Implementing Reed-Solomon

Andrew Brown
Recall

- Reed-Solomon represents messages as polynomials and over-samples them for redundancy.

- An \((n, k, n - k + 1)\) code has
  - \(k\) digit messages
  - \(n\) digit codewords
  - \(n - k + 1\) distance between codewords (at least)
  - \((n - k)/2\) errors before it cannot be decoded
  - \(2s = n - k\)

- In this presentation, all messages and codewords are over the finite field \(GF(2^8)\). This makes byte-oriented implementation easy.
Recall

- Generator Polynomial:
  \[ g(x) = (x - \alpha)(x - \alpha^2) \cdots (x - \alpha^{n-k}) \]
  \( \alpha \) is a generator element in \( GF(2^8) \)

- Encoding Process:
  \( m \) is the message encoded as a polynomial
  \[ m' = mx^{2s} \]
  \[ b = m' \pmod{g} \]
  \[ m' = qg + b \text{ for some } q \]
  \[ c = m' - b \]

- Codewords are multiples of \( g \), and are systematic

- Verifying a codeword is valid is a matter of checking for divisibility by \( g \)
Decoding Procedure Overview

1. Calculate Syndromes

2. **Berlekamp-Massey Algorithm** - calculates the Error Locator Polynomials and Error Evaluator Polynomials

3. **Chien Search** - Finds the error locations using the Error Locator Polynomial

4. **Forney’s Formula** - Finds the error magnitudes using the error evaluator polynomial

5. Correct the Errors
Decoding (Defining Terms)

- **Error Polynomial**

\[ R(x) = C(x) + E(x) \]

\[ E(x) = E_0 + E_1x + \cdots + E_{n-1}x^{n-1} \]

- Has at most \( s \) coefficients that are non-zero

- **Error Positions**

\( j_1, j_2, \cdots, j_s \), each a value between 0 and \( n - 1 \)

- **Error Locations**

\[ X_i = \alpha^{j_i} \]

- **Error Magnitudes**

\[ Y_i = E_{j_i} \]

- Notice that there are \( 2s \) unknowns
Decoding ( Syndromes )

- Step 1: Calculate the first $2s$ syndromes
- Syndromes are defined for all $l$:

$$s_l = \sum_{i=1}^{s} Y_i X_i^l$$

- For the first $2s$, it reduces to:

$$s_l = E(\alpha^l) = \sum_{i=1}^{s} Y_i \alpha^l j_i \quad 1 \leq l \leq 2s$$

- $s_l = R(\alpha^l) = E(\alpha^l)$ for the first $2s$ powers of $\alpha$.
- Equivalent to having $2s$ equations with $2s$ unknowns
Decoding ( Syndromes )

- Encode the syndromes in a generator polynomial:

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

- This can be computed by finding each \( s_i \) from the received codeword for the first \( 2s \) values. That’s all we need though.
Berlekamp-Massey Algorithm

- Input: Syndrome polynomial from the last slide
- Output: Error Locator Polynomial $\sigma(z)$ and Error Evaluator Polynomial $\omega(z)$. Defined as:

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

\[
\omega(z) = \sigma(z) + \sum_{i=1}^{s} zX_i Y_i \prod_{j=1}^{s} (1 - X_j z)
\]

- Notice that the error locations are the inverse roots of $\sigma(z)$. (Roots are $1/X_1, 1/X_2, \cdots 1/X_s$)
Observe the following relation:

\[
\frac{\omega(z)}{\sigma(z)} = 1 + \sum_{i=1}^{s} \frac{zX_i Y_i}{1 - X_i z}
\]

= ...intermediate steps omitted

= \(1 + s(z)\)

Key equation thus states:

\[(1 + s(z))\sigma(z) \equiv \omega(z) \pmod{z^{2s+1}}\]

\(\sigma(z)\) and \(\omega(z)\) have degree at most \(s\)

Key Equation represents a set of \(2s\) equations and \(2s\) unknowns
B-M (procedure)

- B-M iterates $2s$ times
- At each iteration, it produces a pair of polynomials:
  \[(\sigma(l)(z), \omega(l)(z))\]
- where the polynomials satisfy that iteration’s key equation:
  \[(1 + s(z))\sigma(l)(z) \equiv (\mod z^{l+1}) \omega(l)(z)\]
B-M (procedure)

- Once we have
  \[ (\sigma(l)(z), \omega(l)(z)) \]
  for some \( l \). If we’re lucky, they already satisfy the next key equation:
  \[
  (1 + s(z))\sigma(l)(z) \equiv (\text{mod } z^{l+2}) \omega(l)(z)
  \]
  in which case we can set \( \sigma(l+1)(z) = \sigma(l)(z) \) and similarly for \( \omega(z) \).
- However, usually we have an unwanted higher-order term:
  \[
  (1 + s(z))\sigma(l)(z) \equiv (\text{mod } z^{l+2}) \omega(l)(z) + \Delta(l)z^{l+1}
  \]
B-M (procedure)

- $\Delta(l)$ is the non-zero coefficient of $z^{l+1}$ in $(1 + s(z))\sigma(l)(z)$
- Basic idea is to iteratively improve estimates of $\sigma$ and $\omega$
- But since there may be a higher order term, we can’t always just extend to $l + 1$ from iteration $l$
- A complex set of rules determines how to handle different cases
- The next 5 slides describe these cases and how to handle them
\( \Delta(l) \) is the non-zero coefficient in \((1 + s(z))\sigma(l)(z)\)

To find the next iteration’s polynomials, we introduce two more polynomials \( \tau(l)(z) \) and \( \gamma(l)(z) \)

They must satisfy:

\[
(1 + s(z)) \tau(l)(z) \equiv (\mod z^{l+1}) \gamma(l)(z) + z^l
\]

And we have the following rules to derive the next \( \sigma \) and \( \omega \):

\[
\sigma(l+1)(z) = \sigma(l)(z) - \Delta(l)z\tau(l)(z)
\]
\[
\omega(l+1)(z) = \omega(l)(z) - \Delta(l)z\gamma(l)(z)
\]
But how to compute $\tau_{(l+1)}(z)$ and $\gamma_{(l+)}(z)$?

Use one of the following rules:

(A) $\tau_{(l+1)}(z) = z\tau_{(l)}(z)$
$\gamma_{(l+1)}(z) = z\gamma_{(l)}(z)$

(B) $\tau_{(l+1)}(z) = \frac{\sigma_{(l)}(z)}{\Delta_{(l)}}$
$\gamma_{(l+1)}(z) = \frac{\omega_{(l)}(z)}{\Delta_{(l)}}$
One of (A) or (B) is chosen each iteration to minimize the degrees of $\tau_{(l+1)}(z)$ and $\gamma_{(l+1)}(z)$

To choose, define a single value $D(l)$ for each iteration

Choose rule (A) if $\Delta(l) = 0$ or $D(l) > \frac{l+1}{2}$

Choose rule (B) if $\Delta(l) \neq 0$ and $D(l) < \frac{l+1}{2}$

With rule (A) set $D(l+1) = D(l)$

With rule (B) set $D(l+1) = l + 1 - D(l)$

These rules and conditions ensure $0 < D(l+1) \leq l + 1$ and the degrees of $\sigma_{(l+1)}$ and $\omega_{(l+1)}$ are upper-bounded by $D(l+1)$ and degrees of $\tau_{(l+1)}$ and $\gamma_{(l+1)}$ are upper-bounded by $l - D(l)$
But what about when $\Delta(l) \neq 0$ and $D(l) = \frac{l+1}{2}$?

Either rule works, but to do even better, define one last value, a binary value $B(l)$, for each iteration

- When $B(l) = 0$ use rule (A)
- When $B(l) = 1$ use rule (B)

With rule (A) set $B(l+1) = B(l)$
With rule (B) set $B(l+1) = 1 - B(l)$

This keeps the degree inequalities satisfied:

$$\deg \omega(l)(z) \leq D(l) - B(l)$$
$$\deg \gamma(l)(z) \leq l - D(l) - (1 - B(l))$$
All those rules ensure the degrees of $\sigma$ and $\omega$ do not grow too large. Each step they satisfy:

\[
\deg \sigma_{(l)} \leq \frac{l + 1}{2} \\
\deg \omega_{(l)} \leq \frac{l}{2}
\]

Last piece: the initial conditions:

\[
\begin{align*}
\sigma(0)(z) & = 1 \\
\omega(0)(z) & = 1 \\
\tau(0)(z) & = 1 \\
\gamma(0)(z) & = 0 \\
D(0) & = 0 \\
B(0) & = 0
\end{align*}
\]
Decoding: Next Steps

- Now we have the Error Locator Polynomial $\sigma(z)$ and the Error Evaluator Polynomial $\omega(z)$
- Chien’s Search takes $\sigma(z)$ and outputs the error locations/positions ($X_i$ and $j_i$)
- Forney’s Formula takes $\omega(z)$ and the array $X_i$ of error locations outputs the error magnitudes ($Y_i$)
Recall the definition of $\sigma(z)$:

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

Now that we have $\sigma(z)$, finding the array of $X_i$ values is simply a matter of solving for the roots.

The Easy Way: since we’re working over a small field, just test every value.

1. Let $\alpha$ be a generator.
2. Initialize $\{X_i\}$ to the empty set.
3. For $l = 1, 2, \ldots$
   - If $\sigma(\alpha^l) = 0$: add $\alpha^{-l}$ to $\{X_i\}$.
Chien’s Procedure

- But we can do better than evaluating it 255 times!
- If we have computed the $\alpha^l$th evaluation, we get:

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

- Then, computing $\sigma(\alpha^{l+1})$ is an $O(s)$ operation:

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

- The $i$th term in $\sigma(\alpha^{l+1})$ can be computed from the $i$th term in $\sigma(\alpha^l)$ by multiplying that term by $\alpha^i$
Forney’s Formula

Using the Error Evaluator Polynomial $\omega(z)$ and the error locations $\{X_i\}$, the error magnitudes $\{Y_i\}$ can be computed

$$\omega(z) = \sigma(z) + \sum_{i=1}^{s} zX_iY_i \prod_{j=1 \atop j \neq i}^{s}(1 - X_jz)$$

Evaluate at $X_l^{-1}$

$$\omega(X_l^{-1}) = \sigma(X_l^{-1}) + \sum_{i=1}^{s} X_l^{-1}X_iY_i \prod_{j=1 \atop j \neq i}^{s}(1 - X_jX_l^{-1})$$
Forney’s Formula

\[ \omega(X_l^{-1}) = \sigma(X_l^{-1}) + \sum_{i=1}^{s} X_l^{-1}X_i Y_i \prod_{j=1, j\neq i}^{s} (1 - X_j X_l^{-1}) \]

Then simplifies to:

\[ = Y_l \prod_{j=1, j\neq l}^{s} (1 - X_j X_l^{-1}) \]

since \( \sigma(X_l^{-1}) = 0 \)
Forney’s Formula

\[ \omega(X_l^{-1}) = Y_l \prod_{\substack{j=1 \atop j \neq l}}^{s} (1 - X_j X_l^{-1}) \]

Can then be solved for \( Y_l \):

\[ Y_l = \frac{\omega(X_l^{-1})}{\prod_{\substack{j=1 \atop j \neq l}}^{s} (1 - X_j X_l^{-1})} \]

And that can be directly computed. We know all the values on the right hand side!
Putting it all together

- We know:
  - \( \{X_i\} \) The error locations
  - \( \{Y_i\} \) The error magnitudes
- Put them together to build the Error Polynomial \( E(x) \)
- Then subtract to get the codeword!

\[
C(x) = R(x) - E(x)
\]
Reed-Solomon Implementation

The rest of the presentation is about my implementation

- Done in Python with no external libraries or dependencies
- Implemented a Finite Field class for $GF(2^8)$
- Implemented a Polynomial Class for manipulating polynomials
- Implemented the RS algorithms as described
Finite Fields

- Created a Python class that subclasses int
- Instances are integers, which represent the corresponding finite field element when translated to a polynomial

\[ 51 = 00110011 = x^5 + x^4 + x + 1 \]

- Overwrote addition, subtraction, multiplication, division, and exponentiation for finite field arithmetic
- Multiplication defined using an exponentiation table and a logarithm table, pre-generated
Finite Fields (multiplication)

```python
exptable = (1, 3, 5, 15, 17, 51, ... 246, 1)

- This table holds all powers of 3
- exptable[1] = 3
- exptable[255] = 1

logtable = (None, 0, 25, 1, 50, 2, ... 112, 7)

- This table holds all logarithms in base 3
- logtable[3] = 1
- logtable[17] = 4
  (since $3^4 = 17$)
- logtable[0] is an error
```
Finite Fields (multiplication)

exptable = (1, 3, 5, 15, 17, 51, \ldots 246, 1)
logtable = (None, 0, 25, 1, 50, 2, \ldots 112, 7)

These tables together define multiplication like this:

```python
def multiply(a, b):
    x = logtable[a]
    y = logtable[b]
    z = (x + y) % 255
    return exptable[z]
```
Finite Fields (more)

exptable = (1, 3, 5, 15, 17, 51, ... 246, 1)
logtable = (None, 0, 25, 1, 50, 2, ... 112, 7)

Exponentiation and multiplicative inverses also use these tables:

```python
def power(a, b):
    x = logtable[a]
    z = (x * b) % 255
    return exptable[z]

def inverse(a):
    e = logtable[a]
    return exptable[255 - e]
```

Implementing Reed-Solomon – p. 29
Polynomial Class

- Stores numbers from high degree to low degree
- All coefficient math is done using regular Python operators
- Compatible with both integers and field elements as coefficients
- Supports long division and remainders (essential for RS coding)
Since the polynomial class abstracts polynomial math away, encoding boils down to basically:

```python
def encode(m):
    mprime = m * xshift
    b = mprime % g
    c = mprime - b
    return c
```

Implementing Reed-Solomon – p. 31
Decoding is also fairly simple:

def decode(r):
    sz = syndromes(r)
    sigma, omega = berlekamp_massey(sz)
    X, j = chien_search(sigma)
    Y = forney(omega, X)

    # There is a loop to build E here
    ...

    return r - E
Reed Solomon Decoding

- My implementation of those functions are straight up implementations of the math. Nothing surprising.

```python
def syndromes(r):
    s = [GF256int(0)]
    for l in range(1, n-k+1):
        s.append(r.evaluate(GF256int(3)**l))
```

- My Chien Search isn’t actually Chien’s search though, it just evaluates the polynomial 255 times:

```python
p = GF256int(3)
for l in range(1,256):
    if sigma.evaluate( p**l ) == 0:
        X.append( p**(-l) )
        j.append(255 - l)
```
Implementation Notes

- Message to Polynomial translations
  1. “hello”
  3. $104x^4 + 101x^3 + 108x^2 + 108x^1 + 111$

- Messages are effectively left-padded with null bytes
Example

- RS(20,13) code: 13 message bytes and 7 parity bytes. Can correct 3 errors.
- Message: “Hello, world!”
- Codeword: “Hello, world![8d][13][f4][f9][43][10][e5]”
- R: “[00][00][00]lo, world![8d][13][f4][f9][43][10][e5]”
- Decoded: “Hello, world!”

And, to prove this isn’t faked...
As an example, I have written a program that encodes codewords as rows in an image

- Uses RS(255,223)
- Encodes each symbol as a pixel in a grayscale image
- Each row is a codeword

Decodes to:

ALICE’S ADVENTURES IN WONDERLAND
Alice was beginning to get very tired of sitting by her sister on the ...
Demo!

- Since each row is a RS(255,223) codeword, it can handle up to 16 pixel errors per row.
- Drawing 5 px stripes, each of the following still decodes: