WIKIVERSITY

# Reed–Solomon codes for coders

Error correcting codes are a signal processing technique to correct errors. They are nowadays ubiquitous, such as in communications (mobile phone, internet), data storage and archival (hard drives, optical discs CD/DVD/BluRay, archival tapes), warehouse management (barcodes) and advertisement (QR codes).

Reed–Solomon error correction is a specific type of error correction code. It is one of the oldest but it is still widely used, as it is very well defined and several efficient algorithms are now available under the public domain.

Usually, error correction codes are hidden and most users do not even know about them, nor when they are used. Yet, they are a critical component for some applications to be viable, such as communication or data storage. Indeed, a hard drive that would randomly lose data every few days would be useless, and a phone being able to call only on days with a cloud-less weather would be seldom used. Using error correction codes allows to recover a corrupted message into the full original message.

Barcodes and QR codes are interesting applications to study, as they have the specificity of displaying visually the error correction code, rendering these codes readily accessible to the curious user.

In this essay, we will attempt to introduce the principles of Reed–Solomon codes from the point of view of a programmer rather than a mathematician, which means that we will focus more on the practice than the theory, although we will also explain the theory, but only the necessary knowledge for intuition and implementation. Notable references in the domain will be provided, so that the interested reader can dig deeper into the mathematical theory at will. We will provide real-world examples taken from the popular QR code barcode system as well as working code samples. We chose to use Python for the samples (mainly because it looks pretty and similar to pseudocode), but we will try to explain any non-obvious features for those who are not familiar with it. The mathematics involved is advanced in the sense that it is not usually taught below the university level, but it should be understandable to someone with a good grasp of high-school algebra.

We will first gently introduce the intuitions behind error correction codes principles, then in a second section we will introduce the structural design of QR codes, in other words how information is stored in a QR code and how to read and produce it, and in a third section we will study error correction codes via the implementation of a Reed–Solomon decoder, with a quick introduction of the bigger BCH codes family, in order to reliably read damaged QR codes.

Note for the curious readers that extended information can be found in the appendix and on the discussion page.

## Principles of error correction

Before detailing the code, it might be useful to understand the intuition behind error correction. Indeed, although error correcting codes may seem daunting mathematically-wise, most of the mathematical operations are high school grade (with the exception of Galois Fields, but which are in fact easy and common for any programmer: it's simply doing operations on integers modulo a number). However, the complexity of the mathematical ingenuity behind error correction codes hide the quite intuitive goal and mechanisms at play.

Error correcting codes might seem like a difficult mathematical concept, but they are in fact based on an intuitive idea with an ingenious mathematical implementation: **let's make the data structured, in a way that we can "guess" what the data was if it gets corrupted, just by "fixing" the structure**. Mathematically-wise, we use polynomials from the Galois Field to implement this structure.

Let's take a more practical analogy: let's say you want to communicate messages to someone else, but these messages can get corrupted along the way. The main insight of error correcting codes is that, **instead of using a whole dictionary of words, we can use a smaller set of carefully selected words, a "reduced dictionary", so that each word is as different as any other**. This way, when we get a message, we just have to lookup inside our reduced dictionary to **1) detect** which words are corrupted (as they are not in our reduced dictionary); **2) correct** corrupted words by finding the most similar word in our dictionary.

Let's take a simple example: we have a reduced dictionary with only three words of 4 letters: `this`, `that` and `corn`. Let's say we receive a corrupted word: `co**`, where `*` is an erasure. Since we have only 3 words in our dictionary, we can easily compare our received word with our dictionary to find the word that is the closest. In this case, it's `corn`. Thus the missing letters are `rn`.

Now let's say we receive the word `th**`. Here the problem is that we have two words in our dictionary that match the received word: `this` and `that`. In this case, we cannot be sure which one it is, and thus we cannot decode. This means that our dictionary is not very good, and we should replace `that` with another more different word, such as `dash` to maximize the difference between each word. This difference, or more precisely the minimum number of different letters between any 2 words of our dictionary, is called the **maximum Hamming distance** of our dictionary. Making sure that any 2 words of the dictionary share a minimum number of letters at the same position is called **maximum separability**.

The same principle is used for most error correcting codes: we generate a reduced dictionary containing only words with maximum separability (we will detail more how to do that in the third section), and then we communicate only with the words of this reduced dictionary. What Galois Fields provide is the structure (ie, reduced dictionary basis), and Reed–Solomon is a way to automatically create a suitable structure (make a reduced dictionary with maximum separability tailored for a dataset), as well as provide the automated methods to detect and correct errors (ie, lookups in the reduced dictionary). To be more precise, Galois Fields are the structure (thanks to their cyclic nature, the modulo an integer) and Reed–Solomon is the codec (encoder/decoder) based on Galois Fields.

If a word gets corrupted in the communication, that's no big deal since we can easily fix it by looking inside our dictionary and find the closest word, which is probably the correct one (there is however a chance of choosing a wrong one if the input message is too heavily corrupted, but the probability is very small). Also, the longer our words are, the more separable they are, since more characters can be corrupted without any impact.

The simplest way to generate a dictionary of maximally separable words is to make words longer than they really are.

Let's take again our example:

```
t h i s
t h a t
c o r n
```

Append a unique set of characters so that there are no duplicated characters at any of the appended positions, and add one more word to help with the explanation:

```
t h i s a b c d
t h a t b c d e
c o r n c d e f
```

Note that each word in this dictionary differs from every other word by at least 6 characters, so the distance is 6. This allows up to 5 errors in known positions (which are called erasures), or 3 errors in unknown positions, to be corrected.

Assume that 4 erasures occur:

```
t * * * a b * d
```

Then a search of the dictionary for the 4 non-erased characters can be done to find the only entry that matches those 4 characters, since the distance is 5. Here it gives: `t h i s a b c d`

Assume that 2 errors occur as in one of these patterns:

```
t h o s b c d e
```

The issue here is the location of the errors is unknown. The erasures might have happened in any 2 positions meaning that there are $\binom{8}{6}$ or 28 possible sub-sets of 6 characters:

```
t h o s b c * *
t h o s b * d *
t h o s b * * e
...
```

If we do a dictionary search on each of these sub-sequences, we find that there is only one sub-set that matches 6 characters. `t h * * b c d e` matches `t h a t b c d e`.

With these examples, one can see the advantage of redundancy in recovering lost information: redundant characters help you recover your original data. The previous examples show how a crude error correcting scheme could work. Reed–Solomon's core idea is similar, append redundant data to a message based on Galois Field mathematics. The original error correcting decoder was similar to the error example above,

search sub-sets of a received message that correspond to a valid message, and choose the one with the most matches as the corrected message. This isn't practical for larger messages, so mathematical algorithms were developed to perform error correction in a reasonable time.

# QR code structure

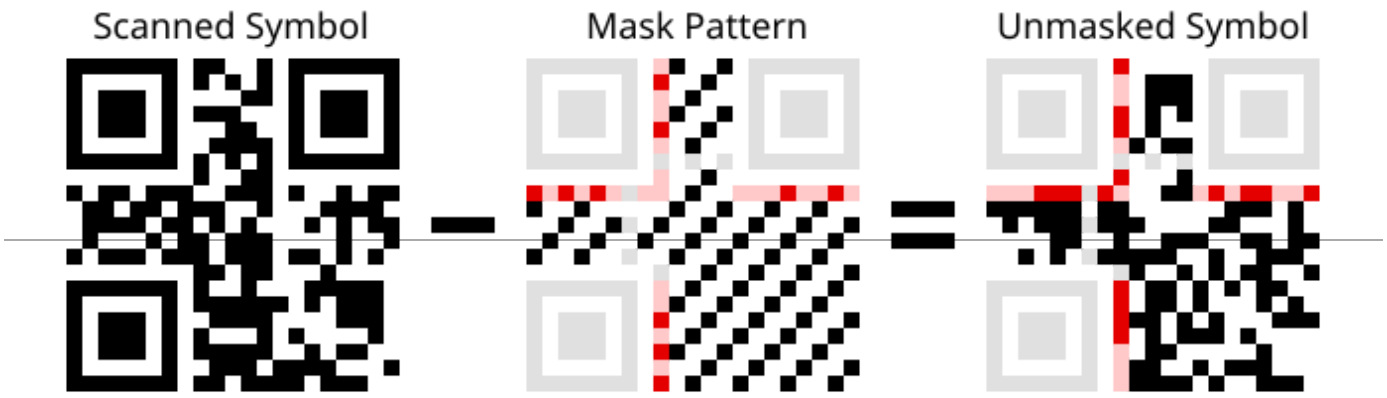
This section introduces the structure of QR codes, which is how data is stored in a QR code. The information in this section is deliberately incomplete. Only the most common features of the small 21×21 size symbols (also known as version 1) are presented here, but see the appendix for additional information.

Here is a QR symbol that will be used as an example. It consists of dark and light squares, known as modules in the barcoding world. The three square locator patterns in the corners are a visually distinctive feature of QR symbols.

## Masking

A masking process is used to avoid features in the symbol that might confuse a scanner, such as misleading shapes that look like the locator patterns and large blank areas. Masking inverts certain modules (white becomes black and black becomes white) while leaving others alone.

In the diagram below, the red areas encode format information and use a fixed masking pattern. The data area (in black and white) is masked with a variable pattern. When the code is created, the encoder tries a number of different masks and chooses the one that minimizes undesirable features in the result. The chosen mask pattern is then indicated in the format information so that the decoder knows which one to use. The light gray areas are fixed patterns which do not encode any information. In addition to the obvious locator patterns, there are also timing patterns which contain alternating light and dark modules.




The masking transformation is easily applied (or removed) using the exclusive-or operation (denoted by a caret ^ in many programming languages). The unmasking of the format information is shown below. Reading counter-clockwise around the upper-left locator pattern, we have the following sequence of bits. White modules represent 0 and black modules represent 1.

```
Input        101101101001011
Mask       ^ 101010000010010
Output       000111101011001
```

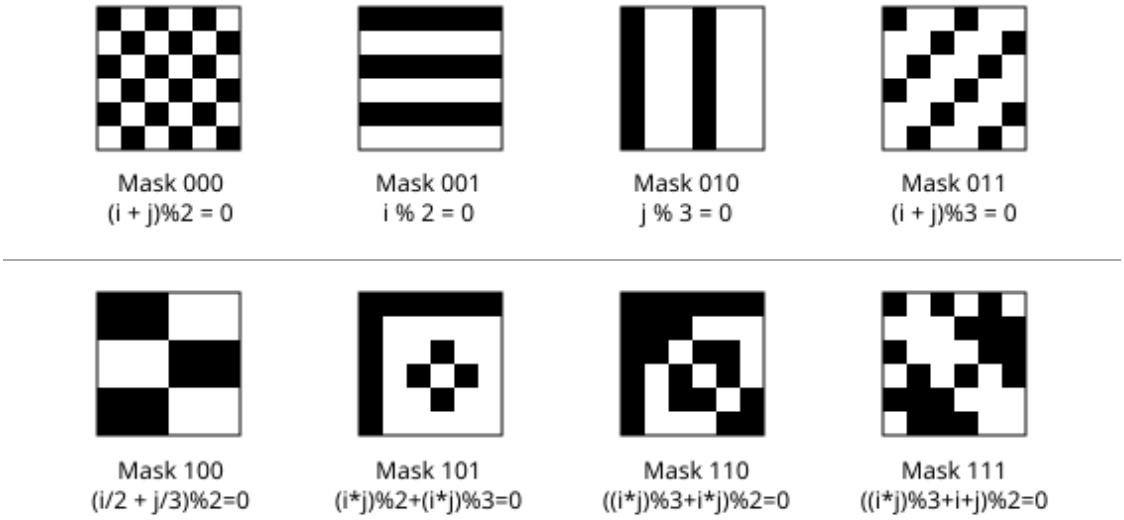
## Formatting information

There are two identical copies of the formatting information, so that the symbol can still be decoded even if it is damaged. The second copy is broken in two pieces and placed around the other two locators, and is read in a clockwise direction (upwards in the lower-left corner, then left-to-right in the upper-right corner).

The first two bits of formatting information give the error correction level used for the message data. A QR symbol this size contains 26 bytes of information. Some of these are used to store the message and some are used for error correction, as shown in the table below. The left-hand column is simply a name given to that level.

| Error Correction Level | Level Indicator | Error Correction Bytes | Message Data Bytes |
|---|---|---|---|
| L | 01 | 7 | 19 |
| M | 00 | 10 | 16 |
| Q | 11 | 13 | 13 |
| H | 10 | 17 | 9 |

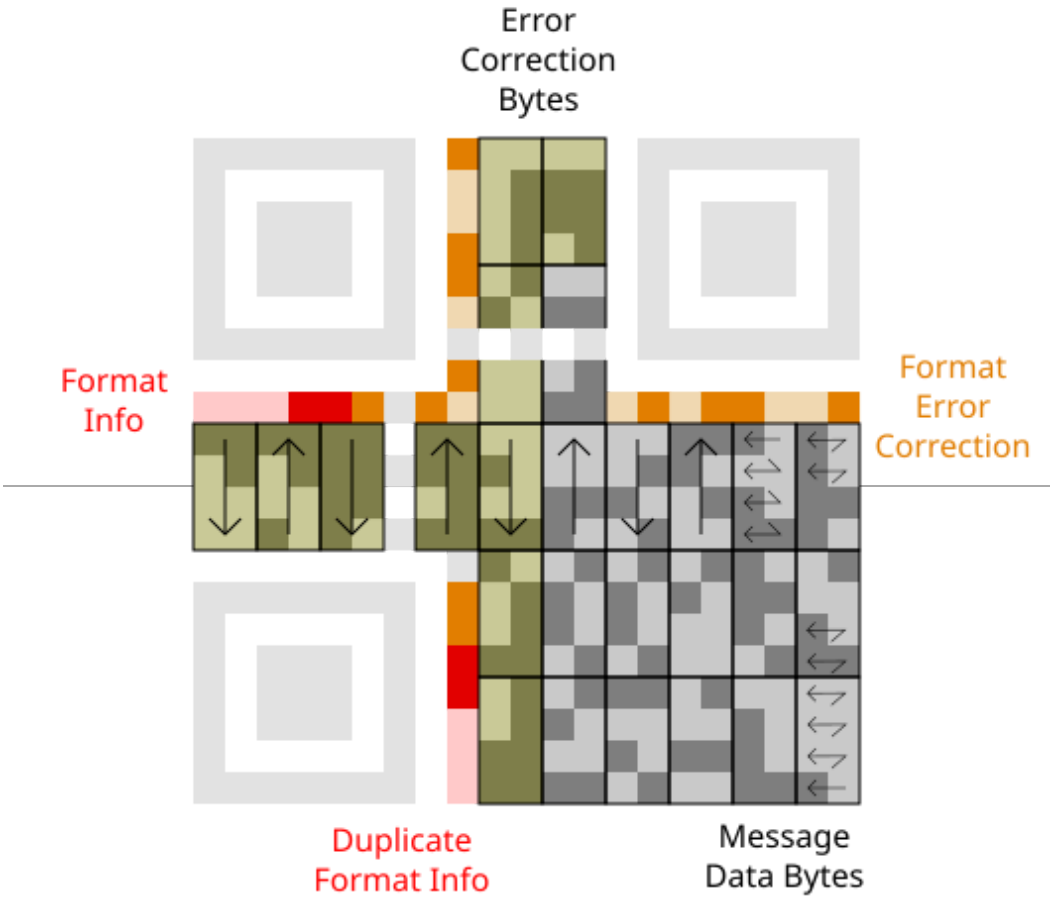
The next three bits of format information select the masking pattern to be used in the data area. The patterns are illustrated below, including the mathematical formula that tells whether a module is black (i and j are the row and column numbers, respectively, and start with 0 in the upper-left hand corner).




The remaining ten bits of formatting information are for correcting errors in the format itself. This will be explained in a later section.

## Message data

Here is a larger diagram showing the "unmasked" QR code. Different regions of the symbol are indicated, including the boundaries of the message data bytes.




Data bits are read starting from the lower-right corner and moving up the two right-hand columns in a zig-zag pattern. The first three bytes are 01000000 11010010 01110101. The next two columns are read in a downward direction, so the next byte is 01000111. Upon reaching the bottom, the two columns after that are read upward. Proceed in this up-and-down fashion all the way to the left side of the symbol (skipping over the timing pattern where necessary). Here is the complete message in hexadecimal notation.

> Message data bytes: 40 d2 75 47 76 17 32 06 27 26 96 c6 c6 96 70 ec
> Error correction bytes: bc 2a 90 13 6b af ef fd 4b e0

## Decoding

The final step is to decode the message bytes into something readable. The first four bits indicate how the message is encoded. QR codes use several different encoding schemes, so that different kinds of messages can be stored efficiently. These are summarized in the table below. After the mode indicator is a length field, which tells how many characters are stored. The size of the length field depends on the specific encoding.

| Mode Name | Mode Indicator | Length Bits | Data Bits |
|---|---|---|---|
| Numeric | 0001 | 10 | 10 bits per 3 digits |
| Alphanumeric | 0010 | 9 | 11 bits per 2 characters |
| Byte | 0100 | 8 | 8 bits per character |
| Kanji | 1000 | 8 | 13 bits per character |

(The length field sizes above are valid only for smaller QR codes.)

Our sample message starts with 0100 (hex 4), indicating that there are 8 bits per character. The next 8 bits (hex 0d) are the length field, 13 in decimal notation. The bits after that can be rearranged in bytes representing the actual characters of the message: 27 54 77 61 73 20 62 72 69 6c 6c 69 67, and additionally 0e c. The first two, hex 27 and 54 are the ASCII codes for apostrophe and T. The whole message is "'Twas brillig" (from w:Jabberwocky#Lexicon).

After the last of the data bits is another 4-bit mode indicator. It can be different from the first one, allowing different encodings to be mixed within the same QR symbol. When there is no more data to store, the special end-of-message code 0000 is given. (Note that the standard allows the end-of-message code to be omitted if it wouldn't fit in the available number of data bytes.)

At this point, we know how to decode, or read, a whole QR code. However, in real life conditions, a QR code is rarely whole: usually, it is scanned via a phone's camera, under potentially poor lighting conditions, or on a scratched surface where part of the QR code was ripped, or on a stained surface, etc.

To make our QR code decoder **reliable**, we need to be able to **correct** errors. The next part of this article will describe how to correct errors, by constructing a BCH decoder, and more specifically a Reed–Solomon decoder.

# BCH codes

In this section, we introduce a general class of error correction codes: the BCH codes, the parent family of modern Reed–Solomon codes, and the basic detection and correction mechanisms.

The formatting information is encoded with a BCH code which allows a certain number of bit-errors to be detected and corrected. BCH codes are a generalization of Reed–Solomon codes (modern Reed–Solomon codes are BCH codes). In the case of QR codes, the BCH code used for the format information is much simpler than the Reed–Solomon code used for the message data, so it makes sense to start with the BCH code for format information.

## BCH error detection

The process for checking the encoded information is similar to long division, but uses exclusive-or instead of subtraction. The format code should produce a remainder of zero when it is "divided" by the so-called generator of the code. QR format codes use the generator 10100110111. This process is demonstrated for the format information in the example code (000111101011001) below.

```
              0001110100110111 )
               000111101011001
                ^ 101001101110
                  010100110111
                   ^ 10100110111
                     00000000000
```

Here is a Python function which implements this calculation.

```
def qr_check_format(fmt):
   g = 0x537 # = 0b10100110111 in python 2.6+
   for i in range(4,-1,-1):
      if fmt & (1 << (i+10)):
         fmt ^= g << i
   return fmt
```

*Python note:* The `range` function may not be clear to non-Python programmers. It produces a list of numbers counting down from 4 to 0 (the code has "-1" because the interval returned by "range" includes the start but not the end value). In C-derived languages, the for loop might be written as `for (i = 4; i >= 0; i--)`; in Pascal-derived languages, `for i := 4 downto 0`.

*Python note 2:* The `&` operator performs bitwise and, while `<<` is a left bit-shift. This is consistent with C-like languages.

This function can also be used to encode the 5-bit format information.

```
encoded_format = (format<<10) + qr_check_format(format<<10)
```

Readers may find it an interesting exercise to generalize this function to divide by different numbers. For example, larger QR codes contain six bits of version information with 12 error correction bits using the generator 1111100100101.

In mathematical formalism, these binary numbers are described as polynomials whose coefficients are integers mod 2. Each bit of the number is a coefficient of one term. For example:

$$10100110111 = 1\,x^{10} + 0\,x^9 + 1\,x^8 + 0\,x^7 + 0\,x^6 + 1\,x^5 + 1\,x^4 + 0\,x^3 + 1\,x^2 + 1\,x + 1 = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$$

If the remainder produced by `qr_check_format` is not zero, then the code has been damaged or misread. The next step is to determine which format code is most likely the one that was intended (ie, lookup in our reduced dictionary).

## BCH error correction

Although sophisticated algorithms for decoding BCH codes exist, they are probably overkill in this case. Since there are only 32 possible format codes, it's much easier to simply try each one and pick the one that has the smallest number of bits different from the code in question (the number of different bits is known as the Hamming distance). This method of finding the closest code is known as exhaustive search, and is possible only because we have very few codes (a code is a valid message, and here there are only 32, all other binary numbers aren't correct).

(Note that Reed–Solomon is also based on this principle, but since the number of possible codewords is simply too big, we can't afford to do an exhaustive search, and that's why clever but complicated algorithms have been devised, such as Berlekamp-Massey.)

```
def hamming_weight(x):
   weight = 0
   while x > 0:
      weight += x & 1
      x >>= 1
   return weight

def qr_decode_format(fmt):
   best_fmt = -1
   best_dist = 15
   for test_fmt in range(0,32):
      test_code = (test_fmt<<10) ^ qr_check_format(test_fmt<<10)
      test_dist = hamming_weight(fmt ^ test_code)
      if test_dist < best_dist:
         best_dist = test_dist
         best_fmt = test_fmt
      elif test_dist == best_dist:
         best_fmt = -1
   return best_fmt
```

The function `qr_decode_format` returns -1 if the format code could not be unambiguously decoded. This happens when two or more format codes have the same distance from the input.

To run this code in Python, first start IDLE, Python's integrated development environment. You should see a version message and the interactive input prompt `>>>`. Open a new window, copy the functions `qr_check_format`, `hamming_weight`, and `qr_decode_format` into it, and save as `qr.py`. Return to the prompt and type the lines following `>>>` below.

```
>>> from qr import *
>>> qr_decode_format(int("000111101011001",2))  # no errors
3
>>> qr_decode_format(int("111111101011001",2))  # 3 bit-errors
3
>>> qr_decode_format(int("111011101011001",2))  # 4 bit-errors
-1
```

You can also start Python by typing `python` at a command prompt.

In the next sections, we will study Finite Field Arithmetics and Reed–Solomon code, which is a subtype of BCH codes. The basic idea (ie, **using a limited words dictionary with maximum separability**) is the same, but since we will encode longer words (256 bytes instead of 2 bytes), with more symbols available (encoded on all 8bits, thus 256 different possible values), we cannot use this naive, exhaustive approach, because it would take way too much time: we need to use cleverer algorithms, and Finite Field mathematics will help us do just that, by giving us a **structure**.

# Finite field arithmetic

## Introduction to mathematical fields

Before discussing the Reed–Solomon codes used for the message, it will be useful to introduce a bit more mathematics.

We'd like to define addition, subtraction, multiplication, and division for 8-bit bytes and always produce 8-bit bytes as a result, so as to avoid any overflow. Naively, we might attempt to use the normal definitions for these operations, and then mod by 256 to keep results from overflowing. And this is exactly what we will be doing, and is what is called a Galois Field 2^8. You can easily imagine why it works for everything, except for division: what is 5/4?

Here's a brief introduction to Galois Fields: a finite field is a set of numbers, and a field needs to have six properties governing addition, subtraction, multiplication and division: Closure, Associative, Commutative, Distributive, Identity and Inverse. More simply put, using a field allows us to study the relationship between numbers of this field, and apply the result to any other field that follows the same properties. For example, the set of reals $\mathbb{R}$ is a field. In other words, mathematical fields studies the structure of a set of numbers.

However, integers $\mathbb{Z}$ aren't a field, because as we said above, not all divisions are defined (such as 5/4), which violates multiplicative inverse property (x such that x*4=5 does not exist). One simple way to fix that is to do modulo using a prime number, such as 257, or any positive integer power of a prime number: in this way, we are guaranteed that x*4=5 exists since we will just wrap around. $\mathbb{Z}$ modulo any prime number is called a Galois Field, and modulo 2 is an extra interesting Galois Field: since an 8-bit string can express a total of 256 = 2^8 values, we say that we use a Galois Field of 2^8, or GF(2^8). In spoken language, 2 is the characteristic of the field, 8 is the exponent, and 256 is the field's cardinality. More information on finite fields can be found here (http://research.swtch.com/field).

Here we will define the usual mathematical operations that you are used to doing with integers, but adapted to GF(2^8), which is basically doing usual operations but modulo 2^8.

Another way to consider the link between GF(2) and GF(2^8) is to think that GF(2^8) represents a polynomial of 8 binary coefficients. For example, in GF(2^8), 170 is equivalent to `10101010 = 1*x^7 + 0*x^6 + 1*x^5 + 0*x^4 + 1*x^3 + 0*x^2 + 1*x + 0 = x^7 + x^5 + x^3 + x`. Both representations are equivalent, it's just that in the first case, 170, the representation is decimal, and in the other case it's binary, which can be thought as representing a polynomial by convention (only used in GF(2^p) as explained here). The latter is often the representation used in academic books and in hardware

implementations (because of logical gates and registers, which work at the binary level). For a software implementation, the decimal representation can be preferred for clearer and more close-to-the-mathematics code (this is what we will use for the code in this tutorial, except for some examples that will use the binary representation).

In any case, try to not confuse the polynomial representing a single GF(2^p) symbol (each coefficient is a bit/boolean: either 0 or 1), and the polynomial representing a list of GF(2^p) symbols (in this case the polynomial is equivalent to the message+RScode, each coefficient is a value between 0 and 2^p and represent one character of the message+RScode). We will first describe operations on single symbol, then polynomial operations on a list of symbols.

## Addition and Subtraction

Both addition and subtraction are replaced with exclusive-or in Galois Field base 2. This is logical: addition modulo 2 is exactly like an XOR, and subtraction modulo 2 is exactly the same as addition modulo 2. This is possible because additions and subtractions in this Galois Field are carry-less.

Thinking of our 8-bit values as polynomials with coefficients mod 2:

```
    0101 + 0110 = 0101 - 0110 = 0101 XOR 0110 = 0011
```

The same way (in binary representation of two single GF(2^8) integers):

$$(x^2 + 1) + (x^2 + x) = 2\,x^2 + x + 1 = 0\,x^2 + x + 1 = x + 1$$

Since `(a ^ a) = 0`, every number is its own opposite, so ($x$ - $y$) is the same as ($x$ + $y$).

Note that in books, you will find additions and subtractions to define some mathematical operations on GF integers, but in practice, you can just XOR (as long as you are in a Galois Field base 2; this is not true in other fields).

Here is the equivalent Python code:

```
def gf_add(x, y):
    return x ^ y

def gf_sub(x, y):
    return x ^ y # in binary galois field, subtraction is just the
same as addition (since we mod 2)
```

## Multiplication

Multiplication is likewise based on polynomial multiplication. Simply write the inputs as polynomials and multiply them out using the distributive law as normal. As an example, 10001001 times 00101010 is calculated as follows.

$$(x^7 + x^3 + 1)\,(x^5 + x^3 + x) = x^7\,(x^5 + x^3 + x) + x^3\,(x^5 + x^3 + x) + 1\,(x^5 + x^3 + x)$$

$= x^{12} + x^{10} + 2\,x^8 + x^6 + x^5 + x^4 + x^3 + x$
$= x^{12} + x^{10} + x^6 + x^5 + x^4 + x^3 + x$

The same result can be obtained by a modified version of the standard grade-school multiplication procedure, in which we replace addition with exclusive-or.

```
       10001001
*      00101010
      10001001
^   10001001
^ 10001001
  1010001111010
```

Note: the XOR multiplication here is carry-less! If you do it with-carry, you will get the wrong result 1011001111010 with the extra term $x^9$ instead of the correct result 1010001111010.

Here is a Python function which implements this polynomial multiplication on single GF(2^8) integers.

Note: this function (and some other functions below) use a lot of bitwise operators such as >> and <<, because they are both faster and more concise to do what we want to do. These operators are available in most languages, they are not specific to Python, and you can get more information about them here (https://wiki.python.org/moin/BitwiseOperators).

```
def cl_mul(x,y):
    '''Bitwise carry-less multiplication on integers'''
    z = 0
    i = 0
    while (y>>i) > 0:
        if y & (1<<i):
            z ^= x<<i
        i += 1
    return z
```

Of course, the result no longer fits in an 8-bit byte (in this example, it is 13 bits long), so we need to perform one more step before we are finished. The result is reduced modulo 100011101 (the choice of this number is explained below the code), using the long division process described previously. In this instance, this is called "modular reduction", because basically what we do is that we divide and keep only the remainder, using a modulo. This produces the final answer 11000011 in our example.

```
  1010001111010
^ 100011101
  0010110101010
  ^ 100011101
    00111011110
    ^ 100011101
      011000011
```

Here is the Python code to do the whole Galois Field multiplication with modular reduction:

```
def gf_mult_noLUT(x, y, prim=0):
    '''Multiplication in Galois Fields without using a precomputed
look-up table (and thus it's slower)
    by using the standard carry-less multiplication + modular
reduction using an irreducible prime polynomial'''

    ### Define bitwise carry-less operations as inner functions ###
    def cl_mult(x,y):
        '''Bitwise carry-less multiplication on integers'''
        z = 0
        i = 0
        while (y>>i) > 0:
            if y & (1<<i):
                z ^= x<<i
            i += 1
        return z

    def bit_length(n):
        '''Compute the position of the most significant bit (1) of
an integer. Equivalent to int.bit_length()'''
        bits = 0
        while n >> bits: bits += 1
        return bits

    def cl_div(dividend, divisor=None):
        '''Bitwise carry-less long division on integers and returns
the remainder'''
        # Compute the position of the most significant bit for each
integers
        dl1 = bit_length(dividend)
        dl2 = bit_length(divisor)
        # If the dividend is smaller than the divisor, just exit
        if dl1 < dl2:
            return dividend
        # Else, align the most significant 1 of the divisor to the
most significant 1 of the dividend (by shifting the divisor)
        for i in range(dl1-dl2,-1,-1):
            # Check that the dividend is divisible (useless for the
first iteration but important for the next ones)
            if dividend & (1 << i+dl2-1):
                # If divisible, then shift the divisor to align the
most significant bits and XOR (carry-less subtraction)
                dividend ^= divisor << i
        return dividend

    ### Main GF multiplication routine ###

    # Multiply the gf numbers
```

```
    result = cl_mult(x,y)
    # Then do a modular reduction (ie, remainder from the division)
with an irreducible primitive polynomial so that it stays inside GF
bounds
    if prim > 0:
        result = cl_div(result, prim)

    return result
```

Result:

```
>>> a = 0b10001001
>>> b = 0b00101010
>>> print bin(gf_mult_noLUT(a, b, 0)) # multiplication only
0b1010001111010
>>> print bin(gf_mult_noLUT(a, b, 0x11d)) # multiplication +
modular reduction
0b11000011
```

Why mod 100011101 (in hexadecimal: 0x11d)? The mathematics is a little complicated here, but in short, 100011101 represents an 8th degree polynomial which is "irreducible" (meaning it can't be represented as the product of two smaller polynomials). This number is called a **primitive polynomial** or irreducible polynomial, or prime polynomial (we will mainly use this latter name for the rest of this tutorial). This is necessary for division to be well-behaved, which is to stay in the limits of the Galois Field, but without duplicating values. There are other numbers we could have chosen, but they're all essentially the same, and 100011101 (0x11d) is a common primitive polynomial for Reed–Solomon codes. If you are curious to know how to generate those prime polynomials, please see the appendix.

Additional infos on the prime polynomial (you can skip): What is a prime polynomial? It is the equivalent of a prime number, but in the Galois Field. Remember that a Galois Field uses values that are multiples of 2 as the generator. Of course, a prime number cannot be a multiple of two in standard arithmetics, but in a Galois Field it is possible. Why do we need a prime polynomial? Because to stay in the bound of the field, we need to compute the modulo of any value above the Galois Field. Why don't we just modulo with the Galois Field size? Because we will end up with lots of duplicate values, and we want to have as many unique values as possible in the field, so that a number always has one and only projection when doing a modulo or a XOR with the prime polynomial.

Note for the interested reader: as an example of what you can achieve with clever algorithms, here is another way to achieve multiplication of GF numbers in a more concise and faster way, using the Russian Peasant Multiplication algorithm (http://www.cut-the-knot.org/Curriculum/Algebra/PeasantMultiplication.shtml):

```
def gf_mult_noLUT(x, y, prim=0, field_charac_full=256,
carryless=True):
    '''Galois Field integer multiplication using Russian Peasant
Multiplication algorithm (faster than the standard multiplication +
modular reduction).
```

```
    If prim is 0 and carryless=False, then the function produces
the result for a standard integers multiplication (no carry-less
arithmetics nor modular reduction).'''
    r = 0
    while y: # while y is above 0
        if y & 1: r = r ^ x if carryless else r + x # y is odd,
then add the corresponding x to r (the sum of all x's corresponding
to odd y's will give the final product). Note that since we're in
GF(2), the addition is in fact an XOR (very important because in
GF(2) the multiplication and additions are carry-less, thus it
changes the result!).
        y = y >> 1 # equivalent to y // 2
        x = x << 1 # equivalent to x*2
        if prim > 0 and x & field_charac_full: x = x ^ prim # GF
modulo: if x >= 256 then apply modular reduction using the
primitive polynomial (we just subtract, but since the primitive
number can be above 256 then we directly XOR).

    return r
```

Note that using this last function with parameters prim=0 and carryless=False will return the result for a standard integers multiplication (and thus you can see the difference between carryless and with-carry addition and its impact on multiplication).

## Multiplication with logarithms

The procedure described above is not the most convenient way to implement Galois field multiplication. Multiplying two numbers takes up to eight iterations of the multiplication loop, followed by up to eight iterations of the division loop. However, we can multiply with no looping by using lookup tables. One solution would be to construct the entire multiplication table in memory, but that would require a bulky 64k table. The solution described below is much more compact.

First, notice that it is particularly easy to multiply by 2=00000010 (by convention, this number is referred to as **α** or the **generator number**): simply left-shift by one place, then exclusive-or with the modulus 100011101 if necessary (why xor is sufficient for taking the mod in this case is an exercise left to the reader). Here are the first few powers of α.

| | | | |
|---|---|---|---|
| $\alpha^0$ = 00000001 | $\alpha^4$ = 00010000 | $\alpha^8$ = 00011101 | $\alpha^{12}$ = 11001101 |
| $\alpha^1$ = 00000010 | $\alpha^5$ = 00100000 | $\alpha^9$ = 00111010 | $\alpha^{13}$ = 10000111 |
| $\alpha^2$ = 00000100 | $\alpha^6$ = 01000000 | $\alpha^{10}$ = 01110100 | $\alpha^{14}$ = 00010011 |
| $\alpha^3$ = 00001000 | $\alpha^7$ = 10000000 | $\alpha^{11}$ = 11101000 | $\alpha^{15}$ = 00100110 |

If this table is continued in the same fashion, the powers of α do not repeat themselves until $\alpha^{255}$ = 00000001. Thus, every element of the field except zero is equal to some power of α. The element **α**, that we define, is known as a primitive element or **generator** of the Galois field.

This observation suggests another way to implement multiplication: by adding the exponents of α.

$$10001001 * 00101010 = \alpha^{74} * \alpha^{142} = \alpha^{74 + 142} = \alpha^{216} = 11000011$$

The problem is, how do we find the power of α that corresponds to 10001001? This is known as the discrete logarithm problem, and no efficient general solution is known. However, since there are only 256 elements in this field, we can easily construct a table of logarithms. While we're at it, a corresponding table of antilogs (exponentials) will also be useful. More mathematical information about this trick can be found here.

```
gf_exp = [0] * 512 # Create list of 512 elements. In Python 2.6+,
consider using bytearray
gf_log = [0] * 256

def init_tables(prim=0x11d):
    '''Precompute the logarithm and anti-log tables for faster
computation later, using the provided primitive polynomial.'''
    # prim is the primitive (binary) polynomial. Since it's a
polynomial in the binary sense,
    # it's only in fact a single galois field value between 0 and
255, and not a list of gf values.
    global gf_exp, gf_log
    gf_exp = [0] * 512 # anti-log (exponential) table
    gf_log = [0] * 256 # log table
    # For each possible value in the galois field 2^8, we will pre-
compute the logarithm and anti-logarithm (exponential) of this
value
    x = 1
    for i in range(0, 255):
        gf_exp[i] = x # compute anti-log for this value and store
it in a table
        gf_log[x] = i # compute log at the same time
        x = gf_mult_noLUT(x, 2, prim)

        # If you use only generator==2 or a power of 2, you can use
the following which is faster than gf_mult_noLUT():
        #x <<= 1 # multiply by 2 (change 1 by another number y to
multiply by a power of 2^y)
        #if x & 0x100: # similar to x >= 256, but a lot faster
(because 0x100 == 256)
            #x ^= prim # subtract the primary polynomial to the
current value (instead of 255, so that we get a unique set made of
coprime numbers), this is the core of the tables generation

    # Optimization: double the size of the anti-log table so that
we don't need to mod 255 to
    # stay inside the bounds (because we will mainly use this table
for the multiplication of two GF numbers, no more).
    for i in range(255, 512):
        gf_exp[i] = gf_exp[i - 255]
    return [gf_log, gf_exp]
```

*Python note:* The `range` operator's upper bound is exclusive, so `gf_exp[255]` is not set twice by the above.

The `gf_exp` table is oversized in order to simplify the multiplication function. This way, we don't have to check to make sure that `gf_log[x] + gf_log[y]` is within the table size.

```
def gf_mul(x,y):
    if x==0 or y==0:
        return 0
    return gf_exp[gf_log[x] + gf_log[y]] # should be
gf_exp[(gf_log[x]+gf_log[y])%255] if gf_exp wasn't oversized
```

## Division

Another advantage of the logarithm table approach is that it allows us to define division using the difference of logarithms. In the code below, 255 is added to make sure the difference isn't negative.

```
def gf_div(x,y):
    if y==0:
        raise ZeroDivisionError()
    if x==0:
        return 0
    return gf_exp[(gf_log[x] + 255 - gf_log[y]) % 255]
```

*Python note:* The `raise` statement throws an exception and aborts execution of the `gf_div` function.

With this definition of division, `gf_div(gf_mul(x,y),y)==x` for any `x` and any nonzero `y`.

Readers who are more advanced programmers may find it interesting to write a class encapsulating Galois field arithmetic. Operator overloading can be used to replace calls to `gf_mul` and `gf_div` with the familiar operators `*` and `/`, but this can lead to confusion as to exactly what type of operation is being performed. Certain details can be generalized in ways that would make the class more widely useful. For example, Aztec codes use five different Galois fields with element sizes ranging from 4 to 12 bits.

## Power and Inverse

The logarithm table approach will once again simplify and speed up our calculations when computing the power and the inverse:

```
def gf_pow(x, power):
    return gf_exp[(gf_log[x] * power) % 255]
```

```
def gf_inverse(x):
    return gf_exp[255 - gf_log[x]] # gf_inverse(x) == gf_div(1, x)
```

## Polynomials

Before moving on to Reed–Solomon codes, we need to define several operations on polynomials whose coefficients are Galois field elements. This is a potential source of confusion, since the elements themselves are described as polynomials; my advice is not to think about it too much. Adding to the confusion is the fact that $x$ is still used as the placeholder. This $x$ has nothing to do with the $x$ mentioned previously, so don't mix them up.

The binary notation used previously for Galois field elements starts to become inconveniently bulky at this point, so I will switch to hexadecimal instead.

> 00000001 $x^4$ + 00001111 $x^3$ + 00110110 $x^2$ + 01111000 $x$ + 01000000 = 01 $x^4$ + 0f $x^3$ + 36 $x^2$ + 78 $x$ + 40

In Python, polynomials will be represented by a list of numbers in descending order of powers of $x$, so the polynomial above becomes `[ 0x01, 0x0f, 0x36, 0x78, 0x40 ]`. (The reverse order could have been used instead; both choices have their advantages and disadvantages.)

The first function multiplies a polynomial by a scalar.

```
def gf_poly_scale(p,x):
    r = [0] * len(p)
    for i in range(0, len(p)):
        r[i] = gf_mul(p[i], x)
    return r
```

*Note to Python programmers:* This function is not written in a "pythonic" style. It could be expressed quite elegantly as a list comprehension, but I have limited myself to language features that are easier to translate to other programming languages.

This function "adds" two polynomials (using exclusive-or, as usual).

```
def gf_poly_add(p,q):
    r = [0] * max(len(p),len(q))
    for i in range(0,len(p)):
        r[i+len(r)-len(p)] = p[i]
    for i in range(0,len(q)):
        r[i+len(r)-len(q)] ^= q[i]
    return r
```

The next function multiplies two polynomials.

```
def gf_poly_mul(p,q):
    '''Multiply two polynomials, inside Galois Field'''
    # Pre-allocate the result array
    r = [0] * (len(p)+len(q)-1)
    # Compute the polynomial multiplication (just like the outer
product of two vectors,
    # we multiply each coefficients of p with all coefficients of
q)
    for j in range(0, len(q)):
        for i in range(0, len(p)):
            r[i+j] ^= gf_mul(p[i], q[j]) # equivalent to: r[i + j]
= gf_add(r[i+j], gf_mul(p[i], q[j]))
                                                    # -- you
can see it's your usual polynomial multiplication
    return r
```

Finally, we need a function to evaluate a polynomial at a particular value of *x*, producing a scalar result. Horner's method is used to avoid explicitly calculating powers of *x*. Horner's method works by factorizing consecutively the terms, so that we always deal with *x^1*, iteratively, avoiding the computation of higher degree terms:

$$01\,x^4 + 0f\,x^3 + 36\,x^2 + 78\,x + 40 = (((01\,x + 0f)\,x + 36)\,x + 78)\,x + 40$$

```
def gf_poly_eval(poly, x):
    '''Evaluates a polynomial in GF(2^p) given the value for x.
This is based on Horner's scheme for maximum efficiency.'''
    y = poly[0]
    for i in range(1, len(poly)):
        y = gf_mul(y, x) ^ poly[i]
    return y
```

There's still one missing polynomial operation that we will need: polynomial division. This is more complicated than the other operations on polynomial, so we will study it in the next chapter, along with Reed–Solomon encoding.

# Reed–Solomon codes

Now that the preliminaries are out of the way, we are ready to begin looking at Reed–Solomon codes.

## Insight of the coding theory

But first, why did we have to learn about finite fields and polynomials? Because this is the main insight of error-correcting codes like Reed–Solomon: instead of just seeing a message as a series of (ASCII) numbers, we see it as **a polynomial** following the very well-defined **rules of finite field arithmetic**. In other words, by representing the data using polynomials and finite fields arithmetic, **we added a structure**

**to the data**. The values of the message are still the same, but this conceptual structure now allows us to operate on the message, on its characters values, using well defined mathematical rules. This structure, that we always know because it's outside and independent of the data, is what allows us to repair a corrupted message.

Thus, even if in your code implementation you may choose to not explicitly represent the polynomials and the finite field arithmetic, these notions are essential for the error-correcting codes to work, and you will find these notions to underlie (even if implicitly) any implementation.

And now we will put these notions into practice!

## RS encoding

### Encoding outline

Like BCH codes, Reed–Solomon codes are encoded by dividing the polynomial representing the message by an irreducible generator polynomial, and then the remainder is the RS code, which we will just append to the original message.

Why? We previously said that the principle behind BCH codes, and most other error correcting codes, is to use a reduced dictionary with very different words as to maximize the distance between words, and that longer words have greater distance: here it's the same principle, first because we lengthen the original message with additional symbols (the remainder) which raises the distance, and secondly because the remainder is almost unique (thanks to the carefully designed irreducible generator polynomial), so that it can be exploited by clever algorithms to deduce parts of the original message.

To summarize, with an approximated analogy to encryption: our **generator polynomial** is our encoding **dictionary**, and **polynomial division** is the operator to **convert** our message using the dictionary (the generator polynomial) into a RS code.

### Exception management

To manage errors and cases where we can't correct a message, we will display a meaningful error message, by raising an exception. We will make our own custom exception so that users can easily catch and manage them:

```
class ReedSolomonError(Exception):
    pass
```

To display an error by raising our custom exception, we can then simply do the following:

```
raise ReedSolomonError("Some error message")
```

And you can easily catch this exception to manage it by using a try/except block:

```
try:
    raise ReedSolomonError("Some error message")
except ReedSolomonError as e:
    pass # do something here
```

### RS generator polynomial

Reed–Solomon codes use a **generator polynomial** similar to BCH codes (not to be confused with the generator number alpha). The generator is the product of factors $(x - \alpha^n)$, starting with $n$=0 for QR codes.

For example: $g_4(x) = (x - \alpha^0)(x - \alpha^1)(x - \alpha^2)(x - \alpha^3)$

The same as $(x + a^i)$ because of GF(2^8).

$g_4(x) = x^4 - (\alpha^3+\alpha^2+\alpha^1+\alpha^0)\,x^3 + ((\alpha^0+\alpha^1)(\alpha^2+\alpha^3)+(\alpha^5+\alpha^1))\,x^2 + (\alpha^6+\alpha^5+\alpha^4+\alpha^3)\,x + \alpha^6$

$g_4(x) = x^4 - (\alpha^3+\alpha^2+\alpha^1+\alpha^0)\,x^3 + (\alpha^2+\alpha^3+\alpha^3+\alpha^4+\alpha^5+\alpha^1)\,x^2 + (\alpha^6+\alpha^5+\alpha^4+\alpha^3)\,x + \alpha^6$

$g_4(x) = x^4 - (\alpha^3+\alpha^2+\alpha^1+\alpha^0)\,x^3 + (\alpha^5+\alpha^4+\alpha^2+\alpha^1)\,x^2 + (\alpha^6+\alpha^5+\alpha^4+\alpha^3)\,x + \alpha^6$

$g_4(x) = \mathtt{01}\,x^4 + \mathtt{0f}\,x^3 + \mathtt{36}\,x^2 + \mathtt{78}\,x + \mathtt{40}$

Here is a function that computes the generator polynomial for a given number of error correction symbols.

```
def rs_generator_poly(nsym):
    '''Generate an irreducible generator polynomial (necessary to
encode a message into Reed-Solomon)'''
    g = [1]
    for i in range(0, nsym):
        g = gf_poly_mul(g, [1, gf_pow(2, i)])
    return g
```

This function is somewhat inefficient in that it allocates successively larger arrays for `g`. While this is unlikely to be a performance problem in practice, readers who are inveterate optimizers may find it interesting to rewrite it so that `g` is only allocated once, or you can compute once and memorize g since it is fixed for a given nsym, so you can reuse g.

### Polynomial division

Several algorithms for polynomial division exist, the simplest one that is often taught in elementary school is long division. This example shows the calculation for the message `12 34 56`.

```
                  ________________ 12 da df
01 0f 36 78 40 ) 12 34 56 00 00 00 00
                 ^ 12 ee 2b 23 f4
                    da 7d 23 f4 00
                  ^ da a2 85 79 84
```

```
                    df a6 8d 84 00
                  ^ df 91 6b fc d9
                       37 e6 78 d9
```

Note: The concepts of polynomial long division apply, but there are a few important differences: When computing the resulting terms/coefficients that will be Galois Field subtracted from the divisor, bitwise carryless multiplication is performed and the result "bitstream" is XORed from the first encountered MSB with the chosen primitive polynomial until the answer is less than the Galois Field value, in this case, 256. The XOR "subtractions" are then performed as usual.

To illustrate the method for one operation (0x12 * 0x36):

```
    00010010 ( 12 )
  x 00110110 ( 36 )
   00110110
00110110
001100001100
 ^100011101   <-- XOR with primitive polynomial value (11D)...
000100110110
  ^100011101  <-- ...until answer is less than 256.
    00101011
       2   b
```

The remainder is concatenated with the message, so the encoded message is `12 34 56 37 e6 78 d9`.

However, long division is quite slow as it requires a lot of recursive iterations to terminate. More efficient strategies can be devised, such as using synthetic division (also called Horner's method, a good tutorial video can be found on Khan Academy (https://www.khanacademy.org/math/algebra2/polynomial_and_rational/synthetic-division/v/synthetic-division)). Here is a function that implements extended synthetic division of GF(2^p) polynomials (extended because the divisor is a polynomial instead of a monomial):

```python
def gf_poly_div(dividend, divisor):
    '''Fast polynomial division by using Extended Synthetic
Division and optimized for GF(2^p) computations
    (doesn't work with standard polynomials outside of this galois
field, see the Wikipedia article for generic algorithm).'''
    # CAUTION: this function expects polynomials to follow the
opposite convention at decoding:
    # the terms must go from the biggest to lowest degree (while
most other functions here expect
    # a list from lowest to biggest degree). eg: 1 + 2x + 5x^2 =
[5, 2, 1], NOT [1, 2, 5]

    msg_out = list(dividend) # Copy the dividend
    #normalizer = divisor[0] # precomputing for performance
    for i in range(0, len(dividend) - (len(divisor)-1)):
```

```
        #msg_out[i] /= normalizer # for general polynomial division
(when polynomials are non-monic), the usual way of using
                                  # synthetic division is to divide
the divisor g(x) with its leading coefficient, but not needed here.
        coef = msg_out[i] # precaching
        if coef != 0: # log(0) is undefined, so we need to avoid
that case explicitly (and it's also a good optimization).
            for j in range(1, len(divisor)): # in synthetic
division, we always skip the first coefficient of the divisior,
                                             # because it's only
used to normalize the dividend coefficient
                if divisor[j] != 0: # log(0) is undefined
                    msg_out[i + j] ^= gf_mul(divisor[j], coef) #
equivalent to the more mathematically correct
                                                               #
(but xoring directly is faster): msg_out[i + j] += -divisor[j] *
coef

    # The resulting msg_out contains both the quotient and the
remainder, the remainder being the size of the divisor
    # (the remainder has necessarily the same degree as the divisor
-- not length but degree == length-1 -- since it's
    # what we couldn't divide from the dividend), so we compute the
index where this separation is, and return the quotient and
remainder.
    separator = -(len(divisor)-1)
    return msg_out[:separator], msg_out[separator:] # return
quotient, remainder.
```

### Encoding main function

And now, here's how to encode a message to get its RS code:

```
def rs_encode_msg(msg_in, nsym):
    '''Reed-Solomon main encoding function'''
    gen = rs_generator_poly(nsym)

    # Pad the message, then divide it by the irreducible generator
polynomial
    _, remainder = gf_poly_div(msg_in + [0] * (len(gen)-1), gen)
    # The remainder is our RS code! Just append it to our original
message to get our full codeword (this represents a polynomial of
max 256 terms)
    msg_out = msg_in + remainder
    # Return the codeword
    return msg_out
```

Simple, isn't it? Encoding is in fact the easiest part in Reed–Solomon, and it's always the same approach (polynomial division). Decoding is the tough part of Reed–Solomon, and you will find a lot of different algorithms depending on your needs, but we will touch on that later on.

This function is quite fast, but since encoding is quite critical, here is an enhanced encoding function that inlines the polynomial synthetic division, which is the form that you will most often find in Reed–Solomon software libraries:

```
def rs_encode_msg(msg_in, nsym):
    '''Reed-Solomon main encoding function, using polynomial
division (algorithm Extended Synthetic Division)'''
    if (len(msg_in) + nsym) > 255: raise ValueError("Message is too
long (%i when max is 255)" % (len(msg_in)+nsym))
    gen = rs_generator_poly(nsym)
    # Init msg_out with the values inside msg_in and pad with
len(gen)-1 bytes (which is the number of ecc symbols).
    msg_out = [0] * (len(msg_in) + len(gen)-1)
    # Initializing the Synthetic Division with the dividend (=
input message polynomial)
    msg_out[:len(msg_in)] = msg_in

    # Synthetic division main loop
    for i in range(len(msg_in)):
        # Note that it's msg_out here, not msg_in. Thus, we reuse
the updated value at each iteration
        # (this is how Synthetic Division works: instead of storing
in a temporary register the intermediate values,
        # we directly commit them to the output).
        coef = msg_out[i]

        # log(0) is undefined, so we need to manually check for
this case. There's no need to check
        # the divisor here because we know it can't be 0 since we
generated it.
        if coef != 0:
            # in synthetic division, we always skip the first
coefficient of the divisior, because it's only used to normalize
the dividend coefficient (which is here useless since the divisor,
the generator polynomial, is always monic)
            for j in range(1, len(gen)):
                msg_out[i+j] ^= gf_mul(gen[j], coef) # equivalent
to msg_out[i+j] += gf_mul(gen[j], coef)

    # At this point, the Extended Synthetic Divison is done,
msg_out contains the quotient in msg_out[:len(msg_in)]
    # and the remainder in msg_out[len(msg_in):]. Here for RS
encoding, we don't need the quotient but only the remainder
    # (which represents the RS code), so we can just overwrite the
quotient with the input message, so that we get
```

```
    # our complete codeword composed of the message + code.
    msg_out[:len(msg_in)] = msg_in

    return msg_out
```

This algorithm is faster, but it's still quite slow for practical use, particularly in Python. There are some ways to optimize the speed by using various tricks, such as inlining (instead of gf_mul, replace by the operation to avoid a call), by precomputing (the logarithm of gen and of coef, or even by generating a multiplication table – but it seems the latter does not work well in Python), by using statically typed constructs (assign gf_log and gf_exp to `array.array('i', [...])`), by using memoryviews (like by changing all your lists to bytearrays), by running it with PyPy, or by converting the algorithm into a Cython or a C extension[1].

This example shows the encode function applied to the message in the sample QR code introduced earlier. The calculated error correction symbols (on the second line) match the values decoded from the QR code.

```
>>> msg_in = [ 0x40, 0xd2, 0x75, 0x47, 0x76, 0x17, 0x32, 0x06,
...             0x27, 0x26, 0x96, 0xc6, 0xc6, 0x96, 0x70, 0xec ]
>>> msg = rs_encode_msg(msg_in, 10)
>>> for i in range(0,len(msg)):
...    print(hex(msg[i]), end=' ')
...
0x40 0xd2 0x75 0x47 0x76 0x17 0x32 0x6 0x27 0x26 0x96 0xc6 0xc6
0x96 0x70 0xec
0xbc 0x2a 0x90 0x13 0x6b 0xaf 0xef 0xfd 0x4b 0xe0
```

*Python version note:* The syntax for the `print` function has changed, and this example uses the Python 3.0+ version. In previous versions of Python (particularly Python 2.x), replace the `print` line with `print hex(msg[i]),` (including the final comma) and `range` by `xrange`.

## RS decoding

### Decoding outline

Reed–Solomon decoding is the process that, from a potentially corrupted message and a RS code, returns a corrected message. In other words, decoding is the process of repairing your message using the previously computed RS code.

Although there is only one way to encode a message with Reed–Solomon, there are lots of different ways to decode them, and thus there are a lot of different decoding algorithms.

However, we can generally outline the decoding process in 5 steps[2][3]:

1. Compute the **syndromes polynomial**. This allows us to analyze what characters are in error using Berlekamp-Massey (or another algorithm), and also to quickly check if the input message is corrupted at all.

2. Compute the erasure/error **locator polynomial** (from the syndromes). This is computed by Berlekamp-Massey, and is a detector that will tell us exactly what characters are corrupted.
3. Compute the erasure/error **evaluator polynomial** (from the syndromes and erasure/error locator polynomial). Necessary to evaluate how much the characters were tampered (ie, helps to compute the magnitude).
4. Compute the erasure/error **magnitude polynomial** (from all 3 polynomials above): this polynomial can also be called the corruption polynomial, since in fact it exactly stores the values that need to be subtracted from the received message to get the original, correct message (i.e., with correct values for erased characters). In other words, at this point, we extracted the noise and stored it in this polynomial, and we just have to remove this noise from the input message to repair it.
5. **Repair the input message** simply by subtracting the magnitude polynomial from the input message.

We will describe each of those five steps below.

In addition, decoders can also be classified by the type of error they can repair: erasures (we know the location of the corrupted characters but not the magnitude), errors (we ignore both the location and magnitude), or a mix of errors-and-erasures. We will describe how to support all of these.

### Syndrome calculation

Decoding a Reed–Solomon message involves several steps. The first step is to calculate the "syndrome" of the message. Treat the message as a polynomial and evaluate it at $\alpha^0$, $\alpha^1$, $\alpha^2$, ..., $\alpha^n$. Since these are the zeros of the generator polynomial, the result should be zero if the scanned message is undamaged (this can be used to check if the message is corrupted, and after correction of a corrupted message if the message was completely repaired). If not, the syndromes contain all the information necessary to determine the correction that should be made. It is simple to write a function to calculate the syndromes.

```
def rs_calc_syndromes(msg, nsym):
    '''Given the received codeword msg and the number of error
correcting symbols (nsym), computes the syndromes polynomial.
    Mathematically, it's essentially equivalent to a Fourrier
Transform (Chien search being the inverse).
    '''
    # Note the "[0] +" : we add a 0 coefficient for the lowest
degree (the constant). This effectively shifts the syndrome, and
will shift every computations depending on the syndromes (such as
the errors locator polynomial, errors evaluator polynomial, etc.
but not the errors positions).
    # This is not necessary, you can adapt subsequent computations
to start from 0 instead of skipping the first iteration (ie, the
often seen range(1, n-k+1)),
    synd = [0] * nsym
    for i in range(0, nsym):
        synd[i] = gf_poly_eval(msg, gf_pow(2,i))
    return [0] + synd # pad with one 0 for mathematical precision
(else we can end up with weird calculations sometimes)
```

Continuing the example, we see that the syndromes of the original codeword without any corruption are indeed zero. Introducing a corruption of at least one character into the message or its RS code gives nonzero syndromes.

```
>>> synd = rs_calc_syndromes(msg, 10)
>>> print(synd)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] # not corrupted message = all 0
syndromes
>>> msg[0] = 0  # deliberately damage the message
>>> synd = rs_calc_syndromes(msg, 10)
>>> print(synd)
[0, 64, 192, 93, 231, 52, 92, 228, 49, 83, 245] # when corrupted,
the syndromes will be non zero
```

Here is the code to automate this checking:

```
def rs_check(msg, nsym):
    '''Returns true if the message + ecc has no error or false
otherwise (may not always catch a wrong decoding or a wrong
message, particularly if there are too many errors -- above the
Singleton bound --, but it usually does)'''
    return ( max(rs_calc_syndromes(msg, nsym)) == 0 )
```

### Erasure correction

It is simplest to correct mistakes in the code if the locations of the mistakes are already known. This is known as **erasure correction**. It is possible to correct one erased symbol (ie, character) for each error-correction symbol added to the code. If the error locations are not known, two EC symbols are needed for each symbol error (so you can correct twice less errors than erasures). This makes erasure correction useful in practice if part of the QR code being scanned is covered or physically torn away. It may be difficult for a scanner to determine that this has happened, though, so not all QR code scanners can perform erasure correction.

Now that we already have the syndromes, we need to compute the locator polynomial. This is easy:

```
def rs_find_errata_locator(e_pos):
    '''Compute the erasures/errors/errata locator polynomial from
the erasures/errors/errata 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.'''
```

```
    e_loc = [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 = product(1 - x*alpha**i) for i in erasures_pos
and where alpha is the alpha chosen to evaluate polynomials.
    for i in e_pos:
        e_loc = gf_poly_mul( e_loc, gf_poly_add([1], [gf_pow(2, i),
0]) )
    return e_loc
```

Next, computing the erasure/error evaluator polynomial from the locator polynomial is easy, it's simply a polynomial multiplication followed by a polynomial division (that you can replace by a list slicing because that's the effect we want in the end):

```
def rs_find_error_evaluator(synd, err_loc, nsym):
    '''Compute the error (or erasures if you supply sigma=erasures
locator polynomial, or errata) evaluator polynomial Omega
       from the syndrome and the error/erasures/errata locator
Sigma.'''

    # Omega(x) = [ Synd(x) * Error_loc(x) ] mod x^(n-k+1)
    _, remainder = gf_poly_div( gf_poly_mul(synd, err_loc), ([1] +
[0]*(nsym+1)) ) # first multiply syndromes * errata_locator, then
do a

# polynomial division to truncate the polynomial to the

# required length

    # Faster way that is equivalent
    #remainder = gf_poly_mul(synd, err_loc) # first multiply the
syndromes with the errata locator polynomial
    #remainder = remainder[len(remainder)-(nsym+1):] # then slice
the list to truncate it (which represents the polynomial), which

# is equivalent to dividing by a polynomial of the length we want

    return remainder
```

Finally, the Forney algorithm is used to calculate the correction values (also called the error magnitude polynomial). It is implemented in the function below.

```
def rs_correct_errata(msg_in, synd, err_pos): # err_pos is a list
of the positions of the errors/erasures/errata
    '''Forney algorithm, computes the values (error magnitude) to
correct the input message.'''
```

```
    # calculate errata locator polynomial to correct both errors
and erasures (by combining the errors positions given by the error
locator polynomial found by BM with the erasures positions given by
caller)
    coef_pos = [len(msg_in) - 1 - p for p in err_pos] # need to
convert the positions to coefficients degrees for the errata
locator algo to work (eg: instead of [0, 1, 2] it will become
[len(msg)-1, len(msg)-2, len(msg) -3])
    err_loc = rs_find_errata_locator(coef_pos)
    # calculate errata evaluator polynomial (often called Omega or
Gamma in academic papers)
    err_eval = rs_find_error_evaluator(synd[::-1], err_loc,
len(err_loc)-1)[::-1]

    # Second part of Chien search to get the error location
polynomial X from the error positions in err_pos (the roots of the
error locator polynomial, ie, where it evaluates to 0)
    X = [] # will store the position of the errors
    for i in range(0, len(coef_pos)):
        l = 255 - coef_pos[i]
        X.append( gf_pow(2, -l) )

    # Forney algorithm: compute the magnitudes
    E = [0] * (len(msg_in)) # will store the values that need to be
corrected (substracted) to the message containing errors. This is
sometimes called the error magnitude polynomial.
    Xlength = len(X)
    for i, Xi in enumerate(X):

        Xi_inv = gf_inverse(Xi)

        # Compute the formal derivative of the error locator
polynomial (see Blahut, Algebraic codes for data transmission, pp
196-197).
        # the formal derivative of the errata locator is used as
the denominator of the Forney Algorithm, which simply says that the
ith error value is given by error_evaluator(gf_inverse(Xi)) /
error_locator_derivative(gf_inverse(Xi)). See Blahut, Algebraic
codes for data transmission, pp 196-197.
        err_loc_prime_tmp = []
        for j in range(0, Xlength):
            if j != i:
                err_loc_prime_tmp.append( gf_sub(1, gf_mul(Xi_inv,
X[j])) )
        # compute the product, which is the denominator of the
Forney algorithm (errata locator derivative)
        err_loc_prime = 1
        for coef in err_loc_prime_tmp:
            err_loc_prime = gf_mul(err_loc_prime, coef)
        # equivalent to: err_loc_prime = functools.reduce(gf_mul,
```

```
err_loc_prime_tmp, 1)

        # Compute y (evaluation of the errata evaluator polynomial)
        # This is a more faithful translation of the theoretical
equation contrary to the old forney method. Here it is an exact
reproduction:
        # Yl = omega(Xl.inverse()) / prod(1 - Xj*Xl.inverse()) for
j in len(X)
        y = gf_poly_eval(err_eval[::-1], Xi_inv) # numerator of the
Forney algorithm (errata evaluator evaluated)
        y = gf_mul(gf_pow(Xi, 1), y)

        # Check: err_loc_prime (the divisor) should not be zero.
        if err_loc_prime == 0:
            raise ReedSolomonError("Could not find error
magnitude")    # Could not find error magnitude

        # Compute the magnitude
        magnitude = gf_div(y, err_loc_prime) # magnitude value of
the error, calculated by the Forney algorithm (an equation in
fact): dividing the errata evaluator with the errata locator
derivative gives us the errata magnitude (ie, value to repair) the
ith symbol
        E[err_pos[i]] = magnitude # store the magnitude for this
error into the magnitude polynomial

    # Apply the correction of values to get our message corrected!
(note that the ecc bytes also gets corrected!)
    # (this isn't the Forney algorithm, we just apply the result of
decoding here)
    msg_in = gf_poly_add(msg_in, E) # equivalent to Ci = Ri - Ei
where Ci is the correct message, Ri the received (senseword)
message, and Ei the errata magnitudes (minus is replaced by XOR
since it's equivalent in GF(2^p)). So in fact here we substract
from the received message the errors magnitude, which logically
corrects the value to what it should be.
    return msg_in
```

*Mathematics note:* The denominator of the expression for the error value is the formal derivative of the error locator polynomial `q`. This is calculated by the usual procedure of replacing each term $c_n\ x^n$ with $n\ c_n\ x^{n-1}$. Since we're working in a field of characteristic two, $n\ c_n$ is equal to $c_n$ when $n$ is odd, and 0 when $n$ is even. Thus, we can simply remove the even coefficients (resulting in the polynomial `qprime`) and evaluate `qprime(x`$^2$`)`.

*Python note:* This function uses [::-1] to inverse the order of the elements in a list. This is necessary because the functions do not all use the same ordering convention (ie, some use the least item first, others use the biggest item first). It also use a list comprehension, which is simply a concise way to write a for loop where items are appended in a list, but the Python interpreter can optimize this a bit more than a loop.

Continuing the example, here we use `rs_correct_errata` to restore the first byte of the message.

```
>>> msg[0] = 0
>>> synd = rs_calc_syndromes(msg, 10)
>>> msg = rs_correct_errata(msg, synd, [0]) # [0] is the list of
the erasures locations, here it's the first character, at position
0
>>> print(hex(msg[0]))
0x40
```

### Error correction

In the more likely situation where the error locations are unknown (what we usually call **errors**, in opposition to **erasures** where the locations are known), we will use the same steps as for erasures, but we now need additional steps to find the location. The Berlekamp–Massey algorithm is used to calculate the error **locator polynomial**, which we can use later on to determine the errors locations:

```
def rs_find_error_locator(synd, nsym, erase_loc=None,
erase_count=0):
    '''Find error/errata locator and evaluator polynomials with
Berlekamp-Massey algorithm'''
    # The idea is that BM will iteratively estimate the error
locator polynomial.
    # To do this, it will compute a Discrepancy term called Delta,
which will tell us if the error locator polynomial needs an update
or not
    # (hence why it's called discrepancy: it tells us when we are
getting off board from the correct value).

    # Init the polynomials
    if erase_loc: # if the erasure locator polynomial is supplied,
we init with its value, so that we include erasures in the final
locator polynomial
        err_loc = list(erase_loc)
        old_loc = list(erase_loc)
    else:
        err_loc = [1] # This is the main variable we want to fill,
also called Sigma in other notations or more formally the
errors/errata locator polynomial.
        old_loc = [1] # BM is an iterative algorithm, and we need
the errata locator polynomial of the previous iteration in order to
update other necessary variables.
    #L = 0 # update flag variable, not needed here because we use
an alternative equivalent way of checking if update is needed (but
using the flag could potentially be faster depending on if using
length(list) is taking linear time in your language, here in Python
it's constant so it's as fast.
```

```
    # 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, which set to 0 the coefficients corresponding to erasures, leaving only the coefficients corresponding to errors.
    synd_shift = len(synd) - nsym

    for i in range(0, nsym-erase_count): # generally: nsym-erase_count == len(synd), except when you input a partial erase_loc and using the full syndrome instead of the Forney syndrome, in which case nsym-erase_count is more correct (len(synd) will fail badly with IndexError).
        if erase_loc: # if an erasures locator polynomial was provided to init the errors locator polynomial, then we must skip the FIRST erase_count iterations (not the last iterations, this is very important!)
            K = erase_count+i+synd_shift
        else: # if erasures locator is not provided, then either there's no erasures to account or we use the Forney syndromes, so we don't need to use erase_count nor erase_loc (the erasures have been trimmed out of the Forney syndromes).
            K = i+synd_shift

        # Compute the discrepancy Delta
        # Here is the close-to-the-books operation to compute the discrepancy Delta: it's a simple polynomial multiplication of error locator with the syndromes, and then we get the Kth element.
        #delta = gf_poly_mul(err_loc[::-1], synd)[K] # theoretically it should be gf_poly_add(synd[::-1], [1])[::-1] instead of just synd, but it seems it's not absolutely necessary to correctly decode.
        # But this can be optimized: since we only need the Kth element, we don't need to compute the polynomial multiplication for any other element but the Kth. Thus to optimize, we compute the polymul only at the item we need, skipping the rest (avoiding a nested loop, thus we are linear time instead of quadratic).
        # This optimization is actually described in several figures of the book "Algebraic codes for data transmission", Blahut, Richard E., 2003, Cambridge university press.
        delta = synd[K]
        for j in range(1, len(err_loc)):
```

```
            delta ^= gf_mul(err_loc[-(j+1)], synd[K - j]) # delta
is also called discrepancy. Here we do a partial polynomial
multiplication (ie, we compute the polynomial multiplication only
for the term of degree K). Should be equivalent to
brownanrs.polynomial.mul_at().
        #print "delta", K, delta, list(gf_poly_mul(err_loc[::-1],
synd)) # debugline

        # Shift polynomials to compute the next degree
        old_loc = old_loc + [0]

        # Iteratively estimate the errata locator and evaluator
polynomials
        if delta != 0: # Update only if there's a discrepancy
            if len(old_loc) > len(err_loc): # Rule B (rule A is
implicitly defined because rule A just says that we skip any
modification for this iteration)
            #if 2*L <= K+erase_count: # equivalent to len(old_loc)
> len(err_loc), as long as L is correctly computed
                # Computing errata locator polynomial Sigma
                new_loc = gf_poly_scale(old_loc, delta)
                old_loc = gf_poly_scale(err_loc, gf_inverse(delta))
# effectively we are doing err_loc * 1/delta = err_loc // delta
                err_loc = new_loc
                # Update the update flag
                #L = K - L # the update flag L is tricky: in
Blahut's schema, it's mandatory to use `L = K - L - erase_count`
(and indeed in a previous draft of this function, if you forgot to
do `- erase_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 `- erase_count` to update L on your own
implementation and see which one works OK without producing wrong
decoding failures.

            # Update with the discrepancy
            err_loc = gf_poly_add(err_loc, gf_poly_scale(old_loc,
delta))

    # Check if the result is correct, that there's not too many
errors to correct
    while len(err_loc) and err_loc[0] == 0: del err_loc[0] # drop
leading 0s, else errs will not be of the correct size
    errs = len(err_loc) - 1
    if (errs-erase_count) * 2 + erase_count > nsym:
        raise ReedSolomonError("Too many errors to correct")    #
too many errors to correct
```

```
    return err_loc
```

Then, using the error locator polynomial, we simply use a brute-force approach called trial substitution to find the zeros of this polynomial, which identifies the error locations (ie, the index of the characters that need to be corrected). A more efficient algorithm called Chien search exists, which avoids recomputing the whole evaluation at each iteration step, but this algorithm is left as an exercise to the reader.

```
def rs_find_errors(err_loc, nmess): # nmess is len(msg_in)
    '''Find the roots (ie, where evaluation = zero) of error
polynomial by brute-force trial, this is a sort of Chien's search
    (but less efficient, Chien's search is a way to evaluate the
polynomial such that each evaluation only takes constant time).'''
    errs = len(err_loc) - 1
    err_pos = []
    for i in range(nmess): # normally we should try all 2^8
possible values, but here we optimize to just check the interesting
symbols
        if gf_poly_eval(err_loc, gf_pow(2, i)) == 0: # It's a 0?
Bingo, it's a root of the error locator polynomial,

# in other terms this is the location of an error
            err_pos.append(nmess - 1 - i)
    # Sanity check: the number of errors/errata positions found
should be exactly the same as the length of the errata locator
polynomial
    if len(err_pos) != errs:
        # couldn't find error locations
        raise ReedSolomonError("Too many (or few) errors found by
Chien Search for the errata locator polynomial!")
    return err_pos
```

*Mathematics note:* When the error locator polynomial is linear (`err_poly` has length 2), it can be solved easily without resorting to a brute-force approach. The function presented above does not take advantage of this fact, but the interested reader may wish to implement the more efficient solution. Similarly, when the error locator is quadratic, it can be solved by using a generalization of the quadratic formula. A more ambitious reader may wish to implement this procedure as well.

Here is an example where three errors in the message are corrected:

```
>>> print(hex(msg[10]))
0x96
>>> msg[0] = 6
>>> msg[10] = 7
>>> msg[20] = 8
>>> synd = rs_calc_syndromes(msg, 10)
>>> err_loc = rs_find_error_locator(synd, nsym)
```

```
>>> pos = rs_find_errors(err_loc[::-1], len(msg)) # find the errors
locations
>>> print(pos)
[20, 10, 0]
>>> msg = rs_correct_errata(msg, synd, pos)
>>> print(hex(msg[10]))
0x96
```

### Error and erasure correction

It is possible for a Reed–Solomon decoder to decode both erasures and errors at the same time, up to a limit (called the Singleton Bound) of `2*e+v <= (n-k)`, where `e` is the number of errors, `v` the number of erasures and `(n-k)` the number of RS code characters (called `nsym` in the code). Basically, it means that for every erasures, you just need one RS code character to repair it, while for every errors you need two RS code characters (because you need to find the position in addition of the value/magnitude to correct). Such a decoder is called an errors-and-erasures decoder, or an **errata decoder**.

In order to correct both errors and erasures, we must prevent the erasures from interfering with the error location process. This can be done by calculating the Forney syndromes, as follows.

```
def rs_forney_syndromes(synd, pos, nmess):
    # Compute Forney syndromes, which computes a modified syndromes
to compute only errors (erasures are trimmed out). Do not confuse
this with Forney algorithm, which allows to correct the message
based on the location of errors.
    erase_pos_reversed = [nmess-1-p for p in pos] # prepare the
coefficient degree positions (instead of the erasures positions)

    # Optimized method, all operations are inlined
    fsynd = list(synd[1:])      # make a copy and trim the first
coefficient which is always 0 by definition
    for i in range(0, len(pos)):
        x = gf_pow(2, erase_pos_reversed[i])
        for j in range(0, len(fsynd) - 1):
            fsynd[j] = gf_mul(fsynd[j], x) ^ fsynd[j + 1]

    # Equivalent, theoretical way of computing the modified Forney
syndromes: fsynd = (erase_loc * synd) % x^(n-k)
    # See Shao, H. M., Truong, T. K., Deutsch, L. J., & Reed, I. S.
(1986, April). A single chip VLSI Reed-Solomon decoder. In
Acoustics, Speech, and Signal Processing, IEEE International
Conference on ICASSP'86. (Vol. 11, pp. 2151-2154). IEEE.ISO 690
    #erase_loc = rs_find_errata_locator(erase_pos_reversed,
generator=generator) # computing the erasures locator polynomial
    #fsynd = gf_poly_mul(erase_loc[::-1], synd[1:]) # then multiply
with the syndrome to get the untrimmed forney syndrome
    #fsynd = fsynd[len(pos):] # then trim the first erase_pos
coefficients which are useless. Seems to be not necessary, but this
reduces the computation time later in BM (thus it's an
```

```
optimization).

    return fsynd
```

The Forney syndromes can then be used in place of the regular syndromes in the error location process.

The function `rs_correct_msg` below brings the complete procedure together. Erasures are indicated by providing `erase_pos`, a list of erasures index positions in the message `msg_in` (the full received message: original message + ecc).

```
def rs_correct_msg(msg_in, nsym, erase_pos=None):
    '''Reed-Solomon main decoding function'''
    if len(msg_in) > 255: # can't decode, message is too big
        raise ValueError("Message is too long (%i when max is 255)"
% len(msg_in))

    msg_out = list(msg_in)     # copy of message
    # erasures: set them to null bytes for easier decoding (but
this is not necessary, they will be corrected anyway, but debugging
will be easier with null bytes because the error locator polynomial
values will only depend on the errors locations, not their values)
    if erase_pos is None:
        erase_pos = []
    else:
        for e_pos in erase_pos:
            msg_out[e_pos] = 0
    # check if there are too many erasures to correct (beyond the
Singleton bound)
    if len(erase_pos) > nsym: raise ReedSolomonError("Too many
erasures to correct")
    # prepare the syndrome polynomial using only errors (ie: errors
= characters that were either replaced by null byte
    # or changed to another character, but we don't know their
positions)
    synd = rs_calc_syndromes(msg_out, nsym)
    # check if there's any error/erasure in the input codeword. If
not (all syndromes coefficients are 0), then just return the
message as-is.
    if max(synd) == 0:
        return msg_out[:-nsym], msg_out[-nsym:]  # no errors

    # compute the Forney syndromes, which hide the erasures from
the original syndrome (so that BM will just have to deal with
errors, not erasures)
    fsynd = rs_forney_syndromes(synd, erase_pos, len(msg_out))
    # compute the error locator polynomial using Berlekamp-Massey
    err_loc = rs_find_error_locator(fsynd, nsym,
erase_count=len(erase_pos))
    # locate the message errors using Chien search (or brute-force
```

```
search)
    err_pos = rs_find_errors(err_loc[::-1] , len(msg_out))
    if err_pos is None:
        raise ReedSolomonError("Could not locate error")    # error
location failed

    # Find errors values and apply them to correct the message
    # compute errata evaluator and errata magnitude polynomials,
then correct errors and erasures
    msg_out = rs_correct_errata(msg_out, synd, (erase_pos +
err_pos)) # note that we here use the original syndrome, not the
forney syndrome

# (because we will correct both errors and erasures, so we need the
full syndrome)
    # check if the final message is fully repaired
    synd = rs_calc_syndromes(msg_out, nsym)
    if max(synd) > 0:
        raise ReedSolomonError("Could not correct message")     #
message could not be repaired
    # return the successfully decoded message
    return msg_out[:-nsym], msg_out[-nsym:] # also return the
corrected ecc block so that the user can check()
```

*Python note:* The lists `erase_pos` and `err_pos` are concatenated with the `+` operator.

This is the last piece needed for a fully operational error-and-erasure correcting Reed–Solomon decoder. If you want to delve more into the inner workings of errata (errors-and-erasures) decoders, you can read the excellent book "Algebraic Codes for Data Transmission" (2003) by Richard E. Blahut.

Mathematics note: in some software implementations, particularly the ones using a language optimized for linear algebra and matrix operations, you will find that the algorithms (encoding, Berlekamp-Massey, etc.) will seem totally different and use the Fourier Transform. This is because this is totally equivalent: when stated in the jargon of spectral estimation, decoding Reed–Solomon consists of a Fourier transform (syndrome computer), followed by a spectral analysis (Berlekamp-Massey or Euclidian algorithm), followed by an inverse Fourier transform (Chien search). See the Blahut book for more info[4]. Indeed, if you are using a programming language optimized for linear algebra, or if you want to use fast linear algebra libraries, it can be a very good idea to use Fourier Transform since it's very fast nowadays (particularly the Fast Fourier Transform or Number Theoretic Transform[5]).

## Wrapping up with an example

Here's an example of how to use the functions you have just made, and how to decode both errors-and-erasures:

```
# Configuration of the parameters and input message
prim = 0x11d
n = 20 # set the size you want, it must be > k, the remaining n-k
```

```
# symbols will be the ECC code (more is better)
k = 11 # k = len(message)
message = "hello world" # input message

# Initializing the log/antilog tables
init_tables(prim)

# Encoding the input message
mesecc = rs_encode_msg([ord(x) for x in message], n-k)
print("Original: %s" % mesecc)

# Tampering 6 characters of the message (over 9 ecc symbols, so we
are above the Singleton Bound)
mesecc[0] = 0
mesecc[1] = 2
mesecc[2] = 2
mesecc[3] = 2
mesecc[4] = 2
mesecc[5] = 2
print("Corrupted: %s" % mesecc)

# Decoding/repairing the corrupted message, by providing the
locations of a few erasures, we get below the Singleton Bound
# Remember that the Singleton Bound is: 2*e+v <= (n-k)
corrected_message, corrected_ecc = rs_correct_msg(mesecc, n-k,
erase_pos=[0, 1, 2])
print("Repaired: %s" % (corrected_message+corrected_ecc))
print(''.join([chr(x) for x in corrected_message]))
```

This should output the following:

```
Original:  [104, 101, 108, 108, 111,  32, 119, 111, 114, 108, 100,
145, 124, 96, 105, 94, 31, 179, 149, 163]
Corrupted: [  0,   2,   2,   2,   2,   2, 119, 111, 114, 108, 100,
145, 124, 96, 105, 94, 31, 179, 149, 163]
Repaired:  [104, 101, 108, 108, 111,  32, 119, 111, 114, 108, 100,
145, 124, 96, 105, 94, 31, 179, 149, 163]
hello world
```

## Conclusion and going further

The basic principles of Reed–Solomon codes have been presented in this essay. Working Python code for a particular implementation (QR codes using a generic Reed–Solomon codec to correct misreadings) has been included. The code presented here is quite generic and can be used for any purpose beyond QR codes where you need to correct errors/erasures, such as file protection, networking, etc. Many variations and refinements of these ideas are possible, since coding theory is a very rich field of study.

If your code is just intended for your own data (eg, you want to be able to generate and read your own QR codes), then you're fine, but if you intend to work with data provided by others (eg, you want to read and decode QR codes of other apps), then this decoder probably won't be enough, because there are some hidden parameters that were here fixed for simplicity (namely: the generator/alpha number and the first consecutive root). If you want to decode Reed–Solomon codes generated by other libraries, you will need to use a **universal** Reed–Solomon codec, which will allow you to specify your own parameters, and even go beyond the field 2^8.

On the complementary resource page, you will find an extended, universal version of the code presented here that you can use to decode almost any Reed–Solomon code, with also a function to generate the list of prime polynomials, and an algorithm to detect the parameters of an unknown RS code. Note that whatever the parameters you use, the repairing capabilities will always be the same: the generated values for the log/antilog tables and for the generator polynomial do not change the structure of Reed–Solomon code, so that you always get the same functionality whatever the parameters. Indeed, modifying any of the available parameter will not change the theoretical Singleton bound which defines the maximal repairing capacity of Reed-Solomon (and in theory of any error correction code).

One immediate issue that you may have noticed is that we can only encode messages of up to 256 characters. This limit can be circumvented by several ways, the three most common being:

- using a higher Galois Field, for example $2^{16}$ which would allow for 65536 characters, or $2^{32}$, $2^{64}$, $2^{128}$, etc. The issue here is that polynomial computations required to encode and decode Reed–Solomon become very costly with big polynomials (most algorithms being in quadratic time, the most efficient being in $n \log n$ such as with number theoretic transform[5]).
- by "chunking", which means that you simply encode your big data stream by chunks of 256 characters.
- using a variant algorithm that includes a packet size such as Cauchy Reed–Solomon (see below).

If you want to go further, there are a lot of books and scientific articles on Reed–Solomon codes, a good starting point is the author Richard Blahut who is notable in the domain. Also, there are a lot of different ways that Reed–Solomon codes can be encoded and decoded, and thus you will find many different algorithms, in particular for decoding (Berlekamp-Massey, Berlekamp-Welch, Euclidian algorithm, etc.).

If you are looking for more performance, you will find in the literature several variants of the algorithms presented here, such as Cauchy–Reed–Solomon. The programming implementation also plays a big role in the performance of your Reed–Solomon codec, which can lead into a 1000x speed difference. For more information, please read the "Optimizing performances" section of the additional resources.

Even if near-optimal forward error correction algorithms are all the rage nowadays (such as LDPC codes, Turbo codes, etc.) because of their great speed, Reed–Solomon is an optimal FEC, which means that it can attain the theoretical limit known as the Singleton bound. In practice, this means that RS can correct up to `2*e+v <= (n-k)` errors and erasures at the same time, where e is the number of errors, v the number of erasures, k the message size, n the message+code size and `(n-k)` the minimum distance. This is not to say that near-optimal FEC are useless: they are unimaginably faster than Reed–Solomon could ever be, and they may suffer less from the cliff effect (which means they may still partially decode parts of the message even if there are too many errors to correct all errors, contrary to RS which will surely fail and

even silently by decoding wrong messages without any detection[6]), but they surely can't correct as many errors as Reed–Solomon. Choosing between a near-optimal and an optimal FEC is mainly a concern of speed.

Lately, the research field on Reed–Solomon has regained some vitality since the discovery of w:List_decoding (not to confuse with soft decoding), which allows to decode/repair more symbols than the theoretical optimal limit. The core idea is that, instead of standard Reed–Solomon which only do a unique decoding (meaning that it always results in a single solution, if it cannot because it's above the theoretical limit the decoder will return an error or a wrong result), Reed–Solomon with list decoding will still try to decode beyond the limit and get several possible results, but by a clever examination of the different results, it's often possible to discriminate only one polynomial that is probably the correct one.

A few list decoding algorithms are already available that allows to repair up to `n - sqrt(n*k)`[7] instead of `2*e+v <= (n-k)`, and other list decoding algorithms (more efficient or decoding more symbols) are currently being investigated.

# Third-party implementations

Here are a few implementations of Reed–Solomon if you want to see practical examples:

- Purely functional pure-Python Reedsolomon library (https://github.com/tomerfiliba/reedsolomon) by Tomer Filiba and LRQ3000, inspired and expanding on this tutorial by supporting more features.
- Object-oriented Reed Solomon library in pure-Python (https://github.com/lrq3000/unireedsolomon) by Andrew Brown and LRQ3000 (same features as Tomer Filiba's lib, but object-oriented so closer to mathematical nomenclatura).
- Reed-Solomon in the Linux Kernel (http://lxr.free-electrons.com/source/lib/reed_solomon/) (with a userspace port here (https://github.com/tierney/reed-solomon), initially ported from Phil Karn's library libfec (http://www.ka9q.net/code/fec) and libfec clone (https://github.com/quiet/libfec)).
- ZXing (Zebra Crossing) (https://github.com/zxing/zxing/), a full-blown library to generate and decode QR codes.
- Speed-optimized Reed-Solomon (https://github.com/catid/wirehair/blob/master/wirehair-mobile/wirehair_codec_8.cpp) and Cauchy-Reed-Solomon (https://github.com/catid/longhair) with lots of comments and an associated blog (http://catid.mechafetus.com/news/news.php) for more details.
- Another high speed-optimized Reed-Solomon (https://github.com/klauspost/reedsolomon) in Go language.
- Port of code in the article (https://github.com/mersinvald/reed-solomon-rs) in Rust language.
- C++ Reed Solomon implementation (https://github.com/mersinvald/Reed-Solomon) with on-stack memory allocation and compile-time changable msg\ecc sizes for embedded, inspired by this tutorial.
- Interleaved Reed Solomon implementation in C++ (https://github.com/NinjaDevelper/ReedSolomon) by NinjaDevelper.
- FastECC, C++ Reed Solomon implementation in O(n log n) using Number Theoretic Transforms (NTT) (https://github.com/Bulat-Ziganshin/FastECC) (open source, Apache License 2.0). Claims to have fast encoding rates even for large data.
- Leopard-RS (https://github.com/catid/leopard), another library in C++ for fast large data encoding, with a similar (but a bit different) algorithm as FastECC.

- Pure Go Implementation (https://github.com/colin-davis/reedSolomon) by Colin Davis (open source, GLPv3 License).
- Shorthair (https://github.com/catid/shorthair), an implementation of error correction code combined with UDP for fast reliable networking to replace the TCP stack or UDP duplication technique (which can be seen as a low efficiency redundancy scheme). Slides (https://github.com/catid/shorthair/blob/master/docs/ErasureCodesInSoftware.pdf) are provided, describing this approach for realtime game networking.
- Pure C Implementation (https://github.com/jackchouchani/reedsolomon) optimised using uint8_t and very efficient.
- hqm rscode (https://github.com/hqm/rscode) ANSI C implementation, for 8-bit symbols

# External links

- w:Reed–Solomon_error_correction
- w:Finite_field_arithmetic
- Short tutorial on Reed-Solomon encoding with an introduction to finite fields (http://research.swtch.com/field)
- A practical tutorial article to implement the core mathematical (galois field) operators (https://www.academia.edu/31243287/Reed_Solomon_Encoding_Simplified_Explanation_for_Programmers).

# References


1. Optimizing a reed-solomon encoder, question on StackOverflow.com http://stackoverflow.com/questions/30363903/optimizing-a-reed-solomon-encoder-polynomial-division
2. Tilavat, V., & Shukla, Y. (2014). Simplification of procedure for decoding Reed–Solomon codes using various algorithms: an introductory survey. International Journal of Engineering Development and Research, 2(1), 279-283.
3. Sarwate, D. V., & Morrison, R. D. (1990). Decoder malfunction in BCH decoders. Information Theory, IEEE Transactions on, 36(4), 884-889.
4. Richard E. Blahut, "Algebraic Codes for Data Transmission", 2003, chapter 7.6 "Decoding in Time Domain"
5. Lin, S. J., Chung, W. H., & Han, Y. S. (2014, October). Novel polynomial basis and its application to reed-solomon erasure codes. In Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on (pp. 316-325). IEEE.
6. Sofair, Isaac. "Probability of miscorrection for Reed-Solomon codes." Information Technology: Coding and Computing, 2000. Proceedings. International Conference on. IEEE, 2000.
7. "Reed-Solomon Error-correcting Codes - The Deep Hole Problem", by Matt Keti, Nov 2012