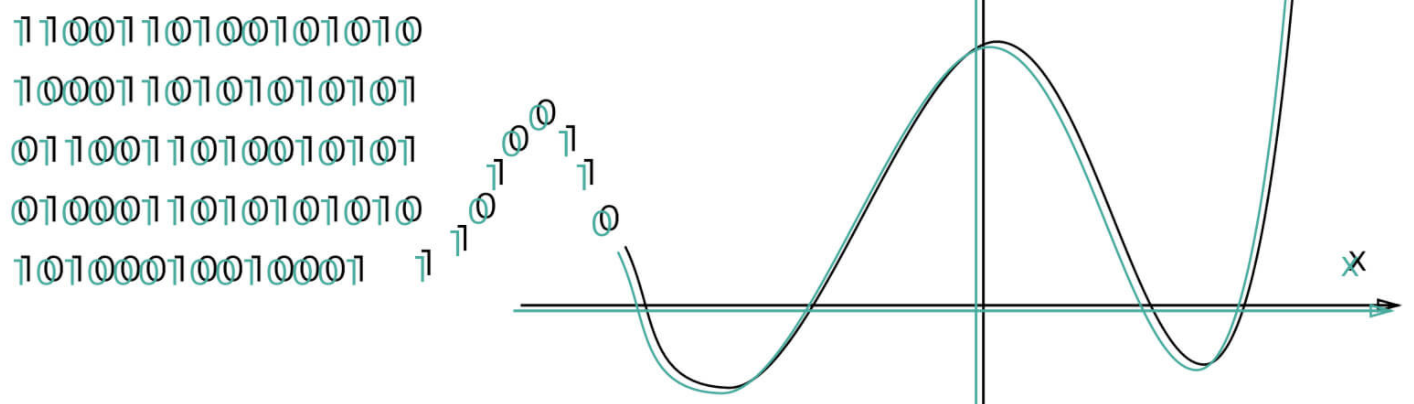
Error Correction Codes

Reed - Solomon:



Israel Zemser - 312246721

Introduction:  
  
Reed - Solomon error correcting codes are one of the oldest codes that were introduced in 1960s by Irving S. Reed and Gustave Solomon . It is an algorithm constructed to use polynomials over data blocks.

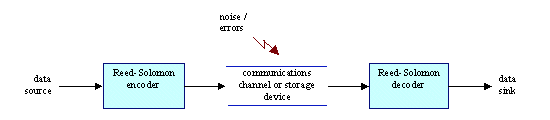
The Reed - Solomon group of algorithms are addressing a common issue which occurs when transferred data is corrupted over the course of the travel to the final destination. The corruption may occur due to malfunctions, interruptions or “noise” in the transmission medium itself - leading to errors.

These error correcting algorithms have many applications. The most prominent of which include consumer technologies:

* Storage devices (CDs, DVDs, Blu-rays discs, etc)
* QRs codes
* Wireless or mobile communications
* High-speed modems (ADSL, xDSL, etc)
* Satellite communication.
* Digital TV.

The Reed–Solomon code is a [n, k, n − k + 1] code. in other words, it is a linear block code of length n with dimension k and minimum Hamming distance Dmin = n-k+1.

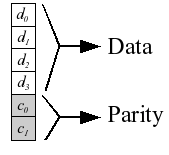
The Reed–Solomon code is optimal in the sense that the minimum distance has the maximum value possible for a linear code of size (n, k). this is known as the Singleton bound. Such a code is also called a maximum distance separable code.



(\*change picture\*)

Our implementation of the Reed - Solomon algorithm:

The RS code consists with two major components: Encoder and Decoder.  
The encoder takes a block of digital data and adds extra "redundant" bits. It first visualized the message as a polynomial instead of representing the data simply as its value, and then activate the polynomial as we shall describe in length bellow.

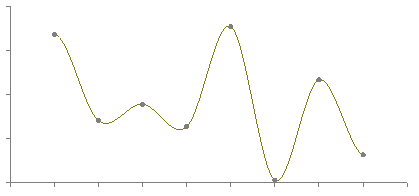


The decoder processes each block and attempts to correct errors and recover the original data. By representing the data as a polynomial, it can use tools from different mathematical fields and employ different methods in order to restore the original message and repair corrupted messages.

The number and type of errors that can be corrected depends on the characteristics of the Reed-Solomon code. Although when the error rate is greater, that decoder will not be able to output the correct result.   
Using a basic decoder one can correct up to (𝑛−𝑘)/2 errors. In our project, we shall implement the List-decoding algorithm, introduced by Madhu Sudan which overcomes that issue and can correct up to 𝑛 − 2𝑘 errors and its improvement that can correct up to

n - 2.

The code relies on a theorem from algebra that states that any k distinct points uniquely determine a polynomial of degree at most k-1 data points.



The RS code operates on a block of data treated as a set of finite field elements called symbols. We shall treat the code as a transferred message and characterize it by the next parameters:

k - the original message length

n - the encoded word length, where k < n.

q - the finite field size (a prime power), where q is greater or equal to n.

Encoder:

**Input**: message, k, n

Execute the following:

* Create finite field q out of n, where q is a prime number such that k < n ≤ q (we select q to be at least 53, our domain size is 52)
* Get the polynomial coefficients: go through the message string and convert each ascii characters to a number in the finite field.
* Build a polynomial F of degree k-1 over the finite field, using the coefficients from the step above.
* Let F be the polynomial which we received by these coefficients.
* Create a codeword of length n, such that for each index i between 0 and n-1,   
  codeword[i] = F[i].

**Output**: codeword of length n.

In other words, our encoder receives the message, its length(k), and the length of message the create(n), and transmit over a communication channel the codeword of n-characters.  
Our implementation considers the given k-bits message as a polynomial of degree k-1, as the message symbols shall represent the polynomial coefficients, in order to create F. We now can activate F on n length vector of coefficients (and take its mod q values) in order to receive our encoded message.

Example:  
Let “abcde” be our message, so our encoder takes as input “abcde”, 5, and “n” of our choosing. let n = 140 for the sake of the example.  
  
So as mentioned above, we first compute q and get 149, as it the smallest prime number that is bigger than n.  
We then create our poly: F(x) = 27 + 28 + 29 + 30+ 31   
Eventually we compute our codeword to equal the concatenation of the polynomial at points [0, n]:  
F(0) mod 149, F(1) mod 149,..., F(139) mod 149.

Corrupt Message:

We emulate the corruption of the message by “injecting errors” into the encoded codeword.

We take as input the number of errors “e”, that we wish to emulate and “inject” them as follows:

Going through the first “e” indexes of the codeword and assigning them a random integer value in the field we have created in the encoder.

Decoder:

**Input**: Encoded message, k, n

Execute the following:s

* Create finite field(q) out of n. (same as in the encoder)
* Build a bivariate polynomial 𝑄(𝑥, 𝑦) =

such that for each index 𝑖 𝜖 (0, . . ., 𝑛 − 1), the pair (,) where =i and =codeword[] the following holds: 𝑄(,) = 0.

* Extract all polynomials s.t:

{𝑃(𝑥) | 𝑦 − 𝑃(𝑥) 𝜖 𝑄(𝑥, 𝑦) }, and P() = for all 𝑖 𝜖 (0, . . ., 𝑛 − 1).

* For every extracted polynomial, map each coefficient to its corresponding symbol, in order to create the decoding list of (at most) k possible messages.

**Output**: Decoded messages list

Details:

In the decoding process, we try to recover the original data k message we have sent by outputting a list of possible messages.

By given n pairs of elements { (𝑥1, 𝑦1 ) …. (𝑥𝑛, 𝑦𝑛) } , where , are value over the field q

and =codeword[].

We would like to find a polynomial 𝑄(𝑥, 𝑦) = such that:

≤ , ≤ and {𝑄(,) = 0 | }.

We shall get all coefficients , by building a matrix A of n equations over the finite field q, which satisfies 𝑄(𝑥𝑖 , 𝑦𝑖) = 0.  
The equations system 𝐴 ∗ 𝑣 = 0 has numerous solutions due to its dimensions. Thus we pick a nonzero vector 𝑣 within the solution space (The kernel space of the solved matrix) of the linear equation system, and construct a polynomial using its values as the coefficients:  
𝑄(𝑥, 𝑦) = s.t: .

By finding the factors of Q over the finite field q we receive the next equation:

𝑄(𝑥, 𝑦) = (𝑥, 𝑦) ∗ (𝑥, 𝑦)… ∗ (𝑥, 𝑦).

Then finding the factors of Q that satisfies the constrains:

{𝑃(𝑥) | 𝑦 − 𝑃(𝑥) 𝜖 𝑄(𝑥, 𝑦) }, and P() = for 𝑖 𝜖 (0, . . ., 𝑛 − 1).

We produce a list of potential original message by going through every such factor

we decode a possible message by mapping the coefficients of the polynomial to its ascii value over the finite field q.

As mentioned in the introduction, our decoder has the ability to correct up to 𝑛 − 2𝑘 errors.

By implementing an improvement, we can decode correctly up to n - 2 errors.

We can do so by changing the degree of x and y in the bivariate polynomial 𝑄(𝑥, 𝑦).

Set to be rounding down of and to .

Test Cases

We define fraction of errors to be .

Choice = 0 => for the bivariant polynomial 𝑄, ≤ & ≤ .

error correction rate is 𝑛 − 2𝑘.

Choice = 1 => for the bivariant polynomial 𝑄, & .

error correction rate is n - 2.

Choice is set by default to 0.

Test #1

Case 1: message “abc” : k = 3, n = 140, errors = 68

Output:

Error Correction Rate = 69.0070426028 and the fraction of errors = 0.485714285714

Success! the message "abc", exists in the decoded list

Case 2: message “abc” : k = 3, n = 140, errors = 106

Output:

Error Correction Rate = 69.0070426028 and the fraction of errors = 0.757142857143

The decoder didn't succeed in correcting the message with number of 106 errors

We observe that parameters n and k are the same we in cases 1 and 2 and have the identical correction rate.

our decoder couldn’t handle fraction of errors that is 0.757142857143 and failed.

Now we want to check how the increment of n affects the result:

Case 3: message “abc” : k = 3, n = 500, errors = 106

Output:

Error Correction Rate = 365.83592135 and the fraction of errors = 0.212 Success! the message "abc", exists in the decoded list

The increment of n helped us to decode successfully the message.

Now lets the check if the decoder would be able to decode successfuly when we increment the error rate to 0.8

Case 4: message “abc” : k = 3, n = 500, errors = 400

Output:

Error Correction Rate = 365.83592135 and the fraction of errors = 0.8 Success! the message "abc", exists in the decoded list

We can infer that when we increase n and the k stays the same the probability to decode the original message increases because the error correction rate increase.

Test #2

Let’s test the improvement of the algorithm with following example

Message = "CoronaVir", k = 9, n = 729, errors = 500

Case 1 : we run the example with default choice = 0

Output:

Error Correction Rate = 243.0 and the fraction of errors = 0.685871056241

The decoder didn't succeed in correcting the message with number of 500 errors

Case 2 : we run the example with choice = 1

Output:

Error Correction Rate = 567.0 and the fraction of errors = 0.685871056241

Success! the message "CoronaVir", exists in the decoded list

We can see that our error correction is higher and indeed the decoder was able to handle with a fraction of errors 0.685871056241 that before the improvement failed.

Test #3

We want to check the constraint on n, which is k < :

we will try to encode the message "BguUniversity" (k=13) with 0 erros to see if the constraint above affects the result.

Case 1: run the decoder with k = 13, n = 100, errors = 0. We notice 13 > .

Output:

Error Correction Rate = 27.8889744907 and the fraction of errors = 0.0 The decoder didn't succeed in correcting the message with number of 0 errors

Case 2: run the decoder with k = 13, n = 600, errors = 0. We notice 13

Output:

Error Correction Rate = 423.364782673 and the fraction of errors = 0.0 Success! the message "BguUniversity", exists in the decoded list

Test #4

Multiple test – we will run the same example multiple times to see if the result are consecutive (we define choice = 1)

Case 1: message = "BguUniversity", k = 13 , n = 600, errors = 423 , times = 5

Output:

Error Correction Rate = 423.364782673 and the fraction of erros = 0.705 Number of success = 5 Number of failure = 0

Case 2: message = "BguUniversity", k = 13 , n = 600, errors = 428 , times = 5

Output:

Error Correction Rate = 423.364782673 and the fraction of erros = 0.713333333333 Number of success = 2 Number of failure = 3

Case 3: message = "BguUniversity", k = 13 , n = 600, errors = 429 , times = 5

Output:

Error Correction Rate = 423.364782673 and the fraction of erros = 0.715 Number of success = 0 Number of failure = 5

We can observe that with the correction rate above the results were different depending on the fraction of errors.

The decoder can handle up to 423 errors in an unambiguous manner, but when we raised the errors above 423 for 428 the decoder succeed only in 1 out of 5 tries and for 429 errors it couldn’t find the original message.

Conclusions:

The improvement of choice 1 lets us handle a higher fraction of errors,

It is better to encode the message into a longer codeword as it increases our chances to decode the original message.

The chosen n (length of the encoded message) should be chosen s.t: k <

**Note**: in order to run the file go to <https://www.sagemath.org/> and open the file via SageWorksheet.

Appendix

* Improvements on the Johnson bound for Reed–Solomon codes (ScienceDirect) -

<https://www.sciencedirect.com/science/article/pii/S0166218X08002758>

* Guruswami-sudan list decoding algorithm -

<https://en.wikipedia.org/wiki/Guruswami%E2%80%93Sudan_list_decoding_algorithm#Algorithm_1_(Sudan's_list_decoding_algorithm)>

* Chapter 4 - Decoding reed Solomon codes –

<http://www.cs.cmu.edu/~venkatg/pubs/papers/listdecoding-NOW.pdf>

* Wikipedia – Reed Solomon error correction –

<https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction>