ENEE 624: Error Correcting Codes
Project 4

Brendan Cain 4/21/21

1 Introduction:

The purpose of this project is to implement a Reed-Solomon decoder using the Berlekamp-Massey, Chien Search, and the Forney Algorithm sequence. Computations are implemented using the functions created in project 3 that are used to do arithmetic operations in $GF(2^m)[x]$. All algorithms are implemented in MATLab code. Function logic and the various algorithms used are described in detail in *Section 2: Implementing A Reed-Solomon Decoder*. The manner in which these functions are tested is described in *Section 3: Testing the Reed-Solomon Decoder*. All MATLab code is published at the end of the respective section in which it was mentioned. All information used in this report was derived from materials provided by Dr. EF Charles LaBerge in the Error Correcting Codes course (*ENEE 624*) at University of Maryland, Baltimore County and "*Error Correction Coding: Mathematical Methods and Algorithms*" by Todd K. Moon.

2 Implementing A Reed-Solomon Decoder:

This section describes the manner of implementation for functions that make up a Reed-Solomon decoder. These functions will be used in the following section to test the decoder and the inscription of all functional code is shown at the end of this section.

The steps for decoding a Reed-Solomon codeword $\in GF(2^m)[x]$ are shown below:

1. Compute the syndromes: Evaluate the received code polynomial $R(x)$ at $2t$ consecutive powers of the primitive root α to get $2t$ syndrome coefficients $S_j \in GF(2^m)$. If all syndromes are $\alpha^{inf} = 0$, then the algorithm quits here because there are no errors detected.
2. Determine the Error Locator Polynomial (ELP): Using the syndromes and the Berlekamp-Massey Algorithm (described in its implemented function), get the error locator polynomial called $\Lambda(x)$. This polynomial is used in the next steps to determine the location of the errors in $R(x)$.
3. Get the roots of the ELP: Do Chien Search (described in its implemented function) to determine all non-zero roots of $\Lambda(x)$. The inverse of these roots corresponds to the symbol locations that have errored in $R(x)$.
4. Determine the Error Evaluator Polynomial (EEP): The Forney method, (described in its implemented function), is used to compute the error evaluator polynomial $\Omega(x)$ which is then utilized to construct the Forney Equation used in the next step.
5. Compute the Error Polynomial: Using $\Omega(x)$ and $\Lambda(x)$, the Forney Equation is constructed. The inputs to the Forney Equation are the roots of $\Lambda(x)$ found in step 3. The output of this equation yields the error value that corresponds to the location represented by the root of $\Lambda(x)$ that was inputted. Using the error values and the error locations, the error polynomial $E(x)$ can be constructed
6. Compute the Corrected Codeword: Adding $E(x)$ to $R(x)$ gives the corrected codeword polynomial $\hat{c}(x)$.

The main function that houses the decoder is called **`RS_Decoder`**. It is mainly responsible for calling the functions that implement the steps above as well as logging algorithm information to the console. This function along with the functions implementing the steps are described below:

function [C_hat, failure] = RS_Decoder(R_GF2m, t, GF, prnt_flag)
This function implements the decoding steps above and calls various helper functions that will be described later in this section. The inputs to this function are as follows: **R_GF2m** is the received codeword polynomial ∈ $GF(2^m)[x]$ represented in power form; **t** is the maximum amount of symbol errors that the code is designed to correct; **GF** is the array of cells enumerating $GF(2^m)$ symbols as elements ∈ $GF(2)[\alpha]$ and is used for passing to helper functions; **print_flag** is an optional parameter that will print out step-by-step algorithm updates to the console. **C_hat** and **failure** are the outputs of the function. **C_hat** is the corrected codeword $\hat{c}(x)$ and **failure** tells the user whether or not decoding was successful. This function catches errors thrown by the functions that implement the steps above. However, if that occurs, this function determines that the codeword is uncorrectable and throws an error of its own to the calling program. The decoder erroring is only possible during steps 2 or 3 from the above procedures and can happen for several reasons that relate to the number of errors being greater than **t**.

function syndromes = get_syndromes(t, R, GF, prnt_flag)
This function computes the syndrome polynomial $S(x)$ returned as the power form polynomial **syndromes**. It is computed from the power form representation of the received codeword polynomial $R(x)$ passed into the function as **R**. The integer value **t** represents the max number of errors that the code is designed to detect and correct. **GF** is the array of cells returned by the **GenerateGF2** function enumerating $GF(2^m)$ symbols as elements ∈ $GF(2)[\alpha]$ and is used for mapping between rectangular form and power form as well as passing to the **EvalPolyGF2** function. **prnt_flag** Is an optional parameter that prints algorithm information to the console. Since this function is used for determining the syndromes for a Reed-Solomon code, the algorithm for computing each syndrome coefficient is done by evaluating the received polynomial $R(x)$ at $2t$ nonzero consecutive powers of α using **EvalPolyGF2**. The resulting output for each evaluation should be a single symbol in $GF(2^m)$. The placement of the $2t$ symbols in the **syndromes** power form polynomial array corresponds to which power of α that $R(x)$ was evaluated at. For example, if $R(x)$ was evaluated at α^2 yielding $R(\alpha^2) = \alpha^y$ for some power y, then the x^2 term of $S(x)$ would appear as $\alpha^y * x^2$. After the **syndromes** array is populated with appropriate syndrome coefficients it is returned to the calling program. Note: if there are no errors found in the received polynomial, all syndromes will evaluate to $\alpha^{inf} = 0$. Also, the x^0 coefficient will always be $\alpha^{inf} = 0$ because the $2t$ consecutive powers of α that are used to evaluate syndromes must all be nonzero.

function lambda = get_error_loc(syn, GF, prnt_flag)
This function implements the Berlekamp-Massey Algorithm for determining the error locator polynomial $\Lambda(x)$. The inputs to this function are as follows: is **syn** the syndrome polynomial $S(x)$; **GF** is the array of cells enumerating $GF(2^m)$ symbols as elements ∈ $GF(2)[\alpha]$; **prnt_flag** Is an optional parameter that prints algorithm steps to the console. Below, in *Figure 1*, is a step-by-step description of the algorithm that attempts to evaluate coefficients of $\Lambda(x)$ by iterating over the $2t$ syndrome coefficients and doing $GF(2^m)[x]$ arithmetic. The output of this function is the error location polynomial $\Lambda(x)$ stored as **lambda** in power form. In the calling decoder function, **RS_Decoder**, an error should be thrown if the degree of $\Lambda(x)$ is greater than t, representing the number of errors the code can correct.

Berlekamp-Massey Algorithm for R-S Codes [Wicker]

1. Compute the syndromes $S_1 \ldots S_r$ for the received codeword. The syndromes will not all be zero, but is possible that one or more will be zero. Retain that syndrome in its appropriate place. Form the syndrome polynomial $S(x) = \sum_{k=1}^{r} S_k x^k = S_1 x + S_2 x^2 + \cdots + S_r x^r$.
2. Initialize the algorithm values: $k = 0, \Lambda^{(0)}(x) = 1, L = 0, T(x) = x$.
3. Set $k = k + 1$. Compute the *discrepancy*, $\Delta^{(k)}$ by subtracting (adding in $GF(2^m)$): $\Delta^{(k)} = S_k - \sum_{i=1}^{L} \Lambda_i^{(k-1)} S_{k-i}$, where $\Lambda_i^{(k-1)}$ is the coefficient of x^i in the $k-1$ (previous) value of $\Lambda(x)$.
4. If $\Delta^{(k)} = 0$, go to Step 8.
5. Use $\Delta^{(k)}$ to modify the connection polynomial, creating a new $\Lambda^{(k)}(x) = \Lambda^{(k-1)}(x) - \Delta^{(k)} T(x)$.
6. If $2L \geq k$, go to Step 8.
7. Set $L = k - L$ and set $T(x) = \frac{\Lambda^{(k-1)}(x)}{\Delta^{(k)}}$. This is a normalizing step.
8. Shift $T(x)$ to the left (higher power): $T(x) = xT(x)$.
9. If $k < r$ go to Step 3
10. The error locator polynomial is $G = \Lambda(x) = \Lambda^{(r)}(x)$, that is, the polynomial resulting from the step $k = r$.

Figure 1

`function error_locs = chien_search(lambda, GF, prnt_flag)`

This function completes the Chien Search algorithm on the error locator polynomial, $\Lambda(x)$, to find the roots of the lambda function ∈ $GF(2^m)$ (the values of x where $\Lambda(x) = \alpha^{inf} = 0 \in GF(2^m)$.) Then it finds the error locations with the inverse of these roots. The inputs to the equation are as follows: **`lambda`** which is the error location polynomial $\Lambda(x)$ in power form; **`GF`** is the array of cells enumerating $GF(2^m)$ symbols as elements ∈ $GF(2)[\alpha]$ needed for arithmetic over $GF(2^m)[x]$; **`prnt_flag`** Is an optional parameter that prints algorithm steps to the console. The Chien Search algorithm attempts to find all non-zero roots of a given polynomial. It evaluates $\Lambda(x)$ at every symbol element ∈ $GF(2^m)$ using the **`EvalPolyGF2`** function. The output of this function is a vector containing the locations of the symbol errors in the $R(x)$ polynomial, stored as **`error_locs`**. This function will throw an error to **`RS_Decoder`** if the number of roots found is greater than the degree of $\Lambda(x)$. If this occurs **`RS_Decoder`** should handle the error by issuing a decoder failure and throwing an error to the calling program.

`function e = evalForney(omega, lambda_prime, X, GF)`

This function is used to evaluate the Forney expression for a given **`X`** root of $\Lambda(x)$. The Forney expression involves the use of the error evaluator polynomial, $\Omega(x)$, represented and passed in as **`omega`** in power form. It also requires the *formal derivative* of the $\Lambda(x)$ polynomial called, $\Lambda'(x)$, which is represented and passed in as **`lambda_prime`** in power form. The formal derivative is computed with the **`FormalDerivGF2`** function over $GF(2^m)[x]$. The calling function **`RS_Decoder`** is responsible for providing the $\Omega(x)$ and $\Lambda'(x)$ polynomials. Formulas for both functions are shown below in *Figure 2* and *Figure 3*, they are computed using $GF(2^m)[x]$ arithmetic. The input **`GF`** is the array of cells enumerating $GF(2^m)$ symbols as elements $\in GF(2)[\alpha]$ needed for arithmetic over $GF(2^m)[x]$. The return value, **`e`**, is the error value $\in GF(2^m)$ for the given input **`X`**. The main decoder function uses these outputs and the error locations given by the roots of $\Lambda(x)$ to construct the error polynomial $E(x)$.

$$\Omega(x) = \Lambda(x) \times \left(S(x) + 1 \right) \bmod x^{r+1}$$

Figure 2

$$e_{i_k} = \frac{-X_i \Omega\left(X_i^{-1}\right)}{\Lambda'\left(X_i^{-1}\right)}$$

Figure 3

Finally, after steps 1 – 5 are completed by the functions described above, **`RS_Decoder`** sums the received polynomial with the error polynomial to get $\hat{c}(x)$ as the corrected codeword **`C_hat`** represented as a power form polynomial $\in GF(2^m)[x]$. The summation over $GF(2^m)[x]$ is as follows $\hat{c}(x) = R(x) + E(x)$. Below, is a sample output of with the step-by-step algorithm updates being displayed to the user. Also below is an inscription of all the MATLab code described above

```
function [C_hat, failure] = RS_Decoder(R_GF2m, t, GF, prnt_flag)
%{
RS_DECODER: implementation of the Berlekamp-Massey algorithm used
for decoding and correcting a received Reed-Solomon codeword in
 GF(2^m).
This function calls other functions to derive these polynomials needed
 for
error detection and correction: the syndrome polynomial = S(x)
(get_syndromes.m), the error locator polynomial = lambda(x)
(get_error_loc.m), the error location polynomial = the inverse roots
 of
lambda(x) (chien_search.m), the error polynomial = e(x) (coeff given
 by
evalForney.m). These functions utilize the GF(2^m) arithmetic
 functions
created in proj3. Additional GF(2^m) functions were created when a
specific computation is needed for a given algorithm.

INPUTS:
 R_GF2m    - the received codeword polynomial in power form to be
 decoded
 t         - an integer value denoting the max amount of errors the
 code is
             able to correct.
 GF        - GF is the array of cells enumerating GF(2^m) symbols as
             elements in GF(2)[a].
 prnt_flag - optional Boolean parameter that defaults to false,
 displays
             algorithm results step-by-step if true, display is
 omitted if
             false.
OUTPUTS:
 C_hat   - The corrected codeword in power format that is returned at
 the
           end of the function.
 failure - Boolean output. If true, decoder failed and C_hat is set to
           equal all -1's because a^-1 represents a^inf = 0 (C_hat
 becomes
           the all zeros codeword)
%}

%see if need to set default value of prnt_flag
if ~exist('prnt_flag','var')
    prnt_flag = false;
end

%set initial conditions
failure = false;
n = size(R_GF2m,2);
m = size(GF{1},2);
k = n-2*t;
if(prnt_flag)
```

```
    fprintf("---------Reed-Solomon Decoding---------\n");
    fprintf("\nPART 1 - Initial Conditions:\n");
    fprintf("  1.) Decoding a (%d,%d) %d-error correcting Reed-Solomon
 Code\n", n, k, t);
    fprintf("    with operations over GF(2^%d)\n", m);
    fprintf("  2.) The received codeword in both power and polynomial
 form:\n");
    print_poly("    R[x]", R_GF2m, true);
    print_poly("    R(x)", R_GF2m, false);
    fprintf("  3.) The array of cells enumerating GF(2^%d) symbols as
\n", m);
    fprintf("    elements in GF(2)[a] is shown below:\n");
    print_poly("    a^inf", GF{1}, true);
    for elm=2:2^m
        fprintf("    a^%d", elm-2);
        print_poly("  ", GF{elm}, true);
    end
end

%get syndromes
if(prnt_flag)
    fprintf("\nPART 2 - Construct the syndrome polynomial S(x):\n");
end
s = get_syndromes(t, R_GF2m, GF, prnt_flag);

%if syndromes are all inf then there are no errors, the codeword
 passes,
%and the algorithm ends here
if(s(:) == -1)
    C_hat = R_GF2m;
    if(prnt_flag)
        fprintf("  2.) After the computation of all syndrome
 coefficients, S(x) is constructed\n");
        fprintf("    in power form as: ");
        print_poly("S[x]", s, true);
        fprintf("  3.) All S(x) coefficients are a^inf and thus no
 errors were detected\n");
        fprintf("  4.) Therefore, the corrected codeword C_hat is set
 to R(x)\n");
        fprintf("    and the algorithm ends:\n");
        print_poly("    C_hat[x]", C_hat, true);
        print_poly("    C_hat(x)", C_hat, false);
    end

    return;
end

if(prnt_flag)
    fprintf("  2.) After the computation of all syndrome coefficients,
 S(x) is constructed:\n");
    print_poly("    S[x]", s, true);
    print_poly("    S(x)", s, false);
    fprintf("  3.) Since some syndrome coefficients are non-zero,
 there are errors in R(x).\n");
```

```
    fprintf("\nPART 3 - Use the Berlekamp-Massey Algrorithm to compute
 lambda(x):\n");
end

%get the error location polynomial

lambda = get_error_loc(s, GF, prnt_flag);
deg_lambda = size(lambda,2) - 1;

%if the degree of lambda is greater than t then the decoder fails and
 the
%algorithm ends here.
if(deg_lambda > t)
    failure = true;
    C_hat = zeros(1,n);
    C_hat(:) = -1;

    if(prnt_flag)
        fprintf("  3.) The error locator polynomial lambda(x) for
 R(x):\n");
        print_poly("   lambda[x]", lambda, true);
        print_poly("   lambda(x)", lambda, false);
        fprintf("  4.) However the degree of lambda(x)(%d) is greater
 than t(%d)\n", deg_lambda, t);
        fprintf("   and so a decoder failure is triggered.");
        fprintf("  5.) C_hat is set to the all zeros code word (a^inf)
 and the algorithm quits\n");
        print_poly("   C_hat[x]", C_hat, true);
    else
        error("The degree of lambda(x)(%d) is greater than t(%d)!
 Decoder Failed!\n", deg_lambda, t);
    end

    return;
end

if(prnt_flag)
    fprintf("  3.) The error locator polynomial lambda(x) for R(x):
\n");
    print_poly("   lambda[x]", lambda, true);
    print_poly("   lambda(x)", lambda, false);
    fprintf("\nPART 5 - Determine the error locations of R(x) using
 Chien Search and lambda(x):\n");
end

%get the error locations via chien search
try
    %could error
    error_locations = chien_search(lambda, GF, prnt_flag);
catch chien_error
    %if the number of errors is less than the degree of lambda then
 the
    %decoder fails and the algorithm ends here.
    failure = true;
```

```
    C_hat = zeros(1,n);
    C_hat(:) = -1;

    if(prnt_flag)
        fprintf("  2.) The Chien Search algorithm abruptly stopped
 with the error message:\n");
        fprintf("    %s", chien_error.message);
        fprintf("  3.) C_hat is set to the all zeros code word (a^inf)
 and the algorithm quits\n");
        print_poly(" C_hat[x]", C_hat, true);
    else
        err_msg = sprintf("The Chien Search algorithm abruptly stopped
 with the error message:\n%sDecoder Failed!\n",chien_error.message);
        error("%s", err_msg);
    end
    return;
end

num_errors = size(error_locations, 2);


s_plus_1 = s;
s_plus_1(end) = 0;
%error evaluator polynomial below...
omega = PolyMultGF2(lambda, s_plus_1, GF);
omega = omega(1,end-2*t:end); %do this to take mod(x^2t+1)

%need derivative of error locator (lambda_prime)
lambda_prime = FormalDerivGF2(lambda, GF);

if(prnt_flag)
    fprintf("\nPART 6 - Determine the error polynomial e(x):\n");
    fprintf(" - In order to determine the error polynomial, first
 determine\n");
    fprintf("   the error evaluator polynomial, omega(x) using the
 Forney method.\n");
    fprintf(" - Also formally derive lambda(x) to get lambda'(x) for
 its\n");
    fprintf("   use in the Forney equation.\n");
    fprintf("  1.) Omega is given by the equation: omega(x) =
 lambda(x)*(S(x) + 1)mod(x^(r+1))\n");
    print_poly("    omega[x]", omega, true);
    print_poly("    omega(x)", omega, false);
    fprintf("  2.) lambda'(x) is computed and shown below:\n");
    print_poly("    lambda'[x]", lambda_prime, true);
    print_poly("    lambda'(x)", lambda_prime, false);
    fprintf("  3.) Use the Forney equation of the form:\n");
    fprintf("    e_i = -X_i*omega(X_i^-1)/lambda'(X_i^-1), where\n");
    fprintf("    e_i is the error value for error i, and X_i is a root
 of \n");
    fprintf("    the error locator polynomial.\n");
    fprintf("  4.) Evaluate the Forney equation for all error
 locations:\n");
end
```

```
e = zeros(1, n); %initialize size of the error polynomial
e(:) = -1; %fill with a^inf

for e_loc = 1:num_errors
    X = error_locations(1, e_loc);
    e_idx = n - X;
    e(1, e_idx) = evalForney(omega, lambda_prime, X, GF);
    if(prnt_flag)
        fprintf("    for error %d at x^%d, the error value is: %d\n",
 e_loc, X, e(1, e_idx));
    end
end

C_hat = PolyAddGF2(e, R_GF2m, GF);


if(prnt_flag)
    fprintf("  5.) Construct the the error polynomial, e(x), using the
 error values\n");
    fprintf("    and the previously determined error locations:\n");
    print_poly("    e[x]", e, true);
    print_poly("    e(x)", e, false);
    fprintf("\nPART 7 - Finally, add e(x) to the received codeword
 polynomial R(x):\n");
    fprintf(" - This gives the corrected codeword polynomial,
 C_hat(x), shown below:\n");
    print_poly("    C_hat[x]", C_hat, true);
    print_poly("    C_hat(x)", C_hat, false);
end

end

---------Reed-Solomon Decoding---------

PART 1 - Initial Conditions:
  1.) Decoding a (7,3) 2-error correcting Reed-Solomon Code
    with operations over GF(2^3)
  2.) The received codeword in both power and polynomial form:
    R[x] = [ 0 6 5 3 Inf 5 5 ]
    R(x) = x^6 + a^6*x^5 + a^5*x^4 + a^3*x^3 + a^5*x + a^5
  3.) The array of cells enumerating GF(2^3) symbols as
    elements in GF(2)[a] is shown below:
    a^inf = [ 0 0 0 ]
    a^0   = [ 0 0 1 ]
    a^1   = [ 0 1 0 ]
    a^2   = [ 1 0 0 ]
    a^3   = [ 0 1 1 ]
    a^4   = [ 1 1 0 ]
    a^5   = [ 1 1 1 ]
    a^6   = [ 1 0 1 ]

PART 2 - Construct the syndrome polynomial S(x):
 - S(x) is constructed as a polynomial with a max degree of 4.
```

- The coefficients of S(x) are computed by evaluating the
  recevied polynomial, R(x) = x^6 + a^6*x^5 + a^5*x^4 + a^3*x^3 +
a^5*x + a^5
  at 2t consecutive powers of alpha (2t=4) over GF(2^3)[x].
- Each evaluated syndrome is the coefficient of the x-term
  whose degree matches the power of alpha that R(x) was
  evaluated at.
 1.) Evaluations of S_1 to S_4 are shown below:
   S_1 = R(a^1) = 0
   S_2 = R(a^2) = a^2
   S_3 = R(a^3) = a^4
   S_4 = R(a^4) = a^2
 2.) After the computation of all syndrome coefficients, S(x) is
constructed:
   S[x] = [2 4 2 Inf Inf]
   S(x) = a^2*x^4 + a^4*x^3 + a^2*x^2
 3.) Since some syndrome coefficients are non-zero, there are errors
in R(x).

