Reed Solomon Encoding: Simplified Explanation for Programmers

Frimpong Twum[1], J.B. Hayfron-Acquah[2], W.W. Oblitey[3], William Morgan-Darko[4]

[1,2,4]*Department of Computer Science, Kwame Nkrumah University of Science and Technology, Kumasi, Ghana*

[3]*Department of Computer Science, Indiana University of Pennsylvania, USA*

***Abstract*- Several articles and books have been written on Reed Solomon Coding, detailing the steps required to arrive at a Reed Solomon (RS) codeword. However, the majority of these materials require a strong understanding of Mathematics in order to make sense of the explanations they offer. This often proves to be a great disadvantage to programmers who come from a background of little or basic Mathematics. The paper presents Reed Solomon Coding in a simplified step-by-step sequence which can be used as guide for those who do not understand the complex Mathematics which other papers provide.**

Keywords - Reed Solomon Encoding, Error detection, Error Correction, Encoding Polynomial, Polynomial Arithmetic, Finite Field, Galois Field, Error Locator Polynomial, Algorithm, Pseudocode

I. INTRODUCTION

In 1960, Irving Reed and Gastowe Solomon proposed a system for encoding and decoding data that is quite versatile and efficient [1]. Reed Solomon encoding works by adding special extra (parity) data (t) to the original data (k) being stored or transmitted [2]. The extra data becomes handy when a portion of the transmitted or stored data (n) gets corrupted or lost and needs to be corrected or regenerated (n = k + t). Reed Solomon recognizes two types of data corruption as follows [1], [2]:

Errors: With an error (data bits altering), the location of the corrupted or modified data is not known and must be computed as the roots of a polynomial referred to as error locator polynomial. The popular methods for Reed Solomon coding can detect any number of corrupt data up to half the number of parity data.

Erasures: With erasures (data loss as a result of deletion), the location of the lost data is known. Hence there is no need to compute the locations. Reed Solomon coding can correct any number of erasures up to the number of parity data.

II. LITERATURE ON REED SOLOMON ENCODING

Reed Solomon (RS) limits the numbers it uses to a finite field (also called Galois Field) [3]. A finite field is a set of numbers with rules such that all additions, subtractions, multiplications and divisions have results within the set [4], [5]. Hence powers, modulus and logarithms also have their results within the finite field. For use in programming, the preferred finite fields have a size that is a power of two [6], [7]. This introduces some interesting properties for the finite field arithmetic. The denotation for such fields is $\mathrm{GF}(2^{\mathrm{m}})$.

https://sites.google.com/site/ijcsis/
ISSN 1947-5500

RS coding enables data errors and erasures to be detected and corrected by appending to the transmitting data, a redundant data in the form of parity that is/are obtained through a long division process by using the coefficients of the encoding polynomial to divide the bytes of the transmitting data to obtain a remainder which is appended to the original data to generate the RS codeword that is used for detection and correction of data corruption [2]. As an example, suppose transmitting a text file with content as 1 2 3 4 5 6 which has respective ASCII representation as 49 50 51 52 53 54. By using an encoding polynomial in GF (256) with parity of 2 given as $x^2 + 6x + 8$ or 1 6 8, we obtain the RS encoded output file as 49 50 51 52 53 54 186 244 through the following process [8]:

The original transmitting message is given as

$$M(x) = 49x^5 + 50x^4 + 51x^3 + 52x^2 + 53x + 54$$

The first step in generating the RS codeword for above message is to multiply the message by x^2 to create memory spaces for storing 2 Forward Error Correction (FEC) codes [3], RS[255, 6] 8, 2 as follows:

$$M(x) = 49x^7 + 50x^6 + 51x^5 + 52x^4 + 53x^3 + 54x^2 + 0x + 0$$

The encoding polynomial of 168 is used to divide into $M(x)$ above through polynomial long division and Galois Field arithmetic as follows:

```
                  49  148  249  204   60  144
      1   6   8 | 49   50   51   52   53   54    0    0
                  49  166  149
                 ---------------
                     148  166   52
                     148   95  212
                    ----------------
                          249  224   53
                          249   44  155
                         ----------------
                               204  174   54
                               204  146   46
                              ----------------
                                    60   24    0
                                    60  136  253
                                   ----------------
                                         144  253    0
                                         144   71  244
                                        ----------------
                                               186  244
```

The RS codeword is finally generated by replacing the values in $M(x)$ where the 2 FEC symbols are with the remainder values obtained from the long division as follows:

$$M(x) = 49x^7 + 50x^6 + 51x^5 + 52x^4 + 53x^3 + 54x^2 + 186x + 244$$

The remainders are used for error detection and recovery.

III. METHODOLOGY

A. Generation of the finite field

The elements of the finite field are the integers 0 through $2^m - 1$. Aside from 0, all the other field elements can be represented as a power of 2. A prime number is used to correct repetitions when the power of 2 exceeds the field size and needs to be wrapped to fit back within the field. An example is shown below of the representation of the finite field $GF(2^3)$ as powers of 2, using the prime number 11 to handle or avoid repetitions.

Powers of 2	Expansion	Field element
2^0	2^0	1
2^1	2^0 x 2 = 1 x 2	2
2^2	2^1 x 2 = 2 x 2	4
2^3	2^2 x 2 = 4 x 2 = 8 ≡ 8 XOR 11	3

https://sites.google.com/site/ijcsis/
ISSN 1947-5500

2^4	2^3 x 2 = 3 x 2	6
2^5	2^4 x 2 = 6 x 2 = 12 ≡ 12 XOR 11	7
2^6	2^5 x 2 = 7 x 2 = 14 ≡ 14 XOR 11	5
2^7	2^6 x 2 = 5 x 2 = 10 ≡ 10 XOR 11	1

For every field size, there is a set of prime numbers which can be used to uniquify the powers of 2 when they fall outside the field range [2]. The prime numbers used are usually found between 2^m and 2^{m+1}. In the case of the example above where the finite field is $GF(2^3)$, prime numbers within the range 2^3 to 2^4 can be used. The prime numbers that qualify are 11 and 13. Another example is shown below, using 13 to handle or avoid repetitions.

Powers of 2	Expansion	Field element
2^0	2^0	1
2^1	2^0 x 2 = 1 x 2	2
2^2	2^1 x 2 = 2 x 2	4
2^3	2^2 x 2 = 4 x 2 = 8 ≡ 8 XOR 13	5
2^4	2^3 x 2 = 5 x 2 = 10 ≡ 10 XOR 13	7
2^5	2^4 x 2 = 7 x 2 = 14 ≡ 14 XOR 13	3
2^6	2^5 x 2 = 3 x 2 = 6	6
2^7	2^6 x 2 = 6 x 2 = 12 ≡ 12 XOR 11	1

As can be seen, using different prime numbers results in different assignments of powers of 2 to the field elements. However, all the elements are accounted for.

Expressing the field elements as powers of 2 allows us to determine the logarithms of the field elements easily. For instance, if $2^5 = 3$, it follows that $\log(3) = 5$. The pseudocode below describes how to populate two arrays, one to hold the field elements in an order that shows the power of two that they correspond to, and the other to hold the logarithms of the field elements.

```
:: function generateExponent
       AndLogarithmArrays()
::     element = 1
::     for i = 0 to m − 2
::         exponent[i] = element
::         log[element] = i
::         element = element × 2
::         if element ≥ field_size
::             element
= element XOR primeNumber
::         end if
::     end for
:: end function
```

B. Arithmetic in the finite field

Addition and subtraction in $GF(2)$ can be summed up as follows

+	0	1
0	0	1
1	1	0

−	0	1
0	0	1
1	1	0

XOR	0	1
0	0	1
1	1	0

Comparing with the XOR operation on bits, it is easy to see that both addition and subtraction boil down to an XOR operation on the field elements. The pseudocode below describes the function to add field elements. The same function works for subtraction.

```
:: function galois_add(x, y)
::     return (x XOR y)
:: end function
```

Multiplication in the finite field is easily performed using the logarithms. From the logarithm rule that

$$\log(a \times b) = \log(a) + \log(b)$$

a change of subject modifies the rule to

$$a \times b = 2^{\log(a)+\log(b)}$$

The pseudocode below describes the function to multiply two field elements

```
:: function galois_multiply(a, b)
::     if a = 0 OR b = 0
::         return 0
::     else
```


https://sites.google.com/site/ijcsis/
ISSN 1947-5500

```
::          return exponent[(log(a)
                + log(b))mod (m − 1)]
::      end if
:: end function
```

Division in the finite field uses a similar logarithm rule:

$$\log(a \div b) = \log(a) - \log(b)$$

Changing the subject gives

$$a \div b = 2^{\log(a) - \log(b)}$$

The pseudocode below describes the function to perform division in the finite field.

```
:: function galois_divide(a, b)
::      if b = 0
::          return null
::      else if a = 0
::          return 0
::      else
::          return exponent[log(a) − log(b)]
```

C. Polynomials in Reed Solomon

Reed Solomon coding treats data as polynomials and manipulates them as such. At the heart of this encoding technique is the production of a message polynomial $M(x)$ that is perfectly divisible by another predetermined polynomial (the encoding polynomial) [9].

The Encoding Polynomial

Generation of the coefficients for the generating polynomial

The Generating polynomial is of the form

$$g(x) = (x + \alpha^1)(x + \alpha^2)(x + \alpha^3) \ldots (x + \alpha^n)$$

where n is the number of parity data being added.

The coefficients of x in the expansion of $g(x)$ will be found using the algorithm below

```
initialise the encoding polynomial to 1
for pow = 1 to n
 create a monomial whose constant is
    alpha raised to pow and the
    coefficient of x is 1
 multiply the monomial by
```

For example, to find the coefficients for $n = 4$, assuming the Galois Field elements are

- $\alpha^1 = a$
- $\alpha^2 = b$
- $\alpha^3 = c$
- $\alpha^4 = d$

Then the looping process will perform the following action

- $pow = 1: \{1\} \times \{a, 1\}$
- $pow = 2: \{1\} \times \{a, 1\} \times \{b, 1\}$
- $pow = 3: \{1\} \times \{a, 1\} \times \{b, 1\} \times \{c, 1\}$
- $pow = 4: \{1\} \times \{a, 1\} \times \{b, 1\} \times \{c, 1\} \times \{d, 1\}$

Thus, we will

1. *initialise the encoding polynomial to* 1
2. *Loop n times* $(i \rightarrow 1$ *to* $n)$

a. *Generate a polynomial of the form* $(x + \alpha^i)$

b. *Multiply the currently generated poynomial by the encoding polynomial*

c. *Save the polynomial product as the new encoding polynomial*

D. Polynomial arithmetic

It has been seen from the finite field arithmetic above that addition and subtraction amount to the same thing in $GF(2)$. The procedure for adding one polynomial to another involves simply performing an XOR operation on the array elements that have the same index. The pseudocode below describes the algorithm for adding two polynomials.

```
:: function polynomialAddition(a[ ], b[ ])
::      longer [ ] = longer of the
            two arrays(a, b)
```


https://sites.google.com/site/ijcsis/
ISSN 1947-5500

```
::      shorter[ ] = shorter of the
                two arrays(a, b)
::      for i = 0 to length of shorter
::          longer[i]
                = shorter[i] XOR longer[i]
::      end for
::      return longer
:: end function
```

Multiplication is based on the principle shown below

$$ax^n \times bx^m = abx^{n+m}$$

The pseudocode below describes a function to multiply two polynomials.

```
:: function polynomialMultiplication(a[ ], b[ ])
::      for i = 0 to length of a
::          for j = 0 to length of b
::              product[i + j]
                 = product[i + j] XOR
                 galois_multiply(a[i], b[j])
::          end for
::      end for
:: end function
```

Division is based on polynomial long division. The pseudocode below describes a function to perform polynomial long division.

```
:: function polynomial_division(a[ ], b[ ])
::      for i = 0 to (length of a
                - length of b)
::          quotient[i]
                = galois_divide(b[0], a[i])
::          for j = 0 to length of b
::              a[i + j] = a[i + j] XOR
             galois_multiply(quotient[i], b[j])
::          end for
::      end for
::      remainder[ ] = subarray of a
          beginning at index (length of a
                - length of b)
::      return quotient, remainder
:: end function
```

The Reed Solomon encoding process requires that the message polynomial be shifted a number of degrees up. The pseudocode below describes a function to shift the polynomial a number of degrees up.

```
:: function polynomial_shift(a[ ], n)
::      for i = 0 to n − 1
::          result[i] = 0
::      end for
::      for i = n to (length of a + n − 1)
::          result[i] = a[length of a − i]
::      end for
:: end function
```

E. Reed Solomon Encoding

The underlying principle of Reed Solomon Encoding is to adjust the original data so it becomes a perfect multiple of another predefined polynomial called the encoding polynomial. The process is summed up in an algorithm as follows:

i. Generate an encoding polynomial
ii. Shift original data polynomial to make space for the parity data (NB. The shift depends on the number of error correction bits required)
iii. Take the remainder from dividing the modified data polynomial by the encoding polynomial.
iv. Subtract the remainder from the modified data to generate a perfect multiple of the encoding polynomial.

The resulting modified data polynomial, called the Reed Solomon Codeword, can be tested later for corruption by checking if it leaves a remainder upon being divided by the encoding polynomial. If there is a remainder, the data is corrupt. An implementation of the algorithm is presented later.

https://sites.google.com/site/ijcsis/
ISSN 1947-5500

IV. IMPLEMENTATION

Both the encoding and decoding processes of the Reed-Solomon algorithm rely on a generator polynomial of the form

$$g(x) = \prod_{i=1}^{n}(x - \alpha^i)$$

where n is the number of parity data being added.

The task of generating the encoding polynomial can be quickly and efficiently carried out using a loop that continually multiplies the next monomial in the $g(x)$ sequence by the previously obtained polynomial. Two functions are required to generate the encoding polynomial.

1. Polynomial Multiplication Function: this function implements polynomial multiplication in code.
2. Encoding Polynomial Function: this function makes use of polynomial multiplication to generate the encoding polynomial using iteration.

Polynomial Multiplication

Polynomial multiplication is based on the distributive property of multiplication over addition, and the multiplicative law of indices. An example is shown below, showing the steps in finding the product of $5x^2 + 8x + 3$ and $3x + 8$.

$$(5x^2 + 8x + 3) \times (3x + 8)$$

Distribute the elements of the first polynomial over the second one

$$= 5x^2(3x + 8) + 8x(3x + 8) + 3(3x + 8)$$

Multiply each element of the first polynomial by each element of the second

$$= (5x^2 \times 3x) + (5x^2 \times 8) + (8x \times 3x) + (8x \times 8) + (3 \times 3x) + (3 \times 8)$$

$$= 15x^3 + 8x^2 + 24x^2 + 64x + 9x + 24$$

Sum the terms of the generated polynomial which have the same degree of x

$$= 15x^3 + 32x^2 + 73x + 24$$

The following pseudocode describes a function to multiply two polynomials. The coefficients of the polynomials are stored in an array with the array index describing the degree of that term in the polynomial.

```
:: function polynomialMultiplication(a[ ], b[ ])
::     for i = 0 to length of a
::         for j = 0 to length of b
::             product[i + j]
                = product[i + j] + (a[i]
                × b[j])
::         end for
::     end for
:: end function
```

Encoding Polynomial Function

The encoding polynomial is the product of the polynomials of the form $(x - 2^i)$ for i values from 1 to the number of parity data. The pseudocode below describes a function to generate the encoding polynomial for n number of parity. The encoding polynomial function simply loops n times, generating a new monomial and multiplying it by the previously obtained encoding polynomial. The encoding polynomial is initialized to $\{1\}$ [9].

```
:: function generateEncodingPolynomial( n)
::     polynomial[ ] = {1}
::     for i = 1 to n
::         polynomial = polynomialMultip_
               _lication(polynomial, {2^i, 1})
```

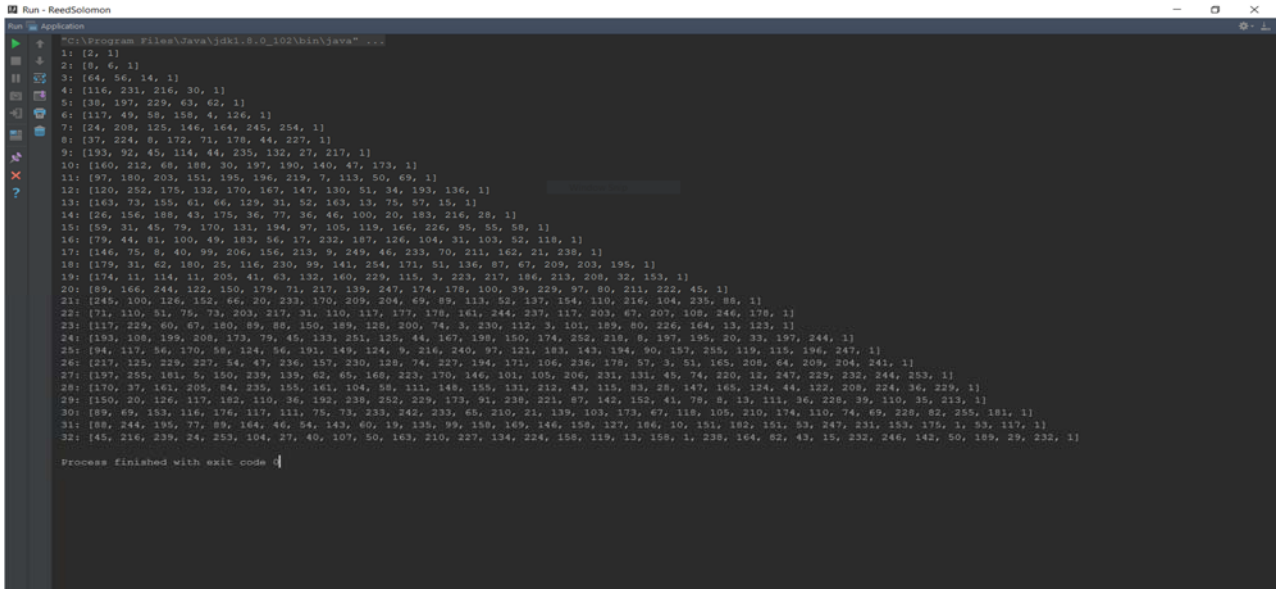

https://sites.google.com/site/ijcsis/
ISSN 1947-5500

```
::      end for
::      return polynomial
:: end function
```

Java Implementation

This algorithm was applied to $GF(256, 285)$ with $n = 32$.

The resulting coefficient values obtained from the Java code implementation execution are as follows:



MANUAL CHECK OF THE ALGORITHM RESULTS

To verify that the results obtained from the Java code implementation execution was right, the first and last coefficient values were computed by hand as follows

Exponents 1 to 32

= {2, 4, 8, 16, 32, 64, 128, 29, 58, 116, 232, 205, 135, 19, 38, 76, 152, 45, 90, 180, 117, 234, 201, 143, 3, 6, 12, 24, 48, 96, 192, 157}

First Coefficient and Last Coefficient

The first coefficient is the sum of all the elements in the exponents array, without combining them with each other. The last coefficient is the product of all the elements in the exponents array, without any additions

Addition Table

old sum	element	new sum = (old sum XOR element)
	2	2
2	4	6
6	8	14
14	16	30

Multiplication Table

Old Product	Element	New Product = (Old Product x Element)
1	2	2
2	4	8
8	8	64
64	16	116
116	32	38
38	64	117

https://sites.google.com/site/ijcsis/
ISSN 1947-5500

30	32	62
62	64	126
126	128	254
254	29	227
227	58	217
217	116	173
173	232	69
69	205	136
136	135	15
15	19	28
28	38	58
58	76	118
118	152	238
238	45	195
195	90	153
153	180	45
45	117	88
88	234	178
178	201	123
123	143	244
244	3	247
247	6	241
241	12	253
253	24	229
229	48	213
213	96	181
181	192	117
117	157	232
117	128	24
24	29	37
37	58	193
193	116	160
160	232	97
97	205	120
120	135	163
163	19	26
26	38	59
59	76	79
79	152	146
146	45	179
179	90	174
174	180	89
89	117	245
245	234	71
71	201	117
117	143	193
193	3	94
94	6	217
217	12	197
197	24	170
170	48	150
150	96	89
89	192	88
88	157	45

Reed Solomon Codeword

One simple way to treat a message as a polynomial is to read the message into an array. The index of the array at which an element is found, is used as the power of x that the element multiplies in the polynomial. For instance, the array

$$M = \{2, 3, 4, 6, 5, 1\}$$

Could easily represent the polynomial

$$M(x) = x^5 + 5x^4 + 6x^3 + 4x^2 + 3x + 2$$

The pseudocode below describes a function for generating the Reed Solomon Codeword from a given message polynomial, using parity value of n.

```
:: function encode_polynomial(M[ ], n)
::      encoding_polynomial[ ]
= generate_encoding_polynomial(n)
::      M = polynomial_shift(M, n)
::      quotient, remainder
                = polynomialDivision(M,
                        encoding_polynomial)
::      codeword = polynomialAddition(M,
                        remainder)
::      return codeword
:: end function
```


https://sites.google.com/site/ijcsis/
ISSN 1947-5500

V. CONCLUSION

Identifying errors that occur in a data transmission and data storage systems is important to ensuring reliability, dependability, and as well as install trust of a data storage system. Reed Solomon (RS) Coding among others as use of checksum, Cyclic Redundancy Check (CRC), Hamming codes, and etc. have been used for achieving this purpose. RS coding enables data errors (alterations and deletions) to be detected and corrected. The presence of a remainder after performing a polynomial division following the RS coding algorithm presented indicates data corruption (ie. data alteration or deletion).

REFERENCES

[1] Wellenzohn, K. (2015). Erasure coding in distributed storage systems. Available at: http://www.inf.unibz.it/dis/teaching/SDB/reports/report_wellenzohn.pdf

[2] Haiman, M. (Date). Notes on Reed Solomon. Available at: https://math.berkeley.edu/~mhaiman/math55/reed-solomon.pdf

[3] cs.cmu.edu (1998). Reed-Solomon Codes. An introduction to Reed-Solomon codes: principles, architecture and implementation. Available at: https://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html

[4] Trench W. F., (2003). Introduction to Real Analysis. Library of Congress Cataloging-in-Publication Data. Available at: http://ramanujan.math.trinity.edu/wtrench/texts/TRENCH_REAL_ANALYSIS.PDF

[5] Wang, J. (2009). Computer Network Security Theory and Practice. Springer

[6] REDTITAN, (2011). Error detection and correction. Available at: http://www.pclviewer.com/rs2/galois.html

[7] Benvenuto, C. J. (2012). Galois Field in Cryptography. Available at: https://www.math.washington.edu/~morrow/336_12/papers/juan.pdf

[8] Hill, T. (2013). Reed Solomon Codes Explained. Available at: https://www.tony-hill.info/app/download/.../Reed+Solomon+Explained+V1-0.pdf

[9] Wikiversity, (2016). Reed-Solomon Codes for Coders. Available at: https://en.wikiversity.org/wiki/Reed%E2%80%93Solomon_codes_for_coders

https://sites.google.com/site/ijcsis/
ISSN 1947-5500