Errata (erasures+errors) Berlekamp-Massey for Reed-Solomon decoding

I am trying to implement a Reed-Solomon encoder-decoder in Python supporting the decoding of both erasures and errors, and that's driving me crazy.

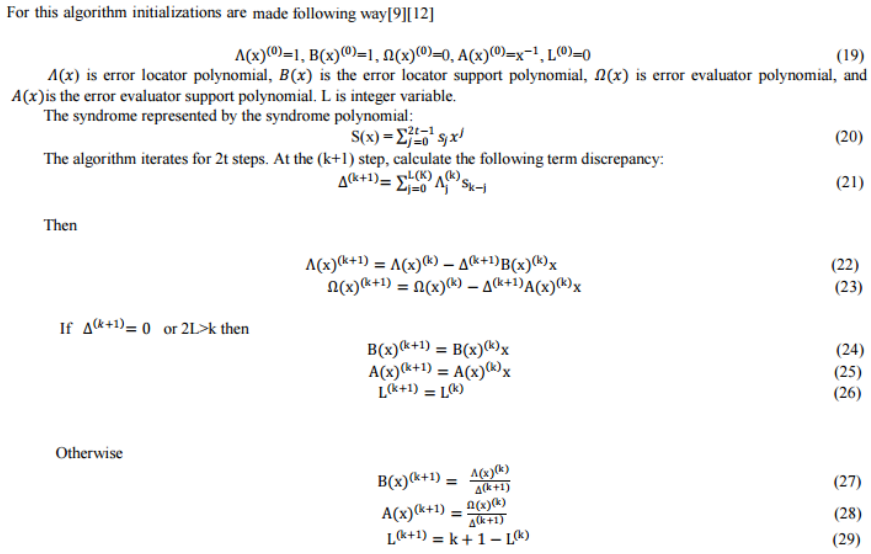
The implementation currently supports decoding only errors or only erasures, but not both at the same time (even if it's below the theoretical bound of 2*errors+erasures <= (n-k)).

From Blahut's papers (here and here), it seems we only need to initialize the error locator polynomial with the erasure locator polynomial to implicitly compute the errata locator polynomial inside Berlekamp-Massey.

This approach partially works for me: when I have 2*errors+erasures < (n-k)/2 it works, but in fact after debugging it only works because BM compute an errors locator polynomial that gets the exact same value as the erasure locator polynomial (because we are below the limit for errors-only correction), and thus it is truncated via galois fields and we end up with the correct value of the erasure locator polynomial (at least that's how I understand it, I may be wrong).

However, when we go above (n-k)/2, for example if n = 20 and k = 11, thus we have (n-k)=9 erased symbols we can correct, if we feed in 5 erasures then BM just goes wrong. If we feed in 4 erasures + 1 error (we are still well below the bound since we have 2*errors+erasures = 2+4 = 6 < 9), the BM still goes wrong.

The exact algorithm of Berlekamp-Massey I implemented can be found in this presentation (pages 15-17), but a very similar description can be found here and here, and here I attach a copy of the mathematical description:



For this algorithm initializations are made following way[9][12]

$$\Lambda(x)^{(0)}=1, B(x)^{(0)}=1, \Omega(x)^{(0)}=0, A(x)^{(0)}=x^{-1}, L^{(0)}=0 \quad (19)$$

$\Lambda(x)$ is error locator polynomial, $B(x)$ is the error locator support polynomial, $\Omega(x)$ is error evaluator polynomial, and $A(x)$is the error evaluator support polynomial. L is integer variable.

The syndrome represented by the syndrome polynomial:

$$S(x) = \sum_{j=0}^{2t-1} s_j x^j \quad (20)$$

The algorithm iterates for 2t steps. At the (k+1) step, calculate the following term discrepancy:

$$\Delta^{(k+1)} = \sum_{j=0}^{L(K)} \Lambda_j^{(k)} s_{k-j} \quad (21)$$

Then

$$\Lambda(x)^{(k+1)} = \Lambda(x)^{(k)} - \Delta^{(k+1)} B(x)^{(k)} x \quad (22)$$
$$\Omega(x)^{(k+1)} = \Omega(x)^{(k)} - \Delta^{(k+1)} A(x)^{(k)} x \quad (23)$$

