ON THE REED-SOLOMON CODES

Nguyen Thi Lan Huong[1], Luu Thi Hiep[2*], Le Le Hang[3], Nguyen Thi Nhung[4], Nguyen Ngo Cong Thanh[5]
[1]TNU - University of Economics and Business Admistration, [2]Thu Dau Mot university
[3]Economics - Technology Industries University
[4]TNU - University of Information and Communication Technology, [5]Deakin University, Australia

ARTICLE INFO

Received: 04/4/2022
Revised: 29/5/2022
Published: 30/5/2022

KEYWORDS

Reed-Solomon codes
Binary codes
Encoder
Decoder
Syndrome decoding
RiBM algorithm
Chien Search algorithm
Forney algorithm

ABSTRACT

The Reed-Solomon (RS) codes are among the most powerful methods to preserve data integrity from errors and erasures for storage or transmission purposes. This coding technique has been proven to be a high performance while maintaining a reasonable cost and productivity. Unlike some coding techniques that enforce data transmission as a sequence of binary numbers, Reed-Solomon encodes the message as non-binary symbols. This gives Reed-Solomon the advantage of handling bursts of errors or even erasure error. It plays a significant role in modern communication systems and many daily life applications. Some known applications of this coding technique are the fault-tolerant systems in CD disks, and the communication protocol in satellites and spaceships. In the paper, we give the basic properties and structures of the Reed-Solomon codes by discussing its mathematics models. The encoding process with the original approach and the modern BCH approaches. For the decoding process, we investigate a wide range of algorithms and techniques, such as Syndrome decoding, RiBM algorithm, Chien search and Forney algorithm. Finally, we present the result is a functional Reed-Solomon encoder and decoder implemented using the MATLAB platform and give examples of encoding and decoding with different messages.

MÃ REED-SOLOMON

Nguyễn Thị Lan Hương[1], Lưu Thị Hiệp[2*], Lê Lệ Hằng[3], Nguyễn Thị Nhung[4], Nguyễn Ngô Công Thành[5]
[1]Trường Đại học Kinh tế và Quản trị kinh doanh – ĐH Thái Nguyên
[2]Đại học Thủ Dầu Một, [3]Trường Đại học Kinh tế Kỹ thuật công nghiệp Hà Nội
[4]Trường Đại học Công nghệ thông tin và truyền thông – ĐH Thái Nguyên, [5]Trường Đại học Daekin, Úc

THÔNG TIN BÀI BÁO

Ngày nhận bài: 04/4/2022
Ngày hoàn thiện: 29/5/2022
Ngày đăng: 30/5/2022

TỪ KHÓA

Mã Reed-Solomon
Mã nhị phân
Bộ mã hóa
Bộ giải mã
Giải mã hội chứng
Thuật toán RiBM
Thuật toán Chien
Thuật toán Forney

TÓM TẮT

Mã Reed-Solomon (mã RS) là một trong những phương pháp mạnh mẽ nhất để bảo vệ tính toàn vẹn của dữ liệu khỏi các lỗi có thể xảy ra trong quá trình lưu trữ hoặc truyền tải. Kỹ thuật mã hóa này đã được chứng minh đạt được hiệu suất cao với chi phí hợp lý. Trong khi các kỹ thuật mã hóa khác truyền dữ liệu dưới dạng một chuỗi số nhị phân, mã Reed-Solomon mã hóa thông điệp dưới dạng một chuỗi ký hiệu. Điều này đem lại cho mã Reed-Solomon lợi thế trong việc xử lý lỗi hàng loạt hoặc thậm chí là lỗi xóa. Nó đóng vai trò quan trọng trong các hệ thống thông tin liên lạc hiện đại và nhiều ứng dụng khác trong cuộc sống. Một số ứng dụng có thể kể đến như là hệ thống chịu lỗi trong đĩa CD và giao thức truyền thông trong vệ tinh và tàu vũ trụ. Trong bài viết này, chúng tôi đưa ra các thuộc tính và cấu trúc cơ bản của mã Reed-Solomon bằng cách thảo luận về các mô hình toán học của nó. Quá trình mã hóa với cách tiếp cận ban đầu và cách tiếp cận BCH hiện đại. Đối với quá trình giải mã, chúng tôi nghiên cứu một loạt các thuật toán và kỹ thuật, chẳng hạn như giải mã hội chứng, thuật toán RiBM, Chien và Forney. Kết quả là một bộ mã hóa và giải mã Reed-Solomon sử dụng nền tảng MATLAB. Chúng tôi đưa ra các ví dụ về mã hóa và giải mã với các thông điệp khác nhau.

DOI: https://doi.org/10.34238/tnu-jst.5808

* Corresponding author. *Email: hieplt@tdmu.edu.vn*

1. Introduction

In the last few decades, communication has been a vital field in engineering, and it is getting ever more interesting and challenging [1]-[4]. There are two important goals to achieve in this field are reliability and efficiency. Depends on the context, one must compromise for the sake of the other, and in most cases, reliability is the priority. In digital communication, there are a wide range of concern over errors, which are mainly occur due to noise, electromagnetic, bandwidth limits, etc.

To provide a better reliability for data transmission or storage, one can use error correction codes. The ideal of error correction is to add redundance to transmitted data such that error can be detected and corrected. There are different techniques, such as Hamming code [4], Golay code [5], etc.

In 1960, I.S. Reed and G. Solomon introduced a family of error-correcting codes that are doubly blessed [6]. The Reed-Solomon code can also be seen as non-binary BCH (Bose-Chaudhuri-Hocquenghem) code and some BCH decoding algorithms can also work for the case of RS code. The codes and their generalizations are useful in practice, and the mathematics that lies behind them is interesting.

Compared to binary cyclic code in which the coefficients of codeword polynomial are all in modulo 2 (either 0 or 1) [7], Reed Solomon codes coefficients are nonbinary, and each symbol of RS codeword can be constructed from multiple bits. This also means that if multiple bits in a symbol are corrupted, it only counts as a single symbol error. An overview of RS encoding and decoding techniques, such as RiBM [8], Chien search and Forney Algorithm [9] are presented in this paper.

2. The construction of RS codes

The original approach of constructing the RS codes is quite straightforward [5]:

A message word of the form $\vec{M} = (M_0, M_1, \cdots, M_{k-1}, M_k)$ where the M_i-th position generates k information symbols taken from the finite field $GF(q)$. Then the polynomial $\mathrm{p}(x) = M_0 + M_1\mathrm{x} + \cdots + M_{k-2}x^{k-2} + M_{k-1}x^{k-1}$ can be constructed accordingly. The RS codeword $\vec{C}$ of message $\vec{M}$ is generated by evaluating $p(x)$ at each of the q element in $GF(q)$:

$\vec{C} = (C_0, C_1, \cdots, C_{q-1}) = [p(0), p(\alpha), p(\alpha^2), \cdots, p(a^{q-1})]$, with α is a primitive element in $GF(q)$

RS codes are denoted by their length n and dimension of k as (n, k) codes. The code length (or block length) n is equivalent to q, since each codewords has q positions. The message length k, also known as the "dimension' of the RS code as the RS codewords are generated from a vector space of dimension k.

The generator matrix is a $n \times k$:

$$\mathrm{G}^{\intercal} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & \alpha & \alpha^2 & \cdots & \alpha^{k-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{n-1} & \alpha^{2(n-1)} & \cdots & \alpha^{(k-1)(n-1)} \end{bmatrix}$$

As can be seen, the generator matrix of RS codes is a Vandermonde matrix, therefore the generated RS codeword are linear, which mean that the sum of any two message of length k is another message of length k.

2.1. RS codes described as BCH codes

The original method of constructing the RS code was eventually replaced with a more modern approach of cyclic code: generator-polynomial. Nowadays, this technique of RS encoding is widely used in studies and research on error correction and communication.

For a symbol size m, a codeword of cyclic RS code from $GF(q = 2^m)$ have the length of $q - 1$, which is one position less than that of the original construction idea.

For the RS code to correct up to t symbols of length $q - 1$ in the finite field $GF(q)$, the generator $G(x)$ is defined as the polynomial:

$$\mathrm{G}(x) = \prod_{j=0}^{2t-1}\left(x - \alpha^{(h+j)}\right) = \sum_{j=0}^{2t-1} G_j x^j$$

Where the integer h is the power of the first consecutive roots of generator $G(x)$, this constant h is usually 1.

To create a RS generator, we can write a function that takes the codeword length n, message length k, number of bits per symbol m, and the primitive polynomial. MATLAB's communication toolbox will be used to provide the Galois Field for RS code:

```
function rs = rsCodeConstruct(n, k, m, primPol)
rs = [];

%% Alpha power
alpha = gf(2, m, primitivePolynomial);
alphaPower = alpha.^(0:n);
alphaPower = uint16(alphaPower.x);

%% Find GF element in alpha power
indexGFE = zeros(1,n+1);
for i = 2:nMax+1
        indexGFE(i) = find(i-1 == alphaPower(1:n));
end

%% Generator Polynomial
        generator = genpoly(n, k, alpha);
        generator = uint16(generaotr.x);

        %% Create alpha array
        nn = 1:2*t;
        alphaSynd = alpha.^(offset+nn-1);
        alphaSyndNum = uint16(alphaSynd.x);

rs.alphaPower = alphaPower;
rs.indexGF = indexGFE;
    rs.generator = fliplr(generator);
    rs.n = n;
    rs.k = k;
    rs.m = m;
    rs.t = (n-k)/2;
    rs.primitivePolynomial = primpol;
rs.alphaSynd = alphaSyndNum;
end

function g = genpoly(k, n, alpha)
    g = 1;
    % Multiplication on galios field is a convolution
    for k = mod(1 : n-k, n)
```

```
            g = conv(g, [1 alpha .^ (k-1)]);
        end
    end
```

2.2. Properties

The RS codes is a $[n, k, n-k+1]$ linear block code, with length n, dimension k and minimum Hamming distance $d = n - k + 1$.

A valid code $C(x)$ of length $q-1$ and dimension k can correct up to $t = \lfloor (q-k+1)/2 \rfloor$ symbol errors.

RS codes are good with burst errors, since any bit errors in a symbol will only treated as one symbol error in terms of correction

For example, a valid RS code is (15, 9):

- The codeword is constructed over $GF(16)$,
- Each symbol is constructed from 4 bit: $m = 4$
- Each codeword contains 15 symbols: $n = 15$,
- There are 9 symbols of which are data: $k = 9$,
- There are 6 symbols of which are used for parity check: $n - k = 6$
- This code can correct up to 3 symbols: $t = (n-k)/2 = 3$
- Codeword generator polynomial:

$$G(x) = x^6 + \alpha^{10}x^5 + \alpha^{14}x^4 + \alpha^4 x^3 + \alpha^6 x^2 + \alpha^9 x + \alpha^6$$

3. Encoding of RS codes

Let the sequence of k message symbols in $GF(2^m)$ be $\vec{m} = (m_0, m_1, \cdots, m_{k-1})$. The message vector can be represented in polynomial form as:

$$\mathrm{M(x)} = M_0 + M_1\mathrm{x} + \cdots + \mathrm{M_{k-1}}x^{k-1}$$

To generate the nonsystematic code polynomial $C(x)$, we multiply the message $M(x)$ by the generator $G(x)$:

$C(x) = M(x)G(x)$,

Or for systematic encoding:

$$C(x) = M(x)x^{2t} + M(x)x^{2t} \; mod \; G(x)$$

For example, we encode the message vector $\vec{v} = (0\ 0\ 0\ 0\ 0\ 0\ 0\ \alpha^{11}\ 0)$ which represented as polynomial $M(x) = \alpha^{11}x$ with the (15,9) RS code above. The codeword is generated by multiply the message polynomial $M(x)$ with the (15,9) RS generator polynomial $C(x)$:

$$C(x) = M(x)G(x) = \alpha^{11}x^7 + \alpha^6 x^6 + \alpha^{11}x^5 + x^4 + \alpha^2 x^3 + \alpha^5 x^2 + \alpha^2 x$$

Which represent the codeword $\vec{C} = (0\ 0\ 0\ 0\ 0\ 0\ 0\ \alpha^{11}\ \alpha^6\ \alpha^{11}\ 1\ \alpha^2\ \alpha^5\ \alpha^2\ 0)$

The decoding process involves recovering the message polynomial $M(x)$ from the received code polynomial $R(x)$. We will discuss the decoder techniques and process in more details later.

The following functions encodes a message vector to a RS systematic codeword using the RS structure provided:

```
function symbols = rsEncoder(m, rs)
    GPL = length(rs.generator);
    z = uint16(zeros(1, GPL - 1));

    for i = 1:rs.k
        xor = bitxor(z(GPLength - 1), m(i));
        gfm = gfMul (xor, rs,generator(1:GPL - 1), rs);
        z = bitxor(gfm, [uint16(0) z(1:GPL - 2)]);
end
```

```
redundancy = fliplr(z(1:GPL - 1));
symbols = [m redundancy]
 end

Multiplying over Galois field:
function [prodVector] = gfMul(x1, x2, rs)
indexGF = rs.indexGF;
      alphaPower = rs.alphaPower;
      alphaIndex = mod((indexGF(x1+1)+indexGF(x2+1)-2), rs.nMax);
      lx1 = uint16( logical(x1) );
      lx2 = uint16( logical(x2) );
      apw = uint16( alphaPower(alphaIndex+1));
      prodVector = lx1.*lx2.*apw;
end
```

4. Decoding of RS codes

4.1. Complete Decoder

A complete decoder process can be described as steps:

1. Compute Hamming distances between the received code and each valid codeword of $RS(n,k)$ code.
2. Chose the code with least Hamming distance value.

This approach to decode is impractical, as there are q^k valid codewords for a $RS(n,k)$, which is an enormous number and would take much time to iterate through each one of them.

4.2. Syndrome calculation

A received polynomial $R(x)$ is a combination of the original codeword and errors: $R(x) = C(x) + E(x)$ where $\mathrm{E}(x) = \sum_{i=0}^{n-1} E_i x^i$ is the error polynomial such that E_i is the error value of the i-th position. Note that the decoder does not know $E(x)$, and its task is to find errors $E(x)$ from the input $R(x)$ then correct the data by subtracting $E(x)$ from $R(x)$.

For the number of errors less than the defined capacity t, the decoder received polynomial as input:

$$\mathrm{R}(x) = \mathrm{C}(x) + \mathrm{R}(x) = R_0 + R_1\mathrm{x} + \cdots + R_{n-1}x^{n-1}$$

The codeword polynomial $C(x)$ is divisible by generator $G(x)$, and $G(a^i) = 0$ for $i = 1,2,\cdots,d-1$. Since $\mathrm{C}(\alpha^i) = \mathrm{M}(\alpha^i)\mathrm{G}(\alpha^i) = 0$ for $i = 1,2,\cdots,d-1$,

$$\mathrm{R}(\alpha^i) = \mathrm{C}(\alpha^i) + \mathrm{E}(\alpha^i) = \mathrm{E}(\alpha^i) = \sum_{j=0}^{n-1} E_j \alpha^{ij}$$

The syndromes S_i of the received polynomial can be computed based on above $d-1$ equations, as following:

$S_i = \mathrm{R}(\alpha^i) = \sum_{j=0}^{n-1} R_j \alpha^{ij}$, for $i = 1,2,\cdots,d-1$

Syndrome polynomial for a codeword up to t errors:

$$S(x) = \sum_{i=1}^{2t} S_i\, x^i$$

If the syndromes are all zero, then the codeword is not corrupted and need no further correction. Otherwise, if there are nonzero syndromes, then the decoder needs to find the number of errors, their locations, and values.

```
function syndr = syndrome(received, rs)
n = rs.n;
        t = rs.t;
        syndr = uint16(zeros(1,2*t));

        for i = 1:n
            syndr = gfMul(syndr, rs.alphaSynd, rs);
            syndr = bitxor(syndr, repmat(received(i), 1, 2*t));
        end
end
```

4.3. RiBM algorithm

RS code can be seen as nonbinary BCH code. For binary codes which is a sub-class of BCH code, the decoding process involves finding the error position and adding 1 to the error to flip the position to the right value. However, for nonbinary codes (such as RS code), the error value is also needed besides its position.

Let the error locator polynomial $\Lambda(x)$ for v unknown number of errors:

$$\Lambda(x) = \sum_{i=0}^{v} \Lambda_i x^i$$

Let the error evaluator $\Omega(x)$ with known syndrome $S(x)$ and error locator $\Lambda(x)$:

$$\Omega(x) = [1 + S(x)]\Lambda(x) \bmod x^{2t+1}$$

Reformulated inversionless Berlekamp–Massey algorithm [7] is used to find error locator polynomial $\Lambda(x)$ and the error evaluator polynomial $\Omega(x)$, implemented as following:

```
function ribm = RiBM(syndrome, rs)
   t = rs.t;
   k = 0;
   zero = uint16(0);

   %% Initialization
   delta = uint16([zeros(1, 3*t), 1, 0]);
   theta = uint16([zeros(1, 3*t),1]);
   gamma = uint16(1);
   theta(1:2*t) = syndrome;
   delta(1:2*t) = syndrome;

   for ii = 1:2*t
      % RiBM.1:
      delta0 = delta;
      delta(1:3*t+1) = bitxor(gfMul(gamma, delta0(2:3*t+2), rs), gfMul(delta0(1), theta, rs));

      % RiBM.2:
      if((delta0(1) ~= zero) && (k >= 0))
         theta(1:3*t) = delta0(2:3*t+1);
         theta(3*t+1) = zero;
         gamma = delta0(1);
         k = -k-1;
      else
         k = k+1;
```

```
    end
end

omega = fliplr(delta(1:t)); % error evaluator polynominal
lambda = delta(t+1:2*t+1);  % error locator polynominal

index = find(lambda(2:t+1));
% Find lambda degree
if isempty(index)
    lambdaDegree = 0;
else
    lambdaDegree = index(length(index));
end

ribm.omega = omega;
ribm.delta = delta;
ribm.theta = theta;
ribm.gamma = gamma;
ribm.lambda = lambda;
ribm.lambdaDegree = lambdaDegree;
end
```

4.4. Chien Search and Forney algorithm

The roots of error locator polynomial $\Lambda(x)$ over finite field $GF(2^m)$ can be found with Chien search method [1]. The polynomial $\Lambda(x)$ is expressed as:

$$\Lambda(x) = \Lambda_0 + \Lambda_1 x + \cdots + \Lambda_t x^t$$

The roots α^i of error locator polynomial $\Lambda(x)$ must satisfy:

$$\Lambda(\alpha^i) = \Lambda_0 + \Lambda_1 \alpha^i + \cdots + \Lambda_t \alpha^{it} = 0$$

After obtaining error locator polynomial $\Lambda(x)$, the number of known errors can be identified as $v = deg\,\Lambda(x)$. Forney algorithm can calculate the error values e at known locations [8] from error evaluator polynomial $\Omega(x)$, error locators X_j and formal derivative polynomial Λ' of Λ:

$$E_{k_j} = -X_j \frac{\Omega(X_j^{-}1)}{\Lambda'(X_j^{-}1)}$$

$\Lambda' = \frac{\sum j=0 \Lambda_j x^j}{x}$; $\Lambda_j = 0$ for j even

Chien search for error position and Forney algorithm for error value are implemented as following:

```
function cf = ChienForney(ribm, rs)
    n = rs.n;
    t = rs.t;
    omega = ribm.omega;
    lambda = ribm.lambda;

    %% Look-up tables shortcuts
    alphaPowerLT = @(x) rs.alphaPower(1+uint16(x));
    % Calculate 1/x
    invertx = uint16(zeros(1,n+1));
    for i = 1:n
```

```
    index = find(i == rs.alphaPower(1:n));
    invertx(1+i) = rs.alphaPower(n+2-index);
end
invertxLT = @(x) invertx(1+uint16(x));

%% Chien search and Forney algorithm
forneyCells2t = gfMul(uint16(1), alphaPowerLT(0), rs);
errorValues = uint16(zeros(1, n));

%% Chien search cells preparation
forneyCells(1:t) = gfMul(omega(1:t), alphaPowerLT(0), rs);
chienCellsEven(1:floor(t/2+1)) = gfMul(lambda(t+3-2*(1:floor(t/2)+1)), alphaPowerLT(0), rs);
chienCellsOdd(1:ceil(t/2)) = gfMul(lambda(t+2-2*(1:ceil(t/2))), alphaPowerLT(0), rs);

alphaForney = alphaPowerLT(t-(1:t));
alphaChienEven = alphaPowerLT(t+2-2*(1:(floor(t/2)+1)));
alphaChienOdd = alphaPowerLT(t+1-2*(1:ceil(t/2)));

roots = 0;

for i = 1:n
    lambdaEven = uint16(0);
    lambdaOdd = uint16(0);
    omegaVal = uint16(0);

    chienCellsEven = gfMul(chienCellsEven, alphaChienEven, rs);
    chienCellsOdd = gfMul(chienCellsOdd, alphaChienOdd, rs);
    forneyCells2t = gfMul(alphaPowerLT(2*t), forneyCells2t, rs);
    forneyCells = gfMul(forneyCells, alphaForney, rs);

    for j=1:floor(t/2)+1
        lambdaEven = bitxor(lambdaEven, chienCellsEven(j));
    end

    for j=1:ceil(t/2)
        lambdaOdd = bitxor(lambdaOdd, chienCellsOdd(j));
    end

    lambdaFull = bitxor(lambdaEven, lambdaOdd);

    for j=1:t
        omegaVal = bitxor(omegaVal, forneyCells(j));
    end

    omegaVal2t = gfMul(omegaVal, forneyCells2t, rs);

    %% Find root of error locator polynomial
    if (lambdaFull == 0) && (lambdaOdd~=0)
        % Calculate error value
        errorValues(i) = gfMul(omegaVal2t, invertxLT(lambdaOdd), rs);
```

```
            roots = roots + 1;
        else
            errorValues(i) = uint16(0);
        end
    end

    cf.errorValues = errorValues;
    cf.roots = roots;

end
```

