The first coding stage is called the outer code and the second is called the inner code. At the receiver, the inner code is decoded first and its output provides the input symbols to the outer decoder. By careful selection of inner and outer coding schemes it becomes possible to approach closer to the theoretical performance of a long random code than can be

achieved with any single stage of coding.

Another so-called 'capacity approaching' technique which has become very common also involves two different coding schemes, but instead of cascading them, they are run in parallel. Typically the info bits are presented to two different coders and the output symbols of both coders are merged and transmitted. At the receiver, the symbols are separated and fed to their respective decoders. The trick then is for the two decoders to compare notes - when they disagree on the decoded message, they exchange information about the likelihood of their tentative decodes and they try again. This continues until they both converge to the same choice of message. For some reason vaguely related to boosting the power of internal combustion engines, this technique is called 'Turbo coding'. I prefer the analogy of two students trying to answer a difficult question. Neither knows the answer for sure but if they are allowed to confer, each will correct to some extent the limited knowledge of the other and together they have a better chance of getting the right answer.

Of course, combining two coding schemes like this doesn't allow you to go beyond the limits proved by Shannon, but it does allow you to get closer than any practical single stage of coding.

Eb/N0

This is a very useful way to refer to the signal/noise ratio of a signal, as measured at the receiving antenna port, or more conveniently, somewhere further along the receiver's signal processing chain but before any demodulation or decoding. It is usually pronounced E B over N zero.

Eb is the signal energy per information bit. This is simply the total energy of the message signal (average received power times message duration) divided by the number of info bits being conveyed.

N0 is the noise power in a 1Hz bandwidth at the signal frequency. This is often called the one-sided noise spectral density.

The ratio Eb/N0 can be thought of as the signal/noise ratio in the bandwidth of an information bit.

Note that both Eb and N0 are independent of the coding scheme being used (if any), and are independent of the modulation method (if any). It doesn't involve the signal bandwidth either. This makes the ratio Eb/N0 extremely useful for fair comparison of widely different coding and modulation schemes and symbol rates. You'll see Eb/N0 used everywhere so it's well worth getting to know it.

In all of my demonstration runs, Eb has been constant because every trial used 10 info bits, sent with the same amplitude, over the same 5 second duration. Likewise N0 has been constant because I've used the same noise level throughout.

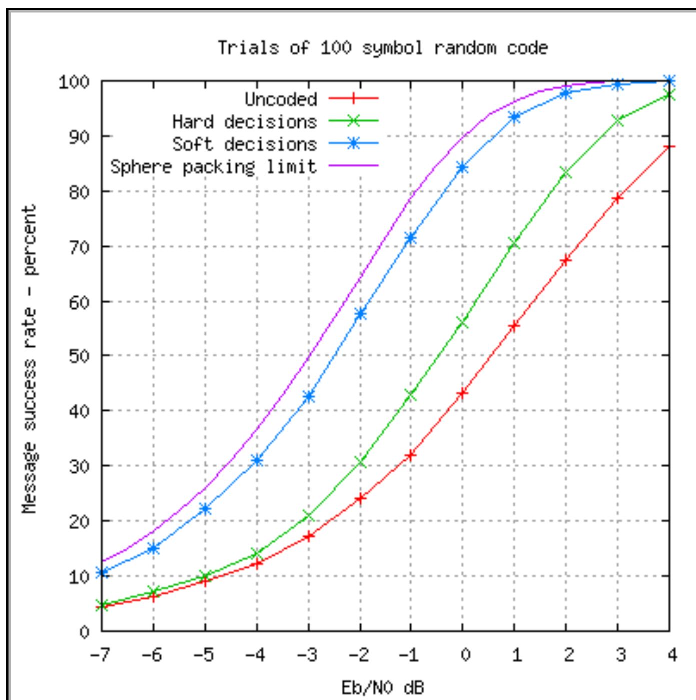
The Eb/N0 ratio is usually given in dB, and for these demonstrations Eb/N0 has been +1.87 dB. Earlier I mentioned that Shannon's theorem proved that longer codes give better performance but only when the signal/noise ratio exceeds a certain level. That critical level corresponds to an Eb/N0 of -1.59dB when using soft decision decoding (Euclidean distance), or +0.37dB when using hard decision decoding (Hamming distance). This limit to Eb/N0 is often called the 'Shannon limit'. If the Eb/N0 of your received signal is above the Shannon limit, you can make the communication as reliable as you like by using a long enough code. If the signal is weaker than the Shannon limit, you can initially obtain some improvement by using a code which is not too long, but with further increase of the code length you reach a point where performance starts to deteriorate.

Coding gain

My original uncoded messages achieved a 65.5% success rate. We can enquire by how much we

need to increase the signal (or reduce the noise) to match the 97.5% success rate of the 100 symbol encoded messages. After some trials with a range of signal levels, it turns out that a 4.0dB increase in signal strength of the uncoded transmission is necessary to achieve 97.5% success rate. We would then say that this 100 symbol random code with soft decision decoding, has a 'coding gain' of 4dB over uncoded messages. The 500 symbol code has a coding gain of 4.5dB, and the 5000 symbol code is only a little better at about 4.8 dB.

By running my 100 symbol code with various levels of noise, I produced the plot below which shows the performance in terms of message success rate for a range of Eb/N0.



From the graph we can immediately read off the coding gain by looking at the horizontal distance in dB between the curves for some particular message success rate. For example, hard decisions give a 70% success rate at Eb/N0 of +1dB and that same success rate occurs with soft decisions at about -1.1dB so we can say that soft decisions have a 2.1dB coding gain over hard decisions. Inspection of the plot shows that soft decisions give about a 2dB gain over hard decisions right across the Eb/N0 range. This 2dB soft/hard gain is typical of all codes.

We can also read off the coding gain of soft decisions versus uncoded messages. This coding gain varies with Eb/N0, being about 3dB at very low signal/noise ratio and increasing when the signal/noise ratio is higher.

For any mention of coding gain to be meaningful, you have to say what Eb/N0 or signal/noise ratio you are looking at, and also what success rate or error rate you are referring to.

You'll also notice I've plotted a mysterious curve called the sphere packing limit. I'll say more about this later but it shows the theoretical best possible performance of a (100, 10) code. My random (100, 10, 38) code is apparently able to get within 0.5dB of this limit.

Error rate

I've used the message success rate in this article as a kind of figure of merit of a code and

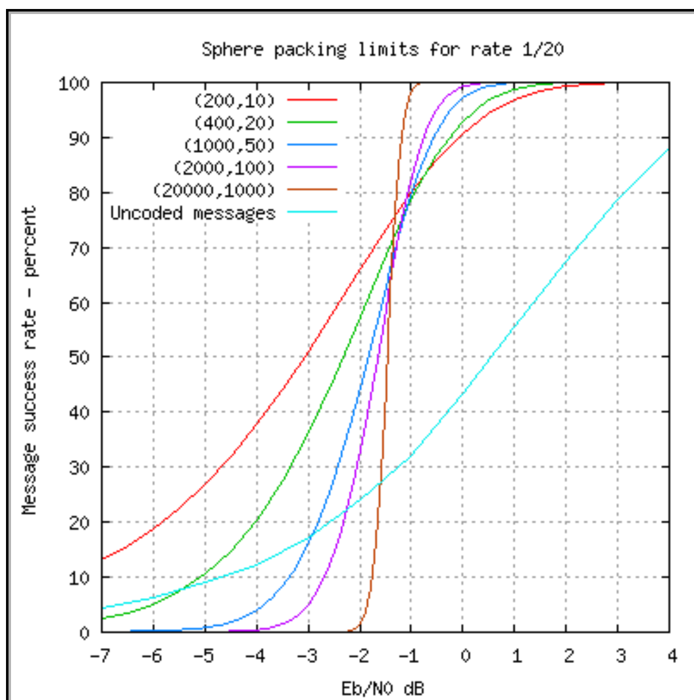
communication link. In the error correction coding literature you'll often see references to the probability that a message is received incorrectly, it is usually referred to as Pw, the 'word error probability'. Pw lies between zero and one. When using hard decisions you'll also come across an error probability for the code symbols themselves, often written as Ps or Pe, the 'symbol error probability'. Ps lies between 0 and 0.5 because even if the symbols are totally lost in the noise, on average half of them will be detected correctly just by chance.

When using soft decisions, there is no symbol error probability because the decoder never makes decisions on the individual symbols, only on the received message as a whole, so with soft decisions we only have a Pw.

Block code performance

With some calculation we can work out the theoretical best possible performance of a (n, k) block code and I'll do this to demonstrate some of the features of Shannon's theorem.

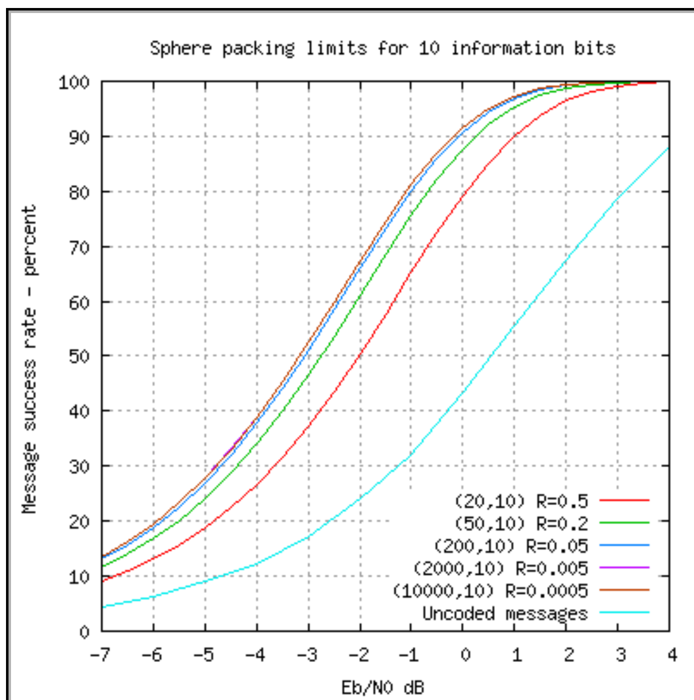
Codewords can be thought of as an array of n symbol amplitudes which are either +1 or -1. A noisy received word is also an n element array of amplitudes which can take on any value thanks to the noise. We can regard these arrays as n dimensional vectors which identify points in an n dimensional space. The minimum Euclidean distance that the soft decoder is looking for, is simply the straight line distance between the point corresponding to the received word and the nearest codeword point. We can try to imagine a sphere around each codeword point and allow these spheres to expand until they are just touching. A received word will decode to the codeword associated with the center of the sphere that it lands in. We might intuitively see that the best performance of a code will be obtained if all the spheres have the same size with the codewords being distributed in some sort of regular manner. With a (n,k) code, we have 2^k different possible messages and therefore 2^k spheres to fit into our n dimensional space. With this geometrical view of the decoding process we can draw on some complicated formulas from the mathematics of sphere packing to work out the best possible code performance for various values of n and k and the code rate k/n.



Above I plot the performance of a range of codes which all have code rate R = k/n = 0.05, so we are looking at how the performance varies with block size and signal to noise ratio Eb/N0 for a constant rate.

As the block size increases the graph steepens, the 'cliff effect' becomes more pronounced, and eventually with an infinite block size we would reach a vertical line at Eb/N0 = -1.59 dB, the Shannon limit. At that extreme, a signal slightly below -1.59 dB will have no chance of decoding but a signal just above that limit will always decode perfectly. So long as the signal is above the Shannon limit we can improve performance at some given code rate by encoding more information bits into larger codewords.

Now let's look at how performance varies with the code rate.



The plots start with rate R = 0.5 and performance improves across the full range of Eb/N0 as we reduce the rate down to 0.0005 (which is about the limit for my sphere packing calculator program). However, most of the gain from reducing the code rate occurs as we go down to R=0.1 and further rate reduction gives very little extra coding gain.

It is clear from these plots that we can improve the performance by increasing the block size for a given code rate R = k/n by increasing both n and k in the same proportion. This is sometimes called increasing the 'complexity' of the code. Or, we can reduce the code rate by increasing the codeword size n while keeping the number of info bits k the same. Increasing the block size is generally the best solution because the bandwidth of the transmission stays the same. Reducing the code rate means we have to use a higher symbol rate and therefore suffer a higher bandwidth if we want to send the same information in the same time.

