The Welch-Berlekamp Algorithm for Correcting Errors in Data

#berlekamp-welsh #error correcting codes #finite fields #graphics #image analysis #mathematics #polynomials #programming #python
2015-09-07

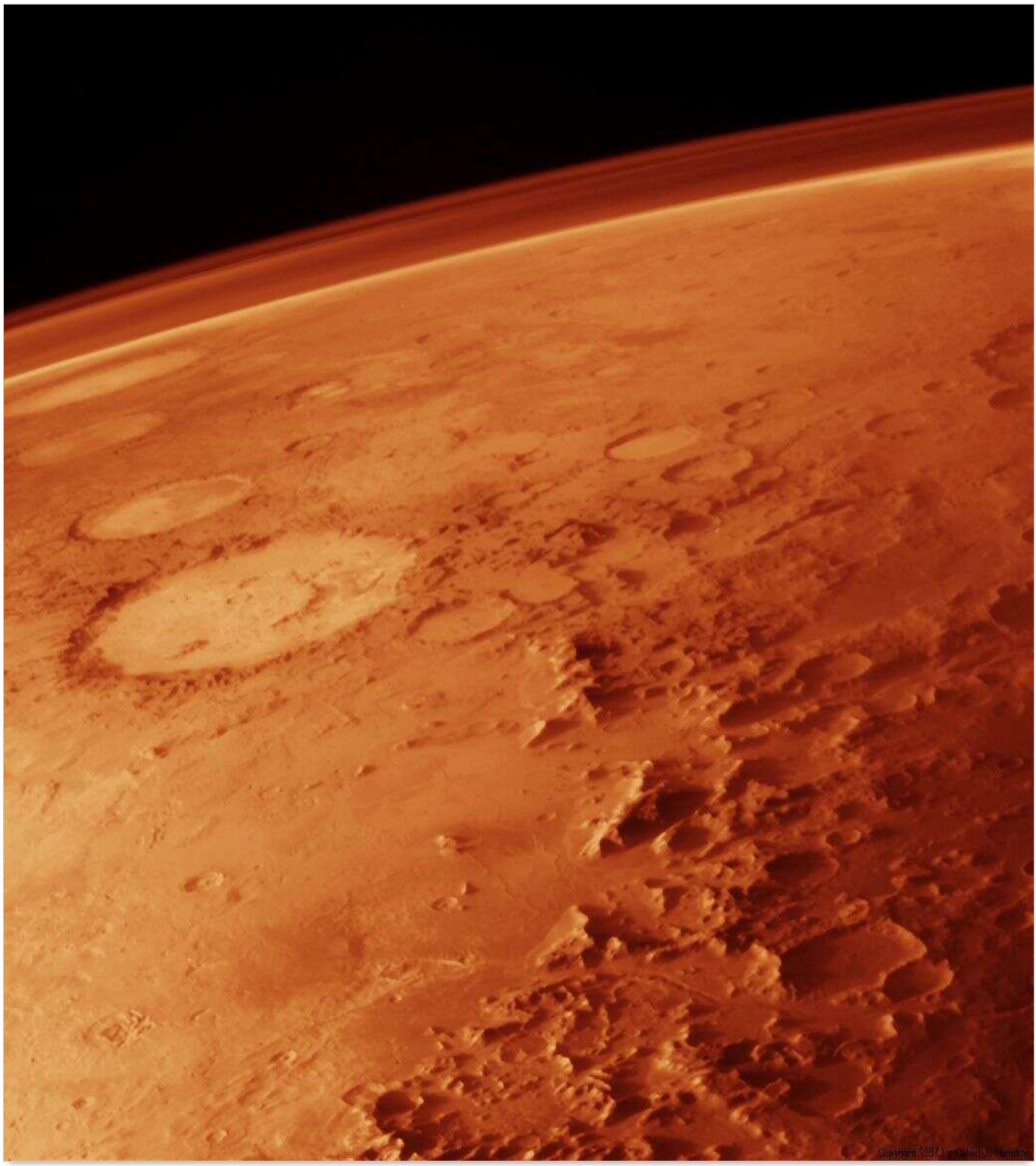
This article was ported from my old Wordpress blog here, If you see any issues with the rendering or layout, please send me an email.

In this post we'll implement Reed-Solomon error-correcting codes and use them to play with codes. In our last post we defined Reed-Solomon codes rigorously, but in this post we'll focus on intuition and code. As usual the code and data used in this post is available on this blog's Github page.

The main intuition behind Reed-Solomon codes (and basically all the historically major codes) is

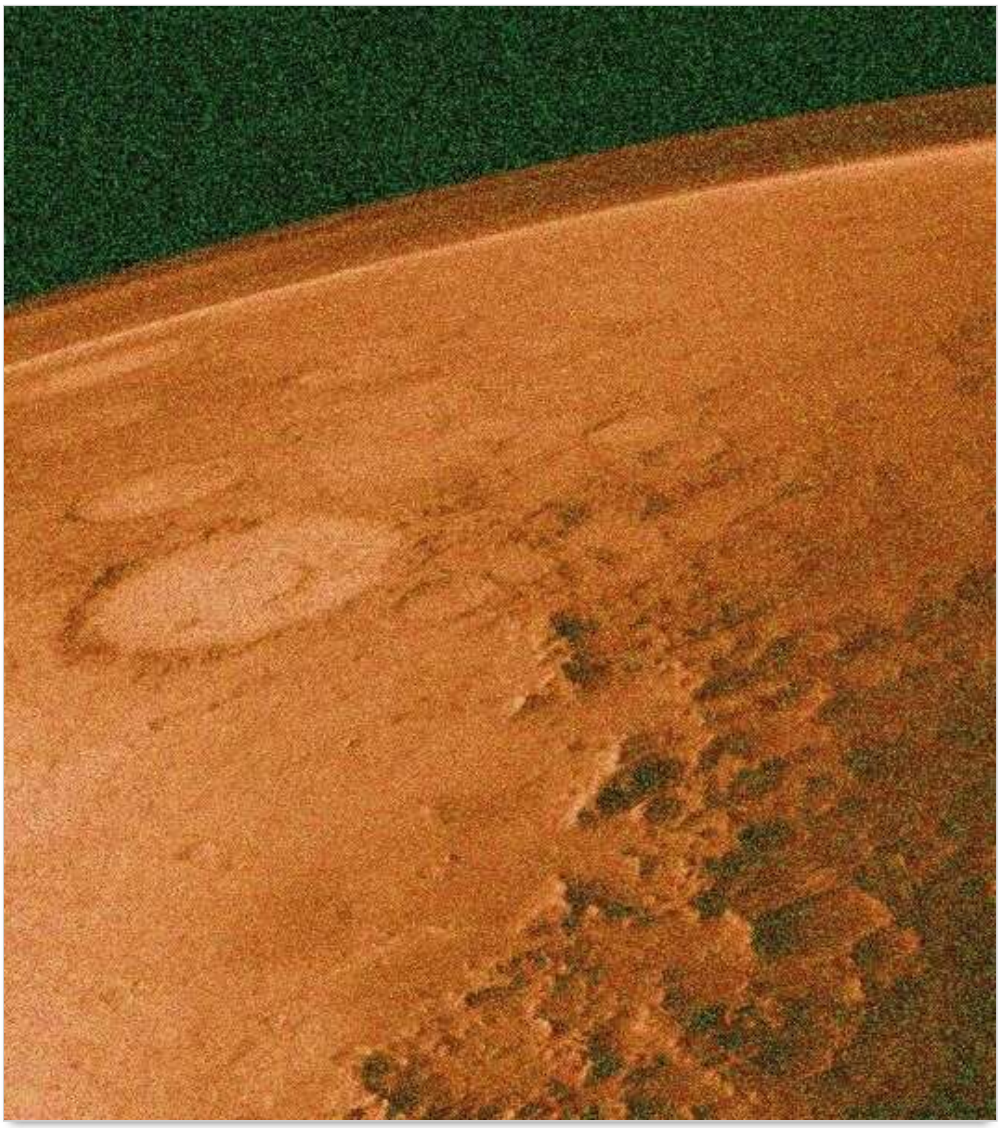
Error correction is about adding redundancy, and polynomials are a really efficient way to do that.

Here's an example of what we'll do in the post. Say you have a space probe flying past Mars taking photographs like this one



Courtesy of NASA's Viking Orbiter.

Unfortunately you know that if you send the images back to Earth via radio waves, the signal will get corrupted by cosmic something-or-other and you'll end up with an image like this.



How can you recover from errors like this? You could do something like repeat each pixel twice in the message so that if one is corrupted the other will get through. But still, every now and then both pixels in a row will be corrupted and it's twice as inefficient.

The idea of error-correcting codes is to find a way to encode a message so that it adds a lot of redundancy without adding too much extra information to the message. The name of the game is to optimize the tradeoff between how much redundancy you get and how much longer the message needs to be, while still being able to efficiently decode the encoded message.

A solid technique turns out to be: use polynomials. Even though you'd think polynomials are too simple (we teach them starting in the 7th grade these days!) they turn out to have remarkable properties. The most important of which is:

if you give me a bunch of points in the plane with different x coordinates, they *uniquely* define a polynomial of a certain degree.

This fact is called *polynomial interpolation*. We used it in a previous post to share secrets, if you're interested.

What makes polynomials great for error correction is that you can take a fixed polynomial (think, the message) and "encode" it as a list of points on that polynomial. If you include enough, then you can get back the original polynomial from the points alone. And the best part, for each two additional points you include above the minimum, you get resilience to one additional error *no matter where it happens in the message*. Another way to say this is, even if some of the points in your encoded message are wrong (the numbers are modified by an adversary or random noise), as long as there aren't too many errors there is an algorithm that can recover the errors.

That's what makes polynomials so much better than the naive idea of repeating every pixel twice: once you allow for three errors you run the risk of losing a pixel, but you had to double your communication costs. With a polynomial-based approach you'd only need to store around six extra pixels worth of data to get resilience to three errors that can happen anywhere. What a bargain!

Here's the official theorem about Reed-Solomon codes:

Theorem: There is an efficient algorithm which, when given points $(a_1, b_1), \ldots, (a_n, b_n)$ with distinct a_i has the following property. If there is a polynomial of degree d that passes through at least $n/2 + d/2$ of the given points, then the algorithm will output the polynomial.

So let's implement the encoder, decoder, and turn the theorem into code!

Implementing the encoder

The way you write a message of length k as a polynomial is easy. Pick a large prime integer p and from now on we'll do all our arithmetic modulo p. Then encode each character $c_0, c_1, \ldots, c_{k-1}$ in the message as an integer between 0 and $p-1$ (this is why p needs to be large enough), and the polynomial representing the message is

$$m(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{k-1} x^{k-1}$$

If the message has length k then the polynomial will have degree $k-1$.

Now to encode the message we just pick a bunch of x values and plug them into the polynomial, and record the (input, output) pairs as the encoded message. If we want to make things simple we can just require that you always pick the x values $0, 1, \ldots, n$ for some choice of $n \leq p$.

A quick skippable side-note: we need p to be prime so that our arithmetic happens in a field. Otherwise, we won't necessarily get unique decoded messages.

Back when we discussed elliptic curve cryptography (ironically sharing an acronym with error correcting codes), we actually wrote a little library that lets us seamlessly represent polynomials with "modular arithmetic coefficients" in Python, which in math jargon is a "finite field." Rather than reinvent the wheel we'll just use that code as a black box (full source in the Github repo). Here are some examples of using it.

```
>>> from finitefield.finitefield import FiniteField
```

```
>>> F13 = FiniteField(p=13)
>>> a = F13(7)
>>> a+9
3 (mod 13)
>>> a*a
10 (mod 13)
>>> 1/a
2 (mod 13)
```

A programming aside: once you construct an instance of your finite field, all arithmetic operations involving instances of that type will automatically lift integers to the appropriate type. Now to make some polynomials:

```
>>> from finitefield.polynomial import polynomialsOver
>>> F = FiniteField(p=13)
>>> P = polynomialsOver(F)
>>> g = P([1,3,5])
>>> g
1 + 3 t^1 + 5 t^2
>>> g*g
1 + 6 t^1 + 6 t^2 + 4 t^3 + 12 t^4
>>> g(100)
4 (mod 13)
```

Now to fix an encoding/decoding scheme we'll call k the size of the unencoded message, n the size of the encoded message, and p the modulus, and we'll fix these programmatically when the encoder and decoder are defined so we don't have to keep carrying these data around.

```
def makeEncoderDecoder(n, k, p):
   Fp = FiniteField(p)
   Poly = polynomialsOver(Fp)

   def encode(message):
      ...

   def decode(encodedMessage):
      ...

   return encode, decode
```

Encode is the easier of the two.

```
def encode(message):
   thePoly = Poly(message)
   return [(Fp(i), thePoly(Fp(i))) for i in range(n)]
```

Technically we could remove the leading `Fp(i)` from each tuple, since the decoder algorithm can assume we're using the first n integers in order. But we'll leave it in and define the decode function more generically.

