Mathematics of QR Codes: History, Theory, and Implementation

Daniel Sun
Mirman School, Los Angeles, USA

Abstract

QR codes are a central part of the world today, bridging the physical and digital worlds through a simple, scannable pattern. Their uses are varied, ranging from displaying menus at a restaurant to product packaging. This essay explains the structure of QR codes, the error-correction mechanisms that ensure reliable decoding, and their variety of encoding modes and versions. It also discusses the concepts of Galois Fields and Reed-Solomon error correction underpinning the code. By exploring the design and inner workings of QR codes, it illustrates how structured patterns can efficiently store and transmit data. A link to sample code for generating QR codes is also provided.

Keywords: QR code, alignment markers, masking pattern, Reed-Solomon error correction algorithm

1 Introduction

A QR code, short for **Quick Response** code, is a two-dimensional, matrix-style barcode composed of black and white pixels. It was invented in 1994 by Masahiro Hara at the Denso Wave automotive company, originally to label automotive parts. [1]

QR codes have become a ubiquitous part of modern life, appearing on everything from product packaging and advertisements to medical forms and academic posters. Their widespread adoption is largely due to their ability to store relatively large amounts of information in a compact, machine-readable format that can be scanned quickly and reliably by a wide

range of devices.

QR codes can store several types of data: numeric (digits only), alphanumeric (digits and capital letters plus a few special characters), binary (general 8-bit binary strings), and Kanji (characters from one of the three scripts used in written Japanese, originating from Chinese characters). [2]

A QR code must incorporate large square patterns located in three corners, except the bottom right. These are **alignment markers** that help the scanning software determine the orientation and alignment of the code.

Although the black and white pixels may appear to form a random pattern, they are carefully arranged. Large areas of uniform color are avoided to improve readability. QR codes use **masking pattern**, which is a mathematical rule that toggles certain pixels (black to white or white to black) in a predictable way. The decoder knows how to reverse this step. The goal is to distribute dark and light areas more evenly, improving contrast and reducing scanning errors.

In this expository essay, we focus on the internal structure and encoding mechanisms of QR codes. We examine the role of alignment of black and white pixels in guiding the scanning process, and explain the mathematical foundation of the error correction algorithm that allows QR codes to retain functionality even when partially obscured or damaged.

2 Historical Overview

Before QR codes were invented, nearly all automated identification systems relied on one-dimensional barcodes. These barcodes, while useful, had significant limitations: they could not store much data and were difficult to scan from certain angles or when damaged. In 1994, the Japanese company Denso Wave sought a more efficient method for labeling and identifying automotive parts.

Masahiro Hara, an engineer at Denso Wave, drew inspiration from the game of Go and proposed a new two-dimensional matrix code. Unlike barcodes, this design could store

much more information and was better suited to incorporating error correction, making the code more robust against damage and scanning misalignment.

However, early versions of the QR code had issues with orientation. If a part was rotated or tilted, the scanner could misread the data. To solve this, Hara and his team added large 7×7 square position detection patterns in three of the code's four corners. These patterns, with their distinctive bullseye appearance, allow scanners to determine accurately the QR code's alignment regardless of rotation.

Interestingly, Denso Wave chose not to enforce patent rights on the QR code, effectively offering it as a gift to the world. However, "QR Code" remains a registered trademark of Denso Wave.

A major turning point in the adoption of QR codes occurred during the outbreak of bovine spongiform encephalopathy (commonly known as mad cow disease) in the late 1990s and early 2000s. The food industry needed a way to trace the origin and manufacturing history of beef products. QR codes, with their ability to store detailed information in a compact format, became a vital tool for traceability.

Since then, QR codes have expanded far beyond industrial use. Today, they are widely used in everyday life, from restaurant menus and product packaging to digital payments. In countries like China, QR codes have become the dominant method of mobile payment. Some tombstones even display QR codes that link to websites sharing the life stories of the deceased.

3 The structure of QR codes

QR codes can store between 17 and 2,953 bytes of data, depending on their version and error correction level. There are 40 versions of QR codes, each with a different size. The smallest, version 1, is a 21×21 grid. Each subsequent version increases the side length by 4 pixels, so a version 40 QR code measures 177×177. The side length of a version x QR code is given by the formula $21 + 4(x - 1)$. Versions 1 through 4 are the most commonly used in everyday applications.

A QR code consists of several fixed components in addition to the data itself:

- **Finder Patterns:** Three 7×7 squares are placed in the top-left, top-right, and bottom-left corners of the matrix, each surrounded by a one-pixel-wide white border. These are known as position markers and are used to locate the corners of the QR code. Only three are used so that the orientation of the code can be determined.