If $\Delta^{(k+1)} = 0$ or $2L>k$ then

$$B(x)^{(k+1)} = B(x)^{(k)} x \quad (24)$$
$$A(x)^{(k+1)} = A(x)^{(k)} x \quad (25)$$
$$L^{(k+1)} = L^{(k)} \quad (26)$$

Otherwise

$$B(x)^{(k+1)} = \frac{\Lambda(x)^{(k)}}{\Delta^{(k+1)}} \quad (27)$$
$$A(x)^{(k+1)} = \frac{\Omega(x)^{(k)}}{\Delta^{(k+1)}} \quad (28)$$
$$L^{(k+1)} = k + 1 - L^{(k)} \quad (29)$$

Now, I have an almost exact reproduction of this mathematical algorithm into a Python code. What I would like is to extend it to support erasures, which I tried by initializing the error locator sigma with the erasure locator:

```
def _berlekamp_massey(self, s, k=None, erasures_loc=None):
    '''Computes and returns the error locator polynomial (sigma) and the
    error evaluator polynomial (omega).
    If the erasures locator is specified, we will return an errors-and-erasures locator
polynomial and an errors-and-erasures evaluator polynomial.
    The parameter s is the syndrome polynomial (syndromes encoded in a
    generator function) as returned by _syndromes. Don't be confused with
    the other s = (n-k)/2

    Notes:
    The error polynomial:
    E(x) = E_0 + E_1 x + ... + E_(n-1) x^(n-1)

    j_1, j_2, ..., j_s are the error positions. (There are at most s
    errors)

    Error location X_i is defined: X_i = a^(j_i)
    that is, the power of a corresponding to the error location

    Error magnitude Y_i is defined: E_(j_i)
    that is, the coefficient in the error polynomial at position j_i

    Error locator polynomial:
    sigma(z) = Product( 1 - X_i * z, i=1..s )
    roots are the reciprocals of the error locations
    ( 1/X_1, 1/X_2, ...)

    Error evaluator polynomial omega(z) is here computed at the same time as sigma, but it
can also be constructed afterwards using the syndrome and sigma (see
_find_error_evaluator() method).
    '''
    # For errors-and-erasures decoding, see: Blahut, Richard E. "Transform techniques for
error control codes." IBM Journal of Research and development 23.3 (1979): 299-315.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.600&rep=rep1&type=pdf and also
a MatLab implementation here: http://www.mathworks.com/matlabcentral/fileexchange/23567-
```

```
reed-solomon-errors-and-erasures-decoder/content//RS_E_E_DEC.m
    # also see: Blahut, Richard E. "A universal Reed-Solomon decoder." IBM Journal of
Research and Development 28.2 (1984): 150-158.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.2084&rep=rep1&type=pdf
    # or alternatively see the reference book by Blahut: Blahut, Richard E. Theory and
practice of error control codes. Addison-Wesley, 1983.
    # and another good alternative book with concrete programming examples: Jiang, Yuan. A
practical guide to error-control coding using Matlab. Artech House, 2010.
    n = self.n
    if not k: k = self.k

    # Initialize:
    if erasures_loc:
        sigma = [ Polynomial(erasures_loc.coefficients) ] # copy erasures_loc by creating
a new Polynomial
        B = [ Polynomial(erasures_loc.coefficients) ]
    else:
        sigma =  [ Polynomial([GF256int(1)]) ] # error locator polynomial. Also called
Lambda in other notations.
        B =    [ Polynomial([GF256int(1)]) ] # this is the error locator support/secondary
polynomial, which is a funky way to say that it's just a temporary variable that will help
us construct sigma, the error locator polynomial
    omega =  [ Polynomial([GF256int(1)]) ] # error evaluator polynomial. We don't need to
initialize it with erasures_loc, it will still work, because Delta is computed using
sigma, which itself is correctly initialized with erasures if needed.
    A =  [ Polynomial([GF256int(0)]) ] # this is the error evaluator support/secondary
polynomial, to help us construct omega
    L =      [ 0 ] # necessary variable to check bounds (to avoid wrongly eliminating the
higher order terms). For more infos, see
https://www.cs.duke.edu/courses/spring11/cps296.3/decoding_rs.pdf
    M =      [ 0 ] # optional variable to check bounds (so that we do not mistakenly
overwrite the higher order terms). This is not necessary, it's only an additional safe
check. For more infos, see the presentation decoding_rs.pdf by Andrew Brown in the doc
folder.

    # Polynomial constants:
    ONE = Polynomial(z0=GF256int(1))
    ZERO = Polynomial(z0=GF256int(0))
    Z = Polynomial(z1=GF256int(1)) # used to shift polynomials, simply multiply your poly
* Z to shift

    s2 = ONE + s

    # Iteratively compute the polynomials 2s times. The last ones will be
    # correct
    for l in xrange(0, n-k):
        K = l+1
        # Goal for each iteration: Compute sigma[K] and omega[K] such that
        # (1 + s)*sigma[l] == omega[l] in mod z^(K)

        # For this particular loop iteration, we have sigma[l] and omega[l],
        # and are computing sigma[K] and omega[K]

        # First find Delta, the non-zero coefficient of z^(K) in
        # (1 + s) * sigma[l]
        # This delta is valid for l (this iteration) only
        Delta = ( s2 * sigma[l] ).get_coefficient(l+1) # Delta is also known as the
Discrepancy, and is always a scalar (not a polynomial).
        # Make it a polynomial of degree 0, just for ease of computation with polynomials
sigma and omega.
        Delta = Polynomial(x0=Delta)

        # Can now compute sigma[K] and omega[K] from
        # sigma[l], omega[l], B[l], A[l], and Delta
        sigma.append( sigma[l] - Delta * Z * B[l] )
        omega.append( omega[l] - Delta * Z * A[l] )

        # Now compute the next B and A
        # There are two ways to do this
        # This is based on a messy case analysis on the degrees of the four polynomials
sigma, omega, A and B in order to minimize the degrees of A and B. For more infos, see
https://www.cs.duke.edu/courses/spring10/cps296.3/decoding_rs_scribe.pdf
        # In fact it ensures that the degree of the final polynomials aren't too large.
        if Delta == ZERO or 2*L[l] > K \
            or (2*L[l] == K and M[l] == 0):
            # Rule A
            B.append( Z * B[l] )
            A.append( Z * A[l] )
            L.append( L[l] )
            M.append( M[l] )

        elif (Delta != ZERO and 2*L[l] < K) \
            or (2*L[l] == K and M[l] != 0):
            # Rule B
            B.append( sigma[l] // Delta )
            A.append( omega[l] // Delta )
            L.append( K - L[l] )
            M.append( 1 - M[l] )

        else:
            raise Exception("Code shouldn't have gotten here")

    return sigma[-1], omega[-1]
```

Polynomial and GF256int are generic implementation of, respectively, polynomials and galois fields over 2^8. These classes are unit tested and they are, normally, bug proof. Same goes for the rest of the encoding/decoding methods for Reed-Solomon such as Forney and Chien search. The full code with a quick test case for the issue I am talking here can be found here: http://codepad.org/l2Qi0y8o

Here's an example output:

```
Encoded message:
hello world□□Ī`>
-------
Erasures decoding:
Erasure locator: 189x^5 + 88x^4 + 222x^3 + 33x^2 + 251x + 1
Syndrome: 149x^9 + 113x^8 + 29x^7 + 231x^6 + 210x^5 + 150x^4 + 192x^3 + 11x^2 + 41x
Sigma: 189x^5 + 88x^4 + 222x^3 + 33x^2 + 251x + 1
Symbols positions that were corrected: [19, 18, 17, 16, 15]
('Decoded message: ', 'hello world', '\xce\xea\x90\x99\x8d\xc4\xaa`>')
Correctly decoded:  True
-------
Errors+Erasures decoding for the message with only erasures:
Erasure locator: 189x^5 + 88x^4 + 222x^3 + 33x^2 + 251x + 1
Syndrome: 149x^9 + 113x^8 + 29x^7 + 231x^6 + 210x^5 + 150x^4 + 192x^3 + 11x^2 + 41x
Sigma: 101x^10 + 139x^9 + 5x^8 + 14x^7 + 180x^6 + 148x^5 + 126x^4 + 135x^3 + 68x^2 + 155x
+ 1
Symbols positions that were corrected: [187, 141, 90, 19, 18, 17, 16, 15]
('Decoded message: ',
'\xf4\x00\x00\x00
```

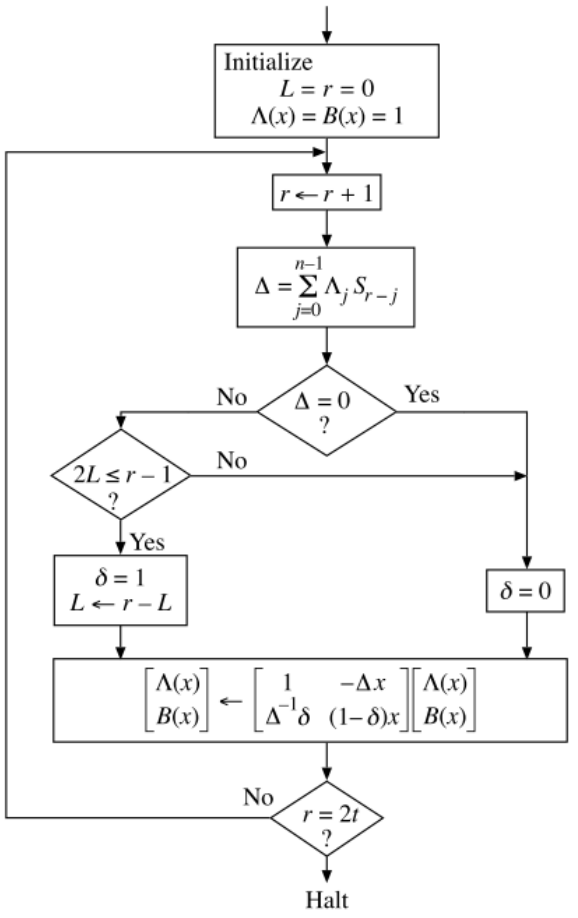


Figure 7.5. The Berlekamp–Massey algorithm

And here is the errata (errors-and-erasures) Berlekamp-Massey algorithm for Reed-Solomon:

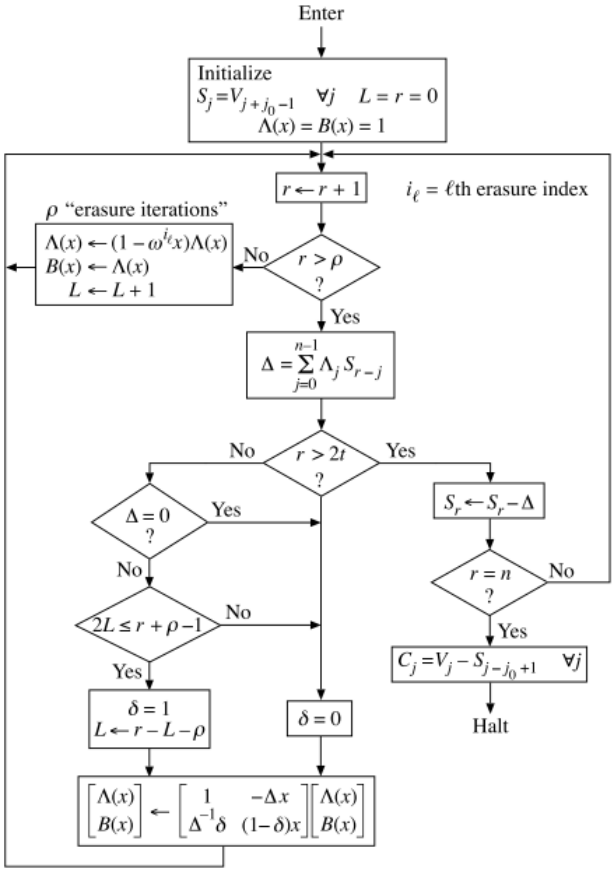


Figure 7.10. An error-and-erasure decoder for BCH codes

As you can see -- contrary to the usual description that you only **need to initialize Lambda, the errors locator polynomial, with the value of the previously computed erasures locator polynomial** -- you also need to skip the first v iterations, where v is the number of erasures. Note that it's not equivalent to skipping the last v iterations: **you need to skip the first v iterations**,

because r (the iteration counter, K in my implementation) is used not only to count iterations but also to generate the correct discrepancy factor Delta.

Here is the resulting code with the modifications to support erasures as well as errors up to `v+2*e <= (n-k)` :

```
def _berlekamp_massey(self, s, k=None, erasures_loc=None, erasures_eval=None,
erasures_count=0):
    '''Computes and returns the errata (errors+erasures) locator polynomial (sigma) and
the
    error evaluator polynomial (omega) at the same time.
    If the erasures locator is specified, we will return an errors-and-erasures locator
polynomial and an errors-and-erasures evaluator polynomial, else it will compute only
errors. With erasures in addition to errors, it can simultaneously decode up to v+2e <=
(n-k) where v is the number of erasures and e the number of errors.
    Mathematically speaking, this is equivalent to a spectral analysis (see Blahut,
"Algebraic Codes for Data Transmission", 2003, chapter 7.6 Decoding in Time Domain).
    The parameter s is the syndrome polynomial (syndromes encoded in a
    generator function) as returned by _syndromes.

    Notes:
    The error polynomial:
    E(x) = E_0 + E_1 x + ... + E_(n-1) x^(n-1)

    j_1, j_2, ..., j_s are the error positions. (There are at most s
    errors)

    Error location X_i is defined: X_i = α^(j_i)
    that is, the power of α (alpha) corresponding to the error location

    Error magnitude Y_i is defined: E_(j_i)
    that is, the coefficient in the error polynomial at position j_i

    Error locator polynomial:
    sigma(z) = Product( 1 - X_i * z, i=1..s )
    roots are the reciprocals of the error locations
    ( 1/X_1, 1/X_2, ...)

    Error evaluator polynomial omega(z) is here computed at the same time as sigma, but it
can also be constructed afterwards using the syndrome and sigma (see
_find_error_evaluator() method).

    It can be seen that the algorithm tries to iteratively solve for the error locator
polynomial by
    solving one equation after another and updating the error locator polynomial. If it
turns out that it
    cannot solve the equation at some step, then it computes the error and weights it by
the last
    non-zero discriminant found, and delays the weighted result to increase the polynomial
degree
    by 1. Ref: "Reed Solomon Decoder: TMS320C64x Implementation" by Jagadeesh Sankaran,
December 2000, Application Report SPRA686

    The best paper I found describing the BM algorithm for errata (errors-and-erasures)
evaluator computation is in "Algebraic Codes for Data Transmission", Richard E. Blahut,
2003.
    '''
    # For errors-and-erasures decoding, see: "Algebraic Codes for Data Transmission",
Richard E. Blahut, 2003 and (but it's less complete): Blahut, Richard E. "Transform
techniques for error control codes." IBM Journal of Research and development 23.3 (1979):
299-315. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.600&rep=rep1&type=pdf
and also a MatLab implementation here:
http://www.mathworks.com/matlabcentral/fileexchange/23567-reed-solomon-errors-and-
erasures-decoder/content//RS_E_E_DEC.m
    # also see: Blahut, Richard E. "A universal Reed-Solomon decoder." IBM Journal of
Research and Development 28.2 (1984): 150-158.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.2084&rep=rep1&type=pdf
    # and another good alternative book with concrete programming examples: Jiang, Yuan. A
practical guide to error-control coding using Matlab. Artech House, 2010.
    n = self.n
    if not k: k = self.k

    # Initialize, depending on if we include erasures or not:
    if erasures_loc:
        sigma = [ Polynomial(erasures_loc.coefficients) ] # copy erasures_loc by creating
a new Polynomial, so that we initialize the errata locator polynomial with the erasures
locator polynomial.
        B = [ Polynomial(erasures_loc.coefficients) ]
        omega =  [ Polynomial(erasures_eval.coefficients) ] # to compute omega (the
evaluator polynomial) at the same time, we also need to initialize it with the partial
erasures evaluator polynomial
        A =  [ Polynomial(erasures_eval.coefficients) ] # TODO: fix the initial value of
the evaluator support polynomial, because currently the final omega is not correct (it
contains higher order terms that should be removed by the end of BM)
    else:
        sigma =  [ Polynomial([GF256int(1)]) ] # error locator polynomial. Also called
Lambda in other notations.
        B =    [ Polynomial([GF256int(1)]) ] # this is the error locator support/secondary
polynomial, which is a funky way to say that it's just a temporary variable that will help
us construct sigma, the error locator polynomial
        omega =  [ Polynomial([GF256int(1)]) ] # error evaluator polynomial. We don't need
to initialize it with erasures_loc, it will still work, because Delta is computed using
sigma, which itself is correctly initialized with erasures if needed.
        A =  [ Polynomial([GF256int(0)]) ] # this is the error evaluator support/secondary
polynomial, to help us construct omega
    L = [ 0 ] # update flag: necessary variable to check when updating is necessary and to
check bounds (to avoid wrongly eliminating the higher order terms). For more infos, see
https://www.cs.duke.edu/courses/spring11/cps296.3/decoding_rs.pdf
```

```
    M = [ 0 ] # optional variable to check bounds (so that we do not mistakenly overwrite
the higher order terms). This is not necessary, it's only an additional safe check. For
more infos, see the presentation decoding_rs.pdf by Andrew Brown in the doc folder.

    # Fix the syndrome shifting: when computing the syndrome, some implementations may
prepend a 0 coefficient for the lowest degree term (the constant). This is a case of
syndrome shifting, thus the syndrome will be bigger than the number of ecc symbols (I
don't know what purpose serves this shifting). If that's the case, then we need to account
for the syndrome shifting when we use the syndrome such as inside BM, by skipping those
prepended coefficients.
    # Another way to detect the shifting is to detect the 0 coefficients: by definition, a
syndrome does not contain any 0 coefficient (except if there are no errors/erasures, in
this case they are all 0). This however doesn't work with the modified Forney syndrome
(that we do not use in this lib but it may be implemented in the future), which set to 0
the coefficients corresponding to erasures, leaving only the coefficients corresponding to
errors.
    synd_shift = 0
    if len(s) > (n-k): synd_shift = len(s) - (n-k)

    # Polynomial constants:
    ONE = Polynomial(z0=GF256int(1))
    ZERO = Polynomial(z0=GF256int(0))
    Z = Polynomial(z1=GF256int(1)) # used to shift polynomials, simply multiply your poly
* Z to shift

    # Precaching
    s2 = ONE + s

    # Iteratively compute the polynomials n-k-erasures_count times. The last ones will be
correct (since the algorithm refines the error/errata locator polynomial iteratively
depending on the discrepancy, which is kind of a difference-from-correctness measure).
    for l in xrange(0, n-k-erasures_count): # skip the first erasures_count iterations
because we already computed the partial errata locator polynomial (by initializing with
the erasures locator polynomial)
        K = erasures_count+l+synd_shift # skip the FIRST erasures_count iterations (not
the last iterations, that's very important!)

        # Goal for each iteration: Compute sigma[l+1] and omega[l+1] such that
        # (1 + s)*sigma[l] == omega[l] in mod z^(K)

        # For this particular loop iteration, we have sigma[l] and omega[l],
        # and are computing sigma[l+1] and omega[l+1]

        # First find Delta, the non-zero coefficient of z^(K) in
        # (1 + s) * sigma[l]
        # Note that adding 1 to the syndrome s is not really necessary, you can do as well
without.
        # This delta is valid for l (this iteration) only
        Delta = ( s2 * sigma[l] ).get_coefficient(K) # Delta is also known as the
Discrepancy, and is always a scalar (not a polynomial).
        # Make it a polynomial of degree 0, just for ease of computation with polynomials
sigma and omega.
        Delta = Polynomial(x0=Delta)

        # Can now compute sigma[l+1] and omega[l+1] from
        # sigma[l], omega[l], B[l], A[l], and Delta
        sigma.append( sigma[l] - Delta * Z * B[l] )
        omega.append( omega[l] - Delta * Z * A[l] )

        # Now compute the next support polynomials B and A
        # There are two ways to do this
        # This is based on a messy case analysis on the degrees of the four polynomials
sigma, omega, A and B in order to minimize the degrees of A and B. For more infos, see
https://www.cs.duke.edu/courses/spring10/cps296.3/decoding_rs_scribe.pdf
        # In fact it ensures that the degree of the final polynomials aren't too large.
        if Delta == ZERO or 2*L[l] > K+erasures_count \
            or (2*L[l] == K+erasures_count and M[l] == 0):
        #if Delta == ZERO or len(sigma[l+1]) <= len(sigma[l]): # another way to compute
when to update, and it doesn't require to maintain the update flag L
            # Rule A
            B.append( Z * B[l] )
            A.append( Z * A[l] )
            L.append( L[l] )
            M.append( M[l] )

        elif (Delta != ZERO and 2*L[l] < K+erasures_count) \
            or (2*L[l] == K+erasures_count and M[l] != 0):
        # elif Delta != ZERO and len(sigma[l+1]) > len(sigma[l]): # another way to compute
when to update, and it doesn't require to maintain the update flag L
            # Rule B
            B.append( sigma[l] // Delta )
            A.append( omega[l] // Delta )
            L.append( K - L[l] ) # the update flag L is tricky: in Blahut's schema, it's
mandatory to use `L = K - L - erasures_count` (and indeed in a previous draft of this
function, if you forgot to do `- erasures_count` it would lead to correcting only 2*
(errors+erasures) <= (n-k) instead of 2*errors+erasures <= (n-k)), but in this latest
draft, this will lead to a wrong decoding in some cases where it should correctly decode!
Thus you should try with and without `- erasures_count` to update L on your own
implementation and see which one works OK without producing wrong decoding failures.
            M.append( 1 - M[l] )

        else:
            raise Exception("Code shouldn't have gotten here")

    # Hack to fix the simultaneous computation of omega, the errata evaluator polynomial:
because A (the errata evaluator support polynomial) is not correctly initialized (I could
not find any info in academic papers). So at the end, we get the correct errata evaluator
polynomial omega + some higher order terms that should not be present, but since we know
that sigma is always correct and the maximum degree should be the same as omega, we can
fix omega by truncating too high order terms.
```

```
    if omega[-1].degree > sigma[-1].degree: omega[-1] =
Polynomial(omega[-1].coefficients[-(sigma[-1].degree+1):])

    # Return the last result of the iterations (since BM compute iteratively, the last
iteration being correct - it may already be before, but we're not sure)
    return sigma[-1], omega[-1]

def _find_erasures_locator(self, erasures_pos):
    '''Compute the erasures locator polynomial from the erasures positions (the positions
must be relative to the x coefficient, eg: "hello worldxxxxxxxxx" is tampered to "h_ll_
worldxxxxxxxxx" with xxxxxxxxx being the ecc of length n-k=9, here the string positions
are [1, 4], but the coefficients are reversed since the ecc characters are placed as the
first coefficients of the polynomial, thus the coefficients of the erased characters are
n-1 - [1, 4] = [18, 15] = erasures_loc to be specified as an argument.'''
    # See:
http://ocw.usu.edu/Electrical_and_Computer_Engineering/Error_Control_Coding/Lecture7.pdf
and Blahut, Richard E. "Transform techniques for error control codes." IBM Journal of
Research and development 23.3 (1979): 299-315.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.600&rep=rep1&type=pdf and also
a MatLab implementation here: http://www.mathworks.com/matlabcentral/fileexchange/23567-
reed-solomon-errors-and-erasures-decoder/content//RS_E_E_DEC.m
    erasures_loc = Polynomial([GF256int(1)]) # just to init because we will multiply, so
it must be 1 so that the multiplication starts correctly without nulling any term
    # erasures_loc is very simple to compute: erasures_loc = prod(1 - x*alpha[j]**i) for i
in erasures_pos and where alpha is the alpha chosen to evaluate polynomials (here in this
library it's gf(3)). To generate c*x where c is a constant, we simply generate a
Polynomial([c, 0]) where 0 is the constant and c is positionned to be the coefficient for
x^1. See https://en.wikipedia.org/wiki/Forney_algorithm#Erasures
    for i in erasures_pos:
        erasures_loc = erasures_loc * (Polynomial([GF256int(1)]) -
Polynomial([GF256int(self.generator)**i, 0]))
    return erasures_loc
```

Note: Sigma, Omega, A, B, L and M are all lists of polynomials (so we keep the whole history of all intermediate polynomials we computed on each iteration). This can of course be optimized because we only really need `Sigma[l]`, `Sigma[l-1]`, `Omega[l]`, `Omega[l-1]`, `A[l]`, `B[l]`, `L[l]` and `M[l]` (so it's just Sigma and Omega that needs to keep the previous iteration in memory, the other variables don't need).

Note2: the update flag L is tricky: in some implementations, doing just like in the Blahut's schema will lead to wrong failures when decoding. In my past implementation, it was mandatory to use `L = K - L - erasures_count` to correctly decode both errors-and-erasures up to the Singleton bound, but in my latest implementation, I had to use `L = K - L` (even when there are erasures) to avoid wrong decoding failures. You should just try both on your own implementation and see which one doesn't produce any wrong decoding failures. See below in the issues for more info.

The only issue with this algorithm is that it does not describe how to simultaneously compute Omega, the errors evaluator polynomial (the book describes how to initialize Omega for errors only, but not when decoding errors-and-erasures). I tried several variations and the above works, but not completely: at the end, Omega will include higher order terms that should have been cancelled. Probably Omega or A the errors evaluator support polynomial, is not initialized with the good value.

However, you can fix that by either trimming the Omega polynomial of the too high order terms (since it should have the same degree as Lambda/Sigma):

```
if omega[-1].degree > sigma[-1].degree: omega[-1] = Polynomial(omega[-1].coefficients[-
(sigma[-1].degree+1):])
```

Or you can totally compute Omega from scratch after BM by using the errata locator Lambda/Sigma, which is always correctly computed:

```
def _find_error_evaluator(self, synd, sigma, k=None):
    '''Compute the error (or erasures if you supply sigma=erasures locator polynomial)
evaluator polynomial Omega from the syndrome and the error/erasures/errata locator Sigma.
Omega is already computed at the same time as Sigma inside the Berlekamp-Massey
implemented above, but in case you modify Sigma, you can recompute Omega afterwards using
this method, or just ensure that Omega computed by BM is correct given Sigma (as long as
syndrome and sigma are correct, omega will be correct).'''
    n = self.n
    if not k: k = self.k

    # Omega(x) = [ Synd(x) * Error_Loc(x) ] mod x^(n-k+1) -- From Blahut, Algebraic codes
for data transmission, 2003
    return (synd * sigma) % Polynomial([GF256int(1)] + [GF256int(0)] * (n-k+1)) # Note
that you should NOT do (1+Synd(x)) as can be seen in some books because this won't work
with all primitive generators.
```

I am looking for a better solution in the following question on CSTheory.

/EDIT: I will describe some of the issues I have had and how to fix them:

- don't forget to init the error locator polynomial with the erasures locator polynomial (that you can easily compute from the syndromes and erasures positions).
- if you can decode errors only and erasures only flawlessly, but limited to `2*errors + erasures <= (n-k)/2`, then you forgot to skip the first v iterations.
- if you can decode both erasures-and-errors but up to `2*(errors+erasures) <= (n-k)`, then you forgot to update the assigment of L: `L = i+1 - L - erasures_count` instead of `L = i+1 - L`. But this may actually make your decoder fail in some cases depending on how you

implemented your decoder, see the next point.

- my first decoder was limited to only one generator/prime polynomial/fcr, but when I updated it to be universal and added strict unit tests, the decoder failed when it shouldn't. It seems Blahut's schema above is wrong about L (the updating flag): it must be updated using `L = K - L` and not `L = K - L - erasures_count`, because this will lead to the decoder failing sometimes even through we are under the Singleton bound (and thus we should be decoding correctly!). This seems to be confirmed by the fact that computing `L = K - L` will not only fix those decoding issues, but it will also give the exact same result as the alternative way to update without using the update flag L (ie, the condition `if Delta == ZERO or len(sigma[l+1]) <= len(sigma[l]):`). But this is weird: in my past implementation, `L = K - L - erasures_count` was mandatory for errors-and-erasures decoding, but now it seems it produces wrong failures. So you should just try with and without on your own implementation and whether one or the other produces wrong failures for you.
- note that the condition `2*L[l] > K` change to `2*L[l] > K+erasures_count`. You may not notice any side effect without adding the condition `+erasures_count` at first, but in some cases the decoding will fail when it shouldn't.
- if you can fix only exactly one error or erasure, check that your condition is `2*L[l] > K+erasures_count` and not `2*L[l] >= K+erasures_count` (notice the `>` instead of `>=`).
- if you can correct `2*errors + erasures <= (n-k-2)` (just below the limit, eg, if you have 10 ecc symbols, you can correct only 4 errors instead of 5 normally) then check your syndrome and your loop inside the BM algo: if the syndrome starts with a 0 coefficient for the constant term x^0 (which is sometimes advised in books), then your syndrome is shifted, and then your loop inside BM must start at `1` and end at `n-k+1` instead of `0:(n-k)` if not shifted.
- If you can correct every symbol but the last one (the last ecc symbol), then check your ranges, particularly in your Chien search: you should not evaluate the error locator polynomial from alpha^0 to alpha^255 but from alpha^1 to alpha^256.

edited 2 mins ago

answered May 26 at 20:35

gaborous

1,933 3 17 31

I have made a mistake in the sourcecode: it's not `2*L[l] >= K` but `2*L[l] > K`, else you will only be able to fix one error only. I edited my answer to fix this mistake. – gaborous Jun 3 at 16:47

Add Another Answer