Matched filtering and MFSK

I described how the soft decision (Euclidean distance) decoder performs a cross correlation of the received signal with each codeword of the codebook in order to select the closest match. Matched filtering is another way to describe the process of cross correlation. With matched filtering we

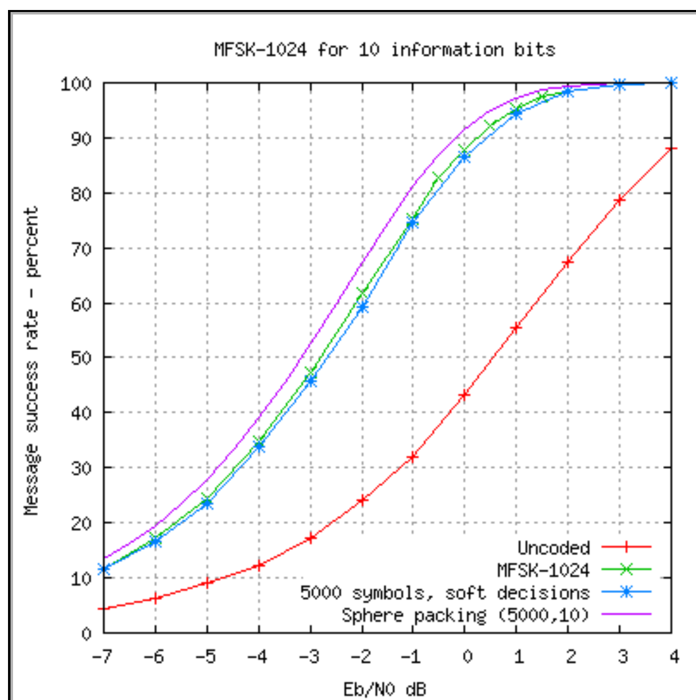
implement the pattern matching process in the frequency domain instead of the time domain (series of amplitude samples). It's exactly the same process, but described in different terms and implemented differently. The cross correlation with a codeword is replaced by a matched filter in which the filter response is the Fourier transform of the codeword. The end result is the same: an output from the filter which measures how close the input signal is to the codeword.

The decoder for my 10 info bit messages can thus be thought of as a bank of 1024 matched filters with each filter tuned to the frequency domain representation of its associated codeword. The decoder presents the received signal to the filter bank and selects the filter which shows the highest output.

This suggests an alternative way to encode the 10 bits of info in the message: associate each of the 1024 possible messages with a sine or cosine wave of a particular frequency and transmit that for the entire 5 second message duration. The encoder needs a table of 512 frequencies, each frequency can be sent as a sine or a cosine to give 1024 possible messages.

Receiving and decoding is quite simple in this case: just run the received signal through a Fourier transform arranged such that the Fourier output bins correspond to the code frequencies. Each Fourier bin contains two independent amplitudes, one for a cosine and the other for a sine. This is called coherent MFSK but it isn't really fundamentally any different to the transmission of strings of binary symbols that I've been demonstrating. The codewords in this case now consist of strings of sinusoidal amplitudes instead of strings of binary amplitudes and that's the only difference. The encoding and soft decision decoding is the same, it is merely a different codebook.

Putting this to the test with a large number of trials across a range of Eb/N0 produces the result shown below.



For comparison I also plot the performance of my 5000 symbol random codewords and the uncoded 10 bit messages. The MFSK sinusoids are actually represented by 5120 amplitude samples so they behave as a (5120,10) code. We can see that MFSK performs about the same as a good random code of similar block size and similar code rate. Both are getting within about 0.5dB of the sphere packing limit.

The reason high order MFSK is not used in practice is because of the sheer number of frequencies which need to be used for realistic block sizes. For example a 30 bit message would need a bank of 2^30 = 1073741824 filters. However, low order MFSK is used in several commonly used radio applications and can also be used as the 'inner' code of a cascaded forward error correction scheme.

List decoding

Why not arrange the decoder to output not just the message with the closest matching codeword, but a short list from say the nearest 10 or so codewords? Then it's up to the operator of the decoder to decide which if any is the correct decode of the signal. This can work well in some applications such as amateur radio. The genuine message is easily seen against the others which are just random strings of characters.

Does this mean that the combination of operator and list decoder can somehow go beyond the Shannon sphere packing limits. Well, no. The information content of the message is reduced. Suppose the code scheme is handling 7 bit ASCII characters, say 5 characters at a time to make 35 info bit blocks with a total of 2^35 = 34359738368 different possible messages. But the operator is expecting say a callsign, perhaps a letter followed by a digit, followed by three more letters. There are just 4569760 possibilities for callsigns in that format, which represents only log2(4569760) = 22.1 bits. Effectively Eb has increased because the total signal energy is now divided into 22.1 bits instead of 35 bits.

As described above, list decoding is no use for sending arbitrary information because the operator has no way to identify the real decode in the list. But if we append a CRC to the info payload before encoding, the operator can use the CRC to select the correct one from the candidate decodes in the list. Automating this, it becomes a cascade of two coding schemes. Long lists can be used (several thousand or even millions) to good advantage with the weakest signals and up to 3dB of extra coding gain can be obtained.

Spread spectrum?

Spread spectrum is not the same as error correcting coding, they are totally different things but are sometimes confused. This is important because in some jurisdictions, spread spectrum operation is not allowed for amateur radio transmissions but error correcting coding certainly is.

Yes, both produce a transmitted signal which occupies a much wider bandwidth than the information rate would require, but that's about the only similarity.

The benefit of spread spectrum is that interfering co-channel signals are scattered into random noise by the receiver's de-spreading. At the same time, disturbances due to multi-path effects are reduced.

We can note that the wider bandwidth of low rate error correcting codes will also benefit in the same way as spread spectrum. We have seen that there may be little extra coding gain obtained from reducing the code rate from say R=0.1 to R=0.01 but the resulting 10 times wider bandwidth may increase immunity to interfering signals and multi-path distortions.

Amateur radio

I've demonstrated that forward error correction works much better if you use a large block size, in other words you encode a lot of information bits into each block. It is better to encode 1000 bits into one block than encode ten blocks each with 100 bits. Large blocks are used in most applications of forward error correction. For example deep space communications with missions such as Cassini and Galileo,

encode blocks of information that are effectively around 10,000 bits. This is ideal if you have great volumes of scientific and image data to convey.

A problem for amateur communications is that most transmissions are relatively short. A typical message might include a callsign, location and signal report, perhaps 100 bits at most. Adding some long winded salutation might stretch that out to 200 or even 300 bits on a Sunday morning. This means that amateur transmissions can't often take advantage of the coding gain of very long blocks. Once you have encoded the whole message into one block, your only further option is to reduce the code rate which eventually brings you to a bandwidth limit.

However, there is one thing that amateur communications have in their favour - a willingness to tolerate higher word error rates. A deep space link or a WiFi connection will expect word error rates far better than 1 in 10,000 whereas the radio amateur will consider a 1 in 100 error rate as pretty solid reception and will be quite happy even if 1 in 10 messages fail to decode correctly.

So a deep space link might use an advanced turbo or LDPC code with a 10,000 symbol block size to operate within 0.5dB of the Shannon limit at a word error rate below 1 in 10,000. The radio amateur will also receive signals down to 0.5dB of the limit but with much smaller block size and an error rate in the order of 1 in 10.

Indeed, the powerful large-block codes used in deep space will not function at all when the signal is say 0.5dB below the Shannon limit. But the radio amateur using a smaller block code can still have some success at even weaker signals. For example from the graphs plotted earlier, you can see that the 100 symbol random code will still succeed 40% of the time at Eb/N0 = -3dB which is 1.4dB below the Shannon limit. It is breaking no laws to do so - it is within the sphere packing limit. The Shannon limit is just the sphere packing limit reached as the block size tends to infinity.

Formulas, Links

I've bottled up a lot of mathematics while preparing this article and placed it all in a ~~useful~~ essential formulas page.

"It might be said that the algebra involved in several places is unusually tedious" ...Claude Shannon, 1959, in his paper Probability of Error for Optimal Codes in a Gaussian Channel in which formulas for error rates are derived from the geometry of sphere packing in n dimensions.

It is fun and very instructive to make up your own programs to demonstrate baseband signaling with uncoded and coded messages.

Below are links to text files listing the random codebooks that I've used in this article. Each file is two columns, the message number 0 to 1023 in the first column, and its corresponding random codeword in the second column.
(100,10,38)
(500,10,224)
(5000,10,2423)
For convenience here is a codebook for uncoded 10 bit transmissions.
(10,10,1)
This allows you to run uncoded messages through exactly the same encode/decode engine as the coded messages.

Paul Nicholson, December 2016. http://abelian.org/