Encoding and Decoding of Reed-Solomon Codes : A MATLAB implementation

Surajkumar Harikumar (EE11B075), Manikandan S (EE11B125)

***Abstract*—In this paper, we show an implementation of a Reed-Solomon encoder and decoder, capable of correcting errors and erasures. The encoder and decoder are designed for any generic Galois Field, but specific parameters of code are specified. Systematic encoding, intuition for converters, Berlekamp-Massey algorithm , Chien's search for root-location, and Forney's algorithm for error magnitude computation are used here. The program gives stage-wise outputs as well as the final-decoded message, and is capable of erasure correction, and can diagnose decoder failure.**

***Index Terms*—Reed-Solomon codes, MATLAB, systematic encoding, Berlekamp-Massey, Erasure decoding**

I. Introduction

In 1960, Irving Reed and Gus Solomon described a new class of error-correcting codes that are now called Reed-Solomon (R-S) codes. These codes have great power and utility, and are today found in many applications from compact disc players to deep-space applications. Reed-Solomon codes are nonbinary cyclic codes with symbols made up of m-bit sequences, where m is any positive integer having a value greater than 2. R-S (n, k) codes on m-bit symbols exist for all n and k for which

$$0 < k < n < 2^m + 2$$

where k is the number of data symbols being encoded, and n is the total number of code symbols in the encoded block. For the most conventional R-S (n, k) code,

$$(n, k) = (2^m - 1, 2^m - 1 - 2t)$$

where t is the symbol-error correcting capability of the code, and $n - k = 2t$ is the number of parity symbols. An extended R-S code can be made up with $n = 2^m$ or $n = 2^m + 1$ but not further. Reed Solomon codes are also called Maximum Distance Separable(MDS) codes since they achieve the largest possible code minimum distance for any linear code with the same encoder input and output block lengths. Let $\delta = 2t + 1$ be the desired design distance.By BCH bound we know that

$$d_{min} \geqslant \delta$$

But $\delta = 2t + 1 = n - k + 1$.Therefore,

$$d_{min} \geqslant n - k + 1$$

We know that by Singleton bound,

$$d_{min} \leqslant n - k + 1$$

Therefore, $d_{min} = n - k + 1$ and hence the RS code is a MDS code.

II. Code specifications

We are required to design a narrow-sense $[n, k]_q$ Reed-Solomon code of length $n = 63$ and design distance $\delta = 15$. We operate over the $\mathbb{F}_{64}$ field. We know that Reed-Solomon codes are MDS (Maximum Distance Seperable) codes. So, given a distance $\delta = 15$, the RS code must satisfy $d = \delta = n - k + 1$. This tells us that $k = n - \delta + 1 = 49$.

So our Reed-Solomon code has parameters $[63, 49, 15]_{64}$. Also, $\mathbb{F}_{64}$ constructed using $X^6 + X^5 + 1$ as the primitive polynomial. So we define out primitive element α such that $\alpha^6 + \alpha^5 + 1 = 0$. The following snippet of code defines the parameters, the primitive polynomial, and the field using MATLAB's, gftuple function. Note that field contains the entire mapping of powers of α. We use this fact later to define converters from Exponent form to Galois Array form, and vice versa. The bulk of encoding is done in the exponent domain, and explicit shifts will be mentioned.

```
p = 2; m = 6;
n=63 ; del = 15; k=49;
% Defining the primitive polynomial and the
    field (regular form)
prim_poly = [1 0 0 0 0 1 1]; % X^6 + X^5 + 1
field = gftuple([-1:p^m-2]',prim_poly,p);
```

We know that the generator polynomial of an $[n, k, \delta]_q$ RS code is given by

$$g(X) = (X + \alpha)(X + \alpha^2) \dots (X + \alpha^{\delta - 1})$$

where α is a primitive element over the field. So, now we show the code to create this $g(X)$ for a given specification. MATLAB's exponent representation of elements maps $-Inf \to \alpha^{-\infty} = 0, 0 \to \alpha^0 = 1, 1 \to \alpha$, and in general $i \to \alpha^i$. The following code snippet makes the generator polynomial. We use MATLAB's gfconv function, which multiples 2 polynomials (specified in increasing power order) over a specified field. ($[1\ 2] \to \alpha + \alpha^2$. So, gfconv([2 3],[2 3],field) would give $\alpha^4 + \alpha^6 X^2 \to [4\ -Inf\ 6]$

```
gx = [1 0];
for i=2:del-1,
    gx = gfconv(gx,[i 0],field);
end
```

The resultant generator polynomial (after execution) is

$$\begin{aligned} g(X) &= [42\ \ 58\ \ 33\ \ 60\ \ 55\ \ 51\ \ 1\ \ 35\ \ 49\ \ 21\ \ 10\ \ 0\ \ 21\ \ 31\ \ 0] \\ &= \alpha^{42} + \alpha^{58} X + \alpha^{33} X^2 + \alpha^{60} X^3 + \alpha^{55} X^4 \\ &\quad + \alpha^{51} X^5 + \alpha^{1} X^6 + \alpha^{35} X^7 + \alpha^{49} X^8 + \alpha^{21} X^9 \\ &\quad + \alpha^{10} X^{10} + \alpha^{0} X^{11} + \alpha^{21} X^{12} + \alpha^{31} X^{13} + X^{14} \end{aligned}$$

III. Systematic Encoder

The systematic encoding scheme used for RS encoding is the same one as used for cyclic and BCH codes. We want section of the codeword to be exactly the message bits, and to use the generator polynomial to create a kind of parity check. We first describe the encoding scheme.

Say the message polynomial is $m(X) = m_0 + m_1 X + \ldots + m_{k-1}X^{k-1}$, a polynomial of degree $< k$. Also, our generator polynomial is $g(X)$. We want the highest k powers of our n-bit codeword $c(X)$ to contain the message bits. So, our highest k-terms are now $X^{n-k}m(X)$. The rest of the $n-k$ terms are given by $X^{n-k}m(X)\ mod\ g(X)$

$$r(X) = X^{n-k}m(X)\ mod\ g(X)$$
$$c(X) = X^{n-k}m(X) + r(X)$$

Since $r(X)$ is taken $modulo\ g(X)$, and $g(X)$ has is of order $n-k$, $r(X)$ is of the form $r(X) = r_0 + r_1 X + \ldots + r_{n-k-1}X^{n-k-1}$. $X^{n-k}m(X)$ occupies the powers from $X^{n-k} \rightarrow X^{n-1}$. So, there is no overlap, and the message polynomial is preserved. This ensures that the codeword is in systematic form.

To put this as code, we accepted input line-by-line and stored in the variable input (I/O not shown here). the input was assumed to be in increasing order of powers of $m(X)$.

We converted the received $99 \rightarrow -Inf$, shifted the message, multiplied with shifter_nk, divided by gx, and added the reminder to the shifted message to obtain the codeword. The program takes input from rx.txt, and writes the codeword outputs to codeword.txt

```
shifter_nk = [ repmat( -Inf,1,del-1 ) 0 ];
input(input==99)=-Inf;
message = input';
shifted_message = gfconv(message,shifter_nk,
    field);
[qx,rx]=gfdeconv(shifted_message,gx,field);
codeword=gfadd(shifted_message,rx,field);
codeword(codeword==-Inf)=99;
```

In Table I we show sample input message bits, and the corresponding systematically encoded codeword.

TABLE I
MESSAGES AND CORRESPONDING CODEWORDS

Message	Codeword
[99]	[99 99 99 99 99 99 99 99 99 99 99 99 99 99 99]
[99 0]	[10 9 0 34 16 1 58 54 8 20 45 52 38 34 99 0]
[0]	[42 58 33 60 55 51 1 35 49 21 10 0 21 31 0]
[2 3 5 0]	[7 18 49 99 23 40 34 99 24 26 19 35 18 41 2 3 5 0]
[1 2 3 11 12 9 0]	[52 49 20 21 16 42 19 43 57 24 59 1 3 1 1 2 3 11 12 9 0]

IV. Exponent- Galois Array Converters and Custom Functions

While implementing the code, we frequently had to switch between the Exponent notation and the Galois Field Array notation. Inputs and outputs were given in exponent form, but the bulk of decoder processing was done with polynomials, which required Galois field arrays. So, we wrote functions which convert Exponent arrays to Galois Field arrays, and vice versa.

To understand how these work, we must first understand how Galois field arrays are represented in MATLAB. A typical Galois field array specifies the generator polynomial used, along with a list of decimal numbers. The decimal number represents the coefficient of a term in the polynomial, with the first element being thee highest power, i.e the polynomial powers are ordered high-to-low (in keeping with MATLAB's standard). The decimal numbers, when converted to binary, tell us the sum of powers of primitive element α over the field which make up the coefficient. Binary numbers are read left-to-right in MATLAB. for instance

```
a=gf([4 5],4,19)

a = GF(2^4) array. Primitive polynomial = D^4+
    D+1 (19 decimal)

Array elements =

          4           5
```

We would interpret $4 \rightarrow 0010 \rightarrow \alpha^2$ and $5 \rightarrow 1010 \rightarrow 1+\alpha^2$. So the polynomial represented by a is $\alpha^2 X + (\alpha^2 + 1)$. We use a lookup table to see that $(\alpha^2+1) = \alpha^8$. So a can also be represented by $\alpha^8 + \alpha^2 X$, which would be $[8\ 2]$. The function we wrote enables this transition.

To understand the conversion, we first analyse the output of the gftuple seen in Table II. For convenience, we take $\mathbb{F}_{16}$ generated using $X^4 + X + 1$.

TABLE II
ESTABLISHING RELATION BETWEEN EXPONENT AND GALOIS FIELD ARRAYS USING GFTUPLE OVER $\mathbb{F}_{16}$ USING $p(X) = X^4 + X + 1$.

Exponent	Index	Gftuple values	Galois relation
$\alpha^{-\infty}$	1	[0 0 0 0]	0
α^0	2	[1 0 0 0]	1
α^1	3	[0 1 0 0]	α
α^2	4	[0 0 1 0]	α^2
α^3	5	[0 0 0 1]	α^3
α^4	6	[1 1 0 0]	$1+\alpha$
α^5	7	[0 1 1 0]	$\alpha+\alpha^2$
α^6	8	[0 0 1 1]	$\alpha^3+\alpha^3$
α^7	9	[1 1 0 1]	$1+\alpha+\alpha^3$
α^8	10	[1 0 1 0]	$1+\alpha^2$
α^9	11	[0 1 0 1]	$\alpha+\alpha^3$
α^{10}	12	[1 1 1 0]	$1+\alpha+\alpha^2$
α^{11}	13	[0 1 1 1]	$\alpha+\alpha^2+\alpha^3$
α^{12}	14	[1 1 1 1]	$1+\alpha+\alpha^2+\alpha^3$
α^{13}	15	[1 0 1 1]	$1+\alpha^2+\alpha^3$
α^{14}	16	[1 0 0 1]	$1+\alpha^3$

The middle columns are taken from gftuple, and the Exponent and Galois relation are extrapolated from what we know about MATLAB representation. We can indeed verify that this

is the relation between element representations over $\mathbb{F}_{16}$. We can use this to easily establish the conversion from Exponent to Galois field representation and vice versa. We clearly see an offset of 2 between the exponent power and the array index, and that the gftuple values correspond exactly to the 4-Tuple representation.

For the Exponent-to-GF, we take the input elements, convert $-Inf \rightarrow -1$, offset all elements by 2, find the value corresponding to the index, convert to decimal, and flip the array (as Exponent is low-high in order of polynomial, and GF array is high-low in order of polynomial). The function takes an exponent array as input, and returns a Galois Field array as output.

```
function result = myexp2gf(in,m,prim)
        a=gftuple([-1:2^m-2]',de2bi(prim),2);
        in(in==-Inf)=-1;
        b = a(in+2,:);
        result = bi2de(b)';
        result=fliplr(result);
        result = gf(result,m,prim);
end
```

For the GF-to-exponent, we essentially do the reverse. We convert the decimal number to binary, and find the index of the gftuple element which gives the current binary number, and subtract 2 to get the exponent array. The function takes a Galois field array as input, and returns an exponent array as output.

```
function result = mygf2exp(in,m,prim)
        a=gftuple([-1:2^m-2]',de2bi(prim),2);
        in=in.x;
        for i =1:size(in,2)
            b=de2bi(in(i));
            b = [ b zeros(1,log2(size(a,1))-
                size(b,2)) ];
            result(i)=find(ismember(a,b,'rows'
                ))-2;
        end
        result(result==-1)=-Inf;
        result=fliplr(result);
end
```

We keep the functions generic by passing the order 2^m and the primitive polynomial as parameters. These functions help us easily convert between the 2 representations, and give us insight into how Galois Field element representations can be generalized.

V. Decoder

A. Computation of the syndrome

In this section we briefly describe the various steps involved in decoding the RS codes. Like in any other linear block code, the first step involved in decoding RS codes is the computation of Syndrome. If $g(x)$ is the generator polynomial of the RS code, we know that

$$g(\alpha) = g(\alpha^2) = ... = g(\alpha^{2t}) = 0$$

It follows that $\mathbf{c} = (c_0, ..., c_{n-1})$with polynomial $c(x) = c_0 + ... + c_{n-1}x^{n-1}$ has

$$c(\alpha) = c(\alpha^2) = ... = c(\alpha^{2t}) = 0$$

For a received polynomial $r(x) = c(x) + e(x)$ we have

$$S_j = r(\alpha^j) = e(\alpha^j) = \sum_{k=0}^{n-1} e_k \alpha^{jk}, j = 1, 2, ...2t$$

The values $S_1, S_2, ..., S_{2t}$ are called the syndromes of the received data.

```
% Converting 99 to polynomial 0 === Exponent
    -Inf
input(input==99)=-Inf;input=input';
erasure_locations = find(input==100)-1;
% Converting 100 to polynomial 0 ===
    Exponent -Inf
input(input==100)=-Inf;input=input';
%myexp2gf reverses the array.
gfinput=myexp2gf(input,m,prim);
gfinput=[zeros(1,n-size(gfinput,2))
    gfinput];
syn1=polyval(gfinput,alphalist');
S1=syn1';
S=gf(S1,m,prim);
```

Now the vector S has the syndrome associated with received vector.

B. Finding the erasure locating polynomial

Let the code have erasures at $j_1, j_2, ..., j_f$.We introduce erasure locators such that

$$Y_1 = \alpha^{j_1} \quad Y_2 = \alpha^{j_2} \quad ... \quad Y_f = \alpha^{j_f}$$

The decoder must find the values of erasures at these locations f_{j_k}.

We begin by creating an erasure locator polynomial

$$\mathrm{T}(x) = \prod_{l=1}^{f}(1 - Y_l x)$$

which is known since the erasures locations are known.

```
    %Finding the erasure locating
        polynomial
erasure_polynomial=gf([0 1],m,prim);
[row n_erasures]=size(erasure_locations);
%n_erasures has the total no of erasures
for i=1:n_erasures,
    ele=myexp2gf(erasure_locations(i),m,
        prim);
    erasure_polynomial=conv(
        erasure_polynomial,[ele 1]);
end
%erasure_polynomial has the erasure
    locator polynomial.
```

C. The Berlekamp-Massey algoritm

At first we create the Syndrome polynomial $S(x)$ as follows

$$S(x) = \sum_{k=0}^{2t-1} S_{k+1}x^k$$

We define the polynomial $E(x) = \mathrm{T}(x)S(x) \quad (mod x^{2t})$ We try to find the so called 'error locator polynomial' $\Lambda(x)$ through a iterative approach. The algorithm used here is Berlekamp-Massey algorithm.The code for implementing the algorithm is given below.

```
x_2t_poly=gf(zeros(1,del),m,prim);
    x_2t_poly(1)=1;
    S_polynomial=fliplr(S);
    [quot E_poly]=deconv(conv(
        erasure_polynomial,S_polynomial),
        x_2t_poly);
    E=fliplr(E_poly);
    %Initializing condition for Berlekamp-
        Messsay algorithm
    L=0; %current LFSR length
    c_x=gf(zeros(1,1+(del-1)/2),m,prim);
    p_x=gf(zeros(1,1+(del-1)/2),m,prim);
    c_x(1)=1;
    p_x(1)=1;
    l=1;
    d_m=gf(1,m,prim);
    %What all should be displayed in the table
         for each iteration
    %k, S(k) d c_x L p_x l d_m
    %c_x and p_x should be in low to high and
        they are already in low to
    %high

    fprintf(fnow,'\nBerlekamp Massey :\n');
    for k=1:del-1,
        temp=gf(0,m,prim);
        for i=1:L,
            temp=temp+c_x(i+1)*E(k-i);
        end
        d=E(k)+temp;
        if d==0
            l=l+1;

        else
            if (2*L >= k)
                x_x=gf(zeros(1,l+1),m,prim);
                x_x(l+1)=1;
                addit=d*(1/d_m)*conv(x_x,p_x);
                c_x=c_x-addit(1:1+(del-1)/2);
                l=l+1;

            else
                t_x=c_x;
                x_x=gf(zeros(1,l+1),m,prim);
                x_x(l+1)=1;
                addit=d*(1/d_m)*conv(x_x,p_x);
                c_x=c_x-addit(1:1+(del-1)/2);
                L=k-L;
                p_x=t_x;
                d_m=d;
                l=1;
            end
        end
    sc = S(k);
    c_2 = mygf2exp(fliplr(c_x),m,prim); c_2(
        c_2==-Inf)=99;
    p_2 = mygf2exp(fliplr(c_x),m,prim); p_2(
        p_2==-Inf)=99;
    fprintf(fnow,'%d & %d & %d & [',k,sc.x,d.x
        );
    fprintf(fnow,[ repmat('%d ',1,size(c_2,2))
         ],c_2);
    fprintf(fnow,'] & %d & [',L);
    fprintf(fnow,[ repmat('%d ',1,size(p_2,2))
         ],p_2);
    fprintf(fnow,'] & %d & %d',l,d_m.x);
    fprintf(fnow,'\n');
    end
    %Now c_x has the error locator polynomial
```

D. Roots of error locator polynomial- Chien search algorithm

In abstract algebra, the Chien search, named after R. T. Chien, is a fast algorithm for determining roots of polynomials defined over a finite field. The most typical use of the Chien search is in finding the roots of error-locator polynomials encountered in decoding Reed-Solomon codes and BCH codes.

Let $\Lambda(x) = \lambda_0 + \lambda_1 x + ...\lambda_k x^k$. Conceptually we may evaluate $\Lambda(\beta)$ for each non-zero β in $\mathrm{GF}(q)$.Those resulting in 0 are roots of the polynomial. The Chien search is based on two observations:

- Each non-zero β may be expressed as α^{i_β} for some i_β where α is a primitive element of $\mathrm{GF}(q)$.i_β is the power number of primitive element α.Thus the powers α^i for $0 \leq i < (q-1)$ cover the entire field(excluding the zero element).
- The following relationship exists:

$$\begin{aligned}\Lambda(\alpha^i) &= \lambda_0 + \lambda_1(\alpha^i) + \lambda_2(\alpha^i)^2 + \cdots + \lambda_t(\alpha^i)^t \\ &\triangleq \gamma_{0,i} + \gamma_{1,i} + \gamma_{2,i} + \cdots + \gamma_{t,i}\end{aligned}$$

$$\begin{aligned}\Lambda(\alpha^{i+1}) &= \lambda_0 + \lambda_1(\alpha^{i+1}) + \lambda_2(\alpha^{i+1})^2 + \cdots + \lambda_t(\alpha^{i+1})^t \\ &= \lambda_0 + \lambda_1(\alpha^i)\,\alpha + \lambda_2(\alpha^i)^2\,\alpha^2 + \cdots + \lambda_t(\alpha^i)^t\,\alpha^t \\ &= \gamma_{0,i} + \gamma_{1,i}\,\alpha + \gamma_{2,i}\,\alpha^2 + \cdots + \gamma_{t,i}\,\alpha^t \\ &\triangleq \gamma_{0,i+1} + \gamma_{1,i+1} + \gamma_{2,i+1} + \cdots + \gamma_{t,i+1}\end{aligned}$$

In other words, we may define each $\Lambda(\alpha^i)$ as the sum of a set of terms $\{\gamma_{j,i} | 0 \leq j \leq t\}$ from which the next set of coefficients may be derived thus: $\gamma_{j,i+1} = \gamma_{j,i}\,\alpha^j$ In this way, we may start at $i = 0$ with $\gamma_{j,0} = \lambda_j$ and iterate through each value of i up to $(q-1)$. If at any stage the resultant summation is zero, i.e.

$$\sum_{j=0}^{t} \gamma_{j,i} = 0$$

then $\Lambda(\alpha^i) = 0$ also, so α_i is a root. In this way, we check every element in the field. When implemented in hardware, this approach significantly reduces the complexity, as all multiplications consist of one variable and one constant, rather than two variables as in the brute-force approach. The code that describes this algorithm is given below.

```
%Chein Search Algorithm
    %Initializing
    error_polynomial=fliplr(c_x);
    fprintf(fnow,'\n Error Locator : ');
    el1 = mygf2exp(error_polynomial,m,prim);
        el1(el1==-Inf)=99;
    dlmwrite('decoderSteps.txt',el1,'delimiter
        ',' ','-append');
    %disp(error_polynomial);
    root_exp=[];
    root_f=[];
    %root_exp1=[];
    %root_f1=[];
    old_c=c_x;
    new_c=c_x;
    prod1=ones(1,1+(del-1)/2);
    prod_exp=[];
    prod_f=[];
    for i=0:(del-1)/2,
        f=myexp2gf(i,m,prim);
```

```
        prod_exp=[prod_exp i];
        prod_f=[prod_f f];
    end
    for i=0:n-1,
        f=myexp2gf(i,m,prim);
        value=0;
        for k=1:1+(del-1)/2,
            value=value+old_c(k);
            new_c(k)=old_c(k)*prod_f(k);
        end
        old_c=new_c;
        if(value==0)
            root_exp=[root_exp i];
            root_f=[root_f f];
        end

    end
```

VI. Finding values of errors and erasures- Forney's method

The next obvious step in decoding after determining the location of errors (and knowing the locations of the erasures) is to find the value of errors and erasures. They are found by Forney's method.We first find a polynomial $\Omega(x)$ such that

$$\Lambda(x)E(x) = \Omega(x) \quad (mod x^{2t})$$

The polynomial

$$\Phi(x) = \Lambda(x)\mathrm{T}(x)$$

called the *combined error/erasure locator polynomial* is then computed. Then if $X_1^{-1}, X_2^{-1}, \cdots, X_v^{-1}$ are roots of the error-locator polynomial and $Y_1^{-1}, Y_2^{-1}, \cdots, Y_v^{-1}$ are roots of erasure-locator polynomial, then

$$e_{ik} = -\frac{\Omega(X_k^{-1})}{\Phi'(X_k^{-1})} \quad and \quad f_{ik} = -\frac{\Omega(Y_k^{-1})}{\Phi'(Y_k^{-1})}$$

The error polynomial is therefore

$$e(x) = \sum_{k=1}^{v} e_{ik}x^{j_k} + \sum_{k=1}^{f} f_{ik}x^{l_k}$$

where $j_1, j_2, \cdots, j_v$ are error locations and $l_1, l_2, \cdots, l_f$ are erasure locations. The decoded codeword $c'(x)$ when r(x) is the received polynomial is therefore

$$c'(x) = r(x) - e(x)$$

```
%Forney's algorithm
    [quot error_eval_polynomial]=deconv(conv(
        E_poly,error_polynomial),x_2t_poly);
    phi_x=fliplr(conv(error_polynomial,
        erasure_polynomial));
    phi_prime_x=gf(zeros(1,length(phi_x)-1),m,
        prim);
    for i=2:length(phi_x),
        for j=1:i-1,
            phi_prime_x(i-1)=phi_prime_x(i-1)+
                phi_x(i);
        end
    end
    phi_polynomial_deri=fliplr(phi_prime_x);
    ei=-polyval(error_eval_polynomial,root_f)
        ./polyval(phi_polynomial_deri,root_f);
    error=gf(zeros(1,n),m,prim);
    fprintf(fnow,'\nError Evaluator : ');
    eval1 = mygf2exp(error_eval_polynomial,m,
        prim); eval1(eval1==-Inf)=99;
    dlmwrite('decoderSteps.txt',eval1,'
        delimiter',' ','-append');
    erasure=gf(zeros(1,n),m,prim);
    inverse_root_f_dec=gf2dec(gf(
        inverse_root_exp,m,prim),m,prim);
    error(inverse_root_f_dec+1)=ei;
    if(n_erasures>0)
    erasure_root_f=fliplr(power(myexp2gf(
        erasure_locations,m,prim),-1));
    fi=-polyval(error_eval_polynomial,
        erasure_root_f)./polyval(
        phi_polynomial_deri,erasure_root_f);
    erasure(erasure_locations+1)=fi;
    end
    decoded_r_f=gfinput-fliplr(error)-fliplr(
        erasure);
    decoded_r_exp=mygf2exp(decoded_r_f,m,prim)
    end
```

NOTE: This implementation of decoding of RS code takes care of decoder failure too. Decoder failure mainly arises if the total number of errors and erasures is greater than $t(t = \frac{\delta-1}{2})$. When such a decoding failure occurs, the decoded vector is made -1 and appropriate decoding failure message is printed.

VII. Sample Decoder - Input and Output

In this section, we demonstrate how the code performs for one sample input. All vectors outputs shown here are in exponent form. Take for instance the received vector, and its exponential representation given below

$$\begin{aligned} r(X) &= [0\ \ 99\ \ 0\ \ 99\ \ 5\ \ 99\ \ 7\ \ 99\ \ 4\ \ 99\ \ 2] \\ &= 1 + X^2 + \alpha^5 X^4 + \alpha^7 X^6 + \alpha^4 X^8 + \alpha^2 X^{10} \end{aligned}$$

We look to the decoderSteps.txt file for the stage-wise output of the program. The corresponding syndrome polynomial is

$$\begin{aligned} s(X) &= [0\ \ 99\ \ 0\ \ 99\ \ 5\ \ 99\ \ 7\ \ 99\ \ 4\ \ 99\ \ 2] \\ &= 1 + X^2 + \alpha^5 X^4 + \alpha^7 X^6 + \alpha^4 X^8 + \alpha^2 X^{10} \end{aligned}$$

The Berlekamp Massey output is given in Table III. The format of the table is as in [1]

