REED-SOLOMON CODING

In telecommunications, Reed–Solomon (RS) codes are a kind of channel coding techniques operating on blocks of data treated as a set of finite field elements, called *symbols*. By adding t redundancy symbols to the data, RS codes can detect (but not correct) any combination of up to t erroneous symbols, or locate and correct up to and including $t/2$ erroneous symbols at unknown locations. As better explained below, RS codes are especially suitable for multiple-burst errors correction, since consecutive corrupted bits affect just a limited number of symbols [4].
In this project, after a short recap of general theory, the error-correction capability of RS codes is discussed and analyzed. First, a MATLAB script (without employing any of its built-in functions) for Reed-Solomon (RS) encoding and decoding is proposed. Then, the same algorithm is adapted for a much more efficient C/C++ implementation.

The developed encoder / decoder pair can handle extended Galois field degrees m from 3 up to 8. The *primitive polynomials* $f(x)$ selected for the definition of each finite field are [1]

$$GF(2^3) \rightarrow f(x) = 1 + x + x^3$$
$$GF(2^4) \rightarrow f(x) = 1 + x + x^4$$
$$GF(2^5) \rightarrow f(x) = 1 + x^2 + x^5$$
$$GF(2^6) \rightarrow f(x) = 1 + x + x^6$$
$$GF(2^7) \rightarrow f(x) = 1 + x^3 + x^7$$
$$GF(2^8) \rightarrow f(x) = 1 + x^2 + x^3 + x^4 + x^8$$

To obtain the mapping table between symbols and m-bit streams (aka basis), the former are converted into polynomials and divided by the primitive polynomial. Then, the remainder of this division represents the basis corresponding to the starting symbol [1]. In **Fig.1** and **Fig.2** the mapping table for $m = 3$ and 4 are depicted. The next and essential step is to create functions for executing basic arithmetic operations (addition, multiplication and power) within each *Galois field* (GF).

INDEX	SYMBOL	BASIS
0	0	000
1	α^0	100
2	α^1	010
3	α^2	001
4	α^3	110
5	α^4	011
6	α^5	111
7	α^6	101

Fig.1 - Mapping table between symbols and basis for $m = 3$

INDEX	SYMBOL	BASIS
0	0	0000
1	α^0	1000
2	α^1	0100
3	α^2	0010
4	α^3	0001
5	α^4	1100
6	α^5	0110
7	α^6	0011

10	α^9	0101
11	α^{10}	1110
12	α^{11}	0111
13	α^{12}	1111
14	α^{13}	1011
15	α^{14}	1001

Fig.2 - Mapping table between symbols and basis for $m = 4$

As well known, RS are nonbinary cyclic codes with symbols made up of m-bit sequences, with $m > 2$. As a special subclass of the BCH ones, the RS codes are typically expressed in the conventional form

$$(n, k) = (2^m - 1,\ 2^m - 1 - 2t)$$

where n is the codeword symbol-length, k the message symbol-length and t the maximum number of symbol errors correctable by the code (and, thus, $2t$ represents the number of parity symbols).
The encoder works in systematic form, appending the redundancy bits at the beginning of the codewords. The parity symbols are calculated by upshifting the input message polynomial by $2t$ positions and then dividing by the *generator polynomial* $g(x)$, which in turn can be easily estimated as

$$g(x) = \prod_{i=1}^{2t} (x - \alpha^i)$$

On the decoder side, after the bit-to-symbol conversion the coefficients of the *syndrome polynomial* $S(x)$ are computed as

$$S_i = r(x) \mid_{\ x=\alpha^i}$$

where i varies from 1 to $2t$ and $r(x)$ represents the received codeword polynomial, which can be thought as the sum of the transmitted codeword polynomial $c(x)$ and the error polynomial $e(x)$. The presence of errors is detected if at least one of these coefficients proves to be different from zero.
In this case, the correction procedure has to determine both location and magnitude of the errors. In order to do so, first the ***Berlekamp-Massey*** algorithm [3] is executed for estimating the *error locator polynomial* $\sigma(x)$ by iteratively computing the *discrepancy* Δ as

$$\Delta^j = \sum_i^E \sigma_i S_{j-i}$$

where σ_i is the *i-th* degree coefficient of $\sigma(x)$, j the current (out of $2t$) iteration of the algorithm and E the number of estimated errors so far.
Moreover, since the polynomial $\sigma(x)$ is related to the error locations L_i of $e(x)$ by the relation

$$\sigma(x) = \prod_{i=1}^{t} (1 - L_i x)$$

it clearly appears that L_i can be obtained as the reciprocal of the roots of $\sigma(x)$ by using the ***Chien search*** algorithm [2]. Next, the *error evaluator polynomial* $\omega(x)$ is introduced by solving the ***Key Equation***

$$[1 + S(x)]\,\sigma(x)\ mod\ x^{2t+1} = \omega(x)$$

since it is essential for the estimation of the error magnitudes M_i of $e(x)$ by means of the ***Forney algorithm*** [2], which states

$$M_i = \frac{\omega(L_i^{-1})}{\prod_{j=1 \neq i}^{t} (1 - L_j L_i^{-1})}$$

Once known the error locations and magnitudes, the error polynomial $e(x)$ can be represented as

$$e(x) = \sum_{i=1}^{t} M_i x^{L_i}$$

and so the transmitted codeword polynomial recovered as $c(x) = r(x) + e(x)$ [NB: addition and subtraction coincide for Galois fields with base 2].
Moreover, the proposed encoder / decoder implementation allows *shortened* versions of RS codes, i.e. with values of n and k respectively smaller than the conventional $2^m - 1$ and $2^m - 1 - 2t$. In this case (assuming k and t fixed) the encoder calculates the redundancy after padding $2^m - 1 - 2t - k$ extra dummy symbols to the actual input message and, then, removes them to retrieve the final shortened codeword. Specularly, the decoder reinserts these extra

Besides the encoder / decoder implementation, the MATLAB script also makes available 3 simulations to test the error correction capability of the code, always assuming binary antipodal modulation, AWGN channel and hard-decision detection:

- **1st** is a single-run simulation where it is possible to specify as parameters the extended Galois field degree *m*, the codeword length *n*, the message length *k* and the *energy-per-bit to noise-spectral-density ratio Eb/N0* over channel and to verify how many bit and symbol errors are introduced by the channel and whether the decoder is able to recover them. An example is displayed in **Fig.3**, where a typical RS(255,223) for $m = 8$ is tested. The expected symbol-error correcting capability of this code is $t = 16$ and, in fact, it proves to able to correct a 15-symbol-corrupted codeword. Moreover, this simulation run shows why RS codes are particularly useful in case of *burst noise*. A symbol error does not depend on how many bits are wrong within it and this is why $\xi > t$ corrupted bits are still recoverable if they all occur in no more than *t* symbols [1].

```
PARAMETERS:
- GF = 2^8
- (n,k,t) = (255,223,16)
- Eb/N0 = 7.5 dB

RESULTS:
- Errors before decoding : 23 out of 2040 bits (15 out of 255 symbols)
- Errors after decoding : 0 out of 1784 bits
```

Fig.3 - Single-run simulation in MATLAB

- **2nd** is a bit-error-rate (BER) vs *Eb/N0* simulation as a function of the maximum number of recoverable errors *t*, which can be derive as

$$t = \frac{n-k}{2}$$

The results (with *n* fixed to its maximum value) are displayed in **Fig.4** and **Fig.5** for $m = 4$ and 5 respectively. The higher *t* (i.e. the lower *k*) the better the coding performances in terms of BER at the expense of an increasing redundancy (or analogously, of a decreasing code-rate *Rc*, defined as *k*/*n*). Moreover, by comparing the two plots it can be highlighted a very important theoretical aspect of RS codes, i.e. that a higher *m* for almost corresponding values of *Rc* (curves with the same colour and marker) produces more sloping trends (i.e. better error performance). However, since it can be demonstrated that decoder complexity exponentially grows with *m*, a practical design and use of RS codes (e.g. on MCUs) shall limit this value.

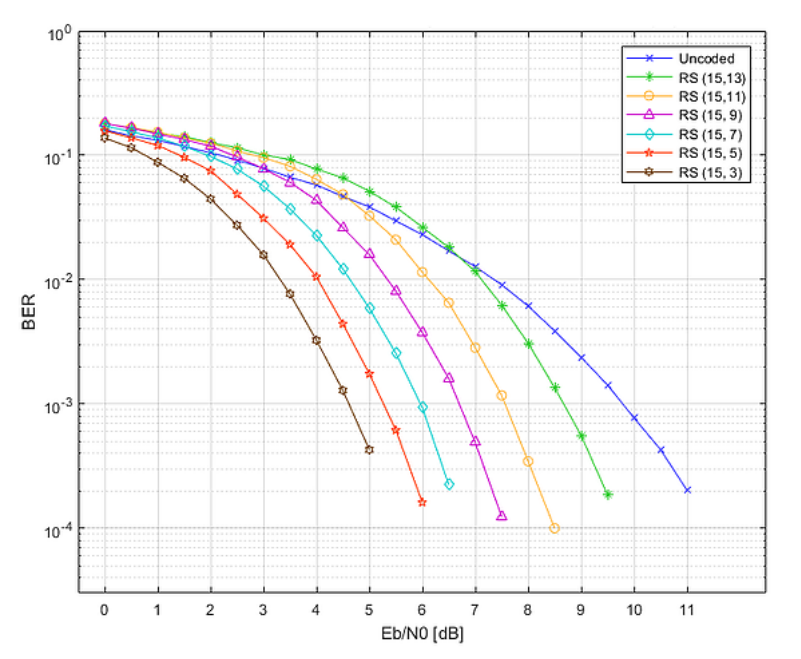
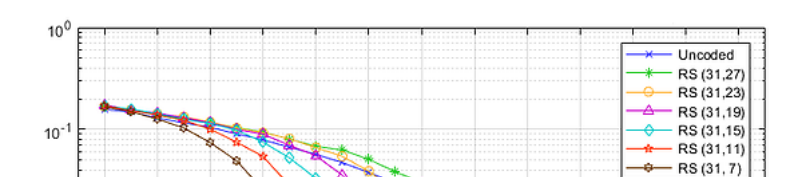


Fig.4 - Coding performance as a function of *t* for $m = 4$



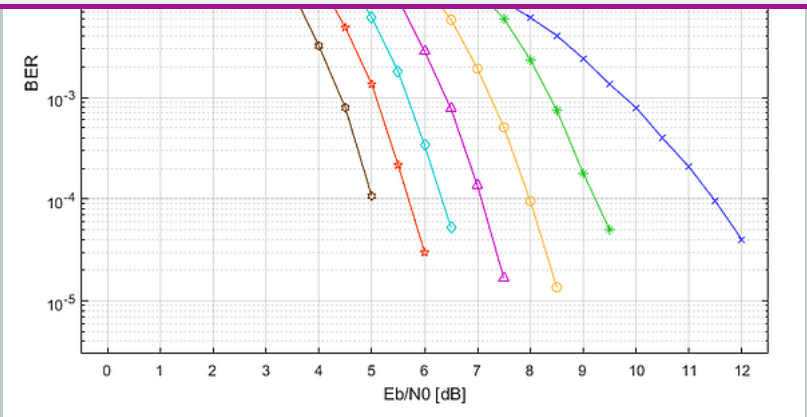


Fig.5 - Coding performance as a function of t for $m = 5$

- **3rd** is a bit-error-rate (BER) vs *Eb/N0* simulation among different shortened codes keeping the number of redundancy symbols (i.e. $2t$) constant. The results (having fixed $m = 6$ and $t = 8$) are displayed in **Fig.6**. Again, lower values of *Rc* (corresponding to lower n and k in this case) assure better BER performance.

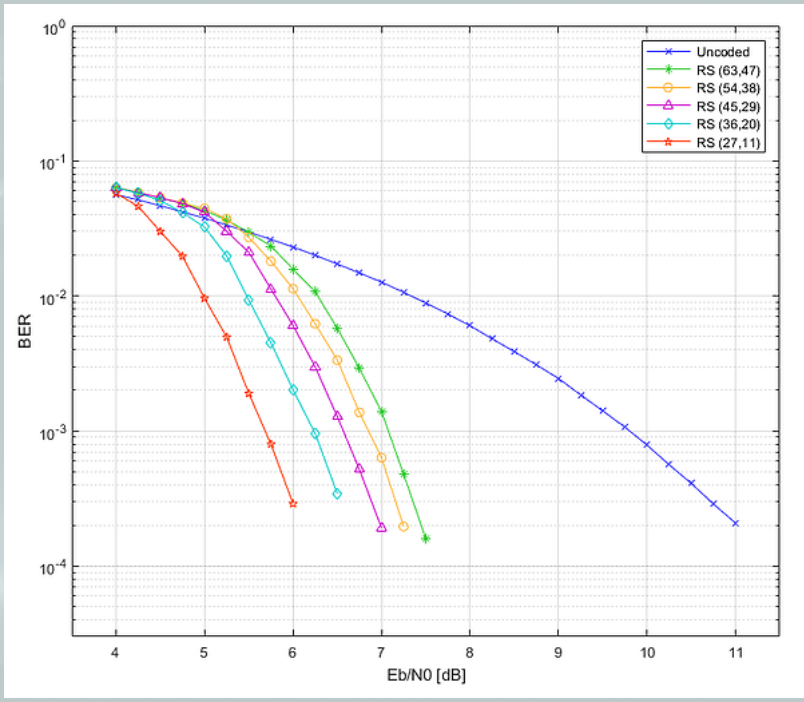


Fig.6 - Coding performance as a function of the shortened length n for $t = 8$ and $m = 6$

After the MATLAB implementation and validation, the encoding and decoding algorithms have been adapted for C/C++ language (only for m = 4 and 8), making it much more efficient and compatible for microcontroller (MCU) operation. In fact, besides other intrinsic advantages, while the MATLAB version works on bits where each of them is by default represented as a double type (i.e. 64 bits), the C/C++ version works on bytes allowing to minimize the memory allocation.

Again, a validation of the code error-correcting capability is provided by simulating a simple *binary symmetric channel* (BSC) and allowing to set as parameter the probability of error *Peb*. An example of this simulation is shown in **Fig.7**, where a source stream (SRC) of 80 bytes for is randomly generated, encoded (ENC) according to the selected m and n parameters, corrupted (ERR) with probability *Peb* and decoded (DEC). The content of these streams are displayed on the left side of the figure in hexadecimal format. Finally, a check of the number of errors before and after the RS decoder is provided.

```
* GF = 2^8
* (n,k,t) = (100,80,10)
* Nmsg = 1
* Peb = 0.010

SRC (80 bytes) :       CE 28 6B 9A 5B A0 E7 BE 60 77
                       E7 6A 9B 48 33 93 3E 4B AA EA
                       41 63 36  3 FC 5F FF B7 F5 79
                       DF C7 F7 23 17 C1 64  4 40 28
```

Fig.7 - Single-run simulation in C/C++

Below there are the links through which the aforementioned codes can be free downloaded. For further details and explanations about the practical implementation of the Reed-Solomon encoder / decoder pair take a look at the comments within these files.

- **Download MATLAB project** : *RS.m*
- **Download C/C++ project** : *RS.cpp*

WARNING#1: The proposed MATLAB simulations may take several hours to be executed, depending on PC performance.
WARNING#2: The *RS.m* project has been developed in MATLAB R2017a. Therefore, the correct operation of the script may not be assured with different software versions.

Last update : 06/11/2018

References

[1] B. Sklar and P.K. Ray, *Digital Communications*, Chapter 8, Pearson Education, 2012
[2] A.Brown, *Implementing Reed-Solomon*, Duke University, 2011
[3] J.Sankaran, *Reed-Solomon Decoder: TMS320C64x Implementation*, Texas Instrument, 2000
[4] Wikipedia - *Reed–Solomon error correction* (*link/a*)

The author graduated in Electronics & Telecommunciation Enginnering in 2016 at University of Bologna, Italy.

Contacts

Mail
filippo.valmori@gmail.com
LinkedIn
https://www.linkedin.com/in/valmorif