```
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*