After we define how the decoder should work in theory we'll run through a simple example step by step. Now on to the decoder.

The decoding algorithm, Berlekamp-Welch

There are a lot of different decoding algorithms for various error correcting codes. The one we'll implement is called the Berlekamp-Welch algorithm, but before we get to it we should mention a much simpler algorithm that will work when there are only a few errors.

To remind us of notation, call k the length of the message, so that $k-1$ is the degree of the polynomial we used to encode it. And n is the number of points we used in the encoding. Call the encoded message M as it's received (as a list of points, possibly with errors).

In the simple method what you do is just randomly pick k points from M, do polynomial interpolation on the chosen points to get some polynomial g, and see if g agrees with most of the points in M. If there really are few errors, then there's a good chance the randomly chosen points won't have any errors in them and you'll win. If you get unlucky and pick some points with errors, then the g you get won't agree with most of M and you can throw it out and try again. If you get *really* unlucky and a bad g does agree with most of M, then you just run this procedure a few hundred times and take the g you get most often. But again, this only works with a small number of errors and while it could be good enough for many applications, don't bet your first-born child's life on it working. Or even your favorite pencil, for that matter. We're going to implement Berlekamp-Welch so you can win someone else's favorite pencil. You're welcome.

Exercise: Implement the simple decoding algorithm and test it on some data.

Suppose we are guaranteed that there are exactly $e < \frac{n-k+1}{2}$ errors in our received message $M = (a_1, b_1, \dots, a_n, b_n)$. Call the polynomial that represents the original message P. In other words, we have that $P(a_i) = b_i$ for all but e of the points in M.

There are two key ingredients in the algorithm. The first is called the *error locator polynomial*. We'll call this polynomial $E(x)$, and it's just defined by being zero wherever the errors occurred. In symbols, $E(a_i) = 0$ whenever $P(a_i) \neq b_i$. If we knew where the errors occurred, we could write out $E(x)$ explicitly as a product of terms like $(x - a_i)$. And if we knew E we'd also be done, because it would tell us where the errors were and we could do interpolation on all the non-error points in M.

So we're going to have to study E indirectly and use it to get P. One nice property of $E(x)$ is the following

$$b_i E(a_i) = P(a_i)E(a_i),$$

which is true for *every* pair $(a_i, b_i) \in M$. Indeed, by definition when $P(a_i) \neq b_i$ then $E(a_i) = 0$ so both sides are zero. Now we can use a technique called *linearization*. It goes like this. The product $P(x)E(x)$, i.e. the right-hand-side of the above equation, is a polynomial, say $Q(x)$, of larger degree $(e + k–1)$. We get the equation for all i:

$$b_i E(a_i) = Q(a_i)$$

Now E, Q, and P are all unknown, but it turns out that we can actually find E and Q efficiently. Or rather, we can't guarantee we'll find E and Q *exactly,* instead we'll find two polynomials that have the same quotient as $Q(x)/E(x) = P(x)$. Here's how that works.

Say we wrote out $E(x)$ as a generic polynomial of degree e and $Q(x)$ as a generic polynomial of degree $e + k - 1$. So their coefficients are unspecified variables. Now we can plug in all the points a_i, b_i to the equations $b_i E(a_i) = Q(a_i)$, and this will form a *linear* system of $2e + k - 1$ unknowns (e unknowns come from $E(x)$ and $e + k - 1$ come from $Q(x)$).

Now we know that this system has *a* good solution, because if we take the true error locator polynomial and $Q = E(x)P(x)$ with the true $P(x)$ we win. The worry is that we'll solve this system and get two different polynomials Q', E' whose quotient will be something crazy and unrelated to P. But as it turns out this will never happen, and any solution will give the quotient P. Here's a proof you can skip if you hate proofs.