- **Alignment Patterns:** Appearing in QR code versions 2 and above, alignment patterns help correct distortion, such as warping or curving in larger codes. These patterns are small 5×5 squares with alternating dark and light modules. The number of alignment patterns is given by $N^2 - 3$, where N ranges from 2 to 6, depending on the version. An $N \times N$ grid is formed using the inner corners of the position markers as vertices, and the alignment patterns are placed at the intersections of this grid.

- **Timing Patterns:** Created by connecting the lines between the inner corners of the finder patterns, these alternate between black and white, beginning with black after the outer strip of white on the finder pattern. As the name implies, these help decide the speed of the scanning.

- **Dark Module:** At the 9th column and 8th row from the bottom, there will always be a black bit.

- **Formatting Information:** Around the Finder Patterns, formatting information is encoded. When decoded, it reveals the masking pattern (one of 8 possibilities) and the error correction level (one of 4 possibilities), which together form 5 bits. An additional 10 bits of error correction are added, and the entire 15-bit sequence is duplicated. This 30-bit block is then inserted into predefined locations in the QR code.

- **Version Information:** For Versions 7 and above, a special Version Information field is included in the QR code. This field encodes the version number using an 18-bit

sequence: 6 bits for the version data and 12 bits for error correction. This 18-bit block is placed in two fixed areas adjacent to the top-right and bottom-left finder patterns, allowing the scanner to identify the version even if part of the QR code is damaged. Versions 1 through 6 do not include this field, as their version can be inferred directly from the matrix size.

The actual data is placed in a zig-zag pattern, starting from the bottom-right corner of the QR code and moving upward, then left in alternating columns. As data is encoded byte by byte, the earlier bytes are placed further along this zig-zag path. If the path encounters a timing pattern, alignment pattern, or any reserved area, the data placement skips over it and continues in the next available pixel.

4 QR Code Generation Process

- QR code data is typically encoded using character sets such as UTF-8 or ISO-8859-1 (Latin-1), which map most common characters to values between 0 and 255. However, QR codes can also encode data in formats like Kanji, alphanumeric characters, or numeric-only strings.

  If the encoded message does not fill the QR code's maximum data capacity, padding is added. First, zero bits are appended until the total length is a multiple of 8. Then, padding bytes are alternately added using the values 236 (binary: 11101100) and 17 (binary: 00010001). These two bytes repeat in sequence until the data area is completely filled. These specific values are chosen because their bit patterns avoid large solid black or white blocks, which are more prone to scanning errors.

- Reed-Solomon error correction is used in QR codes to enhance reliability. Depending on the error correction level and the length of the message, a specific number of error correction bits —- called codewords -— are appended to the message. The message is first divided into multiple blocks, and polynomial division is performed on each block to generate its own set of error correction codewords (i.e., redundancy). These codewords help recover lost or corrupted data. To further improve resilience against localized damage, the data and error-correction bits are sometimes interleaved across the QR code. Section 6 provides additional details on Reed-Solomon error correction.

- Based on the QR code version, the matrix is initialized, and fixed patterns—such as the timing, alignment, and finder patterns—are placed accordingly. The number and placement of alignment patterns depend on the version number. Next, the encoding mode and message length are inserted near the bottom-right corner. The data bits are then embedded in a zigzag pattern, starting from the bottom-right and moving upward in alternating vertical columns. If the pattern encounters an obstruction (such as a functional pattern), the placement skips over it and continues in the next appropriate position. Once all data bits are placed, a 4-bit terminator is added to mark the end of the message. Finally, the error correction codewords are appended.

- One of eight masking patterns is applied to the QR code to minimize large areas of solid black or white, which can interfere with scanning. Each masking pattern is identified by an ID ranging from 0 to 7. The masking process specifically targets and reduces features such as large stripes, uniform patches, and patterns that resemble the finder patterns. These visual artifacts can confuse scanners and disrupt accurate decoding, so selecting the optimal mask is a crucial step in QR code generation.

- To complete the QR code, format, and version information are added. The format information, which includes the error correction level and mask pattern, is encoded in 15 bits and placed in two locations: around the top-left finder pattern and symmetrically near the top-right and bottom-left finder patterns. If the QR code is version 7 or higher, version information is also included. This 18-bit sequence (containing both data and error correction) is placed along the unused areas adjacent to the top-right and bottom-left finder patterns.

5 Mathematical Foundations of QR Code Generation

5.1 Properties of a Galois Field

QR codes utilize a Galois Field with 256 elements. A **Galois Field** (also known as a **finite field**) is a set containing a finite number of elements in which the operations of addition, subtraction, multiplication, and division (excluding division by zero) are defined and satisfy the fundamental properties of a field [3, 4].

The Galois Field is named after Evariste Galois (1811–1832), a brilliant French mathematician who laid the foundations of group and field theory before his untimely death at the age of 20.

The simplest Galois fields are those with a prime number of elements. For instance, the field GF(7) can be constructed using the integers modulo 7. This set satisfies all the field properties.

The key properties of a field are:

- **Commutativity:** $a + b = b + a$ and $a \cdot b = b \cdot a$
- **Associativity:** $a + (b + c) = (a + b) + c$ and $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
- **Identities:** There exist two distinct elements 0 and 1 such that $a+0 = a$ and $a \cdot 1 = a$.
- **Additive Inverses:** For every a, there exists $-a$ such that $a + (-a) = 0$. This element is called the **additive inverse of** a.
- **Multiplicative Inverses:** For every a, there exists a^{-1} such that $a \cdot a^{-1} = 1$. This element is the **multiplicative inverse of** a.
- **Distributivity:** $a \cdot (b + c) = a \cdot b + a \cdot c$.

Importantly, all Galois fields of the same order are isomorphic, meaning they are structurally identical regardless of how they are constructed. Galois fields exist only for orders that are powers of a prime number.

While fields with a prime number of elements are relatively straightforward to construct, fields of non-prime orders, such as GF(4) or GF(256), are more complex. Here's how we construct a few of these:

5.2 Constructing GF(4)

Start with the elements 0 and 1. Introduce a new element, denoted α, which is distinct from both. This gives us the set $0, 1, \alpha, \alpha + 1$. To define addition, we stipulate that each

non-zero element is its own additive inverse, i.e., $a + a = 0$.

To define multiplication, observe that since α is neither 0 nor 1, and the field must close under multiplication, we define: $\alpha^2 = \alpha + 1$. This choice ensures closure and consistency with field properties. The remaining products can be derived using the distributive property.

5.3 Constructing GF(8)

We begin with the same property that every element is its own additive inverse. All elements in the field can be constructed from the basis $0, 1, \alpha, \alpha^2$, resulting in a total of 8 distinct elements. To define multiplication, we specify that $\alpha^3 = \alpha + 1$. The remaining products can then be determined using the distributive property.

5.4 Constructing GF(256)

We define GF(256) with two key properties: for any element x, we have $x + x = 0$, and multiplication is governed by the irreducible polynomial

$$\alpha^8 = \alpha^4 + \alpha^3 + \alpha^2 + 1.$$

These properties ensure that the field satisfies all necessary axioms and allow the entire Galois field to be constructed using the distributive property. In practice, addition is performed as a bitwise XOR operation, where matching bits cancel out. Multiplication involves multiplying each term and reducing the result modulo the defining polynomial to keep powers of α less than 8.

6 Reed-Solomon Error Correction

Before error correction begins, larger QR code versions require the data to be divided into multiple blocks. This block-based approach improves resilience by localizing damage. Once the data is split, error correction codewords are generated.

QR codes use Reed-Solomon error correction over the finite field GF(256). The process begins by determining the required number of error correction codewords based on the specified error correction level. The original data polynomial $D(x)$ is then multiplied by X^n, where n is the number of error correction codewords. This operation shifts the data polynomial to make space for the error correction bits.

Next, a generator polynomial $G(x)$ is constructed. For n error correction codewords, it is defined as:

$$G(x) = (x - \alpha^0)(x - \alpha^1)(x - \alpha^2)\dots(x - \alpha^{n-1}).$$

When expanded, it becomes:

$$G(x) = x^n + g_1 x^{n-1} + g_2 x^{n-2} + \cdots + g_n.$$

The coefficients $g_1, g_2, g_3, \dots, g_n$ are elements of GF(256) and are used in the polynomial division step. For example, the polynomial of the generator for $n = 7$ is:

$$G(x) = x^7 + 87x^6 + 229x^5 + 146x^4 + 149x^3 + 238x^2 + 102x + 21.$$

The shifted data polynomial is then divided by $G(x)$ using polynomial division over GF(256). The remainder $R(x)$ is the error correction polynomial. Its coefficients, which range from 0 to 255, form the error correction codewords.

Finally, the data and error correction codewords are interleaved to distribute them evenly throughout the QR code matrix. This interleaving helps ensure that localized damage affects portions of multiple blocks rather than corrupting a single block entirely.

During scanning, efficient decoding algorithms—such as Berlekamp-Massey—are used to reverse the encoding process. These algorithms compute an error locator polynomial that identifies the positions of errors. Once errors are located, they can be corrected.

Once everything is in order, the QR code is read. The result will be the original data unless the amount of damaged data is more than the error correction level can handle.

7 Historical Note on Reed and Solomon

Irving Reed (1923–2012) and Gustave Solomon (1930–1996) were the inventors of the Reed-Solomon error correction code. Reed earned his Ph.D. from Caltech and later joined MIT Lincoln Laboratory, where he developed the Reed-Muller code in 1954—another foundational error correction scheme. While working at Lincoln Lab, Reed met Solomon, and in 1960, they collaboratively developed the Reed-Solomon code.

Their groundbreaking work, titled *Polynomial Codes over Certain Finite Fields*, introduced the concept of treating data as the coefficients of a polynomial over a finite field [5]. Although their method was powerful, decoding remained a significant challenge. This was resolved nine years later, when Elwyn Berlekamp (1940–2019) and James Massey (1934–2013) developed an efficient algorithm for computing the error locator polynomial. This advancement made Reed-Solomon codes not only theoretically robust but also practically usable, leading to their widespread adoption in digital communication and data storage systems [6,7].

8 Generating QR Codes in Python

Generating a QR Code in Python consists of two main steps: processing the input and placing the processed bits into the matrix.
First, we focus on the processing. Since Python does not feature a built-in function to find the lengths of the error correction bits and data bits for given QR code specifications, we must manually import these tables into Python. After importing them, we ask for the message, version, and error correction level. We then append the encoding and length information to the original data bits to create a bitstream. Padding with 0s, 236, and 17 is applied to prepare for error correction. At this stage, we also check if the message is too long and raise a ValueError if it is.

Now that we have our converted message, we begin Reed-Solomon correction. To speed up multiplication and division, we create exp and log tables over GF(256), enabling multiplication to be treated as addition. After creating the multiplication function, we define the polynomial multiplication function, which uses bitwise XOR and addition to multiply polynomials. Similarly, we define a polynomial division function using our exp and log ta-

bles. Using polymult, we construct the generator polynomial by repeatedly multiplying its factors. We then divide our converted message by the generator polynomial using polydiv and append the result to our data to get the final message.

Now we can begin placing the processed bits into the matrix. We first prepare the fixed patterns: finder patterns, timing patterns, and the dark module, placing a separator around each finder pattern. Then, using a lookup table to determine the locations of the alignment patterns, we place each alignment pattern in its correct location. We also reserve space for the format and version information. At this point, we create a copy of the matrix to preserve these fixed patterns before masking.

Beginning in the bottom-right corner, we place our final message in the matrix, making sure only to occupy unreserved bits. We go through two columns at a time, skipping column 7 as it contains the timing pattern. After placing the bits, we proceed with masking. We hardcode all 8 masking patterns into a single function and use a penalty algorithm to determine which pattern minimizes undesirable patterns such as lookalikes and large monocolor patches. [8] After masking, we restore the fixed patterns using our earlier matrix copy.

To finish, we insert the format and version information into their reserved positions, using binary polynomial division to apply BCH error correction. Once everything is in place, we use matplotlib to display the final QR code.

.

9 Supplemental Materials

You can find the complete Python code that generates the QR code below on our GitHub repository: github.com/QR-Codes. The QR code itself is also scannable and will take you directly to the same GitHub page.

Acknowledgements

I would like to express my sincere gratitude to Dr. Olga Korosteleva, Professor at California State University, Long Beach, for her invaluable guidance and support throughout this

work. Special thanks also go to Mr. Jungseub Rhee, Mr. Benjamin Gross, and Ms. Kara Luna, dedicated mathematics teachers at our school, for their encouragement and insightful feedback. I am grateful to Dr. Harold Reiter, Professor at the University of North Carolina at Charlotte, for his expert advice and mentorship. Finally, I deeply appreciate the unconditional support and understanding of my family members, without whom this journey would not have been possible.

References

[1] Bajaj, B.S. (2024). *Reliability on QR Codes and Reed–Solomon Codes*, arXiv.

[2] Downs, A.S., Sigmon, N.P., and R.E. Klima. (2022). *The Mathematics of QR Codes. Proceedings of the 26th Asian Technology Conference in Mathematics*, 145-159.

[3] Lidl, R. and H. Niederreiter. (1994). *Introduction to Finite Fields and Their Applications*, Cambridge Univ. Press. [4] Cox, D.A. (2012). *Galois Theory*, Hoboken, 2nd ed.

[5] Reed, I.S.and G. Solomon. (1960). Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2), 300-304.

[6] Berlekamp, E.R. (1968). *Algebraic coding theory*, McGraw-Hill.

[7] Massey, J.L. (1969). *Shift register synthesis and BCH decoding*, *IEEE Transactions on Information Theory* , 15(1), 122–127.

[8] International Organization for Standardization, *Information Technology — Automatic Identification and Data Capture Techniques — QR Code Bar Code Symbology Specification*, ISO/IEC 18004:2015, Geneva, Switzerland, 2015.