QR Code Model 2 Structure and Algorithms

Franck Jeannot

Montréal, Canada, June 2024, AB826, v1.0

Abstract

This article reviews the structures and algorithms used in QR codes.

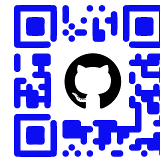
Keywords: QR structure, Galois, Reed-Solomon, Bose-Chaudhuri-Hocquenghem (BCH), fancyqr, ISO 18004

1. Introduction

A **QR** code (**Quick Response** code [1]) is a type of **two-dimensional matrix barcode**, invented in 1994, by Japanese company Denso Wave[1],initially for labelling automobile parts [1].It can encode a wide variety of data types, including numeric, alphabets, special characters and binary data as well. There are 40 QR versions in the QR code standard ISO IEC 18004[2].QR codes were designed to allow high-speed component scanning. The smallest square dot or pixel (series of black and white squares) element of a QR code is called a **module**.

2. QR versions

Below, a QR Version 2 (25×25) made with *pst-barcode* [4] (Fig A, left) and a QR made with *fancyqr* package mixed with github logo (Fig B, right) are displayed:



[1] a division of Denso, which is a subsidiary of the automobile company Toyota Motor Corporation [2]

[2] forty sizes of QR Code symbols are referred to as Version 1, Version 2 ... Version 40,with reference to section 7.3.1 Symbol Versions and sizes from ISO IEC 18004:2000 [3]

For QR Code symbols, symbol versions are referred to in the form Version V-E where V identifies the version number (1 to 40) and E indicates the error correction level (L(low), M(medium), Q(quality), H(high)[3] . There are 4 levels of **Reed–Solomon** error correction, L (7%), M (15%), Q (25%), H(30%). For Micro QR code symbols, symbols versions are referred to in the form MV-E where the letter M indicates the Micro QR Code format and V (with arange of 1 to 4) and E (with values L, M and Q). The largest possible code, Version 40, allowed under the QR code standard is a matrix of 177×177 pixels (or modules), and the smallest, Version 1, is 21×21 pixels. The version of the code gives its size, as the code matrix will be a square of 17 + Version * 4 modules.. Each higher version number comprises 4 additional modules per side. Like with other types of bar codes, it is recommended to have an empty area around the graphic, which makes it easier for devices to read the bar code. This quiet area is ideally 4 modules wide [6].

To simplify the determination of the size of a QR code, it depends on the desired **E**rror **C**orrection **C**apability (**ECC**), as well as the size and type of data intended for inclusion in the QR code. By understanding the type of data (Numeric, Alphanumeric, Binary, Kanji), its size, and the chosen ECC level, we can ascertain the appropriate Version of the QR code. A simplified table example:

Version	Modules	ECC Level	Data bits (mixed)	Numeric	Alphanumeric	Binary
1	21×21	L	152	41	25	17
		M	128	34	20	14
		Q	104	27	16	11
		H	72	17	10	7
2	25×25	L	272	77	47	32
		M	224	63	38	26
		Q	176	48	29	20
		H	128	34	20	14

Figure (1): Simplified table (without Kanji) to determine the version of QR [7].

[3] paragraph 5.3.3 from ISO18004:2015 [5]

3. QR Symbol Structure and components

Each QR Code symbol shall be constructed of nominally square modules set out in a regular square array and shall consist of a encoding region and function patterns, namely finder (Finder Pattern)(**FIP**), separator, timing patterns, and alignment patterns. Herebelow is a Version 2 structure (25 modules X 25 modules) (simplified regarding the Data Units and EC codewords that fill entirely the symbol):

