When information is communicated over faulty channels we would expect some errors in transmission. I may say "four" and you may hear "fault". We humans have fairly successful strategies for correcting such errors. If you hear "twelve equals fault times three" you correct the "fault" to a "four" almost subconsciously.

In the 1950s mathematician Richard Hamming came up with an influential model of unreliable communication. Alice wants to transmit a message to Bob. Alice *encodes* her message as a sequence of symbols called a *codeword* that travel through a channel. The channel can corrupt some symbols in the codeword; they are replaced by other symbols. The objective is for Bob to *decode* Alice's message from the potentially corrupted codeword. We'll assume for now that the codeword symbols are bits (`0` or `1`).

Here is an example. Alice's message consists of a single bit. This bit might indicate an instruction (e.g, $0 = \text{BUY}$, $1 = \text{SELL}$). Its meaning is not important. The *d-repetition code* encodes this bit with d copies of it. For example, the encodings of 0 and 1 under the 3-repetition code are

$$\mathit{Encode}(0) = \texttt{000} \qquad \mathit{Encode}(1) = \texttt{111}.$$

If the channel corrupts any one of the symbols, Bob can still recover the message. The majority of the three symbols will still equal the original message. Bob can recover from up to one error introduced by the channel by taking the majority of the bits in the corrupted codeword.

$$\mathit{Decode}(y_1y_2y_3) = \text{maj}_3(y_1, y_2, y_3).$$

If the channel corrupts one more codeword symbol, Bob will decode incorrectly. There is in fact no decoding algorithm that can recover a one-bit message from (up to) two corruptions of a 3-bit codeword. We will explain why shortly. In general, the d-repetition code can decode a single bit from up to $(d-1)/2$ errors.

How should Alice encode longer messages? One possibility is for her to apply a repetition code to every bit of her message separately. Bob would decode each message block separately. The resulting encoding is still resilient to a single error. In general, Bob cannot correct more than one error. If two errors occur in the same block, the corresponding symbol will be decoded incorrectly (see Figure 1).

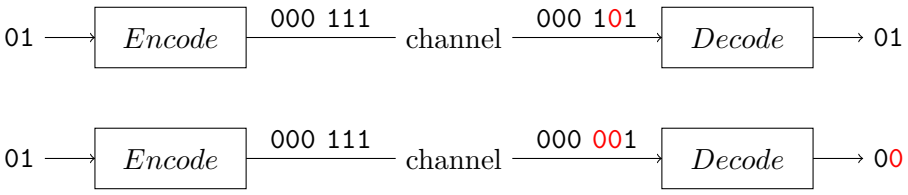


FIGURE 1: Encoding a 2-bit message with a 3-repetition code. The encoding is resilient to (a) one error but (b) not to two errors.

In general, we can make any k-bit message resilient to e errors by encoding each bit with a d-repetition code with $d = 2e+1$. The resulting codeword will have length $n = kd$. For example, if we want to tolerate four errors, the codeword will be nine times longer than the message. Can we do better?

Let's spell out the requirements. An *error-correcting code* is a pair of algorithms *Encode* and *Decode*. *Encode* takes a k-bit message and outputs an n-bit codeword. We say that the code corrects up to e errors if for every message $\mathbf{m}$ and every n-bit string $\mathbf{y}$ that differs from $\mathit{Encode}(\mathbf{m})$ in at most e places, $\mathit{Decode}(\mathbf{y})$ equals $\mathbf{m}$. In the $(2e+1)$-repetition code for k-bit messages, the codeword is $n = k(2e+1)$ bits long.

1 Codes

A few years before Hamming, mathematician Claude Shannon had an important insight about unreliable communication: The error-correcting capabilities of a code do not depend on how encoding and decoding are implemented. The only thing that matters is the set $\mathcal{C}$ of possible codewords. In the 3-repetition code applied to a single bit, the codewords are $\mathcal{C}_1 = \{000, 111\}$. The 3-repetition code applied to 2 bits shown in Figure 1 has four codewords $\mathcal{C}_2 = \{000000, 000111, 111000, 111111\}$. Like Shannon, we will overload the word "code" to also mean the set of possible codewords. Thus we will call $\mathcal{C}_1$ and $\mathcal{C}_2$ *codes* without worrying for now how encoding and decoding work.