PART 3 - Use the Berlekamp-Massey Algrorithm to compute lambda(x):
- The following algorithm is used to construct the error
  locator polynomial, lambda(x), utilizing the syndrome
  coefficients and GF(2^3)[x] arithmetic.
- The roots of lambda(x) will be used to determine the
  locations of the symbol errors in R(x)
 1.) Initialize the algorithm values:
  - k = 0
  - lambda_0(x) = 1
  - L = 0
  - T(x) = x
  - r = 4 (the number of syndromes)
  - syndrome coefficients: S[r] = [2 4 2 Inf]
 2.) Iterate over the following steps until k is equal to r:
---- Iteration 1 ----
lambda_k-1(x) = 1
Step 1.1) Increment k and compute the discrepancy delta_k
  by subtracting the S_k coefficient and the sum of products from
  i=1:L of lambda_k-1[i]*S_k-i
  --Values--
  k = 1
  delta_1 = a^inf
Step 2.1) If delta_1 is a^inf go to Step 6
Step 6.1) Shift coefficients T(x) over by one degree (multiply by x)
  --Values--
  T(x) = T(x)*x = x^2
Step 7.1) If k is less than r, go to Step 1
---- Iteration 2 ----
lambda_k-1(x) = 1
Step 1.2) Increment k and compute the discrepancy delta_k
  by subtracting the S_k coefficient and the sum of products from
  i=1:L of lambda_k-1[i]*S_k-i
  --Values--
  k = 2
  delta_2 = a^2

```
Step 2.2) If delta_2 is a^inf go to Step 6
Step 3.2) Compute lambda_k(x) by adding delta_k*T(x) to lambda_k-1(x)
  --Values--
  delta_k*T(x) = a^2*x^2
  lambda_k-1(x) = 1
  lambda_k(x) = a^2*x^2 + 1
Step 4.2) If 2L is greater that or equal to k, go to Step 6
Step 5.2) Set L and T(x)
  --Values--
 L = k - L = 2
  T(x) = lambda_k-1(x)/delta_k = a^5
Step 6.2) Shift coefficients T(x) over by one degree (multiply by x)
  --Values--
  T(x) = T(x)*x = a^5*x
Step 7.2) If k is less than r, go to Step 1
---- Iteration 3 ----
lambda_k-1(x) = a^2*x^2 + 1
Step 1.3) Increment k and compute the discrepancy delta_k
  by subtracting the S_k coefficient and the sum of products from
  i=1:L of lambda_k-1[i]*S_k-i
  --Values--
  k = 3
  delta_3 = a^4
Step 2.3) If delta_3 is a^inf go to Step 6
Step 3.3) Compute lambda_k(x) by adding delta_k*T(x) to lambda_k-1(x)
  --Values--
  delta_k*T(x) = a^2*x
  lambda_k-1(x) = a^2*x^2 + 1
  lambda_k(x) = a^2*x^2 + a^2*x + 1
Step 4.3) If 2L is greater that or equal to k, go to Step 6
Step 6.3) Shift coefficients T(x) over by one degree (multiply by x)
  --Values--
  T(x) = T(x)*x = a^5*x^2
Step 7.3) If k is less than r, go to Step 1
---- Iteration 4 ----
lambda_k-1(x) = a^2*x^2 + a^2*x + 1
Step 1.4) Increment k and compute the discrepancy delta_k
  by subtracting the S_k coefficient and the sum of products from
  i=1:L of lambda_k-1[i]*S_k-i
  --Values--
  k = 4
  delta_4 = a^5
Step 2.4) If delta_4 is a^inf go to Step 6
Step 3.4) Compute lambda_k(x) by adding delta_k*T(x) to lambda_k-1(x)
  --Values--
  delta_k*T(x) = a^3*x^2
  lambda_k-1(x) = a^2*x^2 + a^2*x + 1
  lambda_k(x) = a^5*x^2 + a^2*x + 1
Step 4.4) If 2L is greater that or equal to k, go to Step 6
Step 6.4) Shift coefficients T(x) over by one degree (multiply by x)
  --Values--
  T(x) = T(x)*x = a^5*x^3
Step 7.4) If k is less than r, go to Step 1
---- Error Locator Algorithm Complete ----
```

```
 Final lambda_k(x) = lambda(x) = a^5*x^2 + a^2*x + 1
  3.) The error locator polynomial lambda(x) for R(x):
   lambda[x] = [ 5 2 0 ]
   lambda(x) = a^5*x^2 + a^2*x + 1

PART 5 - Determine the error locations of R(x) using Chien Search and
 lambda(x):
 - This part attempts to find the roots of lambda(x)
 - The inverse of the roots of lambda(x) correspond to
   the power of the x-term that denotes the location of
   of the symbol error in the received codeword.
  1.) Loop through non-zero powers of the primitive root alpha
    and use them to evaluate the error locator polynomial.
    If lambda(a^?) = 0, that power of alpha is a root.
    1 Root(s) found! lambda(a^3) = 0
    2 Root(s) found! lambda(a^6) = 0
  2.) There were 2 roots of lambda(x) found without any errors.
    Therefore, taking the inverse of the roots gives us
    the error locations in R(x) shown below:
    - error 1: 1/a^3 = a^4. The symbol coefficient
      at x^4 in R(x) has an error
    - error 2: 1/a^6 = a^1. The symbol coefficient
      at x^1 in R(x) has an error

PART 6 - Determine the error polynomial e(x):
 - In order to determine the error polynomial, first determine
   the error evaluator polynomial, omega(x) using the Forney method.
 - Also formally derive lambda(x) to get lambda'(x) for its
   use in the Forney equation.
  1.) Omega is given by the equation: omega(x) = lambda(x)*(S(x) +
 1)mod(x^(r+1))
    omega[x] = [ Inf Inf 3 2 0 ]
    omega(x) = a^3*x^2 + a^2*x + 1
  2.) lambda'(x) is computed and shown below:
    lambda'[x] = [ 2 ]
    lambda'(x) = a^2
  3.) Use the Forney equation of the form:
    e_i = -X_i*omega(X_i^-1)/lambda'(X_i^-1), where
    e_i is the error value for error i, and X_i is a root of
    the error locator polynomial.
  4.) Evaluate the Forney equation for all error locations:
    for error 1 at x^4, the error value is: 3
    for error 2 at x^1, the error value is: 6
  5.) Construct the the error polynomial, e(x), using the error values
    and the previously determined error locations:
    e[x] = [ Inf Inf 3 Inf Inf 6 Inf ]
    e(x) = a^3*x^4 + a^6*x

PART 7 - Finally, add e(x) to the received codeword polynomial R(x):
 - This gives the corrected codeword polynomial, C_hat(x), shown
 below:
    C_hat[x] = [ 0 6 2 3 Inf 1 5 ]
    C_hat(x) = x^6 + a^6*x^5 + a^2*x^4 + a^3*x^3 + a*x + a^5
```

```
function syndromes = get_syndromes(t, R, GF, prnt_flag)
%{
GET_SYNDROMES computes the syndrome polynomial of the recieved
 codeword based off of
 the value of t representing the amount of errors the BCH/R-S code is
 supposed to correct
Inputs:
    t - number of errors the code is supposed to correct
    R - the recieved codeword of length n
    GF - the cell matrix made up of the elements in GF(2^m) whose
    indices are the powers of alpha + 2
Output:
    syndromes - a power form polynomial that holds the evaluation of R
 for a specific
    value a^i in GF(2^m) where i is {1:2*t} (assumes b = 1 or narrow
 sense)
%}

%NOTE: syndrome polynomial represented in power form will always have
 -1
%(inf) in syndromes(1,end), the lowest order coeff, because S(i) goes
 with
%the power of x^i and since i = 1:2*t the first Syndrome value will
%always be S1

%see if need to set default value of prnt_flag
if ~exist('prnt_flag','var')
    prnt_flag = false;
end

%set initial conditions and initialize syndrome vector
num_synd = 2*t;
syndromes = zeros(1, num_synd+1);
syndromes(1, end) = -1;
m = size(GF{1},2);

if(prnt_flag)
    fprintf(" - S(x) is constructed as a polynomial with a max degree
 of %d.\n", num_synd);
    fprintf(" - The coefficients of S(x) are computed by evaluating
 the\n");
    fprintf("   recevied polynomial, ");
    print_poly("R(x)", R, false);
    fprintf("   at 2t consecutive powers of alpha (2t=%d) over GF(2^
%d)[x].\n", num_synd, m);
    fprintf(" - Each evaluated syndrome is the coefficient of the x-
term\n");
    fprintf("   whose degree matches the power of alpha that R(x) was
\n");
    fprintf("   evaluated at.\n");
    fprintf("  1.) Evaluations of S_1 to S_%d are shown below:\n",
 num_synd);
```

```
end

for i = 1:num_synd
    syndromes(end-i) = EvalPolyGF2(R, i, GF);
    if(prnt_flag)
        if(syndromes(end-i) == -1)
            eval_str = "0";
        elseif(syndromes(end-i) == 0)
            eval_str = "1";
        elseif(syndromes(end-i) == 1)
            eval_str = "a";
        else
            eval_str = sprintf("a^%d",syndromes(end-i));
        end
        fprintf("    S_%d = R(a^%d) = %s\n", i, i, eval_str);
    end
end


end
```

Published with MATLAB® R2018b

```
function lambda = get_error_loc(syn, GF, prnt_flag)
%GET_ERROR_LOC does the berlekamp massey algorithm and computes the
 error
%locator polynomial outputs as a power form polynomial
% syn - the syndrome polynomial with a degree of r <= 2t
% GF - the equivalency cell matrix mapping the GF(2) groups to GF(2^m)
%      elements

%see if need to set default value of prnt_flag
if ~exist('prnt_flag','var')
    prnt_flag = false;
end

m = size(GF{1}, 2);
if(prnt_flag)
    fprintf(" - The following algorithm is used to construct the error
\n");
    fprintf("   locator polynomial, lambda(x), utilizing the syndrome
\n");
    fprintf("   coefficients and GF(2^%d)[x] arithmetic.\n", m);
    fprintf(" - The roots of lambda(x) will be used to determine the
\n");
    fprintf("   locations of the symbol errors in R(x)\n");
end

% Step 1: compute syndrome (syndrome polynomial comes in power form
 but
% we just need it as an array because were only going to use S(t*2) to
 S(1)
syn = syn(1, 1:end-1);

% Step 2: initial values for the algorithm
%(this is Step 2 - because Step 1 is computing syndrome)
r = size(syn, 2); %the amount of syndromes

% this polynomial will be populated throughout this algorithm
deg_lambda = 2^size(GF{1},2); %2^m
lambda = zeros(1, deg_lambda+1); %initial value is 1 in power form
lambda(1,1:end-1) = -1;

L = 0; %counter used for computing discrepancy (delta)
T = zeros(1, deg_lambda+1); %T(x) = x in power form
T(1,1:end) = -1;
T(1,end-1) = 0;

if(prnt_flag)
    fprintf("  1.) Initialize the algorithm values:\n");
    fprintf("    - k = 0\n   - ");
    print_poly("lambda_0(x)", lambda, false);
    fprintf("    - L = 0\n   - ");
    print_poly("T(x)",T, false);
    fprintf("    - r = %d (the number of syndromes)\n", r);
```

```
    fprintf("   - syndrome coefficients: ");
    print_poly("S[r]", syn, true);
    fprintf("  2.) Iterate over the following steps until k is equal
 to r:\n");
end

% Step 3: increment k and compute discrepency (delta)
for k = 1:r

    new_lambda = lambda; %this is just incase step 5 gets skipped

    %compute discrepancy or delta
    delta = syn(1, end-(k-1));
    sum = -1; %(sum actually is = 0 for GF(2^m) in power form)
    %get sum from 1 to L of lambda * syn coeff k-i
    for i = 1:L
        %get the product of the i lambda coef and the k-i S coeff
        to_sum = MultGF2(lambda(1,end-i), syn(1, end-(k-i-1)), GF);
        sum = AddGF2(sum, to_sum, GF);
    end
    %since adding and subtracting are the same

    delta = AddGF2(delta, sum, GF);

    if(prnt_flag)
        fprintf("\t---- Iteration %d ----\n\t", k);
        print_poly("lambda_k-1(x)", lambda, false);
        fprintf("\tStep 1.%d) Increment k and compute the discrepancy
 delta_k\n", k);
        fprintf("\t  by subtracting the S_k coefficient and the sum of
 products from\n");
        fprintf("\t  i=1:L of lambda_k-1[i]*S_k-i\n");
        fprintf("\t  --Values--\n");
        fprintf("\t  k = %d\n", k);

        if(delta ~= -1)
            fprintf("\t  delta_%d = a^%d\n",k,delta);
        else
            fprintf("\t  delta_%d = a^inf\n",k);
        end

        fprintf("\tStep 2.%d) If delta_%d is a^inf go to Step 6\n", k,
 k);
    end

% Step 4: if delta is 0 in GF(2^m) go to step 8
    if(delta == -1)

% Step 5: Use delta and T(x) to modify the lambda polynomial
    else
        to_add = PolyMultGF2(delta, T, GF); %delta*T(x)
        %subtract delta*T(x) from the current lambda and create the
        % new lambda polynomial (since subtracting is the same as
        % adding in this field, just add the polynomials together
```

```
        new_lambda = PolyAddGF2(lambda, to_add, GF);
        if(prnt_flag)
            fprintf("\tStep 3.%d) Compute lambda_k(x) by adding
 delta_k*T(x) to lambda_k-1(x)\n", k);
            fprintf("\t  --Values--\n");
            fprintf("\t  ");
            print_poly("delta_k*T(x)", to_add, false);
            fprintf("\t  ");
            print_poly("lambda_k-1(x)", lambda, false);
            fprintf("\t  ");
            print_poly("lambda_k(x)", new_lambda, false);
            fprintf("\tStep 4.%d) If 2L is greater that or equal to k,
 go to Step 6\n", k);
        end
% Step 6: if 2L is greater than or equal to k go to step 8
        if(2*L >= k)

% Step 7: set L and T(x)
        else
            L = k - L;
            %T(x) = quotient of old lambda (just called lambda right
 now) and delta
            [T, ~] = PolyDivGF2(lambda, delta, GF);
            if(prnt_flag)
                fprintf("\tStep 5.%d) Set L and T(x)\n", k);
                fprintf("\t  --Values--\n");
                fprintf("\t L = k - L = %d\n\t  ", L);
                print_poly("T(x) = lambda_k-1(x)/delta_k", T, false);
            end
        end
    end
% Step 8: Shift T(x) over by 1 (multiply by x which in power form is
 [0 -1])
    T = PolyMultGF2(T, [0 -1], GF);
    T = T(1,end-deg_lambda:end);
    if(prnt_flag)
        fprintf("\tStep 6.%d) Shift coefficients T(x) over by one
 degree (multiply by x)\n", k);
        fprintf("\t  --Values--\n");
        fprintf("\t  ");
        print_poly("T(x) = T(x)*x", T, false);
        fprintf("\tStep 7.%d) If k is less than r, go to Step 1\n",
 k);
    end
    lambda = new_lambda; %just incase step 5 isn't skipped or/and
 looping is done
% Step 9: go back to step 3 if k < r
end

% Step 10: return lambda as the error locator polynomial

%Before this happens remove padding -1's or infs
for i=1:deg_lambda+1
    if(lambda(1,i) ~= -1)
```

```
        break;
    end
end

lambda = lambda(1,i:end);

if(prnt_flag)
    fprintf("\t---- Error Locator Algorithm Complete ----\n");
    fprintf("\tFinal lambda_k(x) = ");
    print_poly("lambda(x)", new_lambda, false);
end

end
```

Published with MATLAB® R2018b

```
function error_locs = chien_search(lambda, GF, prnt_flag)
%{
CHIEN_SEARCH Completes the chien search algorithm on the error locator
 polynomial to find the roots of the lambda function in GF(2^m) (the
 values
 of x where lambda = 0 (inf in GF(2^m)) Finds the error locations with
 the
 inverse of these roots
INPUTS:
    lambda - the power form error locator polynomial array whose
 elements
    represent the powers of the alpha coefficients of the polynomial.
    highest degree coeff in lowest index

    GF - Array of cells mapping the binary groups to GF(2^m) alpha
 coeff
OUTPUTS:
    error_locs - a list representing the inverse of roots of the
 lambda polynomial.
    Numbers are powers of alpha. These are the error locations in the
    recieved codeword
%}

%see if need to set default value of prnt_flag
if ~exist('prnt_flag','var')
    prnt_flag = false;
end

deg_lambda = size(lambda, 2) - 1;

%create a roots list, number of roots corresponds to the degree of
 lambda
%polynomial
roots = zeros(1,deg_lambda);
roots(1:end) = -1;
error_locs = roots;

field_len = size(GF,1); %number of elements in the GF(2^m)
root_cnt = 0; %number of non-zero roots found

if(prnt_flag)
    fprintf(" - This part attempts to find the roots of lambda(x)\n");
    fprintf(" - The inverse of the roots of lambda(x) correspond to
\n");
    fprintf("   the power of the x-term that denotes the location of
\n");
    fprintf("   of the symbol error in the received codeword.\n");
    fprintf("  1.) Loop through non-zero powers of the primitive root
 alpha\n");
    fprintf("    and use them to evaluate the error locator
 polynomial.\n");
```

```
    fprintf("    If lambda(a^?) = 0, that power of alpha is a root.
\n");
end

%the indices of GF correlate to the powers of alpha that map to the
 binary
%number stored at that index location. The mapping is a^i-2 = GF{i}
%start at 2 because were looking for non 0 roots
for gf_idx = 2:field_len

    %power of alpha is gf_idx-2 which represents the x_input to the
 lambda
    %function to evaluate at each possible root
    root = gf_idx - 2;
    eval = EvalPolyGF2(lambda, root, GF);
    if(eval == -1)
        root_cnt = root_cnt + 1;
        if(root_cnt > deg_lambda)
            error("Too many roots found for the lambda polynomial of
 degree %d!\n", deg_lambda);
            return;
        end
        roots(root_cnt) = root;
        if(prnt_flag)
            fprintf("    %d Root(s) found! lambda(a^%d) = 0\n",
 root_cnt, root);
        end
    end
end

if(root_cnt ~= deg_lambda)
    error("The number of roots found (%d) is less the degree of
 lambda(x) (%d)!\n", root_cnt, deg_lambda);
end

%get the inverse of the roots, these are the locations of the errors
 in the
%received codeword
if(prnt_flag)
    fprintf("  2.) There were %d roots of lambda(x) found without any
 errors.\n", root_cnt);
    fprintf("    Therefore, taking the inverse of the roots gives us
\n");
    fprintf("    the error locations in R(x) shown below:\n");
end

for i = 1:root_cnt
    error_locs(i) = DivGF2(0,roots(i), GF);
    if(prnt_flag)
        fprintf("    - error %d: 1/a^%d = a^%d. The symbol coefficient
\n", i, roots(i), error_locs(i));
        fprintf("      at x^%d in R(x) has an error\n",
 error_locs(i));
    end
```

```
end


end
```

Published with MATLAB® R2018b

```
function e = evalForney(omega, lambda_prime, X, GF)
%{
EVALFORNEY evaluates the forney expression for a given X error
 location
INPUTS:
    omega - the omega polynomial or the error evaluator polynomial in
 power
            form
    lambda_prime - the formal derivative of the lambda (error locator)
                   polynomial in power form
    X - the error location that the forney expression evaluates at to
        determine the error value
    GF - Array of cells mapping the binary groups to GF(2^m) alpha
 coeff
OUTPUT:
    e - the error value (power of alpha) to be added to the
 coefficient at
        the error location
%}

n = size(GF, 1) - 1; %needed for inversing values of X;

dvd = MultGF2(X, EvalPolyGF2(omega, n-X, GF), GF); %dividened
dvs = EvalPolyGF2(lambda_prime, n-X, GF); %divisor

e = DivGF2(dvd, dvs, GF);

end
```

Published with MATLAB® R2018b

3 Testing the Reed-Solomon Decoder:

The functions from the above section were tested using a MATLab script called **decoder_main.m**. The initial conditions of Homework 7 – Problem 2, Homework 8 – Problem 2, and Homework 8 – Problem 3 were utilized in this script for the purpose of creating test cases implementation of the Reed-Solomon Decoder. The script fully handles any and all errors thrown by the decoder. The script will issue warnings if a specific test case triggers a decoder error and displays test statistics along with decoder error messages at the end of the script. The test cases are shown below.

3.1 Test Cases:

1. Test Set 1: a narrow sense (7,3) two-error correcting Reed-Solomon code
   1.1. A simulated received codeword with 2 errors, via *generation encoding*.
   1.2. A simulated received codeword with 3 errors, via *generation encoding* (should cause decoder error).
2. Test Set 2: a narrow sense (15,9) three-error correcting Reed-Solomon code
   2.1. A simulated received codeword with 3 errors, via *generation encoding*.
   2.2. A simulated received codeword with 3 errors, via *systematic encoding*.
   2.3. A simulated received codeword with 4 errors, via *systematic encoding* (should cause decoder error).

There are three helper functions that were created for the purpose of executing test cases and formatting proper output. Their function descriptions are shown in the following section as well as the code inscriptions and the **decoder_main.m** output.

3.2 Helper Function Descriptions:

Three helper functions are used along with the decoder algorithm functions and the **decoder_main.m** script in order to execute test cases: **do_test()**, **get_message()**, and **print_poly()**. Their descriptions as well as their code inscriptions are shown below.

function [bin, msg_symb] = get_message(code_wrd, GF, G, k)

This function gets the message polynomial $M(x)$ (**msg_symb**) of a corrected codeword as well as the message vector in bits (**msg**). It works with codewords that were generated using the systematic or generator method. If the codeword cannot be decoded it throws an error. The inputs to the function are as follows: **code_wrd** is the corrected codeword $\hat{c}(x) \in GF(2^m)[x]$; **GF** is the array of cells enumerating $GF(2^m)$ symbols as elements $\in GF(2)[\alpha]$ needed for determining the message bits; **G** is the generating polynomial $G(x) \in GF(2^m)[x]$ needed for getting the message polynomial; **k** is an optional parameter needed for getting the message polynomial if the original codeword was generated using systematic encoding. If **k** exists, then the message bits should be extracted using the systematic encoding way.

function do_test(test_name, R_err, GF, G, C, M, sys_gen)

This function executes a single test case, prints out relevant information, and determines the original message bits that were encoded using the above function. It may throw an error if decoding algorithm errors but that will be handled by the main program **decoder_main.m**. The inputs for the function are as follows: **test_name** is a string representing the name of the test; **R_err** is the simulated received polynomial that has errored; **GF** is the array of cells enumerating $GF(2^m)$ symbols as elements $\in$

$GF(2)[\alpha]$; **G** is the generating polynomial $G(x) \in GF(2^m)[x]$ needed for getting the message polynomial by passing to **get_message**; **C** is the original codeword to be compared to the decoded codeword $\hat{c}(x)$; **M** is the original message polynomial to be compared to the decoded message polynomial; **sys_gen** is Boolean value determining whether the original codeword was encoding using the generation method or the systematic method (true = generation method, false = systematic method). This function compares original to decoded codewords and utilizes the **print_poly** function below to print out their symbols in power and polynomial form.

function print_poly(name, polynomial, pwr_form)

This function is responsible for printing a given power form polynomial (**polynomial**) to the console in the form specified by the Boolean variable **pwr_form**. The string argument, **name**, contains the name of the polynomial as function of x that the user wishes to denote the polynomial as in the print statement (*example:* **name="Y(x)"**.) If **pwr_form** is **true**, then the polynomial is printed as power form looking like this: $Y(x) = [\, y_1 \ \ldots \ y_i \ \ldots \ y_l \,]$ where y_i is an element of the **polynomial** vector. When $y_i = -1$, the string **"inf"** is printed. If **pwr_form** is **false**, then the polynomial is printed holding the form of a normal $GF(2^m)[x]$ polynomial: $Y(x) = \alpha^{y}{}_1 * x^{l-1} + \cdots + \alpha^{y}{}_i * x^{l-i} + \cdots + \alpha^{y}{}_{l-1} * x + \alpha^{y}{}_l$ where $i \in \{1{:}\, l\}$, $\alpha^{y}{}_i \in GF(2^m)$, and $l =$ the length of **polynomial**. When $y_i = -1$ the term is skipped; when $y_i = 0$ the **"x"** term is only printed because $\alpha^0 * x^{l-1} = 1 * x^{l-1} = x^{l-1}$; when $y_i = 1$ the coefficient printed is only **"a"** because $\alpha^1 = \alpha$. Both HW 7.2 and HW 8.1 utilize this function.

```
function [bin, msg_symb] = get_message(code_wrd, GF, G, k)
% GET_MESSAGE Gets the message and bits of a codeword that was
%generated using the systematic or generator method. If the
%codeword cannot be decoded it throws an error
%INPUTS:
% code_wrd - the GF2^m representation of a message or codeword
 polynomial
% GF -  the array of cells enumerating GF(2^m) symbols as elements in
 GF(2)[a].
% G - represents the generating polynomial for a
%     code design so that code_wrd can be decoded (into binary) if its
 a
%     received polynomial.
% k - optional parameter, if k exists the encoding is systematic
%OUTPUTS:
% bin - the binary representation of code_wrd in GF(2) or the binary
%       representation of the message that code_wrd represents in
 GF(2)
% msg - the decoded message polynomial

gen_enc = false;
if(~exist('k', 'var'))
    gen_enc = true;
end

m = size(GF{1},2); %symbol length

if(gen_enc)
    [msg_symb, rem] = PolyDivGF2(code_wrd, G, GF);
    if(rem(:) ~= -1)
        error("Could not get the original message polynomial from the
 codeword");
    end
else
    msg_symb = code_wrd(1, 1:k);
end

wrd_len = size(msg_symb, 2);
binary_len = wrd_len*m;

for i = 1:wrd_len
    start = (i-1)*m;
    ending = i*m;
    if(ending > binary_len)
        error("Binary index out of bounds.");
    end
    bin(1,1+start:ending) = GF{msg_symb(i)+2};
end



end
```

```
function do_test(test_name, R_err, GF, G, C, M, sys_gen)
%DO_TEST Completes a signular decoding test, prints out relevant
%information. May throw an error if decoding algorithm errors. Also
%determines the original message bits that were encoded
%INPUTS:
% test_name - a string containing the name of the test
% R_err     - The received codeword polynomial to decode
% t         - the number of errors the code should correct
% GF        - the array of cells enumerating GF(2^m) symbols as
 elements
%             in GF(2)[a].
% G         - The generating polynomial used for getting message bits
% C         - The codeword polynomial used for comparing to the
 corrected
%             codeword (C_hat(x)).
% M         - The original message polynomial
% sys_gen   - Boolean value representing the encoding method
 (generator or
%             systematic). true = generator method, false = systematic
 method


fprintf("\n  --------------%s--------------\n",test_name);

fprintf("   The codeword is received corresponding to R(x) shown
 below.\n");
fprintf("   The codeword R(X) is decoded using the Berlekamp-Massey,
 Chien, Forney process\n");
fprintf("   The orginal codeword is also shown below for comparison:
\n");
if(sys_gen)
    fprintf("   Polynomial forms:\n\t");
    print_poly("C(x)", C, false);
    fprintf("\t");
    print_poly("R(x)", R_err, false);
    fprintf("   Power forms:\n\t");
    print_poly("C[x]", C, true);
    fprintf("\t");
    print_poly("R(x)", R_err, true);
else
    fprintf("   Polynomial forms:\n\t");
    print_poly("C_sys(x)", C, false);
    fprintf("\t");
    print_poly("R(x)", R_err, false);
    fprintf("   Power forms:\n\t");
    print_poly("C_sys[x]", C, true);
    fprintf("\t");
    print_poly("    R(x)", R_err, true);
end

k = size(M,2);
n = size(C,2);
```

```
t = (n-k)/2;

try
    [C_hat, failed] = RS_Decoder(R_err, t, GF);
catch decode_error
    failed = true;
end

if(failed)
    error("%s Failed! The following message is produced:\n%s",
 test_name, decode_error.message);
else
    fprintf("   The received word was decoded!\n");
    fprintf("   Polynomial form:\n\t");
    if(sys_gen)
        print_poly("C_hat(x)", C_hat, false);
        fprintf("   Power form:\n\t");
        print_poly("C_hat[x]", C_hat, true);
        fprintf("   The decoded message is: \n\t");
        %below shouldn't fail because it was already vetted by
 BerlekampMasseyRS
        [msg_bin, msg] = get_message(C_hat, GF, G);
    else
        print_poly("C_sys_hat(x)", C_hat, false);
        fprintf("   Power form:\n\t");
        print_poly("C_sys_hat[x]", C_hat, true);
        fprintf("   The decoded message is: \n\t");
        %below shouldn't fail because it was already vetted by
 BerlekampMasseyRS
        [msg_bin, msg] = get_message(C_hat, GF, G, k);
    end

    print_poly("Message Bits",msg_bin, true);
    fprintf("\t");
    print_poly("M[x]", msg, true);
    fprintf("  Original Message:\n   ");
    print_poly("Orig_M[x]", M, true);
end

end
```

Published with MATLAB® R2018b

```
function print_poly(name, polynomial, pwr_form)
%{
PRINT_POLY: prints the given polynomial to the command window in the
 form
            requested
INPUTS:
    name - a string containing the name of the polynomial ex: "f(x)"
    polynomial - the power form representation of that polynomial
    pwr_form - a boolean variable that tells the function what verison
 to
               print. If true, power form representation printed. If
 false,
               polynomial form is printed
%}


poly_len = size(polynomial,2);

if(pwr_form)
    print_string = sprintf("%s = [", name);
else
    print_string = sprintf("%s = ", name);
end

for i = 1:poly_len
    curr_a = polynomial(i);
    %if power form
    if(curr_a == -1 && pwr_form)
        print_string = strcat(print_string, " Inf");
    elseif(pwr_form)
        curr_string = sprintf(" %d", curr_a);
        print_string = strcat(print_string, curr_string);
    %if polynomial form
    elseif(curr_a ~= -1)
        x_pwr = poly_len - i;
        if(curr_a == 1)
            a_str = "a";
        else
            a_str = sprintf("a^%d", curr_a);
        end
        if(x_pwr == 1)
            x_str = "x";
        else
            x_str = sprintf("x^%d", x_pwr);
        end

        if(curr_a ~= 0 && x_pwr ~= 0)
            curr_string = sprintf("%s*%s + ", a_str, x_str);
        elseif(x_pwr ~= 0)
            curr_string = sprintf("%s + ", x_str);
        elseif(curr_a ~= 0)
            curr_string = sprintf("%s + ", a_str);
```

```
        else
            curr_string = "1 + ";
        end
        print_string = strcat(print_string, curr_string);
    end
end

if(pwr_form)
    fprintf("%s ]\n", print_string);
else
    len = strlength(print_string);
    print_string = eraseBetween(print_string,len-2, len);
    fprintf("%s\n", print_string);
end


end
```

Published with MATLAB® R2018b

ENEE 624: Error Correcting Codes
Project 3

Brendan Cain 4/17/21

1 Introduction:

The purpose of this report is to describe the implementation of computations in $GF(2^m)$ in order to use them for arithmetic operations required to design and execute a Reed-Solomon Code. Computations are implemented using functions in MATLab code. Function logic and the various algorithms used are described in detail in *Section 2: Implementing Galois Field Arithmetic*. The manner in which these functions are tested is described in *Section 3: Testing Implementation of Galois Field Arithmetic*. All MATLab code is published at the end of the respective section in which it was mentioned. All information used in this report was derived from materials provided by Dr. EF Charles LaBerge in the Error Correcting Codes course (*ENEE 624*) at University of Maryland, Baltimore County and "*Error Correction Coding: Mathematical Methods and Algorithms*" by Todd K. Moon.

2 Implementing Galois Field Arithmetic:

This section describes the manner of implementation for functions that are used to do arithmetic operations in $GF(2^m)[x]$ and $GF(2^m)$. These functions will be used in following section and the inscription of function code is shown at the end of this section.

> ***Note:*** It is important to understand how rectangular form and power form are represented throughout these functions and the program. **Rectangular form** is the binary representation of a *polynomial* $\in GF(2)[\alpha]$, where α (*the Greek letter alpha*) is a root of the binary primitive polynomial $\in GF(2)[x]$. The primitive polynomial is used in the function below to generate rectangular form polynomials that map to elements of $GF(2^m)$. Rectangular form is represented in the MATLab code as a binary vector (sometimes called an array) and is used for *addition operations* (bitwise exclusive or) and *mapping sums* to power form using the cell array returned from the function below. **Power form** is slightly different, the elements are an *enumeration of symbols* in $GF(2^m)$ represented as *powers of alpha* (α^y) which are equivalent to the rectangular form representation (this is where the mapping comes in). Power form can be in vector format where it represents a polynomial $\in GF(2^m)[x]$ whose coefficients are symbol elements of $GF(2^m)$. y are the elements in the vector, residing at the index corresponding to the degree of x that α^y is a coefficient of (highest order on the left). Furthermore, the values of y range from (-1) to $(2^m - 2)$ with $-1 = inf$ because $\alpha^{inf} = 0 \in GF(2^m)$ and $\alpha^0 = 1 \in GF(2^m)$.

Figure 1

`function field = GenerateGF2(m, prim_poly, prnt_flag)`

This function is used to generate all the elements of $GF(2^m)$ while also creating a mapping between the *rectangular form* polynomials $\in GF(2)[\alpha]$ and the *power form* symbols of the elements $\in GF(2^m)$. The output is a 2^m by 1 array of cells called **`field`**; within each cell is an **`m`**-element vector representing a unique rectangular form polynomial whose coefficients are elements of $GF(2)$. The binary polynomials are generated using the primitive polynomial passed into the function as **`prim_poly`**. The primitive polynomial is a vector of **`m+1`**-elements that represent an **`m`**-order polynomial $\in GF(2)[x]$. The mapping between power and rectangular forms is done simply by enumerating the rectangular form polynomials

within the **field** array using the index. Individual elements of **field** are polynomials ∈ $GF(2)[\alpha]$ and the indices of the **field** correspond to the power of alpha symbol that the rectangular form vector is equivalent to. In other words, $\alpha^y = field(i)$ where $i = 1{:}\,2^m$ and $y = i + 2$ where y is some power of $\alpha \in GF(2^m)$. Finally, **print_flag** lets the user decide whether or not they wish to print the mapping between power and rectangular form to the console. **true** means that the mapping will be printed, **false** means that the mapping will not be printed. If left empty, the default is **false**.

function sum_out = AddGF2(a1, a2, GF)

This function is used to add two elements over $GF(2^m)$. **a1** and **a2** are power form representations of a single element in $GF(2^m)$. **GF** is the array of cells enumerating $GF(2^m)$ symbols as elements ∈ $GF(2)[\alpha]$ and is used for converting between rectangular form and power form. **sum_out** is the return value containing the power form representation of the sum $\alpha^s = \alpha^{y_1} + \alpha^{y_2} \in GF(2^m)$. The operation is completed by *exclusive or-ing* ($[\alpha^{y_1}]_r \oplus [\alpha^{y_2}]_r$) the rectangular forms of **a1** and **a2** $\in GF(2)[\alpha]$. That rectangular form polynomial is then matched to a power form value in $GF(2^m)$ using **GF** and assigned to **sum_out**.

function prod_out = MultGF2(m1, m2, GF)

The **MultGF2** function is used to multiply two elements over $GF(2^m)$. **m1** and **m2** are power form representations of a single element in $GF(2^m)$. **GF** is the array of cells enumerating $GF(2^m)$ symbols as elements ∈ $GF(2)[\alpha]$ and is used for converting between rectangular form and power form as well as determining (n, m) which is the codeword design. **prod_out** is the return value containing the power form representation of the product $\alpha^{prod} = \alpha^{y_1} * \alpha^{y_2} \in GF(2^m)$. The operation is completed by adding **m1** and **m2** then taking the **n-1** modulus of the sum, **prod_out** is assigned to this value. However, if either **m1** or **m2** is α^{inf} (**-1**) then **prod_out** is **-1**.

function quo = DivGF2(dvd, dvs, GF)

This function is used to divide two elements over $GF(2^m)$. **dvd** and **dvs** are power form representations of the dividend and divisor respectively and are single elements in $GF(2^m)$. **GF** is the array of cells enumerating $GF(2^m)$ symbols as elements ∈ $GF(2)[\alpha]$ and is used for converting between rectangular form and power form as well as determining (n, m) which is the codeword design. **qou** is the return value containing the power form representation of the quotient $\alpha^q = \alpha^{y_{dvd}} / \alpha^{y_{dvs}} \in GF(2^m)$. The operation is completed by taking the **n-1** modulus of the difference between **dvd** and **dvs**, then **quo** is assigned to this value. However, if either **dvd** or **dvs** is α^{inf} (**-1**) then **quo** is **-1**.

function output = EvalPolyGF2(poly, x, GF)

This function evaluates a power form polynomial (**poly**) whose coefficients are ∈ $GF(2^m)$ at a specified value of **x** using operations over $GF(2^m)$. **GF** is the array of cells enumerating $GF(2^m)$ symbols as elements ∈ $GF(2)[\alpha]$ and is used for converting between rectangular form and power form as well as passing to **AddGF2** and **MultGF2** functions when necessary. Remember that power form polynomials are represented as an array of powers of alpha coefficients with the coefficient corresponding to the highest degree of x on the left. Assume a polynomial of the form $Y(x) = \sum_{i=1}^{l} \alpha^{y}{}_i * x^{l-i}$ where l is the overall degree of the polynomial + 1 (*this is the length of the array*). The corresponding power form array would look like this $Y[i] = [\, y_1 \ \ldots \ y_i \ \ldots \ y]$ where i in both examples is the current index of the array. In order to evaluate $Y(x)$ at some $\alpha^x \in GF(2^m)$ simply replace the α^x value at every x yielding the equation $Y(\alpha^{in})$ where α^{in} is represented by **x**. Then evaluate the equation over $GF(2^m)$ using **AddGF2** and **MultGF2**. Computing $(\alpha^{in})^{l-i}$ is done by multiplying the current power of the x-term in

$Y(x)$ by the input argument α^{in} (**x**) and using that value to take the **n-1** modulus. The **output** is the evaluation of the polynomial at **x** representing a single element in $GF(2^m)$.

function cur_prod = PolyMultGF2(A, B, GF)

This function does polynomial multiplication over $GF(2^m)[x]$ for **A** and **B** which are inputs to be multiplied represented as power form polynomials ∈ $GF(2^m)[x]$. **GF** is the array of cells enumerating $GF(2^m)$ symbols as elements ∈ $GF(2)[\alpha]$ and is used for converting between rectangular form and power form as well as passing to **AddGF2** and **MultGF2** functions when necessary. The return value is a power form polynomial **cur_prod** ∈ $GF(2^m)[x]$ whose degree is the sum of degrees of the input polynomials. The output polynomial is computed by element-based multiplication in $GF(2^m)$ where every term of **A** is multiplied by every term of **B**. Then the resulting polynomial is found by summing like terms, this polynomial is stored and returned in **cur_prod**.

function [quo, rem] = PolyDivGF2(dvd, dvs, GF)

This function does polynomial division over $GF(2^m)[x]$ for **dvd** and **dvs** which are inputted as power form polynomials representing the dividend and divisor respectively ∈ $GF(2^m)[x]$. **GF** is the array of cells enumerating $GF(2^m)$ symbols as elements ∈ $GF(2)[\alpha]$ and is used for converting between rectangular form and power form as well as passing to **AddGF2**, **DivGF2**, and **PolyMultGF2** functions when necessary. **quo** and **rem** are the output power form polynomials representing the quotient and remainder polynomials respectively ∈ $GF(2^m)[x]$. This function first determines the degree of the quotient by finding the difference between the degree of the dividend and the degree of the divisor, that value is saved as **deg_quo**. If the degree of the divisor (**deg_dvs**) is greater than the degree of the dividend (**deg_dvd**), then the divisor polynomial must be of higher order than the dividend polynomial and thus polynomial division is not possible. In this case, **rem** is set **dvd** to and **quo** is set to **-1** (meaning quo is $\alpha^{inf} = 0$), then the values are returned. If polynomial division is possible, the algorithm initializes **quo** to a vector of size **deg_quo+1**, then iterates over elements of **quo** via a **for** loop while assigning the computed coefficients to its elements. Simultaneously, **dvd** is updated at every iteration based off the current computed **quo** coefficient, its corresponding x-term, and **dvs**. Coefficients of **quo** are computed over $GF(2^m)$ (*using the **DivGF2** function*) starting from the coefficient corresponding to the highest order x ($i = 1$, because of power form). **quo(i)** is returned from **DivGF2** with the coefficient corresponding to the current highest power of **dvd** (***dvd(i)***) as the dividend argument and the first coefficient of **dvs** (***dvs(1)***) as the divisor argument. Then a polynomial consisting of **quo(i)** and its corresponding x-term is constructed in power form and multiplied by **dvs** over $GF(2^m)[x]$ using the **PolyMultGF2** function. This polynomial is added to the current **dvd** polynomial over $GF(2^m)[x]$ using element-by-element polynomial addition over $GF(2^m)$ (*utilizing the **AddGF2** function*). Computing and updating the dividend polynomial causes the previous dividend coefficient at **dvd(i)** to resolve to $\alpha^{inf} = 0$. Thus, decreasing the degree of **dvd** and possibly causing lower order coefficients of **dvd** to also change. This is due to the use of **quo(i)** and the nature of polynomial multiplication and addition over $GF(2^m)[x]$. Once the power form array of the **quo** polynomial is finished being populated, the resulting **dvd** polynomial is utilized to set the **rem** polynomial (with leading α^{inf} coefficients truncated). If the dividend polynomial is evenly divisible by the divisor polynomial, then the resulting **dvd** polynomial contains only **-1** 's and **rem** is set to equal **-1** or α^{inf}. Finally, **quo** and **rem** are returned to the calling program.

```
function field = GenerateGF2(m, prim_poly, prnt_flag)
%{
GENERATEGF2: Used to generate enumeration of elements within GF(2^m)
 using an irriducible primitive polynomial of GF(2)[x].
INPUTS:
    m - the extension field parameter -> GF(2^m)
    prim_poly - The power form irriducible primitive polynomial of
 GF(2)[x]
    prnt_flag - optional parameter used for printing the enumerated
 field
OUTPUT:
    field - an array of cells with dimensions (2^m,1) whose elements
 are
            the rectangular form representation of polynomial elements
 in
            GF(2^m). The indices of the array correspond to the power of
            alpha (one of the 2^m symbols)that the polynomial element is
            equivalent to in GF(2^m)[x]. For an index (i) in the field
            array, i = e+2 where, e is some exponent of alpha (a^e) that
            exists in GF(2^m)[x]. a^inf symbol is represented as e=-1
 and
            is saved at field{1} (i=1) as [0 0 0 0].
%}

%handle optional parameter
if ~exist('prnt_flag','var')
    prnt_flag = false;
end

p = 2;
n = p^m - 1; %codeword length
syms a;
%create enumeration array of 2^m cells containing [0 0 0 0]
field = mat2cell(zeros(n+1, m), ones(1,n+1), m);
%do algebra, set highest degree term equal to the rest of the
 prim_poly
a_exp_m = prim_poly(2:end);
%create a polynomial GF(2)[x] polynomial that can increase the degree
 of the
%GF(2^m)[x] symbol by 1 when multiplied, ex: a^n * a^1 = a^(n+1)
a_1 = zeros(1, m);
a_1(end-1) = 1;

%generate equivalency matrix where cell index = power + 2 and -1 ==
 inf
for pow = -1:n-1
    if(pow == -1) %set a^inf mapping
        field{pow+2} = zeros(1, m);
    elseif(pow < m) %set mapping for symbols that are already
 represented in a_exp_m
        z = zeros(1, m);
        z(pow+1) = 1;
```

```
        field{pow+2} = flip(z);
    elseif(pow == m) %enumeration for a^m is the rest of prim_poly or
 a_exp_m
        field{pow+2} = a_exp_m;
    else %otherwise the symbol doesn't exist in GF(2)[x] and so must
 be put in terms of GF(2)[x] elements
        %multiply last element by a^1
        curr_cell = conv(field{pow+1},a_1);
        curr_cell = mod(curr_cell, 2); %be sure to mod
        if(curr_cell(end-m) == 1) %if the multiplication created an
 a^m symbol,
            %substitute the enumeration for a^m which should cancel
 some
            %terms through GF(2)[x] addition
            curr_cell = xor(curr_cell(1, end-(m-1):end), a_exp_m);
            field{pow+2} = double(curr_cell);
        else
            %otherwise, set the symbol = to the new polynomial of
 correct
            %size
            field{pow+2} = curr_cell(1, end-(m-1):end);
        end
    end

    %print out the equivalencies in power form
    if(prnt_flag)
        if(pow ~= -1)
            fprintf("a^%d = %s\n", pow, poly2sym(field{pow+2},a));
        else
            fprintf("a^inf = %s\n", poly2sym(field{pow+2},a));
        end
    end
end

end
```

Published with MATLAB® R2018b

```
function sum_out = AddGF2(a1, a2, GF)
%   -a1 and a2 are power form GF(2^m) numbers that will be added
 together via
%    xor
%   -GF is the enumeration of the specifc field
%   -sum_out is the power form of the determined sum

m = size(GF{1},2);
n = 2^m;
if(a1 < -1 || a1 > n-2)
    error("a1 = %d is not a valid power of alpha in GF(2^%d)\n", a1,
 m);
    return
end
if(a2 < -1 || a2 > n-2)
    error("a2 = %d is not a valid power of alpha in GF(2^%d)\n", a2,
 m);
    return
end

add1 = GF{a1+2};
add2 = GF{a2+2};

sum_d = double(xor(add1, add2)); %does sum computation

%finds the equivalent in the GF enumeration
sum_out = cellfun(@(x)isequal(x, sum_d),GF, 'un', 0);
sum_out = find([sum_out{:}] == 1) - 2; %subtracts two to get exp of
 alpha


end
```

Published with MATLAB® R2018b

```
function prod_out= MultGF2(m1, m2, GF)
%MULTGF2:
%   -m1 and m2 are the input parameters that are to be multiplied,
 they are
%   in power form and consist of a single element of GF(2^m)
%   -GF is the enumeration of the GF

m = size(GF{1},2);
n = 2^m;
if(m1 < -1 || m1 > n-2)
    error("a1 = %d is not a valid power of alpha in GF(2^%d)\n", a1,
 m);
    return
end
if(m2 < -1 || m2 > n-2)
    error("a2 = %d is not a valid power of alpha in GF(2^%d)\n", a2,
 m);
    return
end

if(m1 == -1 || m2 == -1)
    prod_out = -1;
else
    prod_out = mod(m1 + m2, n-1);
end

end
```

Published with MATLAB® R2018b

```
function quo = DivGF2(dvd, dvs, GF)
%DIVGF2:
%   -dvd is the divided, dvs is the divisor, the input parameters that
 are to be divided, they are
%   in power form and consist of a single element of GF(2^m)
%   -GF is the enumeration of the GF
%   -quo is the quotient output

m = size(GF{1},2);
n = 2^m;
if(dvd < -1 || dvd > n-2)
    error("dividened = %d is not a valid power of alpha in GF(2^
%d)\n", dvd, m);
    return
end
if(dvs < -1 || dvs > n-2)
    error("divisor = %d is not a valid power of alpha in GF(2^%d)\n",
 dvs, m);
    return
end

if(dvd == -1 || dvs == -1)
    quo = -1;
else
    quo = mod((dvd - dvs) + (n-1),n-1);
end

end
```

Published with MATLAB® R2018b

```
function output = EvalPolyGF2(poly, x, GF)
%EVALPOLYGF2
%   -poly is a power form array polynomial and x is the power form
 input to
%    that polynomial
%   -GF is the enumeration of the given GF
%   -output is the power form output of the function given the input x
%   -Evaluates the polynomial at x using MultGF2 and AddGF2 functions

m = size(GF{1},2);
n = 2^m;
deg = size(poly,2) - 1;
for i=1:deg
    if(x == -1 || poly(1,i) == -1)
        poly(1,i) = -1;
    else
        temp = mod((deg+1-i)*x,(n-1));
        poly(1,i) = MultGF2(temp, poly(1,i),GF);
    end
end
output = poly(1,1);

for i=2:deg+1
    output = AddGF2(output,poly(1,i),GF);
end


end
```

Published with MATLAB® R2018b

```
function cur_prod = PolyMultGF2(A,B, GF)
%   -prod is the return value that gives the power form array of the
 product of
%    A and B. The elements of A and B are the powers of the exponents
 of
%    alpha coefficients (-1 == inf). Their placement represents the
 degree
%    of the xvalue (highest degree on left)
%   -A and B are power form matrix parameters that will be multiplied
 together
%   -GF (nx1 cell) is the power and rectangular form enumeration of
 the given
%   field, the indecies of the cell matrix represent power or pow = i
 - 2, (-1 == inf)
%   -n is the p^m big Galois Field
%Created By Brendan Cain
%Error Correcting Codes HW 7
m = size(GF{1},2);
n = 2^m;

a_len = size(A,2);
b_len = size(B,2);
cur_prod = zeros(1, a_len+b_len-1); %allocate space for return value
cur_prod(:) = -1; %put in terminator value (lets algorithm know this
 space is untouched)
for i = 1:a_len
    for j = 1:b_len
        if(A(1,i) ~= -1 && B(1,j) ~= -1) %only do math if not inf (inf
 times anything is inf)
            pow = i + j - 1; %gives the current place in the product
 array
            val = cur_prod(1, pow); %previous product that has same
 degree
            exp = mod((A(1,i) + B(1,j)),n-1); %current product of
 element multiplication
            if(val == -1) %check to see if touched
                cur_prod(1, pow) = exp;
            else
                %do rectagular form addition (match qith equiv cell)
                new_val = double(xor(GF{val+2}, GF{exp+2}));
                %gets index that the specific value is at
                exp = cellfun(@(x)isequal(x, new_val),GF, 'un', 0);
                exp = find([exp{:}] == 1) - 2; %subtracts two to get
 exp of alpha
                cur_prod(1, pow) = exp;
            end
        end
    end
end

end
```

```
function [quo, rem] = PolyDivGF2(dvd, dvs, GF)
%POLYDIVGF2 does polynomial division in GF(2^m)
%   -dvd is a power form vector that represents the dividened
 polynomial
%   -dvs is a power form vector that represents the divisor polynomial
%   -GF is a list of cells containing the enumeration of the GF(2^m)
%   -quo is a return value that also is a power form vector that
 represents
%    the quotient polynomial
%   -rem is a return valuse that also is a power form vector that
%    represents the remainder polynomial

m = size(GF{1},2);
n = 2^m;
%degree of dividened and divisor
deg_dvd = size(dvd, 2) - 1;
deg_dvs = size(dvs, 2) - 1;
deg_quo = deg_dvd-deg_dvs;
%error handling
if(deg_dvd < 1)
    error("Degree of dividened too small, function is for polynomial
 division in GF(2^%d)\n\", m);
end

quo = zeros(1, deg_quo+1);
quo(:) = -1;

if(deg_dvd >= deg_dvs) %must have larger degree than divisor
    %loop through quotient array and set values
    for i = 1:deg_quo+1
        %loop through dividend and determine coef of quotient at the
        %current power
        quo(i) = DivGF2(dvd(i),dvs(1), GF);
        %determine the size of the power used to modify dividend and
 create
        %a polynomial containing the current quotient term (coeff and
 x
        %power)
        pow = (deg_quo+1) - i;
        temp = zeros(1, pow+1);
        temp(:) = -1;
        temp(1) = quo(i);
        %multiply that polynomial by the divisor and add padding in
 order
        %to modify dividend
        mult = PolyMultGF2(dvs, temp, GF);
        %resize mult/pad beginning with -1
        temp = zeros(1, deg_dvd+1);
        temp(:) = -1;
        temp(1,end-size(mult,2)+1:end) = mult;
        mult = temp;
```

```
        %subtract mult to dvd to modify dividend(adding is subtracting
 in GF(2^m)
        for j = 1:deg_dvd+1
            dvd(j) = AddGF2(dvd(j), mult(j), GF);
        end
        %loop until all quotient terms are filled
    end
    %setting remainder
    if(dvd(:) == -1) %if all coef of the dividend are now a^inf, there
 is no remainder
        rem = -1;
    else
        %otherwise the remainder is whatever is left in the dvd
        idx = find(dvd ~= -1);
        idx = idx(1);
        rem = dvd(1, idx:end);
    end
%set output when the divisor is bigger than the dividened
else
    rem = dvd;
    quo = -1;
end


end
```

Published with MATLAB® R2018b

3 Testing Implementation of Galois Field Arithmetic:

The functions from the above section were used to complete Homework 7 – Problem 2 and Homework 8 – Problem 1 for the purpose of testing the implementation of $GF(2^m)$ and $GF(2^m)[x]$ arithmetic.

3.1 Helper Function Descriptions:

Two helper functions are used along with the Galois Field Arithmetic functions: **`print_poly()`** and **`get_syndromes()`**. Their descriptions as well as their code inscriptions are shown below.

`function print_poly(name, polynomial, pwr_form)`

This function is responsible for printing a given power form polynomial (**`polynomial`**) to the console in the form specified by the Boolean variable **`pwr_form`**. The string argument, **`name`**, contains the name of the polynomial as function of x that the user wishes to denote the polynomial as in the print statement (*example:* **`name="Y(x)"`**.) If **`pwr_form`** is **`true`**, then the polynomial is printed as power form looking like this: $Y(x) = [\, y_1 \ \dots \ y_i \ \dots \ y_l \,]$ where y_i is an element of the **`polynomial`** vector. When $y_i = -1$, the string **`"inf"`** is printed. If **`pwr_form`** is **`false`**, then the polynomial is printed holding the form of a normal $GF(2^m)[x]$ polynomial: $Y(x) = \alpha^{y_1} * x^{l-1} + \cdots + \alpha^{y_i} * x^{l-i} + \cdots + \alpha^{y_{l-1}} * x + \alpha^{y_l}$ where $i \in \{1{:}\, l\}$, $\alpha^{y_i} \in GF(2^m)$, and $l =$ the length of **`polynomial`**. When $y_i = -1$ the term is skipped; when $y_i = 0$ the **`"x"`** term is only printed because $\alpha^0 * x^{l-1} = 1 * x^{l-1} = x^{l-1}$; when $y_i = 1$ the coefficient printed is only **`"a"`** because $\alpha^1 = \alpha$. Both HW 7.2 and HW 8.1 utilize this function.

`function syndromes = get_syndromes(t, R, GF)`

This function computes the syndrome polynomial $S(x)$ returned as the power form polynomial **`syndromes`**. It is computed from the power form representation of the received codeword polynomial $R(x)$ passed into the function as **`R`**. The integer value **`t`** represents the max number of errors that the code is designed to detect and correct. **`GF`** is the array of cells returned by the **`GenerateGF2`** function enumerating $GF(2^m)$ symbols as elements $\in GF(2)[\alpha]$ and is used for mapping between rectangular form and power form as well as passing to the **`EvalPolyGF2`** function. Since this function is used for determining the syndromes for a Reed-Solomon code, the algorithm for computing each syndrome coefficient is done by evaluating the received polynomial $R(x)$ at $2t$ nonzero consecutive powers of α using **`EvalPolyGF2`**. The resulting output for each evaluation should be a single symbol in $GF(2^m)$. The placement of the $2t$ symbols in the **`syndromes`** power form polynomial array corresponds to which power of α that $R(x)$ was evaluated at. For example, if $R(x)$ was evaluated at α^2 yielding $R(\alpha^2) = \alpha^y$ for some power y, then the x^2 term of $S(x)$ would appear as $\alpha^y * x^2$. After the **`syndromes`** array is populated with appropriate syndrome coefficients it is returned to the calling program. Note: if there are no errors found in the received polynomial, all syndromes will evaluate to $\alpha^{inf} = 0$. Also, the x^0 coefficient will always be $\alpha^{inf} = 0$ because the $2t$ consecutive powers of α that are used to evaluate syndromes must all be nonzero. Only HW 8.1 utilizes this function.

```
function print_poly(name, polynomial, pwr_form)
%{
PRINT_POLY: prints the given polynomial to the command window in the
 form
            requested
INPUTS:
    name - a string containing the name of the polynomial ex: "f(x)"
    polynomial - the power form representation of that polynomial
    pwr_form - a boolean variable that tells the function what verison
 to
               print. If true, power form representation printed. If
 false,
               polynomial form is printed
%}


poly_len = size(polynomial,2);

if(pwr_form)
    print_string = sprintf("%s = [", name);
else
    print_string = sprintf("%s = ", name);
end

for i = 1:poly_len
    curr_a = polynomial(i);
    %if power form
    if(curr_a == -1 && pwr_form)
        print_string = strcat(print_string, " Inf");
    elseif(pwr_form)
        curr_string = sprintf(" %d", curr_a);
        print_string = strcat(print_string, curr_string);
    %if polynomial form
    elseif(curr_a ~= -1)
        x_pwr = poly_len - i;
        if(curr_a == 1)
            a_str = "a";
        else
            a_str = sprintf("a^%d", curr_a);
        end
        if(x_pwr == 1)
            x_str = "x";
        else
            x_str = sprintf("x^%d", x_pwr);
        end

        if(curr_a ~= 0 && x_pwr ~= 0)
            curr_string = sprintf("%s*%s + ", a_str, x_str);
        elseif(x_pwr ~= 0)
            curr_string = sprintf("%s + ", x_str);
        elseif(curr_a ~= 0)
            curr_string = sprintf("%s + ", a_str);
```

```
        else
            curr_string = "1 + ";
        end
        print_string = strcat(print_string, curr_string);
    end
end

if(pwr_form)
    fprintf("%s ]\n", print_string);
else
    len = strlength(print_string);
    print_string = eraseBetween(print_string,len-2, len);
    fprintf("%s\n", print_string);
end


end
```

Published with MATLAB® R2018b

3.2 Testing Galois Field Functions Using Homework Problems:

The prompts for Homework 7 – Problem 2 and Homework 8 – Problem 1 are shown in Figure 2 below. They are used to test the functions that implement Galois Field arithmetic. The descriptions of the outputs for each of the main programs (*proj3_7_2.m* and *proj3_8_1.m*) are also shown below along with code inscriptions at the end of this section.

HW07, Problem 2. (EFCL) Let α be a root of the primitive binary polynomial x^4+x^3+1. Design a narrow sense, three error correcting R-S code over $GF(2^4)$ using this primitive polynomial. Provide the generating polynomial, $g(x)$. What is n for this code? What is k? How many bits are in the input message? Encode the message $g(x)=\begin{bmatrix} \alpha & 0 & 0 & 0 & \alpha^5 & 0 & 0 & \alpha^{14} & 1 \end{bmatrix}=\alpha x^8+\alpha^5 x^4+\alpha^{14}x+1$ using the generator polynomial method, and provide the code polynomial, with highest order first. Encode the systematic version of this input and provide code polynomial with highest order first.

HW08, Problem 1: A narrow sense (7,3) Reed-Solomon code has a generator polynomial $g(x)=x^4+\alpha^3x^3+x^2+\alpha x+\alpha^3$. Encode the message $x^2+\alpha^4x+\alpha^2$. Using the resultant code polynomial, evaluate the appropriate syndromes and verify that they are all zero.

Figure 2

Problem 7.2 (*proj3_7_2.m*):

1. For this problem, the required design parameters are p, m, b, t, n, and k because the prompt above specifies that the code design is must be a Reed-Solomon code. Using the prompt above, the values of the design parameters are as follows: $p = 2$ is given by the base field $GF(p) = GF(2)$; $m = 4$ is given by the power of p shown by the secondary field $GF(2^m) = GF(2^4)$ (*this is also the desired symbol length or the number of bits representing any given symbol as a polynomial* $\in GF(2)[\alpha]$); $b = 1$ due to the code being narrow sense; $t = 3$ because it is specified that the code must be able to handle correcting at most three errors; n (*the desired amount symbols in the codeword*) can be computed as $n = p^m - 1 = 2^4 - 1 = 15$; finally, k (*the desired amount symbols in the message*) can be computed as $k = n - 2t = 15 - 2(3) = 9$. From this (*and shown in the output*) we can deduce that the code we are designing is a three-error correcting, $(15,9)$ Reed-Solomon code over $GF(2^4)[x]$. Also, there are 36 bits in an input message made up of nine (4-bit) symbols.

2. Secondly, the primitive polynomial $p(x) = x^4 + x^3 + 1 \in GF(2)[x]$ is used to generate the enumeration of $GF(2^4)$ using the **`GenerateGF2`** function. The mapping generated is shown in Figure 3 below.

Power form $GF(2^4)$ symbol	Polynomial $\in GF(2)[\alpha]$	Rectangular form binary vector
α^{inf}	0	$[0\ 0\ 0\ 0]$
α^0	1	$[0\ 0\ 0\ 1]$
α^1	α	$[0\ 0\ 1\ 0]$
α^2	α^2	$[0\ 1\ 0\ 0]$
α^3	α^3	$[1\ 0\ 0\ 0]$
α^4	α^3+1	$[1\ 0\ 0\ 1]$
α^5	$\alpha^3+\alpha+1$	$[1\ 0\ 1\ 1]$
α^6	$\alpha^3+\alpha^2+\alpha+1$	$[1\ 1\ 1\ 1]$
α^7	$\alpha^2+\alpha+1$	$[0\ 1\ 1\ 1]$
α^8	$\alpha^3+\alpha^2+\alpha$	$[1\ 1\ 1\ 0]$
α^9	α^2+1	$[0\ 1\ 0\ 1]$
α^{10}	$\alpha^3+\alpha$	$[1\ 0\ 1\ 0]$
α^{11}	$\alpha^3+\alpha^2+1$	$[1\ 1\ 0\ 1]$
α^{12}	$\alpha+1$	$[0\ 0\ 1\ 1]$
α^{13}	$\alpha^2+\alpha$	$[0\ 1\ 1\ 0]$
α^{14}	$\alpha^3+\alpha^2$	$[1\ 1\ 0\ 0]$

Figure 3

3. Then the generator polynomial $G(x)$ is created by polynomial multiplication over $GF(2^4)[x]$ by evaluating the product of $n-k$ polynomials being of the form $Y_i(x)=x+i$ where $i=\{1{:}\,n-k\}$. In this problem $n-k=6$ and so $G(x)=\prod_{i=1}^{6}(x+i)$. Then, using the **`PolyMultGF2`** function, the generator polynomial is evaluated to be $G(x)=x^6+\alpha^{12}x^6+x^4+\alpha^2x^3+\alpha^7x^2+\alpha^{11}x+\alpha^6\in GF(2^4)[x]$.

4. Given the message polynomial (shown as $g(x)$ in the prompt but in order to be in accordance with the code it will be called $M(x)$ here) $M(x)=\alpha x^8+\alpha^5x^4+\alpha^{14}x+1\in GF(2^4)[x]$, we can now encode the input symbols using the generator method and systematically. For reference, $M(x)$ in power form representation: $M(x)=[1\ \ Inf\ \ Inf\ \ Inf\ \ 5\ \ Inf\ \ Inf\ \ 14\ \ 0]$

   Generator Encoding Method:

   a. In order to encode the message polynomial using the generator method, the codeword polynomial $C(x)$ must be computed using the **`PolyMultGF2`** function. For generator method encoding, computing $C(x)$ is done by evaluating the following equation: $C(x)=G(x)*M(x)$ using polynomial multiplication over $GF(2^4)[x]$. Calling **`PolyMultGF2`** with $G(x)$ and $M(x)$ as the **`A`** and **`B`** parameters yields a polynomial of degree 14. For the purpose of being concise, the computed codeword will be represented here in power form format instead of the formerly used polynomial format (shown below):
   $C(x)=[1\ \ 13\ \ 1\ \ 3\ \ 9\ \ 7\ \ 14\ \ 5\ \ 6\ \ 11\ \ 8\ \ 5\ \ 11\ \ 13\ \ 6]$

   Systematic Encoding Method:

   b. In order to encode the message polynomial using the systematic method, the codeword polynomial $C_{sys}(x)$ must be computed using the **`PolyMultGF2`**, **`PolyDivGF2`**, and **`AddGF2`** functions. First, $M(x)$ is shifted to the left by $6=n-k$ terms using polynomial multiplication over $GF(2^4)[x]$. Evaluating $M(x)*x^6$ using **`PolyMultGF2`** yields: $M(x)*x^6=[1\ \ inf\ \ inf\ \ inf\ \ 5\ \ inf\ \ inf\ \ 14\ \ 0\ \ inf\ \ inf\ \ inf\ \ inf\ \ inf\ \ inf]$. Then, the

remainder from the polynomial division of $\frac{M(x)*x^6}{G(x)}$ over $GF(2^4)[x]$ is found using the **PolyDivGF2** function. The remainder (denoted as $R[dividend]_{divisor}$) yields the following polynomial in power form $R[M(x)*x^6]_{G(x)} = [2\ 7\ 13\ 5\ 3\ 13]$. This polynomial is appended to the trailing 6 terms of the $M(x)*x^6$ polynomial (This is why $M(x)$ was shifted to the left $6 = n-k$ terms) using element-by-element addition over $GF(2^4)$ using **AddGF2** in order to implement polynomial addition over $GF(2^4)[x]$. After $R[M(x)*x^6]_{G(x)}$ is appended to the $M(x)*x^6$ polynomial the resulting polynomial is $C_{sys}(x) \in GF(2^4)[x]$. In power form: $C_{sys}(x) = [1\ Inf\ Inf\ Inf\ 5\ Inf\ Inf\ 14\ 0\ 2\ 7\ 13\ 5\ 3\ 13]$. Notice how both the $M(x)$ and the $R[M(x)*x^6]_{G(x)}$ polynomials are fully visible in the codeword.

Problem 8.1 (*proj3_8_1.m*):

1. For this problem in HW 8, the primitive polynomial $p(x)$ is provided as well as the addition and multiplication tables over $GF(2^3)$. However, this information is not displayed in the prompt at the start of this section and so it is shown below in Figure 5. Furthermore, even though the arithmetic operation tables are provided, the array of cells, returned by the **GenerateGF2** function that enumerates all $GF(2^3)$ symbols as elements $\in GF(2)[\alpha]$, is still required in order to utilize the functions from Section 2 that implement $GF(2^3)$ arithmetic. The primitive polynomial $p(x) = x^3 + x + 1$ along with $m = 3$ (given by the power of 2 shown by the secondary field $GF(2^m) = GF(2^3)$) are passed into the **GenerateGF2** function in order to generate the enumeration array. The mapping of the returned enumeration array is also shown below in Figure 4.

Power form $GF(2^3)$ symbol	Polynomial $\in GF(2)[\alpha]$	Rectangular form binary vector
α^{inf}	0	$[0\ 0\ 0\ 0]$
α^0	1	$[0\ 0\ 0\ 1]$
α^1	α	$[0\ 0\ 1\ 0]$
α^2	α^2	$[0\ 1\ 0\ 0]$
α^3	$\alpha+1$	$[1\ 0\ 0\ 0]$
α^4	$\alpha^2+\alpha$	$[1\ 0\ 0\ 1]$
α^5	$\alpha^2+\alpha+1$	$[1\ 0\ 1\ 1]$
α^6	α^2+1	$[1\ 1\ 1\ 1]$

Figure 4

Let $p(x) = x^3 + x + 1$ be the primitive polynomial for an implementation of $GF(2^3)$.