The Error Locator polynomial given at the end of the Berlekamp-Massey algorithm is

$$\begin{aligned} \sigma^{\delta-1}(X) &= [0\ \ 20\ \ 54\ \ 13\ \ 1\ \ 40\ \ 30] \\ &= 1 + \alpha^{20}X + \alpha^{54}X^2 + \alpha^{13}X^3 \\ &\quad + \alpha^{1}X^4 + \alpha^{40}X^5 + \alpha^{30}X^6 \end{aligned}$$

The Error-Evaluator polynomial after Chien's search and Forney's algorithm is

$$\begin{aligned} Z_0(X) &= [47\ \ 17\ \ 12\ \ 20\ \ 39\ \ 56] \\ &= 47 + \alpha^{17}X + \alpha^{12}X^2 + \alpha^{20}X^3 \\ &\quad + \alpha^{39}X^4 + \alpha^{56}X^5 \end{aligned}$$

TABLE III
BERLEKAMP-MASSEY DECODING ALGORITHM RUN OVER
$r(X) = 1 + X^2 + \alpha^5 X^4 + \alpha^7 X^6 + \alpha^4 X^8 + \alpha^2 X^{10}$

k	S_k	d_k	$c(X)$	L	$p(X)$	l	d_m
1	55	55	[0 47]	1	[0 47]	1	55
2	37	40	[0 61]	1	[0 61]	2	55
3	9	24	[0 61 14]	2	[0 61 14]	1	24
4	14	18	[0 57 56]	2	[0 57 56]	2	24
5	60	27	[0 57 45 29]	3	[0 57 45 29]	1	27
6	55	7	[0 40 17 22]	3	[0 40 17 22]	2	27
7	50	63	[0 40 26 5 37]	4	[0 40 26 5 37]	1	63
8	32	42	[0 9 50 59 55]	4	[0 9 50 59 55]	2	63
9	8	28	[0 9 57 6 29 53]	5	[0 9 57 6 29 53]	1	28
10	38	56	[0 24 40 42 26 24]	5	[0 24 40 42 26 24]	2	28
11	54	21	[0 24 8 22 14 51 29]	6	[0 24 8 22 14 51 29]	1	21
12	44	21	[0 20 54 13 1 40 30]	6	[0 20 54 13 1 40 30]	2	21
13	48	0	[0 20 54 13 1 40 30]	6	[0 20 54 13 1 40 30]	3	21
14	54	0	[0 20 54 13 1 40 30]	6	[0 20 54 13 1 40 30]	4	21

We use this to find the error magnitudes. Since we also know the error locations, we can frame the error polynomial. This is found to be

$$\begin{aligned} e(X) &= [0 \;\; 99 \;\; 0 \;\; 99 \;\; 5 \;\; 99 \;\; 7 \;\; 99 \;\; 4 \;\; 99 \;\; 2] \\ &= 1 + X^2 + \alpha^5 X^4 + \alpha^7 X^6 + \alpha^4 X^8 + \alpha^2 X^{10} \end{aligned}$$

and the decoded codeword is the all zero codeword, that is

$$\begin{aligned} \hat{c}(X) &= [99 \;\; 99 \;\; 99 \;\; 99 \;\; 99 \;\; 99 \;\; 99 \;\; 99 \;\; 99 \;\; 99] \\ &= 0 + 0X + 0X^2 + \ldots + 0X^{62} \end{aligned}$$

The input received vector had $t = 6$ non-zero positions. Since $\delta = 15$, and RS codes are MDS, 2 codewords are separated by a distance of $d = 15$. So, $t = 6$ or fewer errors will be decoded as the all-zero codeword as this falls within the Hamming sphere of the all-zero codeword.

REFERENCES

[1] Todd. K. Moon, *Error Correction Coding*.
[2] Shu Lin, Daniel J. Costello, *Error Control Coding-Fundamentals and Applications, 2ed*
[3] http://en.wikipedia.org/