The *distance* of a code is the smallest possible number of entries in which two (distinct) codewords differ. $\mathcal{C}_1$ has only two codewords and its distance is 3. In $\mathcal{C}_2$ some codeword pairs differ in 3 entries while others differ in 6 entries. Its distance is also 3.

Suppose Alice and Bob are using a code $\mathcal{C}$ of distance 4 to correct errors. This code must have two codewords c and c' that differ in exactly four symbols, for instance $c = 010001000$ and $c' = 100001011$. There is always some string x that is "in the middle" of the two, for instance $x = 010001011$. Bob could not possibly know whether this x was obtained by corrupting c in the first two positions or by corrupting c' in the last two. Therefore Bob cannot possibly correct from two (worst-case) errors using $\mathcal{C}$.

In general, this argument tells us that a code of distance d cannot correct $d/2$ or more errors. The best we can hope for is $(d-1)/2$. This is a negative insight, but a crucial one. It tells us that to come up with a reasonable code we must at the very least ensure that it has large distance.

The parity check code

Let's start small. Distance 1 is not very interesting. It doesn't require any encoding. At distance 2 something happens: The parity code $\mathcal{P}_n$ consisting of all n-bit strings that XOR to zero has distance 2. For example, $\mathcal{P}_3 = \{000, 011, 101, 110\}$. The code $\mathcal{P}_n$ has distance two because any two strings $c \neq c'$ that differ in one position cannot have the same parity, so they are not both in the code.

The parity code does not have any error-correction capabilities—its distance is just too small—but it can *detect* a single error. If a single bit was flipped by the channel Bob would know that an error in communication occurred by calculating the parity of all the received bits, hence the name parity check code.

The parity check code can detect a single error in communication. But so can a 2-repetition code. What is the advantage in using the parity check code?

You may think that it is impossible to answer this question without explaining how the party check code "works". But you would be wrong to think that. For the purposes of understanding the error-correction capabilities of a code, it does not matter at all how the messages are encoded. It only matters how many messages there are. The number of messages is nothing but the size $|\mathcal{C}|$ of the set of codewords. For example, $\mathcal{P}_2$ encodes four possible messages, which we can identify with the bit strings 00, 01, 10, and 11. Alice and Bob can agree in advance which codeword represents which message.

In general, a code of size 2^k encodes k-bit messages. The code $\mathcal{P}_n$ has size 2^{n-1} as it contains exactly half of all 2^n possible n-bit strings. Therefore k-bit messages are encoded by $n = (k+1)$-bit codewords. In contrast, the 2-repetition code uses up $2k$ codeword bits.

The *rate* of a code is the ratio k/n between the message length and the codeword length. It measures the (fractional) number of message symbols represented by each codeword symbol. The 2-repetition code has rate $1/2$, while the parity check code has rate $1 - 1/(k+1)$. When k is large the latter is almost twice as efficient.

Once we have figured out the rate and distance of a code we can tackle how the encoding actually works. In the parity check code we can take the first k bits of the encoding to be a copy of the message m. The last bit then equals the parity of the bits of $\mathbf{m} = m_1 \cdots m_k$:

$$\mathit{Encode}(\mathbf{m}) = (m_1, \ldots, m_k, m_1 + \cdots + m_k).$$

The sum is modulo 2. This is an example of a *systematic code*: The codeword is a copy of the message followed by some redundant bits that assist error detection or correction.

Hamming's $[7, 4, 3]$ code

Hamming discovered an interesting 7-bit code $\mathcal{H}$ for 4-bit messages:

$$Encode(\mathbf{m}) = (m_1,\ m_2,\ m_3,\ m_4,\ m_1 + m_2 + m_4,\ m_1 + m_3 + m_4,\ m_2 + m_3 + m_4).$$

This code has distance 3. Its distance can be at most 3 because the encodings of `0000` and `1000` differ in three bits (bits $1, 5, 6$). It can be checked by case analysis or computer search that there are no codewords at distance 1 or 2.

We know that a code of distance 3 cannot recover from two or more errors. Can $\mathcal{H}$ handle a single error?

Every code of distance 3 can tolerate up to one error. The decoding algorithm is simple: Round the possibly corrupted codeword $\mathbf{y}$ to the nearest true codeword and output the message that it encodes. For example, suppose Bob receives `0111110`. The nearest codeword to it is `0110110`. This is the encoding of 0110, so Bob would decode to 0110.

To understand why this algorithm works we have to reason by contradiction. Suppose Alice encoded message $\mathbf{m}$ into $\mathbf{c}$. We know that Bob's received word $\mathbf{y}$ differs from *some* $\mathbf{c}$ in at most one bit. Could Bob have decoded $\mathbf{y}$ to some other $\mathbf{c}'$? Bob picks the closest codeword to $\mathbf{y}$ so $\mathbf{c}'$ cannot differ from $\mathbf{y}$ by more than one bit. Therefore $\mathbf{c}$ and $\mathbf{c}'$ can differ in at most two bits. But no two codewords differ in at most two bits.

The same argument tells us that if $\mathcal{C}$ has distance d then correcting up to $(d-1)/2$ errors is in principle possible.

Proposition 1. *If $\mathcal{C}$ has distance d then there exist algorithms Encode and Decode such that for every message $\mathbf{m}$ and for every $\mathbf{y}$ that differs from $Encode(\mathbf{m})$ in at most $(d-1)/2$ bits, $Decode(\mathbf{y})$ equals $\mathbf{m}$.*

Decoding to the nearest codeword always works, but may not particularly practical when the code is large. Hamming's $[7, 4, 3]$ code, for example, has $2^4 = 16$ codewords. For each of them, there are 8 possibly corrupted codewords that can arise from at most 1 bit flip. To implement nearest codeword decoding we might need to write down the map from each of these $16 \cdot 8 = 128$ candidates to its nearest codeword and search for a match:

$$\begin{aligned}
\texttt{0000000} &\to 0000\\
\texttt{0000001} &\to 0000\\
&\vdots\\
\texttt{0111110} &\to 0110\\
&\vdots\\
\texttt{1111111} &\to 1111.
\end{aligned}$$

In general, the list of nearest candidate codewords for n-bit codewords, k-bit messages, and e errors has size $2^k\binom{n}{\leq e}$, where $\binom{n}{\leq e}$ is the number of strings with at most e ones (see Figure 2 (a)). For example, if $k = 10$, $n = 20$, and $e = 5$, the list has more than twenty million items. Isn't there a better way to decode? For the Hamming $[7, 4, 3]$ code, there is.

Algorithm *Decode* for the Hamming $[7, 4, 3]$ code
Input: $\mathbf{y} \in \{0, 1\}^7$ with at most one error

1 Calculate $p_1 = y_1 + y_2 + y_4 + y_5, p_2 = y_1 + y_3 + y_4 + y_6, p_3 = y_2 + y_3 + y_4 + y_7$.
2 Convert $p_1p_2p_3$ into a number i using the rule:
$\texttt{000} \to 0, \texttt{110} \to 1, \texttt{101} \to 2, \texttt{011} \to 3, \texttt{111} \to 4, \texttt{100} \to 5, \texttt{010} \to 6, \texttt{001} \to 7.$
3 If $i \neq 0$, flip the i-th bit of $\mathbf{y}$.
4 Output the first four bits of $\mathbf{y}$.

For example, if $\mathbf{y} = \texttt{0111110}$, $p_1p_2p_3$ will equal $\texttt{111}$, i will be 4. After flipping the fourth bit $\mathbf{y}$ becomes $\texttt{0110110}$, which is the encoding of 0110. This decoding rule may look magical but there is method in the madness. You will explore it in the homework.

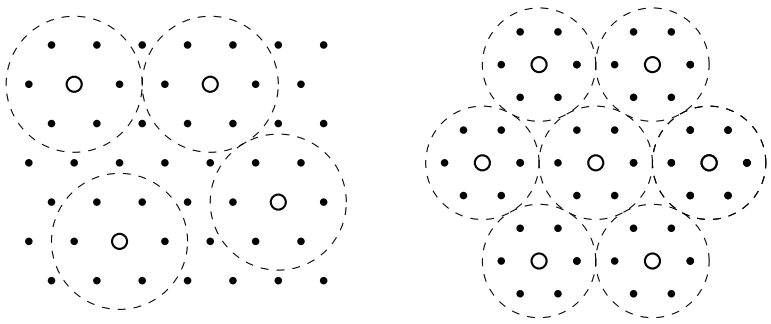


FIGURE 2: Illustration of a code. The white points represent codewords and the black points represent the remaining n-bit strings. (a) In a code of distance 3, all strings that differ from a codeword by at most one bit are distinct. (b) In a 1-perfect code there are no other strings.

Hamming's $[7, 4, 3]$ code has distance 3. Can we encode 5-bit messages into 7-bit strings and preserve the distance? We can't. In Hamming's $[7, 4, 3]$ code all the strings that differ from a codeword in at most one bit are distinct. We already counted that there are 128 such strings. But $128 = 2^7$ so they must account for all strings in $\{0, 1\}^7$. If we try to pack in even one more codeword the distance will be ruined.

This code is an example of a 1-perfect code (see Figure 2 (b)). Codes of distance $2e + 1$ in which every string differs from a codeword in at most e bits are called e-perfect.

2 Reed-Solomon codes

$[7, 4, 3]$ codes exist but $[7, 5, 3]$ codes do not. Coding theorists want to know which values of n (codeword length), k (message length), and d (distance) admit $[n, k, d]$ codes and which do not. This is a difficult question.

So far we have assumed that the codeword symbols are the bits $\texttt{0}$ and $\texttt{1}$. This turns out to be the main source of difficulty. Let's relax this requirement and allow the symbols to come from some larger alphabet. A code over a size-q alphabet is a set of n-symbol strings each of which takes one of q possible values. We will represent these by the numbers 0 to $q - 1$.

For example, $\mathcal{C}_1 = \{\texttt{000}, \texttt{111}, \texttt{222}\}$ is the 3-repetition code over alphabet size 3 (trits). It encodes a message consisting of a single trit. Its distance is 3. The code $\mathcal{C}_2 = \{\texttt{000}, \texttt{012}, \texttt{021}, \texttt{102}, \texttt{111}, \texttt{120}, \texttt{201}, \texttt{210}, \texttt{222}\}$ encodes two trits and has distance 2.

Our alphabets of interest are the finite fields $\mathbb{F}_n$. Like last time, if n is a prime number, the alphabet symbols can be manipulated using arithmetic modulo n. $\mathcal{C}_1$ and $\mathcal{C}_2$ are instances of a very special family of codes.

Definition 2. The codewords of the *Reed-Solomon code* $\mathcal{RS}_{k,n}$ are the evaluations all polynomials p of degree at most $k - 1$ at the elements of $\mathbb{F}_n$ in order.

When n is prime, the codeword is the sequence $(p(0), p(1), \ldots, p(n-1))$. We will assume that $n \geq k$.

$\mathcal{C}_1$ is the Reed-Solomon code $\mathcal{RS}_{1,3}$. The degree-0 polynomials in $\mathbb{F}_3$ are the three constants 0, 1, and 2. When evaluated at inputs 0, 1, and 2 they produce the sequences `000`, `111`, and `222`. $\mathcal{C}_2$ is nothing but $\mathcal{RS}_{2,3}$. The polynomials p are now linear functions $p(x) = m_0 + m_1 x$. The coefficient pair (m_0, m_1) can be chosen in nine possible ways specifying nine polynomials and nine codewords. For example, when $m_0 = 1$ and $m_1 = 2$ the polynomial is $p(x) = 1 + 2x$ and its corresponding codeword is $(p(0), p(1), p(2)) =$ `102`.

In general, $\mathcal{RS}_{k,n}$ contains n^k codewords because a degree-$(k-1)$ polynomial has k coefficients and there are n choices for each coefficient. So it can encode a message consisting of k symbols. This counting argument suggests a natural implementation of the encoding: View the message symbols as coefficients of a polynomial and the codeword symbols as evaluations of the same polynomial.

Algorithm *Encode* for Reed-Solomon codes
Input: $m_0, \ldots, m_{k-1}$ in $\mathbb{F}_n$.
1 Construct the polynomial $p(x) = m_0 + m_1 x + \cdots + m_{k-1} x^{k-1}$.
2 Output the sequence of evaluations $(p(0), p(1), \ldots, p(n-1))$.

How about its distance? As the codeword consists of all evaluations, the distance is the smallest number of places in which any two degree-$(k-1)$ polynomials p and p' differ. We saw in the last lecture that p and p' can match in at most $k-1$ places, so they must differ in the remaining $n-k+1$ places. So the distance is at least $n-k+1$. To summarize,

The Reed-Solomon code $\mathcal{RS}_{k,n}$ has message length k, codeword length n, and distance $n-k+1$.

We only argued that the distance is *at least* $n-k+1$. Can it be greater than that? Not for the Reed-Solomon code or not for any other code!

Singleton bound. There is no code of message length k, codeword length n, and distance greater than $n-k+1$.

Proof. Look at the first $k-1$ symbols of the codeword. If each symbol can take one of q values, there are q^{k-1} possible patterns. The message is k symbols long. There are q^k possible messages. (In the Reed-Solomon code q equals n.) By the pigeonhole principle, there must exist two messages $\mathbf{m}$ and $\mathbf{m}'$ whose encodings $\mathbf{c}$ and $\mathbf{c}'$ share the same $k-1$ first symbols. The codewords $\mathbf{c}$ and $\mathbf{c}'$ can differ in at most the last $n-(k-1) = n-k+1$ symbols. The distance cannot be larger than that. □

Reed-Solomon is optimality codified. These codes have the largest possible distance for any message length k and any codeword length n. The encoding algorithm is fairly simple. How about decoding?

3 Decoding Reed-Solomon codes

An *erasure* is a more benign type of corruption than an error. It is a missing symbol from the codeword at a known position. Let's take the Reed-Solomon code $\mathcal{RS}_{3,5}$ as an example. `22434` is a codeword of this code. Erasures at positions 0 and 2 (we start the count at zero to be consistent with the evaluation points) results in `?2?34`. Can the codeword and message be recovered?

Here is how. The corrupted codeword arose from evaluating some unknown polynomial of degree $3-1=2$ modulo 5. This polynomial has the form $p(x) = m_0 + m_1 x + m_2 x^2$. Even though $p(0)$ and $p(2)$ were erased, the values $p(1) = 2$, $p(3) = 3$ and $p(4) = 4$ provide enough equations to solve for the message (m_0, m_1, m_2):

$$\begin{aligned} m_0 + m_1 + m_2 &= 2 \\ m_0 + m_1 \cdot 3 + m_2 \cdot 3^2 &= 3 \\ m_0 + m_1 \cdot 4 + m_2 \cdot 4^2 &= 4. \end{aligned}$$

Using Gaussian Elimination modulo 5 we recover the unique solution $m_0 = 2, m_1 = 4, m_2 = 1$. The message is 241. Decoding from erasures for Reed-Solomon code is the same as reconstructing in Shamir's sharing scheme. A degree-$(k-1)$ polynomial can be reconstructed from any k evaluations, so this procedure can recover from up to $n-k$ erasures. This is best possible because $\mathcal{RS}_{k,n}$ has distance $n-k+1$.

How about errors? Before we go there I'll describe another algorithm to recover erased symbols. This algorithm is slower and less intuitive but will point us in the right direction for handling errors.

In the corrupted codeword `?2?34`, the values $p(0)$ and $p(2)$ have been erased. Instead of ignoring them, we will fix them to zero by changing the polynomial. Here is how. The polynomial $e(x) = x(x-2)$ evaluates to zero on inputs 0 and 2. Even though $p(0)$ and $p(2)$ are unknown, Bob can evaluate the polynomial $(pe)(x) = p(x)e(x)$ everywhere. Its values are `03042`.

As p has degree at most 2 and e has degree 2, their product pe has degree at most 4. We have five evaluations of pe. We can solve for its coefficients using Gaussian Elimination. The unique solution is $(pe)(x) = x + 4x^2 + 2x^3 + x^4$. We can now recover p by dividing $(pe)(x)$ and $e(x)$ as polynomials (over $\mathbb{F}_5$):

$$p(x) = \frac{(pe)(x)}{e(x)} = \frac{x + 4x^2 + 2x^3 + x^4}{x(x-2)} = 2 + 4x + x^2.$$

The message symbols are 241 as before.

The advantage of this more complicated procedure is that we can extend it to correct errors. The resulting error-correction algoritm was patented by Berlekamp and Welch in 1986. Let's illustrate on an example. The code $\mathcal{RS}_{3,5}$ has distance 3 so it can correct up to one error. Suppose Bob receives the corrupted codeword $y_0y_1y_2y_3y_4 = \texttt{11444}$. He is promised that there is at most one error, but he doesn't know where this error is.

The first insight of Berlekamp and Welch is that assuming the error occurs at position a, the value $p(x)(x-a)$ must match the corrupted codeword symbol y_x multiplied by $(x-a)$. If $x \neq a$ then the symbol y_x was not corrupted so $p(x)$ must equal y_x. Multiplication by $x-a$ does not change this. If x equals a both $p(x)(x-a)$ and $y_x(x-a)$ are zero so they are equal. We can therefore hope to solve for the message by expanding p as $p(x) = m_0 + m_1x + m_2x^2$ and setting up $p(x)(x-a) = y_x(x-a)$ as a system of equations in unknowns a, m_0, m_1, and m_2:

$$\begin{aligned}
m_0 \cdot (-a) &= 1 \cdot (-a) \\
(m_0 + m_1 + m_2) \cdot (1-a) &= 1 \cdot (1-a) \\
&\vdots \\
(m_0 + m_1 \cdot 4 + m_2 \cdot 4^2) \cdot (4-a) &= 4 \cdot (4-a).
\end{aligned}$$

The problem with this system is that it is not linear. The second idea of Berlekamp and Welch is to *linearize* it. Instead of seeking $p(x)$ *and* $e(x) = x - a$ separately, they solve for their product $(pe)(x) = p(x)e(x)$. This polynomial has degree 3 so it has the form $(pe)(x) = b_0 + b_1x + b_2x^2 + b_3x^3$. The equations become linear:

$$\begin{aligned}
b_0 &= 1 \cdot (-a) \\
b_0 + b_1 + b_2 + b_3 &= 1 \cdot (1-a) \\
b_0 + b_1 \cdot 2 + b_2 \cdot 2^2 + b_3 \cdot 2^3 &= 4 \cdot (2-a) \\
b_0 + b_1 \cdot 3 + b_2 \cdot 3^2 + b_3 \cdot 3^3 &= 4 \cdot (3-a) \\
b_0 + b_1 \cdot 4 + b_2 \cdot 4^2 + b_3 \cdot 4^3 &= 4 \cdot (4-a).
\end{aligned}$$

There are five equations in the five unknowns b_0, b_1, b_2, b_3 and a, setting the ground for Gaussian elimination modulo 5. It reveals the unique solution $b_0 = 2, b_1 = 3, b_2 = 4, b_3 = 4, a = 3$. Thus $(pe)(x) = 2 + 3x + 4x^2 + 4x^3$ and $e(x) = x - 3$. To find p all that remains is to divide the two

$$p(x) = \frac{(pe)(x)}{e(x)} = \frac{2 + 3x + 4x^2 + 4x^3}{x - 3} = 1 + x + 4x^2.$$

The message is 114. Its encoding under $\mathcal{RS}_{3,5}$ is `11404`. There was indeed one error in the corrupted codeword.

Algorithm *Decode* (Berlekamp-Welch decoding for the Reed-Solomon code)
Input: A possibly corrupted codeword $\mathbf{y} = (y_0, y_1, \ldots y_{n-1})$ of $\mathcal{RS}_{k,n}$ with at most c errors.
1 Set up the *error-locator* polynomial $e(x) = a_0 + a_1 x + \cdots + a_{c-1}x^{c-1} + x^c$
with unknowns $a_0, \ldots, a_{c-1}$.
2 Set up the product polynomial $(pe)(x) = b_0 + b_1 x + \cdots + b_{k+c-1}x^{k+c-1}$
with unknowns $b_0, \ldots, b_{k+c-1}$.
3 Solve the system of n equations $(pe)(i) = y_i e(i)$, one for each $i \in \mathbb{F}_n$
for $a_0, \ldots, a_{c-1}$ and $b_0, \ldots, b_{k+c-1}$.
4 Divide the polynomials $(pe)(x)$ and $e(x)$ to obtain $p(x)$.
5 Output the coefficients of p.

Theorem 3. *Assuming $c \le (n-k)/2$, Decode outputs the unique message $\mathbf{m}$ whose codeword is within distance c of $\mathbf{y}$.*

As the distance of the code is $n-k+1$, the Berlekamp-Welch algorithm decodes up to half the distance, which is best possible.

Proof. The system $(pe)(i) = y_i e(i), i \in \mathbb{F}_n$ is constructed so it has at least one solution. In this solution $p(x)$ is the polynomial $m_0 + m_1 x + \cdots + m_{k-1}x^{k-1}$, $e(x)$ is the product of terms $(x-a)$ where a ranges over the c corrupted positions,[1] and pe is the product of p and e.

The reason the theorem needs a proof is that there may be other solutions. The algorithm would not know which to pick in step 3, so we need to argue they all lead to the same outcome p. Suppose there is some other solution $(pe)', e'$ whose ratio is different. Then $(pe)/e$ and $(pe)'/e'$ are not the same polynomial, so $(pe)e'$ and $(pe)'e$ are not the same either. But for all n inputs x, $(pe)(x)e'(x) = y_x e(x)e'(x) = (pe)'(x)e(x)$. They are both polynomials of degree at most $k + 2c - 1$ and they match at n inputs. As c is at most $(n-k)/2$, $k + 2c - 1$ is less than n so their difference must equal the zero polynomial. So $(pe)/e$ and $(pe)'/e'$ must both equal p. □

There are many decoding algorithms for the Reed-Solomon code and for other error-correcting codes, but few of them can decode all the way up to half the distance. Berlekamp-Welch stands out in this respect. The algorithm is however expensive to run as it entails setting up and solving a modular system with $k+2c$ unknowns and at least that many equations. Its complexity is cubic in $k + 2c$. A related weakness is that its running time does not scale with the actual number of corruptions in the input string, but only with the preset upper bound c on this number. It is desirable that codewords with few errors decode faster, even though the code is designed to tolerate many of them in the worst case.

Berlekamp-Welch is a Great Algorithm not because it is practical but because it clarifies our thinking about codes. It demonstrates that in principle Reed-Solomon codes can be decoded reasonably efficiently, no ifs and buts. This distinguishes them from a host of other candidates. (It is not unique in this respect, but it is arguably the simplest algorithm among such variants.) Once an engineer knows that a task is feasible in principle, they can look for better algorithms that take into account additional information such as the likelihood of different error patterns. There is indeed an almost endless variety of algorithms for decoding Reed-Solomon and related codes.

[1] If there are fewer than c corruptions, some of the correct values can also be treated as "corruptions" to ensure that the degree of p is exactly c.

4 The long reach of coding theory

Errors in communication are complicated. Some errors could come in bursts like when a cloud obstructs a receiver from a satellite. Others could be random like when some piece of equipment malfunctions. The founders of coding theory consciously decided to ignore these complexities in favor of mathematically simple models of communication. This turned out to be a prescient choice. Today error-correcting codes have scientific impact well beyond fixing errors in communication. Their modern applications could not have been imagined in the 1950s.

Verifying computations In the early 1990s computer scientists were studying efficient verification of long computations. Suppose client Alice has paid the powerful server Bob to do a computation on her behalf. Bob returns an answer. How can Alice be sure that the answer is the actual outcome of the computation and not a made-up answer? Complexity theorists and cryptographers have invented ingenious algorithms for verifying that a computation was done correctly in much less time than it takes to perform it. Computations that take T steps to perform can be verified in time logarithmic in T. Succinct verification is by now a sprawling topic with many variants and optimizations. It is all enabled by error correction.

The execution of a computer program that uses T steps to complete can be represented as a "message" with T symbols. The t-th symbol represents the state of the computation at step t. The computation was executed correctly if the t-th and $(t+1)$-st symbol of this message are consistent with the program execution at all steps $t < T$. Bob can report this message and Alice can try to gain confidence by checking consistency for some random values of t. However, even a single inconsistency can completely derail the computation. Such a failure is unlikely to be detected by a few random spot checks.

The idea of many verification algorithms is to encode the message representing the execution with a suitable error-correcting code. If a malicious Bob tries to "ruin" the computation by corrupting the corresponding codeword then one of two things must happen. Perhaps all the corruptions produced another codeword. This can only happen if the number of corruptions is at least as large as the distance of the code. Provided the distance is sufficiently high, a corruption can then be detected by a random spot check. Otherwise, Bob must have produced something that does not belong to the code. A type of code called a locally testable code admits an algorithm that can (approximately) test membership in the code much faster than it takes to decode or even read the candidate codeword. Computations encoded by locally testable codes are immune to such attacks.

Locally testable codes were invented by computer scientists for the purpose of analyzing computation. They later found cryptographic applications in shared environments like the blockchain where the state of long computations needs to be agreed upon quickly by many participants. The codes used in these applications have much worse rate than the Reed-Solomon code. For a while it appeared that high rate, large distance, and local testability are simultaneously incompatible requirements. A few years ago two groups of mathematicians and computer scientists showed that we can have it all.

Random linear codes and cryptography Reed-Solomon codes are optimal in many ways. The price to pay for this optimality is that they work only over large alphabets. How about codes over bits? Already in the 1950s Gilbert and Varshamov proposed *random linear codes*. k random strings consisting of n bits are independently picked as "basis" codewords. The choice of these basis codewords $\mathbf{c}_1, \ldots, \mathbf{c}_k$ determines the code.

The encoding of a message $\mathbf{m}$ is then the sum modulo 2 of the codewords corresponding to the 1-bits of m: $\mathit{Encode}(\mathbf{m}) = \sum m_i \mathbf{c}_i \pmod 2$. Gilbert and Varshamov showed derived a formula for the minimum distance of most such codes. More than seventy years later there are no known constructions that outperform random codes. Yet it is not known if the Gilbert-Varshamov construction is the best possible.

Random linear codes have a simple encoding algorithm. How about decoding? Decoding from erasures amounts to solving a system of linear equations in modular arithmetic, just like for the Reed-Solomon

code. No comparably efficient algorithm is known to decode from errors. It is commonly believed that this is a difficult problem. This apparent drawback of random linear codes turns out to be an essential feature in the context of data encryption.

Quantum error-correction One of Hamming's motivations for studying error correction was data integrity. We expect that data should be robust to computer storage failures. Error-correcting codes provide a mechanism for protection against such failures.

Classical hardware is reliable enough that computations do not require error-tolerance at the application level. (Nevertheless, error-correcting codes remain ubiquitous in memory, storage, and communication systems.) However, it is widely believed that error-correction will be indispensable for the operation of quantum computers.

Quantum states cannot survive on their own for more than a few instants. To prevent loss of information it is essential that the state of a quantum computer be suitably encoded. Physicists and computer scientists have devised analogues of error-correcting codes for quantum information. Designing hardware that carries out this error correction reliably is perhaps the greatest challenge for quantum computing.