Decoding Reed-Solomon Codes

Bruce Maggs*

October 24^{th}, 2000

Contents

1 Overview of Reed-Solomon Codes

We begin by reviewing the basic elements of the Reed-Solomon system. We are working with polynomials over the field $GF(p^r)$. We are examining an RS system which corrects up to s

*Lecture notes scribed by Amitabh Sinha

errors in a message which originally had length k. The transmitted codeword then has length $n = k + 2s$.

Definition 1.1 *A* **message** *is a sequence $m_0, m_1, \ldots, m_{k-1}$ of elements from $GF(p^r)$. The associated* **message polynomial** *is $m(x) = m_0 + m_1x + m_2x^2 + \ldots + m_{k-1}x^{k-1}$. The* **length** *of the message is k.*

1.1 Reed-Solomon encoding

If α is a generator of $GF(p^r)$, we define a new polynomial $g(x) = (x-\alpha)(x-\alpha^2)(x-\alpha^4)\ldots(x-\alpha^{2s})$ for some fixed s. Now define

$$b(x) = x^{2s}m(x) \bmod g(x)$$

Definition 1.2 *The* **Reed-Solomon codeword** *for the message $m(x)$ is $C(x) = x^{2s}m(x) - b(x)$.*

We have chosen $C(x)$ so that every codeword is a multiple of $g(x)$. Thus the codewords form a k-dimensional subspace of the space of all polynomials of degree $n = k + 2s$ or less. We now simply transmit the coefficients of $C(x)$.

In the previous lecture we showed that $g(x)$ divides $x^n - 1$, and that there is a unique polynomial $h(x)$ of degree $n - 2s$ such that $g(x)h(x) = x^n - 1$. Hence the following definition.

Definition 1.3 *The* **check polynomial** *$h(x)$ of an RS system is the unique polynomial such that $g(x)h(x) = x^n - 1$.*

Let $R(x)$ be the received message. In the previous lecture we showed that there is a simple test to determine if whether $R(x)$ is a codeword, i.e., whether there are no errors in the received codeword. The test is given by the following lemma.

Lemma 1.1 *$R(x)h(x) = 0$ mod $x^n - 1$ iff $R(x)$ is a codeword.*

Thus, when a message arrives we first multiply by the check polynomial $h(x)$ and see if the result is 0 mod $x^n - 1$. If so, then there are no errors, and we can read the coefficients of $m(x)$ directly from the k high-order coefficients of $C(x)$. Otherwise, we have to undertake the more elaborate decoding procedure described in the rest of this lecture.

1.2 Decoding: preliminaries

The received polynomial $R(x)$ can be decomposed as:

$$\begin{array}{ccccc} R(x) & = & C(x) & + & E(x) \\ \uparrow & & \uparrow & & \uparrow \\ \text{received codeword} & & \text{codeword} & & \text{error polynomial} \end{array}$$

Let $E(x) = E_0 + E_1 x + \ldots + E_{n-1}x^{n-1}$ be the expansion of the error polynomial. We will assume that there are at most s errors. If there are more than s errors, all bets are off. Hence at most s of the coefficients E_i are non-zero. We will assume without loss of generality that exactly s of the coefficients are non-zero. It is not difficult to modify the approach described below to handle the case where there are fewer than s errors.

Let $j_1, j_2, \ldots, j_s$ denote the positions of the errors. Clearly each j_i is a distinct integer between 0 and $n-1$. We now define the error locations.

Definition 1.4 *The* **error location** X_i *is defined as* $X_i = \alpha^{j_i}$.

Since α is a generator of $GF(p^r)$, given any X_i, it is possible to determine the unique value j_i such that $X_i = \alpha^{j_i}$ by taking the discrete logarithm of X_i using base α. Hence the error locations $X_1, X_2, \ldots, X_s$ are just another way of representing the indices at which the errors occur. In our lectures on cryptography you heard that taking discrete logarithms was believed to be difficult. That's true and there is no contradiction here. In the end we are not going to take any discrete logarithms to compute the indices at which the errors occur. But we've introduced the X_i because they will prove to be more convenient computationally than using the indices explicitly, and the discussion of the discrete logarithm is just meant to show that the information content in the error locations is equivalent to that of the set of indices. Furthermore, we are working with a small, fixed-size finite set here, typically on the order of 256 elements, so that if we wanted we could easily implement the discrete logarithm using table lookup. In the cryptographic applications, the size of the field is very large – the number of elements might be greater than 10^{100}.

The error magnitudes are simply the non-zero coefficients of the error polynomial.

Definition 1.5 *The* **error magnitude** Y_i *is defined as* E_{j_i}.

Both X_i and Y_i are elements of $GF(p^r)$.

2 Decoder Architecture

2.1 Stages of the decoder

Figure 1 illustrates the main stages of the decoder. It starts with the received codeword $R(x)$, and goes on to output the corrected codeword $C(x)$ assuming there were s or fewer errors. We will now examine each of these layers in detail.

3 Syndrome Calculator

3.1 Syndromes

The first step in decoding a received message $R(x)$ is to compute its syndromes. The syndromes are given by the following definition.

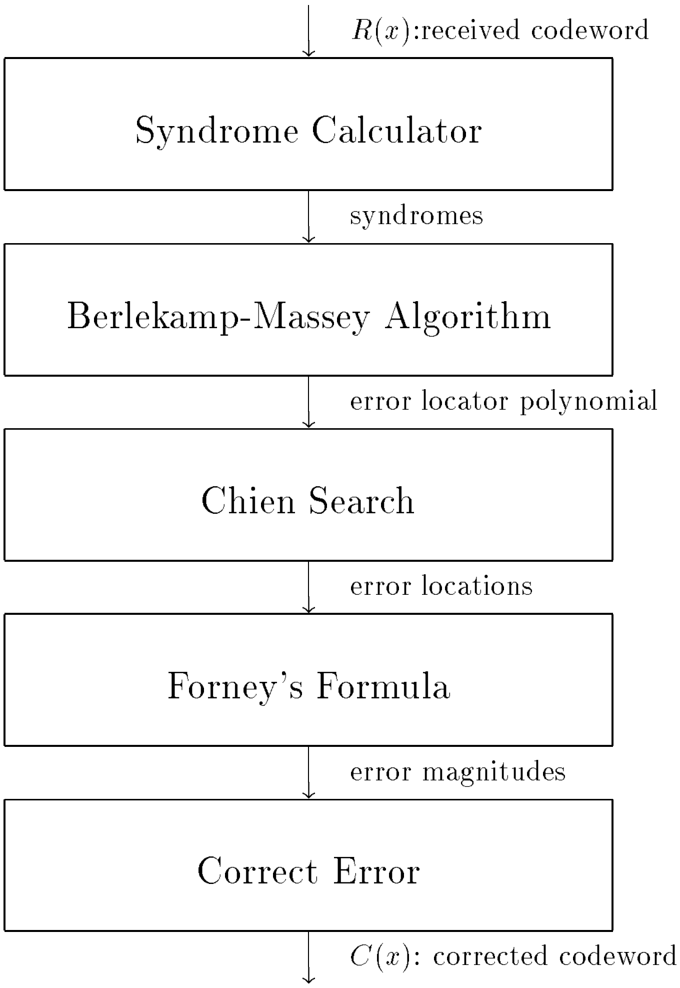

Figure 1: Decoder architecture

Definition 3.1 *The* **syndrome** s_l*, for* $1 \leq l \leq s$*, of the received message is the polynomial* $R(x)$ *evaluated at* α^l*, i.e.,* $s_l = R(\alpha^l)$*.*

Since $C(\alpha^l) = 0$, we have $s_l = R(\alpha^l) = E(\alpha^l)$ for $1 \leq l \leq 2s$. Hence the $2s$ syndromes give us $E(x)$ evaluated at $2s$ distinct points. Note that $E(x)$ is a polynomial with at most s non-zero coefficients. We are going to solve for $E(x)$, given these evaluations.

We note the following relation:

$$s_l = E(\alpha^l) = \sum_{i=1}^{s} Y_i \alpha^{l j_i} = \sum_{i=1}^{s} Y_i X_i^l$$

We can now define the **syndrome polynomial** using the syndromes as:

$$s(z) = \sum_{i=1}^{\infty} s_i z^i$$

Note that although $s(z)$ has a countably infinite number of terms, the relationship that $s_l = R(\alpha^l) = E(\alpha^l)$ only holds for the first $2s$ coefficients. It just turns out to be convenient to encode these coefficients in a generator function $s(z)$.

4 Berlekamp-Massey Algorithm

4.1 Error locator and evaluator polynomials

Having computed the syndromes, we will now solve for the the error locator and error evaluator polynomials. Once we have these polynomials, we will be able to find the error locations and the error magnitudes.

Definition 4.1 *The* **error locator** *polynomial* $\sigma(z)$ *is defined as:*

$$\sigma(z) = \prod_{i=1}^{s} (1 - X_i z)$$

Notice that the roots of $\sigma(z)$ are precisely the reciprocals of the error locations, that is, the roots are $1/X_1, 1/X_2, \ldots, 1/X_s$.

Definition 4.2 *The* **error evaluator** *polynomial* $\omega(z)$ *is defined as:*

$$\omega(z) = \sigma(z) + \sum_{i=1}^{s} z X_i Y_i \prod_{\substack{j=1 \\ j \neq i}}^{s} (1 - X_j z)$$

The *Berlekamp-Massey* procedure essentially solves for these two polynomials. We first derive an expression relating the three polynomials $s(z), \sigma(z)$ and $\omega(z)$. This is called the *key equation*.

$$\begin{aligned}
\frac{\omega(z)}{\sigma(z)} &= 1 + \sum_{i=1}^{s} \frac{zX_iY_i}{1 - X_iz} \\
&= 1 + \sum_{i=1}^{s} zX_iY_i \sum_{l=0}^{\infty} X_i^l z^l && \text{using the expansion } \tfrac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots \\
&= 1 + \sum_{l=1}^{\infty} \sum_{i=1}^{s} Y_i X_i^l z^l && \text{rearranging sums} \\
&= 1 + \sum_{l=1}^{\infty} s_l z^l && \text{by definition of } s_l \\
&= 1 + s(z)
\end{aligned}$$

Definition 4.3 *The* **key equation** *states that:*

$$(1 + s(z))\sigma(z) = \omega(z) \bmod z^{2s+1}$$

Note that although $s(z)$ may have more than $2s$ non-zero coefficients, the unknown polynomials $\sigma(z)$ and $\omega(z)$ have degree at most s, and so our solutions for $\sigma(z)$ and $\omega(z)$ better also have degree at most s.

Unfortunately, there does not seem to be any easy method to compute $\sigma(z)$ and $\omega(z)$ directly. The Berlekamp-Massey algorithm is an iterative procedure that generates a series of $2s$ pairs of polynomials $(\sigma^{(1)}(z), \omega^{(1)}(z)), (\sigma^{(2)}(z), \omega^{(2)}(z)), \ldots, (\sigma^{(2s)}(z), \omega^{(2s)}(z))$, where the last pair in the series is actually $(\sigma(z), \omega(z))$. At each iteration, we require that the key equation is satisfied:

$$(1 + s(z))\sigma^{(l)}(z) = \omega^{(l)}(z) \bmod z^{l+1}$$

Suppose we have computed $(\sigma^{(l)}(z), \omega^{(l)}(z))$ for some l. If we are lucky, the polynomials also satisfy the next key equation:

$$(1 + s(z))\sigma^{(l)}(z) = \omega^{(l)}(z) \bmod z^{l+2}$$

If this were the case, we could simply set $(\sigma^{(l+1)}(z), \omega^{(l+1)}(z)) = (\sigma^{(l)}(z), \omega^{(l)}(z))$. Unfortunately, we are not always so lucky. We usually have the following situation.

$$(1 + s(z))\sigma^{(l)}(z) = \omega^{(l)}(z) + \Delta_1^{(l)} z^{l+1} \bmod z^{l+2}$$

As we will see, the degree of $\omega^{(l)}(z)$ is at most l, so that $\Delta_1^{(l)}$ is the non-zero coefficient of z^{l+1} in

$$(1 + s(z))\sigma^{(l)}(z).$$

4.2 Two more polynomials

To get around this difficulty, we introduce two more polynomials $\tau^{(l)}(z)$ and $\gamma^{(l)}(z)$. We require that these polynomials also have degree no more than l. We also want them to satisfy the following equation:

$$(1+s(z))\tau^{(l)}(z) = \gamma^{(l)}(z) + z^l \bmod z^{l+1}$$

We now have the following rules for deriving $\sigma^{(l+1)}(z)$ and $\omega^{(l+1)}(z)$ from $\sigma^{(l)}(z)$, $\omega^{(l)}(z)$, $\tau^{(l)}(z)$, and $\gamma^{(l)}(z)$:

$$\begin{aligned}\sigma^{(l+1)}(z) &= \sigma^{(l)}(z) - \Delta_1^{(l)} z\tau^{(l)}(z)\\ \omega^{(l+1)}(z) &= \omega^{(l)}(z) - \Delta_1^{(l)} z\gamma^{(l)}(z)\end{aligned}$$

Note that to compute each of the new functions requires only a scalar multiplication (by $\Delta_1^{(l)}$), a shift (multiplication by z), and a subtraction. It is easy to check that the new pair $(\sigma^{(l+1)}(z), \omega^{(l+1)}(z))$ satisfies the key equation. Next we'll see that it is also not computationally difficult to compute $\tau^{(l+1)}(z)$ and $\gamma^{(l+1)}(z)$.

There are two natural and simple expressions for computing the pair of polynomials ($\tau^{(l+1)}(z)$, $\gamma^{(l+1)}(z)$) given the polynomials at iteration l.

$$\begin{aligned}(A)\quad \tau^{(l+1)}(z) &= z\tau^{(l)}(z)\\ \gamma^{(l+1)}(z) &= z\gamma^{(l)}(z)\\[1em] (B)\quad \tau^{(l+1)}(z) &= \frac{\sigma^{(l)}(z)}{\Delta_1^{(l)}}\\ \gamma^{(l+1)}(z) &= \frac{\omega^{(l)}(z)}{\Delta_1^{(l)}}\end{aligned}$$

Notice either choice requires only a scalar multiplication or a shift. It can be easily checked that whichever of these formulae we use, the following holds:

$$(1+s(z))\tau^{(l+1)}(z) = \gamma^{(l+1)}(z) + z^{l+1} \bmod z^{l+2}$$

We will choose either (A) or (B) so as to minimize the degrees of $\tau^{(l+1)}(z)$ and $\gamma^{(l+1)}(z)$. The exact choice of the formula to be used depends on a messy case analysis on the degrees of the four polynomials $\sigma^{(l)}(z), \omega^{(l)}(z), \tau^{(l)}(z), \gamma^{(l)}(z)$. We will now briefly overview this "mess". We essentially need to be able to deal with accidental cancellations of the high order terms.

To achieve this, Berlekamp and Massey introduce an upper bound $D(l)$ on the degrees of the polynomials with the invariant that

$$\begin{aligned}\text{degree } \sigma^{(l)}, \text{ degree } \omega^{(l)} &\leq D(l)\\ \text{degree } \tau^{(l)}, \text{ degree } \gamma^{(l)} &\leq l - D(l)\end{aligned}$$

This is a nice theoretical trick. When it is difficult to analyze the some quantity exactly, introduce a bound on that quantity that can be carefully controlled.

We use rule (A) if $\Delta_1^{(l)} = 0$ or $D(l) > \frac{l+1}{2}$, and in this case we have $D(l+1) = D(l)$. We use rule (B) if $\Delta_1^{(l)} \neq 0$ and $D(l) < \frac{l+1}{2}$, in which case we have $D(l+1) = l+1-D(l)$. These choices guarantee that $0 \leq D(l+1) \leq l+1$, and that degrees of $\sigma^{(l+1)}$ and $\omega^{(l+1)}$ are upper-bounded by $D(l+1)$, and the degrees of $\tau^{(l+1)}(z)$ and $\gamma^{(l+1)}(z)$ are upper-bounded by $l-D(l)$. It is easy to see that $0 \leq D(l) \leq l$. Note, however, that when we are all done the degree of $\sigma^{(2s)}(z) = \sigma(z)$ must be at most s, as does the degree of $\omega^{(s)}(z)$. So we are going to have to tighten things up a bit.

Notice that the above rules don't tell us how to handle the case when $\Delta_1^{(l)} \neq 0$ and $D(l) = \frac{l+1}{2}$. It turns out that either of the choices (A) and (B) will ensure that degree $\sigma^{(l+1)}$, degree $\omega^{(l+1)} \leq D(l+1)$ and degree $\tau^{(l+1)}$, degree $\gamma^{(l+1)} \leq l+1-D(l+1)$. But we can do even better. In this case we can ensure that one of the following strict inequalities hold.

$$\begin{aligned} \text{degree } \omega^{(l+1)} &< D(l) \\ \text{degree } \gamma^{(l+1)} &< l - D(l) \end{aligned}$$

To take advantage of this, we introduce one more parameter $B(l)$, which takes on only one of the two values, either 0 or 1. We then use rule (A) if $\Delta_1^{(l)} \neq 0, D(l) = \frac{l+1}{2}$ and $B(l) = 0$. We use rule (B) if $\Delta_1^{(l)} \neq 0, D(l) = \frac{l+1}{2}$ and $B(l) = 1$. When using rule (A), we update $B(l+1) = B(l)$, and when using rule (B) we update $B(l+1) = 1 - B(l)$.

This keeps the degree inequalities satisfied:

$$\begin{aligned} \text{degree } \omega^{(l)}(z) &\leq D(l) - B(l) \\ \text{degree } \gamma^{(l)}(z) &\leq l - D(l) - (1 - B(l)) \end{aligned}$$

We will not prove it here, but the rules given above ensure that the degrees of $\sigma^{(l)}(z)$ and $\omega^{(l)}(z)$ do not grow to bee too large, as the following lemma shows.

Lemma 4.1

$$\begin{aligned} \textit{degree } \sigma^{(l)} &\leq \frac{l+1}{2} \\ \textit{degree } \omega^{(l)} &\leq \frac{l}{2} \end{aligned}$$

This lemma ensures that the degree of $\sigma(z) = \sigma^{(2s)}(z)$ is at most s, as is the degree of $\omega(z)$.

We are now ready to define the initial conditions:

$$\begin{aligned} \sigma^{(0)}(z) &= 1 \\ \omega^{(0)}(z) &= 1 \\ \tau^{(0)}(z) &= 1 \\ \gamma^{(0)}(z) &= 0 \\ D(0) &= 0 \\ B(0) &= 0 \end{aligned}$$

This completes the Berlekamp-Massey algorithm to compute the polynomials $\sigma(z)$ and $\omega(z)$.

5 Chien Search

Having solved for $\sigma(z)$, we want to compute the error locations X_i. We recall the definition of the error locator polynomial:

$$\sigma(z) = \prod_{i=1}^{s}(1 - X_i z)$$

We first notice that the roots of $\sigma(z)$ are in fact $1/X_i$. So it would seem natural to just compute the roots and take reciprocals.

5.1 Chien's procedure

However, computing the roots of the polynomial may be very inefficient. Chien's idea begins by recalling that we are working over a small finite field. So we can just enumerate all the elements of the field, and test whether the polynomial evaluates to zero.

Chien's algorithm

1. Let α be a generator of $GF(p^r)$.
2. Initialise $\{X_i\}$ to the empty set.
3. For $l = 1, 2, \ldots$
   If $\sigma(\alpha^l) = 0$, add α^{-l} to $\{X_i\}$.

If $\sigma(\alpha^l) = 0$, then α^l is a root of $\sigma(z)$, so that α^{-l} is an error location. Note that Chien's algorithm gives us the indices in which the errors occur, $j_1, j_2, \ldots, j_s$ directly. When we add α^{-l} to $\{X_i\}$, we can also add l to $\{j_i\}$. Thus we don't have to compute a discrete logarithm. Also note that we don't really have to bother computing α^{-l}. Since we are working over a small finite field we can simply do a table lookup.

It is also easy to compute $\sigma(\alpha^{l+1})$ from $\sigma(\alpha^l)$ quickly, which makes this procedure efficient. Notice that if

$$\sigma(z) = 1 + \sigma_1 z + \sigma_2 z^2 + \sigma_3 z^3 + \cdots + \sigma_s z^s,$$

then

$$\sigma(\alpha^l) = 1 + \sigma_1 \alpha^l + \sigma_2 \alpha^{2l} + \sigma_3 \alpha^{3l} + \cdots + \sigma_s \alpha^{sl},$$

and

$$\sigma(\alpha^{l+1}) = 1 + \sigma_1 \alpha^{l+1} + \sigma_2 \alpha^{2l+2} + \sigma_3 \alpha^{3l+3} + \cdots + \sigma_s \alpha^{sl+s}.$$

Hence the ith term in $\sigma(\alpha^{l+1})$ can be computed from the ith term in $\sigma(\alpha^l)$ by multiplying that term by α^i.

6 Forney's Formula

Now that we have found the error locations $\{X_i\}$, we just need to compute the error magnitudes $\{Y_i\}$.

6.1 Error magnitude computation

Forney's formula tells us that if we evaluate $\omega(z)$ at X_l^{-1}, we get

$$\begin{aligned}
\omega(X_l^{-1}) &= \sigma(X_l^{-1}) + \sum_{i=1}^{s} X_l^{-1} X_i Y_i \prod_{\substack{j=1 \\ j \neq i}}^{s} (1 - X_j X_l^{-1}) \\
&= Y_l \prod_{\substack{j=1 \\ j \neq l}}^{s} (1 - X_j X_l^{-1})
\end{aligned}$$

This is true because $\sigma(X_l^{-1}) = 0$ since X_l^{-1} is a root of $\sigma(z)$.

We can now compute Y_l.

$$\begin{aligned}
Y_l &= \frac{\omega(X_l^{-1})}{\prod_{\substack{j=1 \\ j \neq l}}^{s} (1 - X_j X_l^{-1})} \\
&= \frac{X_l^s \omega(X_l^{-1})}{X_l \prod_{\substack{j=1 \\ j \neq l}}^{s} (X_l - X_j)} \\
&= \frac{\tilde{\omega}(X_l)}{X_l \prod_{\substack{j=1 \\ j \neq l}}^{s} (X_l - X_j)}
\end{aligned}$$

where we define $\tilde{\omega}(z) = z^s \omega(1/z)$.

Chien's algorithm gave us the indices of the errors, and Forney's formula gave us the error magnitudes. Thus, we have a solution for the error polynomial $E(x)$. We can now compute the codeword $C(x)$ using the formula $C(x) = R(x) - E(x)$, and the decoding is complete.