# NOTES ON REED-SOLOMON CODES

MARK HAIMAN

Reed-Solomon codes are examples of *error correcting codes*, in which redundant information is added to data so that it can be recovered reliably despite errors in transmission or storage and retrieval. The error correction system used on CD's and DVD's is based on a Reed-Solomon code. These codes are also used on satellite links and other communications systems.

We will first discuss the general principles behind error correcting codes. Then we will explain one type of Reed-Solomon code, and discuss the algorithms for encoding using this code, and for decoding with error correction.

## 1. Error correcting and error detecting codes

In an error correcting code, a message $M$ is encoded as a sequence of symbols $a_1 a_2 \dots a_n$, called a *codeword*. The set of possible symbols is fixed in advance; for instance each symbol might be a byte (8 bits) of binary data. The code incorporates some redundancy, so that if some of the symbols in a codeword are changed, we can still figure out what the original message must have been. Usually, the number of symbols in the codeword, $n$, is also fixed, so each message carries a fixed amount of data. To encode larger amounts of data, one would break it up into a number of messages and encode each one separately.

The simplest example of an error correcting code is the *triple-redundancy* code. In this code, the message $M$ consists of a single symbol $a$, and we encode it by repeating the symbol three times, as $aaa$. Suppose one symbol in the codeword is changed, so we receive a word $baa$ or $aba$ or $aab$. We can still recover the original symbol $a$ by taking a majority vote of the three symbols received. If errors result in two symbols being changed, we might receive a word like $abc$. In that case we can't correct the errors and recover the original message, but we can at least detect the fact that errors occurred. However, with this code we cannot always detect two errors. If the code word was $aaa$, two errors might change it to something like $bba$. This received word would be miscorrected by the majority vote decoding algorithm, giving the wrong message symbol $b$.

By repeating the message more times, we can achieve higher rates of error correction: with five repetitions we can correct two errors per codeword, and with seven repetitions we can correct three. These simple-minded error correcting codes are very inefficient, however, since we have to store or transmit three, five or seven times the amount of data in the message itself. Using more sophisticated mathematical techniques, we will be able to achieve greater efficiency.

1.1. **Hamming distance.** An obvious strategy for decoding any error correcting code is, if we receive a word $B$ that is not a codeword, to find the codeword $A$ that is "closest" to $B$, and then decode $A$ to get the message. For example, majority-vote decoding in the triple-redundancy code follows this strategy. If we receive a word such as $baa$, the closest codeword is $aaa$, since it differs from the received word in only one place, while any other codeword $xxx$ differs from the

*Date*: February, 2003.

received word *baa* in at least two places. It is useful to make a precise definition of this concept of "closeness."

**Definition 1.** The *Hamming distance* $d(A, B)$ between two words $A = a_1a_2 \dots a_n$ and $B = b_1b_2 \dots b_n$ is the number of places where they differ. In other words $d(A, B)$ is the number of indices $i$ such that $a_i \neq b_i$.

The Hamming distance satisfies the following inequality, called the *triangle inequality* (because it is also satisfied by the lengths of the sides of a triangle):

$$d(A, C) \leq d(A, B) + d(B, C) \qquad \text{for all words } A,\ B,\ C. \tag{1}$$

To see why the triangle inequality holds, observe that we can change $A$ into $B$ by altering $d(A, B)$ symbols, and then change $B$ into $C$ by altering another $d(B, C)$ symbols. So we can change $A$ into $C$ by altering at most $d(A, B) + d(B, C)$ symbols (perhaps fewer, since in changing $B$ to $C$ we might undo some of the changes from $A$ to $B$).

The *standard decoding algorithm* is as follows. Given a received word $B$, find the codeword or codewords $A$ that minimize the Hamming distance $d(A, B)$. If there is just one such codeword $A$, decode it to get the message. If there is more than one such codeword, report that an uncorrectable error has been detected. Our first theorem tells us how many errors a code can correct or detect using the standard decoding algorithm.

**Theorem 1.** *Let $d$ be the minimum Hamming distance between pairs of distinct codewords of a code $C$. If $d \geq 2e + 1$, the standard decoding algorithm for $C$ can correct up to $e$ errors. If $d \geq 2e + 2$, it can correct up to $e$ errors and detect $e + 1$ errors.*

*Proof.* Suppose we cannot correct up to $e$ errors. This means there is a codeword $A$, and some combination of at most $e$ errors that changes $A$ to a word $B$ which is either miscorrected or uncorrectable. The fact that we can change $A$ to $B$ with at most $e$ errors means that $d(A, B) \leq e$. If $B$ is miscorrected or uncorrectable, it means there is a word $C \neq A$ with $d(C, B) \leq d(A, B)$, and therefore $d(C, B) \leq e$. By the triangle inequality, we have $d(A, C) \leq 2e$, contradicting the hypothesis that $d \geq 2e + 1$.

For the second part of the theorem, observe that if $d \geq 2e + 2$ then we can correct up to $e$ errors by the first part. Suppose that we cannot also detect $e + 1$ errors. Then there is a codeword $A$ and a combination of $e + 1$ errors that changes $A$ to a word $B$ which will be miscorrected. For that to happen there must be a codeword $C$ with $d(B, C) \leq e$. Since $d(A, B) = e + 1$, the triangle inequality gives $d(A, C) \leq 2e + 1$, contradicting the hypothesis $d \geq 2e + 2$. □

*Example.* Consider the $n$-fold redundancy code, where the message is a single symbol $a$, and the corresponding codeword is $n$ copies $aa \dots a$. The Hamming distance between any two distinct codewords is $d(aa \dots a, bb \dots b) = n$, so the minimum distance $d$ for this code is equal to $n$. If $n$ is odd, equal to $2e + 1$, then we can correct $e$ errors, as we have seen before: triple-redundancy corrects one error, quintuple redundancy corrects two, and so on. If $n$ is even, equal to $2e + 2$, then we can also detect $e + 1$ errors. So the quadruple redundancy code can correct one error, just like the triple redundancy code, but it can also detect any two errors, which the triple redundancy code cannot do.

There are several practical difficulties in using Theorem 1 to construct error correcting codes. The first difficulty is that of determining the minimum Hamming distance $d$ between codewords.

For a completely arbitrary code, the only way to do this is by comparing every two codewords. However, for practical applications, we want to design codes in which a substantial amount of data is contained in each message. This means that the number of possible messages, and therefore the number of possible codewords, will be enormous. It then becomes impractical to find the minimum Hamming distance by comparing all possible pairs of codewords.

A second difficulty, which is related to the first one, is that we need an efficient means of encoding. If our code is completely arbitrary, the only way to encode is by means of a table listing the codeword for every message. Once again, if we wish to design a code in which each message message contains a substantial amount of data, and so there are a huge number of possible messages, it becomes impractical to keep a table of codewords for all the possible messages.

The third difficulty is that the standard decoding algorithm is impractical to carry out for an arbitrary code. If we receive a word $B$ that is not a codeword, and we want to locate the closest codeword $A$, the best we can do is to first try every possible one-symbol change to $B$, then every possible two-symbol change, and so on, until we find one or more codewords. This could require a very long search to decode just one single message.

The difficulty of encoding is easily overcome with the help of *linear codes*, which we will discuss in the next section. However, not all linear codes help much with the difficulties involved in determining the minimum distance and efficient decoding. For that, we need codes with even more special properties, known generally as *algebraic codes*. The Reed-Solomon codes, which we will study in the last sections, are examples of algebraic codes.

## 2. Linear codes

For the linear codes we are going to study, the code symbols $a$ will be integers (mod $p$), for a fixed prime number $p$. We use the notation

$$\mathbb{Z}_p = \text{set of all congruence classes (mod } p).$$

To be definite, we represent every integer by its remainder (mod $p$), so there are $p$ different symbols $a \in \mathbb{Z}_p$, represented by the $p$ numbers $0, 1, 2, \ldots, p-1$. Using modular arithmetic, we can add, subtract, and multiply these symbols. Because $p$ is prime, every symbol $a \neq 0$ has a multiplicative inverse $a^{-1}$, so we can also "divide" symbols.

The set $\mathbb{Z}_p$ with its arithmetic operations is an example of what mathematicians call a *field*: a system with addition, subtraction, multiplication, and division (except by zero) obeying the usual identities of algebra. Since it is a finite set, $\mathbb{Z}_p$ is a *finite field*. In more advanced algebra courses, other examples of finite fields are constructed, which may also be used as the set of symbols for a linear code. It turns out that there is a finite field with $p^d$ elements for every prime number $p$ and every positive integer $d$. The Reed-Solomon codes most commonly encountered in practice are based on the field with $2^8 = 256$ elements, so the symbols can be conveniently represented by 8-bit binary bytes. To keep things simple, in these notes we will only study codes based on $\mathbb{Z}_p$. However, the same concepts apply to codes based on other finite fields.

A $1 \times n$ matrix—that is, a matrix with one row—is called a *row vector*. A word $a_1 a_2 \ldots a_n$ with symbols $a_i \in \mathbb{Z}_p$, can (and will) be represented by the row vector

$$\begin{bmatrix} a_1 & a_2 & \ldots & a_n \end{bmatrix}.$$

**Definition 2.** Fix positive integers $m < n$, and an $m \times n$ matrix $\mathbf{C}$ with entries in $\mathbb{Z}_p$. The code in which a message vector $\begin{bmatrix} x_1 & x_2 & \dots & x_m \end{bmatrix}$ is encoded by the code vector

$$\begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & \dots & x_m \end{bmatrix} \cdot \mathbf{C}$$

is called a *linear code* with $m$ message symbols and $n$ codeword symbols.

For a linear code to be any use at all, we must be sure that the encoding function from message vectors $\mathbf{x}$ to code vectors $\mathbf{a} = \mathbf{xC}$ is one-to-one, that is, we can recover the message from the code. One way to ensure this is to use a matrix $\mathbf{C}$ that is in *normal form*, meaning that the first $m$ columns of $\mathbf{C}$ form an $m \times m$ identity matrix:

$$\mathbf{C} = \begin{bmatrix} \mathbf{I}_m \mid \mathbf{P} \end{bmatrix},$$

where $\mathbf{P}$ is an $m \times (n-m)$ matrix. Then from the rules of matrix multiplication we deduce that the message $\mathbf{a} = \mathbf{xC}$ will have the form $\mathbf{a} = \begin{bmatrix} \mathbf{x} \mid \mathbf{c} \end{bmatrix}$, where $\mathbf{c} = \mathbf{xP}$. In other words, the codeword is just the message word, follow by some additional *check symbols* $\mathbf{c}$. In this case, we can obviously recover $\mathbf{x}$ from $\mathbf{a}$ if there are no errors, just by looking at the first $m$ symbols.

One important point about linear codes is that by their very definition, there is an easy algorithm for encoding a message: just multiply it by the code matrix $\mathbf{C}$. A second important point is that for linear codes there is a theorem that allows us to determine the minimum Hamming distance between codewords without checking every possible pair. Before giving the theorem, we first need the following lemma, which expresses a key property of linear codes.

**Lemma 1.** *If* $\mathbf{a}$ *and* $\mathbf{b}$ *are two codewords in a linear code* $C$*, then* $\mathbf{a}+\mathbf{b}$ *and* $\mathbf{a}-\mathbf{b}$ *are also codewords, and more generally so is* $c\,\mathbf{a} + d\,\mathbf{b}$*, for any scalars* $c$*,* $d$ *in the base field* $\mathbb{Z}_p$*.*

*Proof.* By definition, there exist message vectors $\mathbf{x}$ and $\mathbf{y}$ such that $\mathbf{a} = \mathbf{xC}$ and $\mathbf{b} = \mathbf{yC}$. Then from the rules of matrix multiplication we see that

$$c\,\mathbf{a} + d\,\mathbf{b} = (c\,\mathbf{x} + d\,\mathbf{y})\mathbf{C}.$$

Now $c\,\mathbf{x} + d\,\mathbf{y}$ is again a $1 \times m$ row vector, and thus a possible message vector, so therefore $c\,\mathbf{a} + d\,\mathbf{b}$ is a code vector. To get the special cases $\mathbf{a} \pm \mathbf{b}$ take $c = 1$ and $d = \pm 1$. □

Now for the theorem about Hamming distance.

**Theorem 2.** *The minimum Hamming distance* $d$ *between codewords in a linear code* $C$ *is equal to the minimum number of non-zero symbols in a code vector* $\mathbf{a} \neq \mathbf{0}$ *of* $C$*.*

*Proof.* Let $\mathbf{b}$ and $\mathbf{c}$ be distinct code vectors with minimum distance $d(\mathbf{b}, \mathbf{c}) = d$. Then $\mathbf{a} = \mathbf{b} - \mathbf{c}$ is a code vector, by Lemma 1, and the number of non-zero symbols in $\mathbf{a}$ is equal to $d$. This shows that the minimum number of non-zero symbols in a code vector is less than or equal to $d$. To prove the opposite inequality, note that Lemma 1 implies that $\mathbf{0}$ is a code vector. For every code vector $\mathbf{a}$, we have $d(\mathbf{a}, \mathbf{0})$ equal to the number of non-zero symbols in $\mathbf{a}$, so this number is greater than or equal to $d$. □

**Definition 3.** The *weight* of a code vector is the number of non-zero symbols in it. The weight of a linear code is the minimum weight of a non-zero code vector. Thus Theorem 2 says that the minimum Hamming distance in a linear code is equal to its weight. Combining this with Theorem 1, we see that a linear code with odd weight $2e + 1$ can correct $e$ errors, and a linear code with even weight $2e + 2$ can correct $e$ errors and detect $e + 1$ errors.

## 3. Reed-Solomon codes

All linear codes offer the advantages that they are easy to encode, and the minimum Hamming distance reduces to a simpler concept, the weight. However, the weight of an arbitrary linear code is still not easy to determine. Also, the mere fact that a code is linear doesn't help all that much in applying the standard decoding algorithm. In order to compute the weight and decode more easily, we need to use linear codes with special properties. Usually the special properties are based on algebra, in which case the code is called an *algebraic code*. The Reed-Solomon codes that we will now define are examples of algebraic codes.

**Definition 4.** Let $p$ be a prime number and let $m \leq n \leq p$. The *Reed-Solomon code* over the field $\mathbb{Z}_p$ with $m$ message symbols and $n$ code symbols is defined as follows. Given a message vector $\begin{bmatrix} x_1 & x_2 & \dots & x_m \end{bmatrix}$, let $P(t)$ be the polynomial

$$P(t) = x_m t^{m-1} + x_{m-1} t^{m-2} + \cdots + x_2 t + x_1$$

with coefficients given by the message symbols. Thus $P(t)$ is a polynomial of degree at most $m-1$ in one variable $t$, with coefficients in $\mathbb{Z}_p$. Then the code vector $\mathbf{a}$ for this message vector is the list of the first $n$ values of the polynomial $P(t)$:

$$\mathbf{a} = \begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix} = \begin{bmatrix} P(0) & P(1) & \dots & P(n-1) \end{bmatrix}$$

(evaluated using modular arithmetic in $\mathbb{Z}_p$).

From this definition it might not be obvious that Reed-Solomon codes are in fact linear codes. However, using the definition of matrix multiplication, we can deduce the following linear presentation of the Reed-Solomon code.

**Theorem 3.** *The Reed-Solomon code over $\mathbb{Z}_p$ with $m$ message symbols and $n$ code symbols is the linear code with matrix*

$$\mathbf{C} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 0 & 1 & 2 & \dots & n-1 \\ 0^2 & 1^2 & 2^2 & \dots & (n-1)^2 \\ \vdots & \vdots & \vdots & & \vdots \\ 0^{m-1} & 1^{m-1} & 2^{m-1} & \dots & (n-1)^{m-1} \end{bmatrix},$$

*all entries taken* (mod $p$).

Although the code matrix $\mathbf{C}$ for a Reed-Solomon code is not in normal form, we will see shortly that it does define a one-to-one encoding function.

Next we want to compute the weight of a Reed-Solomon code. To do this we first have to recall some facts about polynomials that you learned in high school algebra. Of course, when you learned algebra in high school, you learned about polynomials with coefficients in the field of real numbers (or maybe the field of rational numbers). Now we are working with polynomials that have coefficients in $\mathbb{Z}_p$. However, what you learned in high school is still valid in this situation, because $\mathbb{Z}_p$ is a field, and the facts we will now recall depend only on laws of algebra that hold in every field. Here they are.

**Lemma 2.** *If $a$ is a root of a polynomial $P(t)$, that is, if $P(a) = 0$, then $P(t)$ can be factored as $P(t) = (t-a)Q(t)$. Note that the degree of $Q(t)$ is one less than the degree of $P(t)$.*

*Proof.* You probably learned the proof of this in high school, although you might not have realized at the time that what you were learning was the proof of a theorem. I'll just remind you quickly how it works. Regardless of whether or not $a$ is a root of $P(t)$, you can always divide $P(t)$ by $(t-a)$ using long division, obtaining a quotient polynomial $Q(t)$ and a remainder $c$, which is a constant. Then $P(t) = (t-a)Q(t) + c$, and substituting $t = a$ in this identity shows that $c = P(a)$. So if $a$ is a root of $P(t)$ then $c = 0$, and the result follows. □

**Theorem 4.** *If $P(t)$ has $d$ distinct roots and is not identically zero, then $P(t)$ has degree at least $d$.*

*Proof.* Suppose $a_1, \ldots, a_d$ are distinct roots of $P(t)$. By Lemma 2, we can factor $P(t) = (t - a_d)Q_1(t)$, and substituting $t = a_i$ into this for $i \neq d$ shows that $a_1, \ldots, a_{d-1}$ are roots of $Q_1(t)$. In turn we can factor $Q_1(t) = (t - a_{d-1})Q_2(t)$, where $a_1, \ldots, a_{d-2}$ are roots of $Q_2(t)$. Repeating this process we eventually get $P(t) = (t - a_d)(t - a_{d-1}) \cdots (t - a_1)Q_d(t)$. If $P(t)$ is not identically zero, then neither is $Q_d(t)$, so the factorization shows that $P(t)$ has degree at least $d$. □

Now we are ready to determine the weights of our codes.

**Theorem 5.** *The weight of the Reed-Solomon code over $\mathbb{Z}_p$ with $m$ message symbols and $n$ code symbols is equal to $n - m + 1$.*

*Proof.* First we'll show that the weight is at least $n-m+1$. By the definition of the Reed-Solomon code, this is equivalent to saying that for every non-zero polynomial $P(t)$ of degree at most $m-1$, when we evaluate it at the elements $0, 1, \ldots, n-1$ of $\mathbb{Z}_p$, we will get at least $n - m + 1$ non-zero values. Supposing to the contrary that we get at most $n-m$ non-zero values, we see that $P(t)$ has at least $m$ roots among the elements $0, 1, \ldots, n-1$. Note that $n \leq p$ by definition, so these roots are distinct elements of $\mathbb{Z}_p$. But by Theorem 4 this is impossible if $P(t)$ has degree less than $m$ and is not identically zero.

To show that the weight is at most $n - m + 1$ and complete the proof, we just have to find an example of a code vector with weight exactly $n - m + 1$. Again by the definition of the code, this is equivalent to finding an example of a polynomial $P(t)$ with degree at most $m - 1$ such that exactly $m - 1$ of the values $P(0), \ldots, P(n-1)$ are zero. An example of such a polynomial is $P(t) = t(t-1)(t-2)\cdots(t-(m-2))$. □

**Corollary 1.** *A Reed-Solomon code with $m$ message symbols and $n = m + 2e$ code symbols can correct $e$ errors.*

*Example.* A Reed-Solomon code with 15 code symbols can transmit 11 message symbols, while correcting errors in any two of the code symbols. By contrast, using a quintuple redundancy code to achieve two-error correction, we can only transmit 3 message symbols with 15 code symbols.

To complete up our discussion of encoding, we will now show, as promised earlier, that the encoding function is one-to-one. This amounts to saying that if two message polynomials $P_1(t)$ and $P_2(t)$ give the same code vector, that is, if

$$P_1(i) = P_2(i) \quad \text{for } i = 0, 1, \ldots, n-1,$$

then we must have $P_1(t) = P_2(t)$ identically. To see this, observe that $P_1(t) - P_2(t)$ is a polynomial of degree less than $m$, and it has at least $n$ distinct roots in $\mathbb{Z}_p$, namely $0, 1, \ldots, n-1$. Since $m \leq n$ by definition, we must have $P_1(t) - P_2(t) = 0$, by Theorem 4.

## 4. Decoding Reed-Solomon codes

What makes Reed-Solomon codes really useful is the existence of a simple algorithm for correcting errors and decoding. The algorithm we will discuss was discovered by Berlekamp and Welch. Rather than publish it in a mathematical or engineering journal, they patented it in 1983. More on that at the end.

The discussion in this section will always refer to the Reed-Solomon code over $\mathbb{Z}_p$ with

$$\begin{aligned} m &= \text{number of message symbols,} \\ n &= \text{number of code symbols} \\ &= m + 2e, \\ e &= \text{number of errors the code can correct.} \end{aligned}$$

The decoding procedure relies on two lemmas, which we can prove using Theorem 4.

**Lemma 3.** *Let the transmitted code vector be*

$$\begin{bmatrix} P(0) & P(1) & \dots & P(n-1) \end{bmatrix},$$

*and the received vector be*

$$\begin{bmatrix} R_0 & R_1 & \dots & R_{n-1} \end{bmatrix}.$$

*Assume there are at most $e$ errors, that is, at most $e$ values $i$ such that $R_i \neq P(i)$. Then there exist non-zero polynomials*

$$\begin{aligned} &E(t) \text{ of degree} \leq e \\ &Q(t) \text{ of degree} \leq m + e - 1, \end{aligned}$$

*such that*

$$Q(i) = R_i E(i) \quad \text{for all } i = 0, 1, \dots, n. \tag{2}$$

*Proof.* Let $\{i_1, i_2, \dots, i_k\}$ be the set of error positions, that is, $i$ is a member of this set if $R_i \neq P(i)$. Let

$$\begin{aligned} E(t) &= (t - i_1)(t - i_2) \cdots (t - i_k), \\ Q(t) &= P(t)E(t). \end{aligned}$$

Clearly $E(t)$ has degree $k$, which is less than or equal to $e$ by assumption. Since $P(t)$ has degree $\leq m - 1$, it also follows that $Q(t)$ has degree $\leq m + e - 1$.

To see why equation (2) holds for every $i$, consider two cases. The first case is that $i$ is not an error position. Then $R_i = P(i)$, so $Q(i) = P(i)E(i) = R_i E(i)$. The second case is that $i$ is an error position, so it belongs to the set $\{i_1, i_2, \dots, i_k\}$. From our definition of $E(t)$ we see that in this case $E(i) = 0$, and therefore $Q(i) = P(i)E(i) = 0 = R_i E(i)$. So equation (2) holds in either case. ☑

Equation (2) is called the *key equation* for the decoding algorithm. Lemma 3 tells us that it has a non-zero solution. However, the solution we wrote down to prove the lemma can only be calculated explicitly if we already know the error positions. In practice, we have to solve the key equation by interpreting it as a system of linear equations for the unknown coefficients of the polynomials $E(t)$ and $Q(t)$. The lemma guarantees that a non-zero solution of this linear system must exist.

The next lemma shows that after solving the key equation, we can calculate the original message polynomial $P(t)$.

**Lemma 4.** *If $E(t)$ and $Q(t)$ satisfy the key equation (2) in Lemma 3, and the number of errors is at most $e$, then $Q(t) = P(t)E(t)$. Hence we can compute $P(t)$ as $Q(t)/E(t)$.*

*Proof.* In the course of proving Lemma 3 we showed that there exists a solution of (2) with $Q(t) = P(t)E(t)$, but now we must show that *every* solution has this property.

Observe that both $Q(t)$ and $P(t)E(t)$ are polynomials of degree $\leq m + e - 1$, and hence so is their difference $Q(t) - P(t)E(t)$. If $i$ is not an error position, then $P(i) = R_i$, and therefore $Q(i) - P(i)E(i) = 0$. There are at least $n - e$ non-error positions, so the polynomial $Q(t) - P(t)E(t)$ has at least $n - e = m + e$ distinct roots in $\mathbb{Z}_p$. Since its degree is less than $m + e$, we must have $Q(t) - P(t)E(t) = 0$ identically, by Theorem 4. □

Based on these two lemmas we can describe the decoding algorithm.

*Decoding algorithm.* Suppose the received vector is

$$\begin{bmatrix} R_0 & R_1 & \dots & R_{n-1} \end{bmatrix}.$$

Introduce variables $u_0, u_1, \dots, u_{m+e-1}$ and $v_0, v_1, \dots, v_e$ to stand for the coefficients of $Q(t)$ and $E(t)$, so

$$\begin{aligned} Q(t) &= u_{m+e-1}t^{m+e-1} + u_{m+e-2}t^{m+e-2} + \cdots + u_1 t + u_0, \\ E(t) &= v_e t^e + v_{e-1}t^{e-1} + \cdots + v_1 t + v_0. \end{aligned}$$

For each $i = 0, 1, \dots, n-1$, substitute $t = i$ in $Q(t)$ and $E(t)$ to evaluate the key equation

$$Q(i) = R_i E(i).$$

This results in a system of $n$ linear equations for the unknown coefficients $u_j$ and $v_k$. Find a non-zero solution of this system, and plug in the result to get $Q(t)$ and $E(t)$ explicitly. Finally, divide $Q(t)$ by $E(t)$ to recover the message polynomial $P(t)$.

What might go wrong if there are more than $e$ errors in the received vector? The first main step in the algorithm is to solve $n$ linear equations for the $n + 1$ unknowns $u_j$ and $v_k$. It is a general theorem of linear algebra that this is always possible. The second main step is to divide $Q(t)$ by $E(t)$ to find $P(t)$. On doing this we might find that there is a remainder, and therefore no solution of $Q(t) = P(t)E(t)$. If that occurs, then Lemma 4 implies that there must have been more than $e$ errors. We cannot correct them, but at least we have detected them. Not every combination of more than $e$ errors can be detected, of course—some will simply result in incorrect decoding.

## 5. An improvement to the decoding algorithm

Most of the work in the decoding algorithm lies in solving $n$ linear equations for the $n + 1$ unknowns $u_j$ and $v_k$, a straightforward but fairly complex task that must be repeated for every message vector. In practice one often wants to decode data at high rates, so a reduction in the complexity per message is desirable. In this section we will see how to use some ideas from polynomial interpolation theory to simplify decoding.

The general theory described in the previous sections can be applied to Reed-Solomon codes over any finite field, but the method I will now describe is a special one for the fields $\mathbb{Z}_p$. We begin with a classical technique for predicting all the values of a polynomial function from the first few.

**Definition 5.** The *first difference* of a sequence

$$a_1,\ a_2,\ a_3,\ \ldots$$

is the sequence

$$\Delta a_i = a_{i+1} - a_i.$$

The first difference of the first differences is called the *second difference* and denoted $\Delta^2 a_i$; similarly we have third differences $\Delta^3 a_i$ and so on.

Here is an example. We'll start with the sequence of integer cubes and below it write its successive differences.

$$\begin{aligned} a&\colon 0,\ 1,\ 8,\ 27,\ 64,\ 125,\ 216,\ldots \\ \Delta a&\colon 1,\ 7,\ 19,\ 37,\ 61,\ 91,\ldots \\ \Delta^2 a&\colon 6,\ 12,\ 18,\ 24,\ 30,\ldots \\ \Delta^3 a&\colon 6,\ 6,\ 6,\ 6,\ldots \\ \Delta^4 a&\colon 0,\ 0,\ 0,\ldots \end{aligned}$$

The obvious thing to notice here is that the fourth difference is identically zero. This is a general fact.

**Theorem 6.** *If $f(0), f(1), \ldots$ is the sequence of values of a polynomial of degree $d$, then its first difference $\Delta f(i) = g(i)$ is given by a polynomial $g$ of degree less than $d$, or is zero if $d = 0$.*

*Proof.* If $d = 0$ then $f$ is constant, so the first differences are clearly zero. If $f(x) = x^d$ then its first difference is given by $(x+1)^d - x^d$. Now $(x+1)^d$ is a polynomial of degree $d$ with highest term $x^d$, so subtracting $x^d$ leaves a polynomial of degree less than $d$. If $f(x)$ is a general polynomial of degree $d$, say

$$f(x) = c_d x^d + c_{d-1} x^{d-1} + \cdots + c_0,$$

then

$$\Delta f(x) = c_d \Delta x^d + c_{d-1} \Delta x^{d-1} + \cdots \Delta c_0.$$

By our calculation of $\Delta x^k$, this is a sum of polynomials of degree less than $d$ and so is itself a polynomial of degree less than $d$. □

Repeated application of the preceding theorem (or more precisely, mathematical induction) gives the following corollary.

**Corollary 2.** *If $k > d$ then the $k$-th difference $\Delta^k f(i)$ of a polynomial $f$ of degree $d$ is identically zero.*

This result provides a way of solving the key equation (2) in the decoding algorithm for $E(t)$, without having to compute $Q(t)$ at the same time.

**Corollary 3.** *Let the received vector be*

$$\begin{bmatrix} R_0 & R_1 & \ldots & R_{n-1} \end{bmatrix},$$

*and let $a^{(j)}$ be the sequence*

$$0^j R_0,\ 1^j R_1, \ldots,\ (n-1)^j R_{n-1}.$$

*Note that the $(m+e)$-th difference $\Delta^{m+e}a^{(j)}$ is a sequence of length $n-(m+e)=e$. Let $\mathbf{B}$ be the $e \times (e+1)$ matrix whose $j$-th column contains the entries*

$$b_{ij} = \Delta^{m+e} a_i^{(j)}$$

*of the sequence $\Delta^{m+e}a^{(j)}$, for $j = 0, 1, \ldots, e$. Then the coefficients $v_k$ of the polynomial $E(t)$ in the decoding algorithm are given by the matrix equation*

$$\mathbf{B} \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_e \end{bmatrix} = \mathbf{0}. \tag{3}$$

*Proof.* There isn't much to prove here, beyond unraveling what all the notation means. Since $Q(t)$ is a polynomial of degree $m+e-1$ or less, the $(m+e)$-th difference of the sequence of values $Q(i)$ must be zero. Therefore the key equation gives

$$\Delta^{(m+e)}(R_i E(i)) = 0.$$

Writing this out in terms of the coefficients of the polynomial $E(t)$ leads directly to the matrix equation (3). □

In most applications the error correction number $e$ is much smaller than the code length $n$, so solving the system (3) of $e$ linear equations in $e+1$ unknowns is much quicker than solving the $n$ equations in $n+1$ unknowns resulting from the original key equation (2). However, we still have to find $Q(t)$ in order to compute $P(t)$. At this point we know the values $Q(0), \ldots, Q(n-1)$, since the key equation gives

$$Q(i) = R_i E(i)$$

and we have found $E(t)$. So our remaining difficulty is to compute the coefficients of $Q(t)$ from its values. This ancient problem has a beautiful solution in terms of differences.

**Theorem 7.** *Let $C_k(x)$ be the degree $k$ polynomial*

$$C_k(x) = x(x-1)\cdots(x-k+1)/k!,$$

*with $C_0(x)$ defined to be 1. Let $f(x)$ be any polynomial of degree at most $d$. Then $f(x)$ can be computed from its successive differences by the formula*

$$f(x) = f(0)C_0(x) + \Delta f(0)C_1(x) + \cdots + \Delta^d f(0)C_d(x). \tag{4}$$

*Proof.* We prove the theorem by mathematical induction on $d$. The base case $d = 0$ is trivial.

For the general case, since $\Delta f(x)$ has degree less than $d$ we can use induction to apply formula (4) to $\Delta f(x)$, obtaining

$$\Delta f(x) = \Delta f(0)C_0(x) + \Delta^2 f(0)C_1(x) + \cdots + \Delta^{d+1} f(0)C_d(x).$$

Let $g(x)$ denote the expression on the right-hand side of formula (4). By direct calculation we can show that $\Delta C_k(x) = C_{k-1}(x)$ for all $k > 0$, and of course $\Delta C_0(x) = 0$. Using this we can calculate

$$\Delta g(x) = \Delta f(0)C_0(x) + \Delta^2 f(0)C_1(x) + \cdots + \Delta^{d+1} f(0)C_d(x),$$

which is equal to $\Delta f(x)$. In other words, $\Delta(f(x) - g(x)) = 0$, so $f(x) - g(x)$ is a constant. Now $C_k(0) = 0$ for all $k > 0$ and $C_0(0) = 1$, so $g(0) = f(0)$. This shows that the constant $f(x) - g(x)$ is equal to zero, proving the formula $f(x) = g(x)$. □

*Example.* Looking back at the difference table for $f(x) = x^3$ in our earlier example, we see that $f(0) = 0$, $\Delta f(0) = 1$, $\Delta^2 f(0) = 6$, $\Delta^3 f(0) = 6$. According to the formula, we should have

$$x^3 = C_1(x) + 6C_2(x) + 6C_3(x),$$

which you can check is true by doing the algebra.

A bit of care is required when using formula (4) on polynomials with coefficients in $\mathbb{Z}_p$. In the definition of $C_k(x)$, we must understand the factor $1/k!$ to mean the inverse of $k!$ (mod $p$). This inverse exists as long as $k < p$, and the formula works fine for polynomials of degree less than $p$. It does not work for higher degree polynomials. However, the polynomial $Q(t)$ we are trying to find in the decoding algorithm has degree $m + e - 1$, which is less than $n$, and $n \leq p$ by definition. Putting it all together, we get the improved algorithm.

*Improved decoding algorithm.* Given the received vector

$$\begin{bmatrix} R_0 & R_1 & \dots & R_{n-1} \end{bmatrix},$$

compute the matrix $\mathbf{B}$ in Corollary 3, and find a non-zero solution of the resulting $e$ linear equations for the $e+1$ unknown coefficients $v_j$ of $E(t)$. Compute the sequence of values $Q(i) = R_i E(i)$ and its difference table to find $Q(0)$, $\Delta Q(0)$, $\dots$, $\Delta^{m+e-1} Q(0)$. Finally use formula (4) to compute $Q(t)$, and divide by $E(t)$ as before to find $P(t)$.

*Example.* We'll use a Reed-Solomon code over $\mathbb{Z}_7$ with $m = 3$ message symbols and $n = 7$ code symbols to correct $e = 2$ errors. Say the message vector is

$$\begin{bmatrix} 2 & 3 & 4 \end{bmatrix}.$$

The message polynomial is then

$$P(t) = 4t^2 + 3t + 2,$$

and the code vector $\begin{bmatrix} P(0) & P(1) & \dots & P(6) \end{bmatrix}$ is

$$\begin{bmatrix} 2 & 2 & 3 & 5 & 1 & 5 & 3 \end{bmatrix}.$$

All the arithmetic here is (mod 7), of course. Suppose we receive the vector with two errors as

$$\begin{bmatrix} 2 & 2 & 6 & 5 & 3 & 5 & 3 \end{bmatrix}.$$

This is our vector $\begin{bmatrix} R_0 & R_1 & \dots & R_6 \end{bmatrix}$. Following the recipe in Corollary 3, with $m + e = 5$, we compute the fifth difference sequences

$$\begin{aligned} \Delta^5 R_i &= \begin{bmatrix} 2 & 0 \end{bmatrix} \\ \Delta^5 i R_i &= \begin{bmatrix} 5 & 5 \end{bmatrix} \\ \Delta^5 i^2 R_i &= \begin{bmatrix} 0 & 2 \end{bmatrix}, \end{aligned}$$

yielding the matrix equation

$$\begin{bmatrix} 2 & 5 & 0 \\ 0 & 5 & 2 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \end{bmatrix} = \mathbf{0}.$$

A nonzero solution (mod 7) is

$$\begin{bmatrix} v_0 \\ v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix},$$

giving $E(t) = t^2 + t + 1$. Now we compute the sequence $Q(i) = R_i E(i)$ to be

$$2,\ 6,\ 0,\ 2,\ 0,\ 1,\ 3.$$

Its difference table is

$$\begin{array}{ccccccc} 2 & 6 & 0 & 2 & 0 & 1 & 3 \\ 4 & 1 & 2 & 5 & 1 & 2 & \\ 4 & 1 & 3 & 3 & 1 & & \\ 4 & 2 & 0 & 5 & & & \\ 5 & 5 & 5. & & & & \end{array}$$

Note that the next row would be the fifth difference, which is automatically zero because that was the condition we imposed to solve for $E(t)$. From the first column of the difference table and formula (4), we get

$$Q(t) = 2C_0(t) + 4C_1(t) + 4C_2(t) + 4C_3(t) + 5C_4(t),$$

and expanding out each $C_k(t)$ (mod 7), we find that this is equal to

$$Q(t) = 4t^4 + 2t^2 + 5t + 2 = (4t^2 + 3t + 2)(t^2 + t + 1).$$

As expected, the polynomial $E(t) = t^2 + t + 1$ factors out exactly, and the quotient $Q(t)/E(t) = 4t^2 + 3t + 2$ is the original message polynomial $P(t)$, recovered correctly despite the two errors.

The most complicated part of this whole procedure is the computation of the polynomials $C_k(t)$. Observe, however, that if we intended to decode many messages with the same code, we could compute these polynomials once and for all in advance, and quickly look up the result each time.

## 6. Can you patent a mathematical algorithm?

Historically, U.S. courts held that "laws of nature, natural phenomena, and abstract ideas" could not be patented. Mathematical algorithms were considered to be abstract ideas and therefore unpatentable. A computer program is a form of mathematical algorithm, and so, although the code for an individual program could be protected by copyright, the underlying algorithm could not be patented.

In the 1980's and 90's, patent attorneys for software companies began to file for patents on inventions consisting in part or entirely of algorithms, in a gradual effort to get the patent office and the courts to expand the definition of patentable inventions until it would include pure algorithms. In one famous case, ATT in 1988 patented an algorithm for linear programming developed by N. Karmarkar, a researcher at its Bell laboratories. Karmarkar's algorithm was an important mathematical development and the effort to patent it created much controversy, particularly because before the patent was applied for, Karmarkar and ATT boasted about how well the algorithm worked while refusing to reveal any details. The earlier Berlekamp-Welch patent seems to have attracted less attention.

When these patents were granted in the 80's, their validity was doubtful, as courts then still held mathematical algorithms *per se* to be unpatentable. However, the broadening of the definition has continued to the point that under U.S. law today, there are effectively no barriers to patenting

software and algorithms. If an algorithm has "some type of practical application," the courts no longer consider it an unpatentable abstract idea.

In my opinion and that of many other mathematicians, and also software developers, the present state of affairs is a bad one. In mathematics, claims of ownership over ideas serve no useful purpose and only impede progress. In commerce and industry, patents are supposed to serve the public interest by providing a reward for innovation. However, many observers have made the case that software patents have the opposite overall effect, stifling innovation by threatening in effect to make much non-commercial software development illegal.

Patents and copyrights are often spoken of today as *intellectual property*. This rather loaded term carries with it the suggestion that it is natural or desirable for individuals or corporations to own ideas and control their dissemination and use, prohibiting their sharing in the public domain. Historically, however, the basis of patent law is not the concept of natural property but that of a social contract, in which society grants to inventors a temporary monopoly on the use of their inventions in return for publicizing those inventions and as an incentive to invent. It's interesting to note that the basis for the authority of Congress to establish patents and copyrights is the following clause in Article I, Section 8 of the U.S. Constitution, which explicitly specifies the temporary nature of such rights and the social interest that they should serve:

> The Congress shall have the power... To promote the Progress of Science and useful Arts, by securing for limited Times to Authors and Inventors the exclusive Right to their respective Writings and Discoveries;

When patents on algorithms, software, genes, and life forms begin to obstruct more than to promote the "Progress of Science and useful Arts," perhaps the rules governing them need reconsideration.

Here are some sources of additional information and opinions on this subject

`http://lpf.ai.mit.edu`
(League for Programming Freedom web site)

`www.softwarepatents.co.uk`
(UK Resource on Software Patents: web site about UK and European software patents also discusses the U.S. experience)

`http://swpat.ffii.org/index.en.html`
(Foundation for a Free Information Infrastructure: German web site about European software patents)

## 7. Exercises

1. Consider $F(t) = 2t^5 + t^3 + 8t^2 + 1$ and $G(t) = t^2 + 3$ as polynomials with coefficients in $\mathbb{Z}_{11}$. Use long division to find a quotient $Q(t)$ and remainder $R(t)$ such that $F(t) = Q(t)G(t) + R(t)$ and $R(t)$ has smaller degree than $G(t)$.

2. Show that the polynomials $C_k(x)$ in Theorem 7 are uniquely determined by the properties (i) $C_k(x)$ has degree $k$, (ii) $C_k(i) = 0$ for $i = 0, 1, \ldots, k-1$, (iii) $C_k(k) = 1$.

3. Use the properties of $C_k(x)$ in Exercise 2 to show that $\Delta C_k(x) = C_{k-1}(x)$ for $k > 0$.

4. Show that a polynomial $f(x)$ has integer values $f(n)$ for every integer $n$ if and only if it can be written as

$$f(x) = a_0 C_0(x) + a_1 C_1(x) + \cdots + a_d C_d(x)$$

for some $d$ and integer coefficients $a_0, a_1, \ldots, a_d$.

5. Show that for integers $n \geq 0$, the value $C_k(n)$ is equal to the binomial coefficient $\binom{n}{k}$.

6. If you know that $1, 1, 2, 3, 3$ is the beginning of the sequence of values $f(0), f(1), f(2), \ldots$ of a polynomial of degree 3, find the next four terms in the sequence. [Hint: make a difference table, then extend each row four more terms, starting at the bottom.]

7. Find the polynomial $f(x)$ that produces the sequence in Exercise 6.

8. The *Hamming code* of order $k$ is a linear code over $\mathbb{Z}_2$ with $n = 2^k - 1$ code symbols and $m = 2^k - k - 1$ message symbols, given by the code matrix

$$\mathbf{C} = \left[\mathbf{I}_m \mid \mathbf{B}\right],$$

where $\mathbf{B}$ is an $m \times k$ matrix whose rows are all the possible row vectors of length $k$ with entries 0 and 1, and at least two 1's (note that the number of possible vectors is equal to $m$).

(a) Write out the code matrix for a Hamming code of order 3.

(b) Show that the Hamming code of order $k$ has weight 3, and therefore can correct one error. [Hint: this amounts to showing that every non-zero vector obtained as a sum of rows in $\mathbf{C}$ has at least three 1's in it.]

9. Show that the Hamming codes in Exercise 8 are *perfect* single-error correcting codes: every vector is either a code vector or has Hamming distance 1 from a code vector. This implies that no single-error correcting code over $\mathbb{Z}_2$ of the same length $n$ can have more message symbols than the Hamming code.

10. Show that a Reed-Solomon code with 1 message symbol and $n$ code symbols is just an $n$-fold redundancy code.

11. (a) Write out the code matrix $\mathbf{C}$ for the two-error correcting Reed-Solomon code over $\mathbb{Z}_7$ with 7 code symbols (the code used in the example in Section 5).

(b) Use the code matrix to encode the same message vector $\begin{bmatrix}2 & 3 & 4\end{bmatrix}$ as in the example, and check that your answer agrees with the one there.

12. For a linear code with $m \times n$ code matrix $\mathbf{C} = \left[\mathbf{I}_m|\mathbf{P}\right]$, show that a received vector $\mathbf{r}$ is a code vector (*i.e.*, has no errors) if and only if $\mathbf{rS} = \mathbf{0}$, where

$$\mathbf{S} = \left[\frac{\mathbf{P}}{-\mathbf{I}_{n-m}}\right].$$

13. Using a single-error correcting Reed-Solomon code over $\mathbb{Z}_{11}$ with 3 message symbols and 5 code symbols, you receive the vector $\begin{bmatrix}9 & 2 & 9 & 1 & 7\end{bmatrix}$. Where is the error and what were the transmitted code vector and the original message polynomial?

14. The *data rate* of an error correcting code is the fraction (number of message symbols)/(number of code symbols). The *error rate* of a transmission channel is the fraction of the code symbols transmitted in error. What is the maximum data rate achievable by a Reed-Solomon code if it must be able to tolerate a maximum error rate of 1/3? How is this better than a triple-redundancy code?