Proof. Say you have two pairs of solutions to the system, (Q_1, E_1) and (Q_2, E_2), and you want to show that $Q_1/E_1 = Q_2/E_2$. Well, they might not be divisible, but we can multiply the previous equation through to get $Q_1E_2 = Q_2E_1$. Now we show two polynomials are equal in the same way as always: subtract and show there are too many roots. Define $R(x) = Q_1E_2–Q_2E_1$. The claim is that $R(x)$ has n roots, one for every point (a_i, b_i). Indeed,

$$R(a_i) = (b_iE_1(a_i))E_2(a_i)–(b_iE_2(a_i))E_1(a_i) = 0$$

But the degree of $R(x)$ is $2e + k–1$ which is less than n by the assumption that $e < \frac{n-k+1}{2}$. So $R(x)$ has too many roots and must be the zero polynomial, and the two quotients are equal.

□

So the core python routine is just two steps: solve the linear equation, and then divide two polynomials. However, it turns out that no python module has any decent support for solving linear systems of equations over finite fields. Luckily, I wrote a linear solver way back when and so we'll adapt it to our purposes. I'll leave out the gory details of the solver itself, but you can see them in the source for this post. Here is the code that sets up the system

```
def solveSystem(encodedMessage):
   for e in range(maxE, 0, -1):
      ENumVars = e+1
      QNumVars = e+k
      def row(i, a, b):
         return ([b * a**j for j in range(ENumVars)] +
```

```
                    [-1 * a**j for j in range(QNumVars)] +
                    [0]) # the "extended" part of the linear system

        system = ([row(i, a, b) for (i, (a,b)) in enumerate(encodedMessage)] +
                  [[0] * (ENumVars-1) + [1] + [0] * (QNumVars) + [1]])
                    # ensure coefficient of x^e in E(x) is 1

        solution = someSolution(system, freeVariableValue=1)
        E = Poly([solution[j] for j in range(e + 1)])
        Q = Poly([solution[j] for j in range(e + 1, len(solution))])

        P, remainder = Q.__divmod__(E)
        if remainder == 0:
            return Q, E

    raise Exception("found no divisors!")

def decode(encodedMessage):
    Q,E = solveSystem(encodedMessage)

    P, remainder = Q.__divmod__(E)
    if remainder != 0:
        raise Exception("Q is not divisibly by E!")

    return P.coefficients
```

A simple example

Now let's go through an extended example with small numbers. Let's work modulo 7 and say that our message is

2, 3, 2 (mod 7)

In particular, $k = 3$ is the length of the message. We'll encode it as a polynomial in the way we described:

$$m(x) = 2 + 3x + 2x^2 (\mod 7)$$

If we pick $n = 5$, then we will encode the message as a sequence of five points on $m(x)$, namely $m(0)$ through $m(4)$.

[[0, 2], [1, 0], [2, 2], [3, 1], [4, 4]] (mod 7)

Now let's add a single error. First remember that our theoretical guarantee says that we can correct any number of errors up to $\frac{n+k-1}{2}$–1, which in this case is $[(5+3-1)/2]-1 = 2$, so we can definitely correct one error. We'll add 1 to the third point, giving the received corrupted message as

[[0, 2], [1, 0], [2, 3], [3, 1], [4, 4]] (mod 7)

Now we set up the system of equations $b_i E(a_i) = Q(a_i)$ for all (a_i, b_i) above. Rewriting the equations as $b_i E(a_i) - Q(a_i) = 0$, and adding as the last equation the constraint that the coefficient of x^e is 1, so that we get a "generic" error locator polynomial of the right degree. The columns represent the variables, with the last column being the right-hand-side of the equality as is the standard for Gaussian elimination.

```
# e0 e1 q0 q1 q2 q3
[
  [2, 0, 6, 0, 0, 0, 0],
  [0, 0, 6, 6, 6, 6, 0],
  [3, 6, 6, 5, 3, 6, 0],
  [1, 3, 6, 4, 5, 1, 0],
  [4, 2, 6, 3, 5, 6, 0],
  [0, 1, 0, 0, 0, 0, 1],
]
```

