Implementation of a Reed-Solomon Encoder and Decoder in MATLAB

Jayanth Ramesh, EE11B121  S.Sundara Rajan, EE11B130

Abstract—This project implements a narrow-sense Reed-Solomon Encoder and Decoder of length $n = 63$, and designed distance $\delta = 15$ in the field $GF(64)$ constructed using the primitive polynomial $x^6 + x^5 + 1$. The encoder accepts one or more inputs from the file "msg.txt" and outputs to a file named "codeword.txt". The inputs are given in exponential form. The input to the decoder is from "rx.txt" and the output is written to "decoderOut.txt". The decoder takes in inputs containing errors and erasures in exponential format. It displays all the syndromes, then performs Berlekamp-Massey algorithm to find the error-locator, does a Chien search to find its roots, and finally finds the error-vector using Forney’s algorithms. It prints out the stage-wise output of Berlekamp-Massey, the error locator and error-value evaluator polynomials, the decoded codeword and the message on separate lines. It also displays if the decoder fails to correct errors, displaying the reason for the same.

I. ENCODING

Reed-Solomon codes can be encoded like any other cyclic code. Given a message vector $m = (m_0, m_1, ..., m_{k-1})$, we can form the corresponding message polynomial $m(x) = m_0 + m_1x + m_2x^2 + ... m_{k-1}x^{k-1}$, where all $m_i \in GF(n)$. The systematic encoding process is

$$c(x) = m(x)x^{n-k} - R_g(x)[m(x)x^{n-k}]$$  (1)

where the $R_g(x)$ denotes the operation of taking remainder when divided by $g(x)$.

The inputs are given in the exponential form, which is converted to the polynomial form using a user-defined MATLAB function on Galois Field.

Multiplication of Polynomials in the Galois Field is realized by convolving the two sequences(coefficients of the polynomials). The generator polynomial $g(x)$ is obtained by repeated multiplication of polynomials. Also $m(x)x^{n-k}$ is obtained by the same technique.

Sample code:

```matlab
>> a=gf([1,2,3],3);
>> b=gf([0,2],3);
>> c=conv(a,b);
c = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)
Array elements =
0 2 4
The remainder operation is implemented by deconvolving and taking the second term of the output. The following code will give the reminder of $\frac{x^2+2x+3}{x+2}$
```

Sample code:

```matlab
>> a=[1 2 3];
>> b=[1 2];
>> [q,r]=deconv(a,b);
>msg=[1:49]; > r
r =
0 0 3
```

And finally the codeword is converted to exponential form using another user defined function.

Sample input:

```matlab
msg=[1:49];
```

Sample output:

Note: "result" is the output of our encoder, while "result2" is the output of MATLAB in-built RS encoder. We see that both the outputs are same except that both the message and parity bits are flipped in both.

```
result =
Columns 1 through 13
50 51 52 53 54 55 56
57 58 59 60 61 62
Columns 14 through 26
0 1 2 3 4 5 6
7 8 9 10 11 12
Columns 27 through 39
13 14 15 16 17 18 19
20 21 22 23 24 25
Columns 40 through 52
26 27 28 29 30 31 32
33 34 35 36 37 38
Columns 53 through 63
39 40 41 42 43 44 45
46 47 48 49
result2 =
Columns 1 through 13
0 62 61 60 59 58 57
56 55 54 53 52 51
Columns 14 through 26
50 49 48 47 46 45 44
43 42 41 40 39 38
Columns 27 through 39
37 36 35 34 33 32 31
30 29 28 27 26 25
Columns 40 through 52
24 23 22 21 20 19 18
```
The LFSR relationship between the syndromes and the error-locator polynomial is:

\[ S_j = - \sum_{i=1}^{v} \Lambda_i S_{j-i}, \quad j = v + 1, \ldots, 2t \]  

We want to determine the polynomial with the least degree, i.e. we want to determine the shortest LFSR satisfying the given set of equations.

We calculate the connection polynomial \( \Lambda(x) \) formed at the \( k \)-th iteration and determine the discrepancy \( d_k \) by subtracting the syndromes at the \( k \)-th and \( (k-1) \)-th iteration. If this is zero, then \( \Lambda[1](x) = \Lambda[k-1](x) \), and the length of the LFSR remains the same. If there is some non-zero discrepancy, then the connection polynomial is updated using

\[ \Lambda[k](x) = \Lambda[k-1](x) + Ax^l \Lambda[m-1](x) \]

where \( l = k - m \) and \( A = d_{m-1} d_k \).

Two important results regarding the LFSR have been proved in the book by Todd.K.Moon which we state here:

- If the LFSR can produce \( \{S_1, S_2, \ldots, S_{k-1}\} \) but not \( \{S_1, S_2, \ldots, S_k\} \), then length of the LFSR that produces the latter sequence satisfies \( L_k \geq k - L_{k-1} \).
- In the update procedure, if \( \Lambda[k](x) \neq \Lambda[k-1](x) \), then a new LFSR can be found whose length is \( L_k = \text{max}(L_{k-1}, k - L_{k-1}) \). Therefore, if \( 2L_{k-1} \geq k \), then the connection polynomial is updated, but there is no change in length.

Also a polynomial \( p(x) \) is maintained to indicate the “previous connection polynomial”. While updating the connection polynomial, this \( p(x) \) is used and is updated whenever \( 2L \neq k \).

The crucial last step has been accomplished by the given code snippet:

```matlab
if 2*L>=k
    x_=zeros(1,2*t);
    x_(l+1)=1;
    prod=conv(x_,p);
    c=c-d*(dmˆ(-1))*prod(1:2*t);
    l=l+1;
else
    tp=c;
    x_=zeros(1,2*t);
    x_(l+1)=1;
    prod=conv(x_,p);
    c=c-d*(dmˆ(-1))*prod(1:2*t);
    l=k-L;
    p=tp;
    dm=d;
    l=1;
```

**D. The Berlekamp-Massey Algorithm**

The Berlekamp-Massey algorithm is a beautiful way to find the best fit for a set of equations involving an LFSR relationship. It has a computational complexity of \( O(v^2) \).

The particular way of implementing the Berlekamp-Massey in our decoder is what is mentioned in the book by Todd.K.Moon.
roots. It is quite easy to implement but there is a problem associated with this, which results in a decoder failure and is dealt with in that section.

The following code does a Chien search

```matlab
for i=0:n-1
    temp1=gf(0,m,prim_poly);
    for j=1:L+1
        temp1=temp1+c(j)*(gf(2,m,prim_poly)^(i*j));
    end
    if temp1==0
        roots(k)=gf(2,m,prim_poly)^i;
        k=k+1;
    end
end
```

F. Forney’s Algorithm to estimate error and erasure values

Having found the error locations, the key equation is

\[ \Omega(x) = S(x)\Gamma(x)\Lambda(x) \pmod{x^{2t}} \]  

(6)

and the polynomial called the combined error/erasure locator polynomial is obtained by \( \Phi(x) = \Lambda(x)\Gamma(x) \). Then error and erasure values are obtained by

\[ e_{i_k} = -\frac{\Omega(x_k^{-1})}{\Phi(x_k^{-1})} \]  

(7)

\[ f_{i_k} = -\frac{\Omega(Y_k^{-1})}{\Phi(Y_k^{-1})} \]  

(8)

The following code computes one of the error values, this should be done for all error and erasure locations.

```matlab
templ=gf(0,m,prim_poly);
for j=1:2*t
    templ=templ+W(j)*(roots(i)^(j-1));
end
temp2=gf(0,m,prim_poly);
for j=1:P_size
    temp=gf(0,m,prim_poly);
    for k=1:j
        temp=temp+P(j+1)*(roots(i)^(j-1));
    end
    temp2=temp2+temp;
end
e(X_exp(i)+1)=-(temp1/temp2);
```

III. Decoder failure

There can be two cases of decoder failure:

- Any RS code is an MDS code, so there can never be a case where the number of errors alone (excluding the erasures) will exceed \( t \). The decoder will fail if there are some erasures along with errors such that \( e + 2v > 2t \), where \( e \) is the number of erasures and \( v \) is the number of errors, in which case the reason is "too many errors and erasures for the decoder to handle."

- Given a set of syndrome values, the Berlekamp will find the best suited polynomial that will satisfy the given syndrome, say \( \Lambda(x) \) is the error locator polynomial obtained. Whilst performing Chien search, we assume that the \( \Lambda(x) \) found earlier has a number of roots equal to that of the degree \( \deg(\Lambda(x)) \). i.e \( \Lambda(x) = (x-a)(x-b)(x-c)... \) where \( a, b, c \) are in the field. So if the \( \Lambda(x) \) obtained doesn’t have \( 'k' \) roots, \( k = \deg(\Lambda(x)) \), then the decoder has detected \( k \) errors, but is not able to find the location of some or all of its roots. If such a scenario arises, then Chien search and hence the decoder will fail because of "its inability to locate the positions of all the errors".

IV. Conclusion

The project implements an efficient RS encoder and decoder from the scratch and uses only the Galois Field operations already present in the MATLAB environment. All other functions are user-defined. To test the correctness of the decoder (the encoder has been tested already using the RSenc), we refer to a solved example given in the book by TODD.K.MOON, namely example 6.20. The received vector is the input to the decoder and the output is as follows:

```
x = 12 99 6 11 0 11 100
   100 99 6 99 5
T = GF(2^4) array. Primitive polynomial = D^4+D+1
   Array elements =
   1    7    13
s = GF(2^4) array. Primitive polynomial = D^4+D+1
   Array elements =
   1    0    10
   15    4    5
W = GF(2^4) array. Primitive polynomial = D^4+D+1
   Array elements =
   1    9    11
   0    0    0
e_exp =
   Columns 1 through 13
   Columns 14 through 15
   99    99
rx =
   Columns 1 through 13
   12    99    6    11    0    11    14
```

...
Apart from this Example 6.18 in Moon has also been tested and verified for each step of Berlekamp-Massey and all other outputs.

REFERENCES

[1] Todd K. Moon, *Error-Correction Coding*