4.5. Decoding function implementation

The complete RS code decoder using implemented Syndrome calculation, RiBM algorithm, Chien search and Forney algorithm:

```
function decoder = rsDecoder(receivedCode, rsStructure)
    receivedCode = uint16(receivedCode);
    n = rs.n;
    t = rs.t;

    %% Syndrome Calculation
    syndr = syndrome(receivedCode, rs);

    %% RiBM algorithm
    ribm = RiBM(syndr, rs);

    %% Chien search and Forney Algorithm
    chienForney = ChienForney(ribm, rs);

    %% Error Correction
    corrected = receivedCode(1:(n-2*t));
    if ribm.lambdaDegree == chienForney.roots
        % Correct symbols
        corrected = bitxor(receivedCode, chienForney.errorValues);
    end

    decoder.received = receivedCode;
    decoder.syndrome = syndr;
    decoder.RiBM = ribm;
    decoder.chienForney = chienForney;
    decoder.corrected = corrected;
    decoder.msg = decoder.corrected(1:(n-2*t));

end
```

6. Evaluation with examples

RS(7,3)

A RS code of $m = 3$ bits symbol size, $k = 3$ symbols message size, codeword size of $n = 7$ symbols and capable of correcting $t = (n - k)/2 = 2$ symbol errors. The encoder takes m bits to form a symbol, and k symbols to form a message. The encoder then calculates $2t = 4$ added

symbols which are appended to the message, the result is a codeword corresponding to the message.

Primitive Polynomial:

$$p(x) = x^3 + x^2 + 1$$

primitivePoly = 13; % x^3 + x^2 +1 -> [1 1 0 1] -> 13 in decimal

The generator:

$$G(x) = x^4 + 2x^3 + 2x^2 + 7x + 6$$

rsConstruct = rsCodeConstruct(n, k, m, primitivePoly);

rsConstruct.generatorPolynomial

>>> [6 7 2 2 1]

The message polynomial:

$$M(x) = x^2 + 4x + 1$$

msg = [1 4 1]'

Codeword (systematic):

$$C(x) = M(x)x^{2t} + \left(M(x)x^{2t}\ mod\ G(x)\right) = x^6 + 4x^5 + x^4 + 2x^3 + 7x^2 + 0x + 1$$

encodedmsg = rsEncoder(message, rsStructure)

>>> [1 4 1 2 7 0 1]

RS(7,1)

An inefficient RS code that only have $k = 1$ message symbol and $2t = 6$ redundancy symbols, and able to correct up to $t = 3$ error positions in a codeword of length $n = 7$. This RS code is constructed over $GF(2^3)$ that require $m = 3$ bits to form a symbol.

Primitive Polynomial:	[1 0 1 1]
Generator Polynomial:	[2 4 5 7 3 6 1]
Message:	[3]
Codeword:	[3 7 5 4 2 1 6]

RS(15,9)

This codeword need to have $m = 4$ bits to represent one symbol, as this RS code is constructed over $GF(2^4)$ using the primitive polynomial $p = x^4 + x + 1$. One codeword of RS(15,9) code have the length of $n = 15$ symbols, $k = 9$ in which are used to store the actual message. This code can correct up to $t = (n - k)/2 = 3$ symbol errors, by using $2t = 6$ redundancy symbols.

Primitive Polynomial:	[1 1 0 0 1]
Generator polynomial:	[01 03 04 02 15 10 01]
Message:	[12 15 01 13 13 08 04 03 03]
Codeword:	[12 15 01 13 13 08 04 03 03 00 10 01 04 12 13]

7. Known applications and developments

It is true that RS codes are the most used digital error-correction code. That is because of the Compact Disc uses two RS codes for error correction and concealment. The special properties of theses codes allowed the sound to be regenerated by the player in high quality.

A pair of so called “cross-interleaved” RS codes are used in the CD systems as sequences of nonbinary codewords symbols and need to be translated into a string of bits to be used in a binary channel [10]. Supposed that a noise burst corrupts some of consecutive bit on the transmitting channel, it means that the error bits are contained within a few of nonbinary symbols. For each of those noise bursts, the encoder only needs to correct a few of symbol errors, compared to long error bit strings. RS codes also allows to efficiently regenerate bytes from erasure error. As implemented in the compact disc error control system: the cross-interleaved RS will use the first code declare byte erasure, and the second to correct them. Due to this characteristic, the sound

can be reproduced by the CD player even if there are damages to the surface of the disc, such as scratches, dust particles, fingerprints, cracks, etc.

However, the most significant use case of RS coding was one of its applications for deep-space exploration. In 1977, the Voyager II has implemented the RS code with 255 bytes length and 223 bytes dimension, which able to correct 16 bytes errors to communicate with Earth base in its exploration mission [11]. Previously, the very first implementation of RS codes was for satellite telecommunication systems and for military use in the 70s.

In 1960s, the structure which involving multilevel of coding was found, called concatenated codes, could exponentially reduce the probability of error at in information rate no larger than channel capacity. The RS code was used as the "outer" code due to the Viterbi bursty decoder output, and the "inner" convolutional code to ensure the maximum likelihood of achieving a good error probability [12]. In this way, the RS "outer" code with its powerful error correcting capability can correct the output of the convolutional decoder in the concatenated coding system.

8. Conclusions

A generalized structure of RS code was implemented in MATLAB, based on cyclic code fundamentals to store RS code parameters length n, symbol size m, message length k, correction capability t and calculate the Galois field, Generator polynomial. An encoder then takes this RS structure to generate a unique systematic codeword from a message by multiply it with the generator polynomial. After receiving a codeword, the encoder first attempts to correct the symbols and then removes the redundant symbols from the codeword to return the original message.

Some examples of known RS code applications and their impacts to the world are stated: CD discs, space exploration, concatenated codes, etc.

In this paper, we provide some results about Reed-Solomon codes. We also encode and decode such codes by using the MATLAB platform. We give examples of encoding and decoding different messages.

Acknowledgement

This research is funded by Thu Dau Mot University, Binh Duong Province, Vietnam under grant number DT.21.2-018.

REFERENCES

[1] S. Y. Korabelshchikova, B. F. Melnikov, S. V. Pivneva, and L. V. Zyablitseva, "Linear codes and some their applications," 2018, vol. 1096: IOP Publishing, 1 ed., p. 012174.

[2] R. W. McEliece and L. Swanson, "Reed-Solomon codes and the exploration of the solar system," in Reed-Solomon Codes and Their Applications (S. B. Wicker and V. K. Bhargava, eds.), pp. 25–40. Piscataway, NJ: IEEE Press, 1994.

[3] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147-160, 1950, doi: 10.1002/j.1538-7305.1950.tb00463.x.

[4] S. B. Wicker and V. K. Bhargava, *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[5] M. Greferath, "Golay Codes," Wiley Encyclopedia of Telecommunications, 2003. [Online]. Available: https://doi.org/10.1002/0471219282.eot371. [Accessed Oct. 15, 2021].

[6] S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300-304, 1960.

[7] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379-423, 1948, doi: 10.1002/j.1538-7305.1948.tb01338.x.

[8] D. V. Sarwate and N. R. Shanbhag, "High-speed architectures for Reed-Solomon decoders," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 5, pp. 641-655, 2001, doi: 10.1109/92.953498.

[9] R. E. Blahut, *Algebraic Codes for Data Transmission*. Cambridge: Cambridge University Press, 2003.

[10] B. W. Stephen and K. B. Vijay, "Reed-Solomon Codes and the Compact Disc," in *Reed-Solomon Codes and Their Applications*: IEEE, 1994, pp. 41-59.

[11] J. Uri. "45 years ago: Viking 1 Touches Down on Mars," NASA. [Online]. Available: https://www.nasa.gov/feature/45-years-ago-viking-1-touches-down-on-mars. [Accessed Oct. 15, 2021].

[12] Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260-269, 1967, doi: 10.1109/TIT.1967.1054010.