Addition Table for GF(2^3)

+	Inf	0	1	2	3	4	5	6
Inf	Inf	0	1	2	3	4	5	6
0	0	Inf	3	6	1	5	4	2
1	1	3	Inf	4	0	2	6	5
2	2	6	4	Inf	5	1	3	0
3	3	1	0	5	Inf	6	2	4
4	4	5	2	1	6	Inf	0	3
5	5	4	6	3	2	0	Inf	1
6	6	2	5	0	4	3	1	Inf

Multiplication Table for GF(2^3)

x	Inf	0	1	2	3	4	5	6
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
0	Inf	0	1	2	3	4	5	6
1	Inf	1	2	3	4	5	6	0
2	Inf	2	3	4	5	6	0	1
3	Inf	3	4	5	6	0	1	2
4	Inf	4	5	6	0	1	2	3
5	Inf	5	6	0	1	2	3	4
6	Inf	6	0	1	2	3	4	5

Figure 5

2. Given the generator polynomial $G(x) = x^4 + \alpha^4 x^3 + x^2 + \alpha x + \alpha^3$ and the message polynomial $M(x) = x^2 + \alpha^4 x + \alpha^2$ (both shown in Figure 2 at the beginning of this section), the codeword polynomial can be resolved using the generator method (described in problem 7.2) yielding $C(x) = x^6 + \alpha^6 x^5 + \alpha^2 x^4 + \alpha^3 x^3 + \alpha x + \alpha^5 \in GF(2^3)[x]$.

3. Using the resultant $C(x)$ polynomial, treat it like a received code word and compute the syndrome polynomial $S(x)$ using the **`get_syndromes`** function. Additionally, compute the value for **`t`** given by $k = n - 2t$. Since n and k are given in Figure 2 at the start of the section as $(7,3)$ respectively, the value for **`t`** is found to be $t = 2$. That value is passed into the **`get_syndromes`** function along with the power form of $C(x) \in GF(2^3)[x]$. The polynomial returned from the function has a degree of at most $2t$ because that is the max amount of possible syndrome coefficients for the $(7,3)$ Reed-Solomon code $\in GF(2^3)$. The returned syndrome polynomial is stored in the $S(x)$ power form polynomial vector.

4. From the output, the syndrome polynomial constructed is $S(x) = [Inf \;\; Inf \;\; Inf \;\; Inf \;\; Inf]$. The MATLab code (inscribed below) looks through every element in the power form polynomial vector and prints whether or not there were any errors detected in the received codeword. Since the original codeword $C(x)$ was artificially used as the received codeword, the no-error syndrome polynomial is to be expected. As $S(x)$ reflects, all coefficients of the syndrome polynomial should be $\alpha^{inf} = 0$ in order for the received codeword to be error-free. Otherwise, the between 1 and t symbols of the received codeword polynomial errored.

```
%Brendan Cain
%project 3 part 2

%Below is the completion of hw 7.2 using the functions completed
%for part 1 of the project

%HW 7
%number 2
fprintf("-----------------HW 7.2-----------------\n");
syms x;
prim = x^4 + x^3 + 1;
p = 2;
m = 4;
t = 3; %three error correcting code
n = p^m - 1; %codeword length
k = n - 2*t; %message length
prim_poly = sym2poly(prim); %array representation of prim variable
fprintf("Design a narrow sense, three error correcting R-S code over
 GF(2^4)\n");
fprintf("using the primitive polynomial: ");
print_poly("p(x)", prim_poly, true);
%generate equivalency matrix where cell index = power - 2 and -1 ==
 inf
fprintf("GF(2^4)[x] requires an equivalency matrix for every element
 in the field\n");
fprintf("That mapping is generated and shown below:\n");
equiv_cell = GenerateGF2(m, prim_poly, true);

fprintf("\nBoth methods for generating a codeword require the
 generating polynomial g(x).\n");
fprintf("In order to create this polynomial, factors of g must be
 multiplied together in power form\n");
fprintf("(in GF(2^4)[x]), the factors to be multiplied are x^i
 elements with i=[1:n-k].\n");
fprintf("Since n-k = 6, g(x) = x^1 * x^2 * x^3 * x^4 * x^5 * x^6\n");
%creating the generating polynomial using the second method
%the factors of G to be multiplied together in power form
%the placements of elements are powers of x, the value of the elements
 are
%the powers of alpha that are the x's coefficients
G = {[0 1], [0 2], [0 3], [0 4], [0 5], [0 6]};

while(size(G,2) > 1)
    G = {PolyMultGF2(G{1}, G{2}, equiv_cell), G{3:end}}; %appends G
 matrix
end

G = [G{:}]; %G is now the minimal generating polynomial
fprintf("\nUsing the equivalency matrix and polynomial multiplication
 in GF(2^4)[x]\n");
fprintf("the generating polynomial g(x) for the 3 error correcting
 code is shown below:\n");
```

```
print_poly("g(x)", G, true);%G is now the generating polynomial for
 the RS code (Made using the 2nd method)

fprintf("\n------Generator Polynomial Method------\n");

fprintf("1.) This is a %d-error correcting, (%d,%d) R-S code over
 GF(%d^%d).\n", t, n,k,p,m);
fprintf("There are %d bits in an input message made up of %d (%d-bit)
 messages\n",k*m, k,m);
M = [1 -1 -1 -1 5 -1 -1 14 0]; %message polynomial

%print M
fprintf("The message polynomial to be encoded is shown below:\n");
print_poly("m(x)", M, true);
%compute encoded codeword
fprintf("2.) the generating polynomial is multiplied by the message
 polynomial\n");
fprintf("to get the codeword shown below: \n");
C = PolyMultGF2(G, M, equiv_cell);
print_poly("c(x)", C, true);

fprintf("\n------Systematic Encoding Method------\n");
disp("Below is the encoding of the systematic version of this code:
\n");
disp("1.) First we shift the message polynomial over x^n-k bits");
fprintf("In this case, n-k=%d so the shifted codeword looks like
 this...\n", n-k);
shift = [0 -1 -1 -1 -1 -1 -1]; %this equals x^6
shifted = PolyMultGF2(M, shift, equiv_cell); %this is the shifted
 value after the computation

print_poly("m(x)*x^6", shifted, true); %display the shifted message
 polynomial

disp("2.) the remainder of the polynomial division of (m(x)*x^6)/g(x)
 is found as...");
[~, rem] = PolyDivGF2(shifted, G, equiv_cell);

print_poly("r(x)", rem, true);

disp("3.) r(x) is appended to the end of the shifted message
 polynomial, giving the");
disp("systematic encoding of the codeword (shown below).");

sys_code = shifted;
sys_code(1,end-(n-k)+1:end) = rem;

print_poly("c_sys(x)", sys_code, true);

disp("Notice how the message polynomial is fully visible as the
 highest order symbols in the systematic word");

-----------------HW 7.2-----------------
Design a narrow sense, three error correcting R-S code over GF(2^4)
```

using the primitive polynomial: p(x) = [1 1 0 0 1]
GF(2^4)[x] requires an equivalency matrix for every element in the field
That mapping is generated and shown below:
a^inf = 0
a^0 = 1
a^1 = a
a^2 = a^2
a^3 = a^3
a^4 = a^3 + 1
a^5 = a + a^3 + 1
a^6 = a + a^2 + a^3 + 1
a^7 = a + a^2 + 1
a^8 = a + a^2 + a^3
a^9 = a^2 + 1
a^10 = a + a^3
a^11 = a^2 + a^3 + 1
a^12 = a + 1
a^13 = a + a^2
a^14 = a^2 + a^3

Both methods for generating a codeword require the generating polynomial g(x).
In order to create this polynomial, factors of g must be multiplied together in power form
(in GF(2^4)[x]), the factors to be multiplied are x^i elements with i=[1:n-k].
*Since n-k = 6, g(x) = x^1 * x^2 * x^3 * x^4 * x^5 * x^6*

Using the equivalency matrix and polynomial multiplication in GF(2^4)[x]
the generating polynomial g(x) for the 3 error correcting code is shown below:
g(x) = [0 12 0 2 7 11 6]

------Generator Polynomial Method------
1.) This is a 3-error correcting, (15,9) R-S code over GF(2^4).
There are 36 bits in an input message made up of 9 (4-bit) messages
The message polynomial to be encoded is shown below:
m(x) = [1 Inf Inf Inf 5 Inf Inf 14 0]
2.) the generating polynomial is multiplied by the message polynomial to get the codeword shown below:
c(x) = [1 13 1 3 9 7 14 5 6 11 8 5 11 13 6]

------Systematic Encoding Method------
Below is the encoding of the systematic version of this code:\n
1.) First we shift the message polynomial over x^n-k bits
In this case, n-k=6 so the shifted codeword looks like this...
*m(x)*x^6 = [1 Inf Inf Inf 5 Inf Inf 14 0 Inf Inf Inf Inf Inf Inf]*
*2.) the remainder of the polynomial division of (m(x)*x^6)/g(x) is found as...*
r(x) = [2 7 13 5 3 13]
3.) r(x) is appended to the end of the shifted message polynomial, giving the

```
%Brendan Cain
%project 3 part 2

%Below is the completion of hw 8.1 using the functions completed
%for part 1 of the project

fprintf("-----------------HW 8.1-----------------\n");

syms x;
prim = x^3 + x + 1;
prim_poly = sym2poly(prim);
m = 3;
t = 2;
p = 2;
n = p^m - 1;
k = n - 2*t;
disp("Let p(x) be the primitive polynomial for the implementation of a
 narrow sense");
disp("(7,3) Reed-Solomon code shown below:");
fprintf("Polynomial form:\n\tp(x) = ");
disp(prim);
fprintf("Power form:\n\t");
print_poly("p(x)", prim_poly, true);
fprintf("\nGenerating mapping between binary groups in GF(2^3) and
 powers of alpha for arithmetic...\n\n");
equiv_cell = GenerateGF2(m, prim_poly);
fprintf("Given generator G(x) and message M(x) (both shown below),
 encode the codeword to send.\n");
fprintf("Then using that resulting polynomial, compute appropriate
 syndromes and verify that they are all zero\n");
G = [0 3 0 1 3]; % g(x) = x^4 + a^3x^3 + x^2 + ax + a^3
M = [0 4 2]; % m(x) = x^2 + a^4 + a^2
fprintf("Polynomial forms:\n\t");
print_poly("G(x)", G, false);
fprintf("\t");
print_poly("M(x)", M, false);
fprintf("Power forms:\n\t");
print_poly("G(x)", G, true);
fprintf("\t");
print_poly("M(x)", M, true);
fprintf("\nThen after encoding (generator method), the codeword C(x)
 is:\n");
% generator method of encoding
C = PolyMultGF2(G, M, equiv_cell);
fprintf("Polynomial form:\n\t");
print_poly("C(x)", C, false);
fprintf("Power form:\n\t");
print_poly("C(x)", C, true);
R = C; %received codeword


%computing syndrome:
```

```
fprintf("\nNow compute the syndromes and verify that they are all 0...
\n");
syn = get_syndromes(t, R, equiv_cell);
no_error = true;

for i = 1:size(syn, 2)
    if(syn(i) ~= -1)
        no_error = false;
        break;
    end
end

if(no_error)
    fprintf("All syndromes are a^inf = 0 in GF(2^m), no errors
 detected.\n");
else
    fprintf("Syndromes are not all inf and so error correction is
 required.\n");
end
print_poly("s(x)", syn, true);
fprintf("\n");

-----------------HW 8.1-----------------
Let p(x) be the primitive polynomial for the implementation of a
 narrow sense
(7,3) Reed-Solomon code shown below:
Polynomial form:
 p(x) = x^3 + x + 1

Power form:
 p(x) = [ 1 0 1 1 ]

Generating mapping between binary groups in GF(2^3) and powers of
 alpha for arithmetic...

Given generator G(x) and message M(x) (both shown below), encode the
 codeword to send.
Then using that resulting polynomial, compute appropriate syndromes
 and verify that they are all zero
Polynomial forms:
 G(x) = x^4 + a^3*x^3 + x^2 + a*x + a^3
 M(x) = x^2 + a^4*x + a^2
Power forms:
 G(x) = [ 0 3 0 1 3 ]
 M(x) = [ 0 4 2 ]

Then after encoding (generator method), the codeword C(x) is:
Polynomial form:
 C(x) = x^6 + a^6*x^5 + a^2*x^4 + a^3*x^3 + a*x + a^5
Power form:
 C(x) = [ 0 6 2 3 Inf 1 5 ]

Now compute the syndromes and verify that they are all 0...
All syndromes are a^inf = 0 in GF(2^m), no errors detected.
```

```
s(x) = [ Inf Inf Inf Inf Inf ]
```

Published with MATLAB® R2018b

```
systematic encoding of the codeword (shown below).
c_sys(x) = [ 1 Inf Inf Inf 5 Inf Inf 14 0 2 7 13 5 3 13 ]
Notice how the message polynomial is fully visible as the highest
 order symbols in the systematic word
```

Published with MATLAB® R2018b