Then we do row-reduction to get

```
[
  [1, 0, 0, 0, 0, 0, 5],
  [0, 1, 0, 0, 0, 0, 1],
  [0, 0, 1, 0, 0, 0, 3],
  [0, 0, 0, 1, 0, 0, 3],
  [0, 0, 0, 0, 1, 0, 6],
  [0, 0, 0, 0, 0, 1, 2]
]
```

And reading off the solution gives $E(x) = 5 + x$ and $Q(x) = 3 + 3x + 6x^2 + 2x^3$. Note in particular that the $E(x)$ given in this solution is the true error locator polynomial, but it is not guaranteed to be so! Either way, the quotient of the two polynomials is exactly $m(x) = 2 + 3x + 2x^2$ which gives back the original message.

There is one catch here: how does one determine the value of e to use in setting up the system of linear equations? It turns out that an upper bound on e will work just fine, so long as the upper bound you use agrees with the theoretical maximum number of errors allowed (see the Singleton bound from last time). The effect of doing this is that the linear system ends up with some number of free variables that

you can set to arbitrary values, and these will correspond to additional shared roots of $E(x)$ and $Q(x)$ that cancel out upon dividing.

A larger example

Now it's time for a sad fact. I tried running Welch-Berlekamp on an encoded version of the following tiny image:

And it didn't finish after running all night.

Berlekamp-Welch is a slow algorithm for decoding Reed-Solomon codes because it requires one to solve a large system of equations. There's at least one equation for each pixel in a black and white image! To get around this one typically encodes blocks of pixels together into one message character (since p is larger than $n > k$ there is lots of space), and apparently one can balance it to minimize the number of equations. And finally, a nontrivial inefficiency comes from my implementation of everything in Python without optimizations. If we rewrote everything in C++ or Go and fixed the prime modulus, we would likely see reasonable running times. There are also asymptotically *much* faster methods based on the fast Fourier transform, and in the future we'll try implementing some of these. For the dedicated reader, these are all good follow-up projects.

For now we'll just demonstrate that it works by running it on a larger sample of text, the introductory paragraphs of To Kill a Mockingbird:

```
def tkamTest():
   message = '''When he was nearly thirteen, my brother Jem got his arm badly bro
ken at the elbow.  When it healed, and Jem's fears of never being able to play fo
otball were assuaged, he was seldom   self-conscious about his injury. His left a
rm was somewhat shorter than his right; when he stood or walked, the back of his
hand was at right angles to his body, his thumb parallel to his thigh. He   could
n't have cared less, so long as he could pass and punt.'''

   k = len(message)
   n = len(message) * 2
   p = 2087
   integerMessage = [ord(x) for x in message]

   enc, dec, solveSystem = makeEncoderDecoder(n, k, p)
   print("encoding...")
   encoded = enc(integerMessage)

   e = int(k/2)
```

```
print("corrupting...")
corrupted = corrupt(encoded[:], e, 0, p)

print("decoding...")
Q,E = solveSystem(corrupted)
P, remainder = (Q.__divmod__(E))

recovered = ''.join([chr(x) for x in P.coefficients])
print(recovered)
```

Running this with unix `time` produces the following:

```
encoding...
corrupting...
decoding...
When he was nearly thirteen, my brother Jem got his arm badly broken at the elbo
w. When it healed, and Jem's fears of never being able to play football were assu
aged, he was seldom self-conscious about his injury. His left arm was somewhat sh
orter than his right; when he stood or walked, the back of his hand was at right
angles to his body, his thumb parallel to his thigh. He couldn't have cared less,
so long as he could pass and punt.

real    82m9.813s
user    81m18.891s
sys 0m27.404s
```

So it finishes in “only” an hour or so.

In any case, the decoding algorithm is an interesting one. In future posts we’ll explore more efficient algorithms and faster implementations.

Until then!

Posts in this series:

- A Proofless Introduction to Coding Theory
- Hamming’s Code
- The Codes of Solomon, Reed, and Muller
- The Welch-Berlekamp Algorithm for Correcting Errors in Data

---

Want to respond? Send me an email, post a webmention, or find me elsewhere on the internet.

DOI: https://doi.org/10.59350/nbs03-gnh20

← Previous: The Čech Complex and the Vietoris-Rips Complex

Next: The Boosting Margin, or Why Boosting Doesn't Overfit →