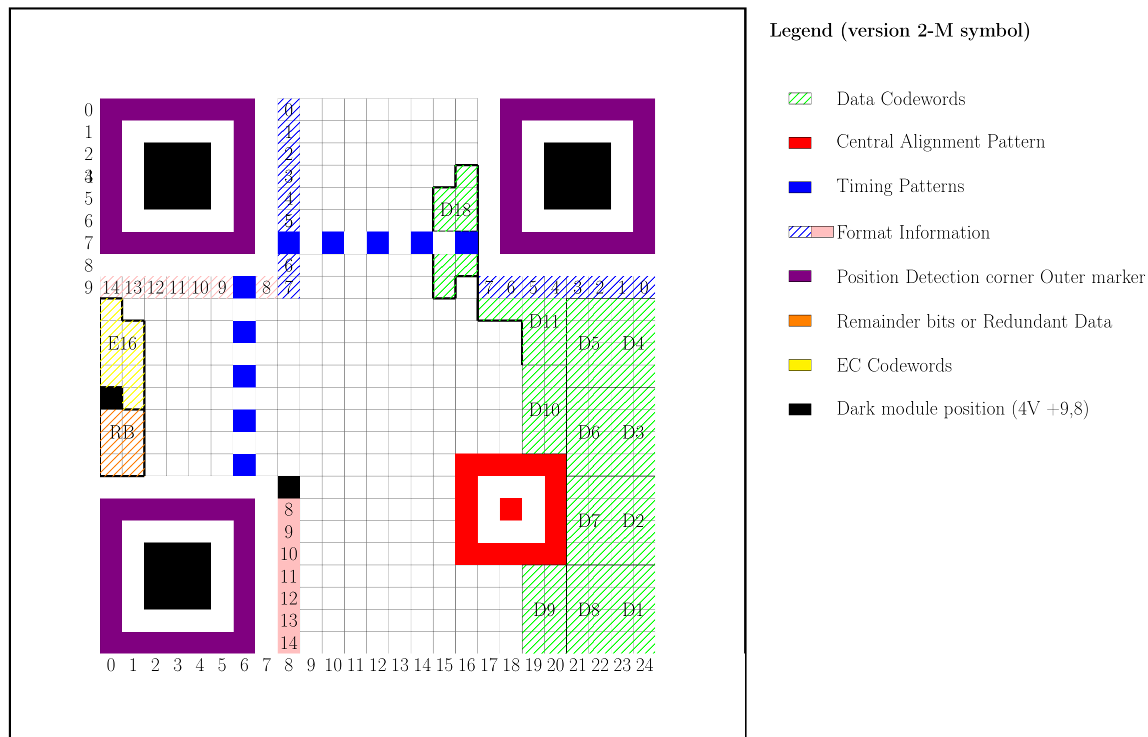


Figure (2): Standard QR code structure main components Version 2-M (medium)

QR code Model 1 and 2 have a square shape and on its three corners that are typically square-shaped patterns—finder patterns (FP), which are used to locate the code and to determine its dimensions and rotation [8]. Essential components:

1. Timing patterns (TP): they interconnect finder patterns and are formed by sequence of alternating dark and light modules and used to determine the size of a module, the number of rows and columns, and possible distortion of a code
2. Format information: contains additional information such as used error correction level (4 options) or a mask patterns number (7 options), which are required for decoding a QR Code.

3. Quiet zone: blank margin around the QR code
4. Extension patterns: markers for the alignment of the QR code (model 1)
5. Alignment patterns: markers for the alignment of the QR code (models 2
6. Version information: data giving the QR code size, for instance 25 x 25 modules (models 2 and 2005)

4. QR Format information and BCH algorithm

The format information, as specified by ISO 18004, includes a 15-bit sequence with 5 data bits and 10 **Bose-Chaudhuri-Hocquenghem (BCH)** error correction bits calculated using the (15, 5) BCH code.

Error Correction Level	Binary Indicator
L (Low)	01
M (Medium)	00
Q (Quartile)	11
H (High)	10

Table 1: Error Correction Levels and their Binary Indicators in QR Codes

In a QR code model 2, the Format Information is repeated twice for redundancy and improved readability (refer to Figure (2): pink and blue Format information zones). It is placed in two locations: a. Around the top-left finder pattern b. Split between the bottom-left and top-right finder patterns

Reading the Format Information:
To identify which zones contain what information:
The complete 15-bit sequence can be read from the top-left zone. The bottom-left and top-right zones together form a second copy of the same 15-bit sequence. Both copies are XORed with a fixed mask pattern to improve readability. The QR code reader software uses both copies of the Format Information for verification. If one copy is damaged or unreadable, the other can be used to retrieve the necessary information.

5 Data Information Bits		10 BCH Error Correction / parity Bits
Error Correction Level (2 bits)	Mask Pattern (3 bits)	
XXXXX		XXXXXXXXXX

Table 2: 15-bit Format Information sequence in QR code as (15,5) BCH code

A (15,5) BCH code is an error-correcting code that encodes 5 data bits into a 15-bit codeword [9]. Let us implement a simplified Python function for the generator polynomial for the (15,5) BCH code:

```
def bch_generator(data_bits):
    # Generator polynomial coefficients for g(x) = x^10 + x^8 + x^5 + x^4 + x^2 + x + 1
    generator_poly = [1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1]

    # Append 10 zeros to the data_bits (since the generator polynomial is of degree 10)
    data_bits_extended = data_bits + [0] * 10

    # Perform polynomial division (modulo 2)
    for i in range(len(data_bits)):
        if data_bits_extended[i] == 1:  # Only if the bit is 1, we perform XOR with generator_poly
            for j in range(len(generator_poly)):
                data_bits_extended[i + j] ^= generator_poly[j]

    # The remainder is the parity bits
    parity_bits = data_bits_extended[-10:]

    # The final codeword is the concatenation of the original data_bits and the parity_bits
    codeword = data_bits + parity_bits

    return codeword

# Example usage
data_bits = [1, 0, 1, 1, 0]  # Example 5 data bits
codeword = bch_generator(data_bits)
print("data_bits:", data_bits)
print("Codeword:", codeword)
```

Listing 1: Python function for the generator polynomial for the BCH code

```
$ python bch_generator.py
data_bits: [1, 0, 1, 1, 0]
Codeword: [1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0]
```

Listing 2: Python function for the generator polynomial for the BCH code - results

The generator polynomial for the (15,5) BCH code is:

$$g(x) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$$

The generator polynomial is used to generate the 15-bit codewords from the 5 data bits by multiplying the data polynomial by $g(x)$. The resulting 15-bit codeword includes 5 data bits and 10 parity bits, which are used for error detection and correction.

5. QR models and history

QR Code is an International Standard ISO/IEC 18004. QR Code is only used by the GS1 system to encode the GS1 Digital Link URI syntax. For full technical aspects of QR Code, see ISO/IEC 18004 [10].
Models:

1. QR Code Model 1 was the original specification for QR Code and is described in AIM ITS 97-001 International Symbology Specification-QR Code
2. QR Code Model 2 was an enhanced form of the symbology with additional features (primariliy the addition of alignement patterns to assist navigation in larger symbols), and was the basis of the first edition of version ISO/IEC 18004:2000 [3] then updated with ISO/IEC 18004:2015.

6. QR hexadecimal versions

The hexadecimal codes in the extract below [11] represents the raw version bits that specify different versions of QR codes, specifically for versions 7 through 40. In QR code technology, the version of a QR code determines its size and data capacity. Versions range from 1 to 40, with higher versions corresponding to larger QR codes that can store more data. Each hexadecimal value in the array encodes the version information for a specific QR code version:

```
/**
  * See ISO 18004:2006 Annex D.
  * Element i represents the raw version bits that specify version i +
    7
  * https://github.com/zxing/zxing/blob/master/core/src/main/java/com/
    google/zxing/qrcode/decoder/Version.java
  */
 private static final int[] VERSION_DECODE_INFO = {
      0x07C94, 0x085BC, 0x09A99, 0x0A4D3, 0x0BBF6,
      0x0C762, 0x0D847, 0x0E60D, 0x0F928, 0x10B78,
      0x1145D, 0x12A17, 0x13532, 0x149A6, 0x15683,
      0x168C9, 0x177EC, 0x18EC4, 0x191E1, 0x1AFAB,
      0x1B08E, 0x1CC1A, 0x1D33F, 0x1ED75, 0x1F250,
      0x209D5, 0x216F0, 0x228BA, 0x2379F, 0x24B0B,
```

```
13        0x2542E, 0x26A64, 0x27541, 0x28C69
14    };
```

Listing 3: zxing/qrcode/decoder/Version.java extract

The index of each value in the array corresponds to the QR code version minus 7. For example, the first value (0x07C94) represents version 7, the second (0x085BC) represents version 8, and so on. These hexadecimal values are used in the process of decoding QR codes. When a QR code reader scans a code, it uses these values to determine the version of the QR code being read.

7. Alignements patterns

QR Code alignment patterns are defined in the table of ISO/IEC 18004:2000 Annex E [12]. In order to calculate the positions of the coordinates we can use the provided below python function for an approximation depending on versions. The coordinates provide the center of each alignement pattern.

We consider n as the width of the QR code, $m+1$ is the length of the sequence, the 3 rd formula is from the ideal condition $n = 6.5 \cdot 2 + md$, and the name of the c means a correction (based on annex E of ISO IEC 18004:2015). Each row is an arithmetic sequence except the first element 6. The difference d of this sequence is a function of the version v :

$$\begin{aligned} n &= 4v + 17 \\ m &= \lfloor v/7 \rfloor + 1 \\ d &= 2 \left\lceil \frac{\lceil (n-13)/m - 1/2 \rceil}{2} \right\rceil \end{aligned} \tag{1}$$

```python
def get_alignment_positions(version):
# Interpolate end points to get point
# Round to nearest int by adding half
# of divisor before division
# Floor-divide by number of intervals
# to complete interpolation
# Round down to even integer
    positions = []
    if version > 1:
        n_patterns = version // 7 + 2
        first_pos = 6
        positions.append(first_pos)
```

```
        matrix_width =  4 * version + 17
        last_pos = matrix_width - 1 - first_pos
        second_last_pos = (
            (first_pos + last_pos * (n_patterns - 2)
            + (n_patterns - 1) // 2)

            // (n_patterns - 1)

            ) & -2
        pos_step = last_pos - second_last_pos
        second_pos = last_pos - (n_patterns - 2) * pos_step
        positions.extend(range(second_pos, last_pos + 1, pos_step))
    return positions

for version in range(1, 40 + 1): # 1 to 40 inclusive
    print("V%d: %s" % (version, get_alignment_positions(version)))
```

Listing 4: Python function to get center alignement coordinates

```
$ python calculating-the-position-of-qr-code-alignment-patterns.py
1: []
V2: [6, 18]
V3: [6, 22]
V4: [6, 26]
V5: [6, 30]
V6: [6, 34]
....
V38: [6, 32, 58, 84, 110, 136, 162]
V39: [6, 26, 54, 82, 110, 138, 166]
V40: [6, 30, 58, 86, 114, 142, 170]
```

Listing 5: Python approximation function to get center alignement coordinates

For the Version 2, the numbers are val1=6 and val2=18. This means that the alignment patterns are to be placed at (6, 6), (6, 18), (18, 6) and (18, 18). However, we do not put alignment patterns on top of finder patterns or separators, so for a version 2 we will just consider a center with coordinates (18,18):

Version	Va1	Val2	(X,Y) coord.
QR Version 2	6	18	(18,18)
QR Version 3	6	22	..

Table 3: QR Code Versions and Center Module Positions

8. QR and related algorithms

Various algorithms are used to implement QR codes or analyze related topics. Some important algorithms include:

1. **Viola-James**' object detection framework (as used in article "*Fast Component-Based QR Code Detection in Arbitrarily Acquired Images*"
2. **Reed-Solomon** algorithm for error corrections
3. **Bose-Chaudhuri-Hocquenghem (BCH)** for QR data processing
4. **Berlekamp-Massey** algorithm for finding error locator polynomials [13]
5. **Lay-Wang-2015** for *Rectification of QR-Code Images Using the Parametric Cylindrical Surface Model* (example on bottles) [14]
6. **Base 45 Data encoding** (rfc9285) [15] for more compact QR code encoding
7. **Bresenham algorithm** [16]
8. **Peterson-Gorenstein-Zierler Algorithm** (refering to BCH) [17]

8.1. Viola-Jones framework

The Viola-Jones framework (2001) [18] detects objects by extracting Haar-like features, which are digital image features resembling **Haar wavelets** [4]. It uses integral images for quick feature calculation and a cascade of classifiers to efficiently discard non-object regions, ensuring rapid and accurate object detection. This method was applied in (Belussi, 2012) in article "*Fast Component-Based QR Code Detection in Arbitrarily Acquired Images*" [20] code detection.

8.2. Bose-Chaudhuri-Hocquenghem (BCH)

Bose-Chaudhuri-Hocquenghem [21] form a class of cyclic error-correcting codes that are constructed using polynomials over a finite field (also called a Galois field). BCH codes in QR codes correct errors [3] by encoding data with extra parity bits, ensuring accurate data retrieval despite possible corruption. Advanced details on BCH can be found in "Error Control Coding" by (Lin, 2004) [22].

[4] Orthogonal functions forming basis for wavelet transformations in signal processing [19]

9. Reed-Solomon Error Correction

Reed-Solomon error correction is a powerful method used in QR codes to ensure data integrity. Reed-Solomon codes are a group of error-correcting codes that were introduced by Irving S. Reed and Gustave Solomon in 1960 [23]. They are used to detect and correct multiple symbol errors. Reed-Solomon codes are particularly useful in digital communications and storage, such as CDs, DVDs, QR codes, and data transmission technologies.

10. Galois field

A Galois field , named after the mathematician Évariste Galois [24], is a finite field containing a set number of elements where arithmetic operations such as addition, subtraction, multiplication, and division (except by zero) are closed within the set. Galois fields are denoted as $\mathbb{F}_q$, where q is a power of a prime number.

11. Reed-Solomon Algorithm Essentials

11.1. Key Concepts

1. **Symbols and Codewords**: Reed-Solomon codes work with symbols, which are groups of bits. 2. A **codeword** is a sequence of symbols. 3. **Field**: The algorithm operates over a finite field, typically denoted as $\mathbb{F}_{2^m}$ [5], where m is a positive integer. 4. **Generator Polynomial**: The generator polynomial $g(x)$ is used to generate codewords.

Encoding Process

To encode a message using Reed-Solomon codes, the message is represented as a polynomial $m(x)$ and then multiplied by the generator polynomial $g(x)$ to produce the encoded message $c(x)$.

$$c(x) = m(x) \cdot g(x)$$

[5] $\mathbb{F}q$ represents a finite field, also known as a **Galois field**, where q is a power of a prime number. Specifically, $\mathbb{F}2^m$ denotes a finite field with 2^m elements, used for efficient arithmetic in coding theory and cryptography.

Practical example for encoding

We suppose that we have a message represented by the polynomial $m(x) = x^2 + 1$ and a generator polynomial $g(x) = x^2 + x + 1$.

- $m(x) = 1 \cdot x^2 + 0 \cdot x + 1$
- $g(x) = 1 \cdot x^2 + 1 \cdot x + 1$

The codeword polynomial $c(x)$ is calculated as:

$$c(x) = m(x) \cdot g(x) = (x^2 + 1) \cdot (x^2 + x + 1)$$

Expanding this product:

$$c(x) = x^4 + x^3 + x^2 + x^2 + x + 1 = x^4 + x^3 + 2x^2 + x + 1$$

Since we are in a finite field $\mathbb{F}_2$, where $2 \equiv 0$:

$$c(x) = x^4 + x^3 + x + 1$$

So, the codeword polynomial $c(x)$ is $x^4 + x^3 + x + 1$.

Decoding Process

The decoding process involves finding the error locations and magnitudes. The key equations used in the decoding process include the Syndrome polynomial $S(x)$ and the Error locator polynomial $\sigma(x)$.

1. **Syndrome Polynomial**: The syndrome polynomial is calculated using the received polynomial $r(x)$ and is defined as:

$$S(x) = r(\alpha^i) \quad \text{for} \quad i = 0, 1, \ldots, 2t - 1$$

where α is a primitive element of the finite field, and t is the number of errors the code can correct.

2. **Error Locator Polynomial**: The error locator polynomial $\sigma(x)$ is found using the Berlekamp-Massey algorithm or the Euclidean algorithm. It is used to locate the positions of errors in the received message.

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

Once the error locations are identified, the error values can be determined, and the original message can be reconstructed.

12. Reed-Solomon examples

We define a simplified algorithm as below:

1/ Generator Polynomial - Generates the generator polynomial g(x) needed for encoding
2/ Initialization - Sets up the Galois Field by initializing the exponential and logarithm tables
3/ Generator Polynomial Generates the generator polynomial g(x) needed for encoding
4/ Encode Message - Encodes the message polynomial $m(x)$ by multiplying it with x^t and computing the remainder when divided by $g(x)$
5/ Output Codeword - Combines the message and parity symbols to form the final codeword $c(x)$.

We can then define corresponding functions:

Function	**Details**
gf_poly_mul	Multiply two polynomials in GF(2^8)
gf_poly_add(p, q)	Add two polynomials in GF(2^8)
rs_generator_poly(nsym)	Generate a generator polynomial for RS encoding.
rs_encode_msg(msg_in, nsym)	Encode a message using Reed-Solomon codes.

Table 4: Function Descriptions for Reed-Solomon simplified Algorithm example

Generator Polynomial: $g(x) = \prod_{i=0}^{t-1}(x + \alpha^i)$
Codeword Polynomial: $c(x) = m(x) \cdot x^t + r(x)$, $r(x) = (m(x) \cdot x^t) \mod g(x)$

Algorithm 1 Reed-Solomon Encoding simplified Algorithm example

```
1: function GF_POLY_MUL(p, q)
2:     r ← [0] × (len(p) + len(q) − 1)
3:     for j = 0 to len(q) − 1 do
4:         for i = 0 to len(p) − 1 do
5:             r[i + j] ← r[i + j] ⊕ gf_mul(p[i], q[j])
6:         end for
7:     end for
8:     return r
9: end function
10: function GF_POLY_ADD(p, q)
11:     r ← [0] × max(len(p), len(q))
12:     for i = 0 to len(p) − 1 do
13:         r[i + len(r) − len(p)] ← p[i]
14:     end for
15:     for i = 0 to len(q) − 1 do
16:         r[i + len(r) − len(q)] ← r[i + len(r) − len(q)] ⊕ q[i]
17:     end for
18:     return r
19: end function
20: function RS_GENERATOR_POLY(nsym)
21:     g ← [1]
22:     for i = 0 to nsym − 1 do
23:         g ← gf_poly_mul(g, [1, gf_exp[i]])
24:     end for
25:     return g
26: end function
27: function RS_ENCODE_MSG(msg_in, nsym)
28:     init_tables()
29:     gen ← rs_generator_poly(nsym)
30:     msg_out ← [0] × (len(msg_in) + nsym)
31:     msg_out[0 : len(msg_in)] ← msg_in
32:     for i = 0 to len(msg_in) − 1 do
33:         coef ← msg_out[i]
34:         if coef ≠ 0 then
35:             for j = 0 to len(gen) − 1 do
36:                 msg_out[i + j] ← msg_out[i + j] ⊕ gf_mul(gen[j], coef)
37:             end for
38:         end if
39:     end for
40:     msg_out[0 : len(msg_in)] ← msg_in
41:     return msg_out
42: end function
```

Simplified implementation of the Reed-Solomon algorithm in Python. This function focuses on encoding a message with a given generator polynomial and a finite field. For simplicity, this code will work in $GF(2^8)$ using precomputed tables for multiplication and logarithms

```
# Precomputed tables for GF(2^8)
gf_exp = [0] * 512
gf_log = [0] * 256

def init_tables(prim=0x11d):
    """ Initialize the exponential and logarithm tables for GF(2^8).
    """
    x = 1
    for i in range(255):
        gf_exp[i] = x
        gf_log[x] = i
        x <<= 1
        if x & 0x100:
            x ^= prim
    for i in range(255, 512):
        gf_exp[i] = gf_exp[i - 255]

def gf_mul(x, y):
    """ Multiply two numbers in GF(2^8). """
    if x == 0 or y == 0:
        return 0
    return gf_exp[gf_log[x] + gf_log[y]]

def gf_poly_mul(p, q):
    """ Multiply two polynomials in GF(2^8). """
    r = [0] * (len(p) + len(q) - 1)
    for j in range(len(q)):
        for i in range(len(p)):
            r[i + j] ^= gf_mul(p[i], q[j])
    return r

def gf_poly_add(p, q):
    """ Add two polynomials in GF(2^8). """
    r = [0] * max(len(p), len(q))
    for i in range(len(p)):
        r[i + len(r) - len(p)] = p[i]
    for i in range(len(q)):
        r[i + len(r) - len(q)] ^= q[i]
    return r

def rs_generator_poly(nsym):
```

```
41      """ Generate a generator polynomial for RS encoding. """
42      g = [1]
43      for i in range(nsym):
44          g = gf_poly_mul(g, [1, gf_exp[i]])
45      return g
46  
47  def rs_encode_msg(msg_in, nsym):
48      """ Encode a message using Reed-Solomon codes. """
49      init_tables()
50      gen = rs_generator_poly(nsym)
51      msg_out = [0] * (len(msg_in) + nsym)
52      msg_out[:len(msg_in)] = msg_in
53  
54      for i in range(len(msg_in)):
55          coef = msg_out[i]
56          if coef != 0:
57              for j in range(len(gen)):
58                  msg_out[i + j] ^= gf_mul(gen[j], coef)
59  
60      msg_out[:len(msg_in)] = msg_in
61      return msg_out
62  
63  # Example usage
64  if __name__ == "__main__":
65      message = [32, 91, 11, 98, 56]  # example message
66      nsym = 10  # number of error correction symbols
67      encoded_message = rs_encode_msg(message, nsym)
68      print("Original message:", message)
69      print("Number of error correction symbols:", nsym)
70      print("Encoded message:", encoded_message)
```

Listing 6: Simplified implementation of the Reed-Solomon algorithm

Then we get:

```
1  $ python galois.py
2  Original message: [32, 91, 11, 98, 56]
3  Number of error correction symbols: 10
4  Encoded message: [32, 91, 11, 98, 56, 107, 33, 43, 244, 102, 30, 52,
       87, 107, 207]
```

Listing 7: Simplified implementation of the Reed-Solomon algorithm - exampls results

References

[1] Wikipedia, QR code.
URL https://en.wikipedia.org/wiki/QR_code

[2] qrworld.
URL https://www.britannica.com/technology/QR-Code

[3] iso18004-2000, International Organization for Standardization, ISO/IEC Standard 18004: Information Technology – Automatic Identification and Data Capture Techniques – QR Code Bar Code Symbology Specification, Geneva, Switzerland, 2000.
URL https://github.com/yansikeim/QR-Code/blob/master/ISO%20IEC%2018004%202015%20Standard.pdf

[4] Terry Burton, Herbert Voß, pst-barcode, A PSTricks package for drawing barcodes; v.0.19 .
URL https://ctan.mirror.globo.tech/graphics/pstricks/contrib/pst-barcode/doc/pst-barcode-doc.pdf

[5] International Organization for Standardization, Information technology – automatic identification and data capture techniques – qr code bar code symbology specification, International Standard ISO/IEC 18004:2015, ISO/IEC, Geneva, Switzerland (September 2015).
URL https://www.iso.org/standard/62021.html

[6] QR (Quick Response) codes.
URL https://www.prepressure.com/library/technology/qr-code

[7] Information capacity and versions of the QR Code.
URL https://www.qrcode.com/en/about/version.html

[8] L. Karrach, E. Pivarciova, P. Božek, Identification of qr code perspective distortion based on edge directions and edge projections analysis, Journal of Imaging 6 (2020) 67. doi:10.3390/jimaging6070067.
URL https://www.researchgate.net/publication/342859848_Identification_of_QR_Code_Perspective_Distortion_Based_on_Edge_Directions_and_Edge_Projections_Analysis

[9] BCH codes.
URL https://web.ntpu.edu.tw/~yshan/BCH_code.pdf

[10] gs1ca.org, Barcoding for Designers, Printers and Packagers.
URL https://gs1ca.org/gs1ca-components/documents/Barcoding-for-Designers-Printers-and-Packagers.pdf

[11] QR decoder library.
URL https://github.com/zxing/zxing/blob/master/core/src/main/java/com/google/zxing/qrcode/decoder/Version.java

[12] Alignment pattern locations.
URL https://www.thonky.com/qr-code-tutorial/alignment-pattern-locations

[13] QR decoder library.
URL https://github.com/dlbeer/quirc/blob/master/lib/decode.c

[14] K.-T. Lay, L.-J. Wang, C.-H. Wang, Rectification of qr-code images using the parametric cylindrical surface model, in: 2015 International Symposium on Next-Generation Electronics (ISNE), IEEE, 2015. doi:10.1109/isne.2015.7132033.

[15] Base45 Data Encoding.
URL https://datatracker.ietf.org/doc/html/rfc9285

[16] Wikipedia, Bresenham's algorithm.
URL https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

[17] BCH codes.
URL https://simoneparisotto.com/math/misc/qrcode/qrcode.pdf

[18] P. A. Viola, M. J. Jones, Rapid object detection using a boosted cascade of simple features., in: Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2001), IEEE Computer Society, 2001, pp. 511–518. doi:10.1109/CVPR.2001.990517.
URL https://www.researchgate.net/publication/3940582_Rapid_Object_Detection_using_a_Boosted_Cascade_of_Simple_Features

[19] A. Haar, Theorie der orthogonalen Funktionensysteme, Giesecke & Devrient, 1910.

[20] L. F. F. Belussi, N. S. T. Hirata, Fast component-based qr code detection in arbitrarily acquired images, Journal of Mathematical Imaging and Vision 45 (3) (2012) 277–292. doi:10.1007/s10851-012-0355-x.

[21] BCH code.
URL https://en.wikipedia.org/wiki/BCH_code

[22] S. Lin, D. J. Costello, Error Control Coding: Fundamentals and Applications, 2nd Edition, Pearson-Prentice Hall, Upper Saddle River, NJ, 2004.

[23] Reed, Irving S. and Solomon, Gustave, Polynomial codes over certain finite fields, Journal of the Society for Industrial and Applied Mathematics 8 (2) (1960) 300–304.
URL https://sites.math.rutgers.edu/~zeilberg/akherim/reed.pdf

[24] Galois, Évariste and Liouville, Joseph, Oeuvres mathématiques d'Évariste Galois, Bachelier, Paris, 1846.
URL https://gallica.bnf.fr/ark:/12148/bpt6k9800489w/f22.item