Building a QR Code Generator from Scratch

A Mathematical and Algorithmic Tutorial in Python

Tutorial Document

Based on ISO/IEC 18004 Standard

December 4, 2025

Abstract

This tutorial provides a comprehensive guide to implementing a QR code generator from first principles. We cover the complete mathematical foundations including Galois field arithmetic, Reed-Solomon error correction, and BCH codes. Each section includes theoretical background, mathematical derivations, and complete Python implementations. By the end of this tutorial, you will understand exactly how QR codes work and have built your own generator without relying on external libraries.

Contents

1 Introduction

QR (Quick Response) codes are two-dimensional matrix barcodes invented in 1994 by Masahiro Hara at Denso Wave, a subsidiary of Toyota. Unlike traditional barcodes that store information in one dimension, QR codes use a pattern of black and white modules arranged in a square grid to encode data in two dimensions.

1.1 Overview of the QR Code Generation Process

The QR code generation process consists of these main steps:

1. **Data Analysis**: Determine the optimal encoding mode (numeric, alphanumeric, byte, or kanji)
2. **Data Encoding**: Convert the input data into a binary bitstream
3. **Error Correction**: Generate Reed-Solomon error correction codewords
4. **Structure Final Message**: Interleave data and error correction blocks
5. **Module Placement**: Place the data bits into the QR code matrix along with function patterns
6. **Data Masking**: Apply mask patterns to improve readability
7. **Format and Version Information**: Add metadata about error correction level and mask pattern

1.2 QR Code Versions and Capacity

QR codes come in 40 versions, where version 1 is 21×21 modules and each subsequent version adds 4 modules per side:

$$\text{Size} = 4V + 17 \tag{1}$$

where V is the version number (1-40). Version 40 is 177×177 modules.

Table 1: QR Code Version Sizes and Approximate Capacities (Error Correction Level L)

Version	Size	Numeric	Alphanumeric	Bytes
1	21×21	41	25	17
2	25×25	77	47	32
5	37×37	202	122	84
10	57×57	652	395	271
40	177×177	7,089	4,296	2,953

1.3 Error Correction Levels

QR codes support four error correction levels, each providing different recovery capabilities:

1.4 References for This Section

- ISO/IEC 18004:2015 - QR Code bar code symbology specification
- Wikipedia: QR Code (`https://en.wikipedia.org/wiki/QR_code`)
- Thonky's QR Code Tutorial (`https://www.thonky.com/qr-code-tutorial/`)

Table 2: Error Correction Levels

Level	Recovery Capacity	Use Case
L (Low)	$\sim 7\%$	Maximum data capacity
M (Medium)	$\sim 15\%$	General purpose
Q (Quartile)	$\sim 25\%$	Industrial applications
H (High)	$\sim 30\%$	Harsh environments, logos

2 Mathematical Foundations: Galois Field Arithmetic

The heart of QR code error correction relies on arithmetic in **Galois Fields** (finite fields). QR codes use $GF(2^8)$, also written as $GF(256)$, which contains exactly 256 elements.

2.1 What is a Galois Field?

Definition 2.1 (Galois Field)**.** A Galois Field $GF(p^n)$ is a finite field containing exactly p^n elements, where p is a prime number (called the characteristic) and n is a positive integer. For QR codes, we use $GF(2^8)$ with $p = 2$ and $n = 8$.

In $GF(2^8)$, elements are represented as polynomials of degree at most 7 with coefficients in $GF(2) = \{0, 1\}$:

$$a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \tag{2}$$

where each $a_i \in \{0, 1\}$. This maps naturally to 8-bit bytes.

2.2 The Primitive Polynomial

To define multiplication in $GF(256)$, we need an **irreducible primitive polynomial**. QR codes use:

$$p(x) = x^8 + x^4 + x^3 + x^2 + 1 \tag{3}$$

In binary, this is $100011101_2 = 285_{10}$ (the standard notation includes the x^8 term).

Note

The primitive polynomial defines how we "wrap around" when multiplication would produce a result with degree 8 or higher. We take the remainder when dividing by $p(x)$.

2.3 Addition in GF(256)

Addition in $GF(2^8)$ is performed coefficient-wise modulo 2, which is simply the XOR operation:

$$a + b = a \oplus b \tag{4}$$

Example 2.1. Let $a = 83 = 01010011_2$ and $b = 202 = 11001010_2$:

$$a + b = 01010011_2 \oplus 11001010_2 \tag{5}$$
$$= 10011001_2 = 153 \tag{6}$$

Important

In $GF(2^8)$, subtraction is identical to addition! Since $-1 \equiv 1 \pmod 2$, we have $a - b = a + b = a \oplus b$.

2.4 Multiplication in GF(256)

Multiplication is polynomial multiplication modulo the primitive polynomial $p(x)$.

2.4.1 Direct Polynomial Multiplication

Given two polynomials $a(x)$ and $b(x)$:

1. Multiply them as regular polynomials (using XOR for coefficient addition)
2. If the result has degree ≥ 8, reduce modulo $p(x)$

Example 2.2. Multiply $a = 83$ and $b = 202$ in $GF(256)$:

First, express as polynomials:

$$a(x) = x^6 + x^4 + x + 1 \tag{7}$$

$$b(x) = x^7 + x^6 + x^3 + x \tag{8}$$

Multiply and reduce modulo $p(x) = x^8 + x^4 + x^3 + x^2 + 1$.

2.4.2 Logarithm Table Method (Efficient)

Since $GF(256)^* = \{1, 2, \ldots, 255\}$ is a cyclic group, every non-zero element can be expressed as a power of a primitive element α (typically $\alpha = 2$):

$$a = \alpha^{\log_\alpha(a)} \tag{9}$$

Then multiplication becomes:

$$a \cdot b = \alpha^{(\log_\alpha(a) + \log_\alpha(b)) \mod 255} \tag{10}$$

2.5 Python Implementation: GF(256) Arithmetic

```python
"""
Galois Field GF(256) Implementation for QR Codes
Based on the primitive polynomial: x^8 + x^4 + x^3 + x^2 + 1 (0x11d)
"""

class GF256:
    """Galois Field GF(2^8) arithmetic for QR codes."""

    PRIMITIVE_POLY = 0x11d  # x^8 + x^4 + x^3 + x^2 + 1 = 285

    def __init__(self):
        # Build exponential and logarithm tables
        self.exp_table = [0] * 512  # Extended for convenience
        self.log_table = [0] * 256

        self._build_tables()

    def _build_tables(self):
        """Build exp and log lookup tables using alpha = 2."""
        x = 1
        for i in range(255):
            self.exp_table[i] = x
            self.exp_table[i + 255] = x  # Duplicate for modulo ease
            self.log_table[x] = i

            # Multiply by alpha (2) with reduction
            x = self._multiply_no_table(x, 2)
```

```python

        self.log_table[0] = -1  # log(0) is undefined

    def _multiply_no_table(self, a, b):
        """
        Multiply two GF(256) elements without using tables.
        Uses Russian peasant multiplication with polynomial reduction.
        """
        result = 0
        while b > 0:
            if b & 1:  # If lowest bit is set
                result ^= a  # Add (XOR) a to result
            b >>= 1
            a <<= 1
            if a & 0x100:  # If degree >= 8
                a ^= self.PRIMITIVE_POLY  # Reduce modulo primitive
        return result

    def add(self, a, b):
        """Addition in GF(256) is XOR."""
        return a ^ b

    def subtract(self, a, b):
        """Subtraction in GF(256) is the same as addition."""
        return a ^ b

    def multiply(self, a, b):
        """Multiply two GF(256) elements using log tables."""
        if a == 0 or b == 0:
            return 0
        return self.exp_table[self.log_table[a] + self.log_table[b]]

    def divide(self, a, b):
        """Divide a by b in GF(256)."""
        if b == 0:
            raise ZeroDivisionError("Division by zero in GF(256)")
        if a == 0:
            return 0
        return self.exp_table[(self.log_table[a] - self.log_table[b]) % 255]

    def power(self, a, n):
        """Raise a to the power n in GF(256)."""
        if a == 0:
            return 0 if n > 0 else 1
        return self.exp_table[(self.log_table[a] * n) % 255]

    def inverse(self, a):
        """Find multiplicative inverse of a in GF(256)."""
        if a == 0:
            raise ZeroDivisionError("No inverse for 0")
        # a^(-1) = a^254 since a^255 = 1
        return self.exp_table[255 - self.log_table[a]]


# Create a global instance for convenience
gf = GF256()


def demo_gf256():
    """Demonstrate GF(256) operations."""
    print("=== GF(256) Arithmetic Demo ===\n")

    a, b = 83, 202
```

```
    print(f"a = {a}, b = {b}")
    print(f"a + b = {a} XOR {b} = {gf.add(a, b)}")
    print(f"a * b = {gf.multiply(a, b)}")
    print(f"a / b = {gf.divide(a, b)}")
    print(f"a^10 = {gf.power(a, 10)}")
    print(f"a^(-1) = {gf.inverse(a)}")

    # Verify: a * a^(-1) = 1
    print(f"\nVerification: a * a^(-1) = {gf.multiply(a, gf.inverse(a))}")

    # Show first few elements of exp table
    print(f"\nFirst 10 powers of alpha=2:")
    for i in range(10):
        print(f"  alpha^{i} = {gf.exp_table[i]}")


if __name__ == "__main__":
    demo_gf256()
```

Listing 1: Complete GF(256) Implementation

2.6 References for This Section

- Wikiversity: Reed-Solomon codes for coders (`https://en.wikiversity.org/wiki/Reed-Solomon_codes_for_coders`)
- Wikipedia: Finite field arithmetic (`https://en.wikipedia.org/wiki/Finite_field_arithmetic`)
- Research: Finite Field Arithmetic and Reed-Solomon Coding (`https://research.swtch.com/field`)

3 Polynomial Operations in GF(256)

Reed-Solomon codes work with polynomials whose coefficients are elements of $GF(256)$. We need to implement polynomial arithmetic for error correction.

3.1 Polynomial Representation

A polynomial is represented as a list of coefficients, where index i corresponds to the coefficient of x^i:

$$p(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n \tag{11}$$

is stored as `[c_0, c_1, c_2, ..., c_n]`.

3.2 Polynomial Operations

3.2.1 Addition

Add corresponding coefficients using GF(256) addition (XOR):

$$(a + b)(x) = \sum_i (a_i \oplus b_i) x^i \tag{12}$$

3.2.2 Multiplication

Convolution of coefficient sequences:

$$(a \cdot b)(x) = \sum_k \left(\sum_{i+j=k} a_i \cdot b_j \right) x^k \tag{13}$$

where multiplication of coefficients is in $GF(256)$.

3.2.3 Division

Polynomial long division, giving quotient $q(x)$ and remainder $r(x)$ such that:

$$a(x) = q(x) \cdot b(x) + r(x) \tag{14}$$

3.3 Python Implementation: Polynomial Operations

```python
"""
Polynomial operations over GF(256) for Reed-Solomon encoding.
"""

class Polynomial:
    """Polynomial with coefficients in GF(256)."""

    def __init__(self, coefficients, gf_instance):
        """
        Initialize polynomial with coefficients.
        coefficients[i] is the coefficient of x^i.
        Leading zeros are trimmed.
        """
        self.gf = gf_instance
        # Trim leading zeros (from the highest degree end)
        self.coeffs = list(coefficients)
        while len(self.coeffs) > 1 and self.coeffs[-1] == 0:
            self.coeffs.pop()

    @property
    def degree(self):
        """Return the degree of the polynomial."""
        return len(self.coeffs) - 1

    def __repr__(self):
        terms = []
        for i, c in enumerate(self.coeffs):
            if c != 0:
                if i == 0:
                    terms.append(f"{c}")
                elif i == 1:
                    terms.append(f"{c}x")
                else:
                    terms.append(f"{c}x^{i}")
        return " + ".join(terms) if terms else "0"

    def evaluate(self, x):
        """Evaluate polynomial at x using Horner's method."""
        result = 0
        for coeff in reversed(self.coeffs):
            result = self.gf.add(self.gf.multiply(result, x), coeff)
        return result

```

```
    def add(self, other):
        """Add two polynomials."""
        # Pad shorter polynomial with zeros
        max_len = max(len(self.coeffs), len(other.coeffs))
        a = self.coeffs + [0] * (max_len - len(self.coeffs))
        b = other.coeffs + [0] * (max_len - len(other.coeffs))

        result = [self.gf.add(a[i], b[i]) for i in range(max_len)]
        return Polynomial(result, self.gf)

    def multiply(self, other):
        """Multiply two polynomials."""
        result = [0] * (len(self.coeffs) + len(other.coeffs) - 1)

        for i, a in enumerate(self.coeffs):
            for j, b in enumerate(other.coeffs):
                product = self.gf.multiply(a, b)
                result[i + j] = self.gf.add(result[i + j], product)

        return Polynomial(result, self.gf)

    def scale(self, scalar):
        """Multiply polynomial by a scalar."""
        result = [self.gf.multiply(c, scalar) for c in self.coeffs]
        return Polynomial(result, self.gf)

    def divide(self, divisor):
        """
        Divide self by divisor, returning (quotient, remainder).
        Uses polynomial long division in GF(256).
        """
        if divisor.coeffs == [0] or len(divisor.coeffs) == 0:
            raise ZeroDivisionError("Division by zero polynomial")

        # Work with copies in descending order (high degree first)
        dividend = list(reversed(self.coeffs))
        div = list(reversed(divisor.coeffs))

        if len(dividend) < len(div):
            return Polynomial([0], self.gf), self

        quotient = []

        for i in range(len(dividend) - len(div) + 1):
            # Coefficient of quotient term
            coeff = self.gf.divide(dividend[i], div[0])
            quotient.append(coeff)

            # Subtract divisor * coeff from dividend
            for j in range(len(div)):
                subtract = self.gf.multiply(div[j], coeff)
                dividend[i + j] = self.gf.subtract(dividend[i + j], subtract)

        # Remainder is remaining non-zero terms
        remainder = list(reversed(dividend[len(dividend) - len(div) + 1:]))
        quotient = list(reversed(quotient))

        return Polynomial(quotient, self.gf), Polynomial(remainder, self.gf)

    def mod(self, divisor):
        """Return self mod divisor (the remainder)."""
        _, remainder = self.divide(divisor)
        return remainder
```

```python


def demo_polynomials():
    """Demonstrate polynomial operations."""
    gf = GF256()

    # Create polynomials
    p1 = Polynomial([1, 2, 3], gf)  # 1 + 2x + 3x^2
    p2 = Polynomial([4, 5], gf)     # 4 + 5x

    print("=== Polynomial Operations Demo ===\n")
    print(f"p1(x) = {p1}")
    print(f"p2(x) = {p2}")
    print(f"\np1 + p2 = {p1.add(p2)}")
    print(f"p1 * p2 = {p1.multiply(p2)}")

    q, r = p1.divide(p2)
    print(f"\np1 / p2 = {q} remainder {r}")

    # Evaluate at x=2
    print(f"\np1(2) = {p1.evaluate(2)}")


if __name__ == "__main__":
    demo_polynomials()
```

Listing 2: Polynomial Operations in GF(256)

4 Reed-Solomon Error Correction

Reed-Solomon (RS) codes are the error-correcting codes used in QR codes. They work by adding redundant codewords that allow reconstruction of lost or corrupted data.

4.1 Theory of Reed-Solomon Codes

Definition 4.1 (Reed-Solomon Code)**.** An RS code RS(n, k) over $GF(q)$ encodes k data symbols into n codewords, where $n - k$ symbols are error correction. It can correct up to $\lfloor (n-k)/2 \rfloor$ symbol errors.

For QR codes:

- The field is $GF(256)$ ($q = 256$)
- Each symbol is one byte (8 bits)
- The number of EC codewords depends on version and EC level

4.2 The Generator Polynomial

The Reed-Solomon generator polynomial of degree t (where t is the number of EC codewords) is:

$$g(x) = \prod_{i=0}^{t-1}(x - \alpha^i) = \prod_{i=0}^{t-1}(x + \alpha^i) \tag{15}$$

In $GF(256)$, subtraction equals addition, so $(x - \alpha^i) = (x + \alpha^i)$.

Example 4.1. For 7 error correction codewords (Version 1, EC Level L):

$$g(x) = (x + \alpha^0)(x + \alpha^1)(x + \alpha^2)(x + \alpha^3)(x + \alpha^4)(x + \alpha^5)(x + \alpha^6) \tag{16}$$
$$= (x + 1)(x + 2)(x + 4)(x + 8)(x + 16)(x + 32)(x + 64) \tag{17}$$

Expanding (all arithmetic in $GF(256)$):

$$g(x) = x^7 + \alpha^{87}x^6 + \alpha^{229}x^5 + \alpha^{146}x^4 + \alpha^{149}x^3 + \alpha^{238}x^2 + \alpha^{102}x + \alpha^{21} \tag{18}$$

4.3 Encoding Process

Given a message polynomial $m(x)$ of degree $k - 1$:

1. Multiply $m(x)$ by x^t to make room for EC codewords
2. Divide by $g(x)$ to get remainder $r(x)$
3. The encoded message is $m(x) \cdot x^t + r(x)$

This is called **systematic encoding** because the original message appears unchanged at the beginning.

4.4 Mathematical Formulation

$$\text{Dividend: } d(x) = m(x) \cdot x^t \tag{19}$$
$$\text{Division: } d(x) = q(x) \cdot g(x) + r(x) \tag{20}$$
$$\text{Codeword: } c(x) = d(x) - r(x) = d(x) + r(x) \tag{21}$$

Since addition and subtraction are the same in $GF(2^8)$, and $c(x)$ is divisible by $g(x)$, any valid codeword will have $c(x) \mod g(x) = 0$.

4.5 Python Implementation: Reed-Solomon Encoder

```python
"""
Reed-Solomon Error Correction Encoder for QR Codes.
"""

class ReedSolomonEncoder:
    """Reed-Solomon encoder for QR code error correction."""

    def __init__(self, gf_instance):
        self.gf = gf_instance
        self._generator_cache = {}

    def build_generator(self, num_ec_codewords):
        """
        Build generator polynomial for given number of EC codewords.

        g(x) = (x - alpha^0)(x - alpha^1)...(x - alpha^(n-1))
        """
        if num_ec_codewords in self._generator_cache:
            return self._generator_cache[num_ec_codewords]

        # Start with g(x) = 1
        gen = Polynomial([1], self.gf)

        for i in range(num_ec_codewords):
```

```
            # Multiply by (x + alpha^i)
            # This is [alpha^i, 1] as coefficients for alpha^i + x
            factor = Polynomial([self.gf.exp_table[i], 1], self.gf)
            gen = gen.multiply(factor)

        self._generator_cache[num_ec_codewords] = gen
        return gen

    def encode(self, data, num_ec_codewords):
        """
        Encode data bytes with Reed-Solomon error correction.

        Args:
            data: List of data bytes (integers 0-255)
            num_ec_codewords: Number of error correction codewords to generate

        Returns:
            List of error correction codewords
        """
        generator = self.build_generator(num_ec_codewords)

        # Create message polynomial m(x) * x^n
        # Coefficients are in order from x^0 to highest degree
        # We want the data at the high-degree end
        message_coeffs = [0] * num_ec_codewords + list(data)
        message = Polynomial(message_coeffs, self.gf)

        # Alternative approach: polynomial division
        # The remainder is our EC codewords

        # Perform division to get remainder
        remainder = self._divide_for_remainder(data, generator,
    num_ec_codewords)

        return remainder

    def _divide_for_remainder(self, data, generator, num_ec):
        """
        Compute remainder of message polynomial divided by generator.
        Uses the shift-register approach for efficiency.
        """
        # Create extended message with space for EC bytes
        result = list(data) + [0] * num_ec
        gen_coeffs = list(reversed(generator.coeffs))  # High degree first

        # For each data byte
        for i in range(len(data)):
            coeff = result[i]
            if coeff != 0:
                # XOR generator polynomial (scaled by coeff) into result
                for j in range(len(gen_coeffs)):
                    result[i + j] ^= self.gf.multiply(gen_coeffs[j], coeff)

        # The last num_ec bytes are the remainder (EC codewords)
        return result[-num_ec:]


def demo_reed_solomon():
    """Demonstrate Reed-Solomon encoding."""
    gf = GF256()
    rs = ReedSolomonEncoder(gf)

    print("=== Reed-Solomon Encoding Demo ===\n")
```

```python

    # Example from QR code specification
    # Version 1-M: 16 data codewords, 10 EC codewords
    data = [16, 32, 12, 86, 97, 128, 236, 17, 236, 17, 236, 17, 236, 17, 236, 17]
    num_ec = 10

    print(f"Data codewords ({len(data)}): {data}")
    print(f"Number of EC codewords: {num_ec}")

    # Build generator polynomial
    generator = rs.build_generator(num_ec)
    print(f"\nGenerator polynomial degree: {generator.degree}")
    print(f"Generator coefficients: {generator.coeffs}")

    # Encode
    ec_codewords = rs.encode(data, num_ec)
    print(f"\nError correction codewords: {ec_codewords}")

    # The complete encoded message
    complete = data + ec_codewords
    print(f"\nComplete codeword ({len(complete)} bytes): {complete}")


if __name__ == "__main__":
    demo_reed_solomon()
```

Listing 3: Reed-Solomon Encoder

4.6 References for This Section

- Wikipedia: Reed-Solomon error correction (https://en.wikipedia.org/wiki/Reed-Solomon_error_correction)
- Thonky: Error Correction Coding (https://www.thonky.com/qr-code-tutorial/error-correction-co
- DEV Community: QR Code Generator Part III (https://dev.to/maxart2501/let-s-develop-a-qr-co

5 Data Encoding Modes

QR codes support four primary encoding modes, each optimized for different types of data.

5.1 Mode Indicators

Each mode has a 4-bit indicator:

Table 3: Encoding Mode Indicators

Mode	Indicator (Binary)	Indicator (Decimal)
Numeric	0001	1
Alphanumeric	0010	2
Byte	0100	4
Kanji	1000	8

5.2 Character Count Indicator

The character count indicator length varies by version:

Table 4: Character Count Indicator Bit Lengths

Mode	V1-9	V10-26	V27-40
Numeric	10	12	14
Alphanumeric	9	11	13
Byte	8	16	16
Kanji	8	10	12

5.3 Numeric Mode

Encodes digits 0-9. Groups of 3 digits are converted to 10-bit binary, groups of 2 to 7-bit, and single digits to 4-bit.

$$\text{bits}(d_1 d_2 d_3) = \text{binary}(d_1 \times 100 + d_2 \times 10 + d_3) \tag{22}$$

5.4 Alphanumeric Mode

Encodes: 0-9, A-Z (uppercase only), space, $, %, *, +, -, ., /, :

Character values are assigned 0-44, and pairs of characters encode to 11 bits:

$$\text{bits}(c_1, c_2) = 45 \times \text{value}(c_1) + \text{value}(c_2) \tag{23}$$

Table 5: Alphanumeric Character Values

0-9	A	B	...	Z	(space)	$	%	*	+
0-9	10	11	...	35	36	37	38	39	40

5.5 Byte Mode

Encodes any 8-bit byte. Default encoding is ISO-8859-1, but UTF-8 can be used (not all readers support it).

Each character is simply encoded as its 8-bit value.

5.6 Python Implementation: Data Encoding

```python
"""
Data encoding modes for QR codes.
"""

# Mode indicators
MODE_NUMERIC = 0b0001
MODE_ALPHANUMERIC = 0b0010
MODE_BYTE = 0b0100
MODE_KANJI = 0b1000
MODE_TERMINATOR = 0b0000

# Alphanumeric character table
ALPHANUMERIC_TABLE = {
    '0': 0, '1': 1, '2': 2, '3': 3, '4': 4,
    '5': 5, '6': 6, '7': 7, '8': 8, '9': 9,
    'A': 10, 'B': 11, 'C': 12, 'D': 13, 'E': 14,
    'F': 15, 'G': 16, 'H': 17, 'I': 18, 'J': 19,
    'K': 20, 'L': 21, 'M': 22, 'N': 23, 'O': 24,
    'P': 25, 'Q': 26, 'R': 27, 'S': 28, 'T': 29,
```

```
    'U': 30, 'V': 31, 'W': 32, 'X': 33, 'Y': 34,
    'Z': 35, ' ': 36, '$': 37, '%': 38, '*': 39,
    '+': 40, '-': 41, '.': 42, '/': 43, ':': 44
}


def get_character_count_bits(version, mode):
    """Get the number of bits for the character count indicator."""
    if version <= 9:
        table = {MODE_NUMERIC: 10, MODE_ALPHANUMERIC: 9,
                 MODE_BYTE: 8, MODE_KANJI: 8}
    elif version <= 26:
        table = {MODE_NUMERIC: 12, MODE_ALPHANUMERIC: 11,
                 MODE_BYTE: 16, MODE_KANJI: 10}
    else:
        table = {MODE_NUMERIC: 14, MODE_ALPHANUMERIC: 13,
                 MODE_BYTE: 16, MODE_KANJI: 12}
    return table[mode]


def detect_mode(data):
    """Detect the most efficient encoding mode for the data."""
    # Check if all numeric
    if all(c.isdigit() for c in data):
        return MODE_NUMERIC

    # Check if all alphanumeric
    if all(c in ALPHANUMERIC_TABLE for c in data):
        return MODE_ALPHANUMERIC

    # Default to byte mode
    return MODE_BYTE


def encode_numeric(data):
    """Encode numeric data. Returns list of bits."""
    bits = []

    # Process in groups of 3
    i = 0
    while i < len(data):
        if i + 3 <= len(data):
            # Three digits -> 10 bits
            value = int(data[i:i+3])
            bits.extend(int_to_bits(value, 10))
            i += 3
        elif i + 2 <= len(data):
            # Two digits -> 7 bits
            value = int(data[i:i+2])
            bits.extend(int_to_bits(value, 7))
            i += 2
        else:
            # One digit -> 4 bits
            value = int(data[i])
            bits.extend(int_to_bits(value, 4))
            i += 1

    return bits


def encode_alphanumeric(data):
    """Encode alphanumeric data. Returns list of bits."""
    bits = []
```

```python

    # Process in pairs
    i = 0
    while i < len(data):
        if i + 2 <= len(data):
            # Pair -> 11 bits
            v1 = ALPHANUMERIC_TABLE[data[i]]
            v2 = ALPHANUMERIC_TABLE[data[i + 1]]
            value = 45 * v1 + v2
            bits.extend(int_to_bits(value, 11))
            i += 2
        else:
            # Single character -> 6 bits
            value = ALPHANUMERIC_TABLE[data[i]]
            bits.extend(int_to_bits(value, 6))
            i += 1

    return bits


def encode_byte(data):
    """Encode byte data. Returns list of bits."""
    bits = []

    # Convert to bytes (UTF-8 or ISO-8859-1)
    if isinstance(data, str):
        data = data.encode('utf-8')

    for byte in data:
        bits.extend(int_to_bits(byte, 8))

    return bits


def int_to_bits(value, length):
    """Convert integer to list of bits with specified length."""
    return [(value >> (length - 1 - i)) & 1 for i in range(length)]


def bits_to_bytes(bits):
    """Convert list of bits to list of bytes."""
    # Pad to multiple of 8
    while len(bits) % 8 != 0:
        bits.append(0)

    bytes_list = []
    for i in range(0, len(bits), 8):
        byte = 0
        for j in range(8):
            byte = (byte << 1) | bits[i + j]
        bytes_list.append(byte)

    return bytes_list


def encode_data(data, version, mode=None):
    """
    Encode data for QR code.

    Returns: List of bits including mode indicator and character count.
    """
    if mode is None:
        mode = detect_mode(data)
```

```python

    bits = []

    # Mode indicator (4 bits)
    bits.extend(int_to_bits(mode, 4))

    # Character count indicator
    count_bits = get_character_count_bits(version, mode)
    char_count = len(data.encode('utf-8') if mode == MODE_BYTE else data)
    bits.extend(int_to_bits(char_count, count_bits))

    # Data encoding
    if mode == MODE_NUMERIC:
        bits.extend(encode_numeric(data))
    elif mode == MODE_ALPHANUMERIC:
        bits.extend(encode_alphanumeric(data))
    else:  # MODE_BYTE
        bits.extend(encode_byte(data))

    return bits


def demo_encoding():
    """Demonstrate data encoding."""
    print("=== Data Encoding Demo ===\n")

    # Test numeric
    data_num = "12345"
    bits_num = encode_data(data_num, 1, MODE_NUMERIC)
    print(f"Numeric '{data_num}': {len(bits_num)} bits")

    # Test alphanumeric
    data_alpha = "HELLO WORLD"
    bits_alpha = encode_data(data_alpha, 1, MODE_ALPHANUMERIC)
    print(f"Alphanumeric '{data_alpha}': {len(bits_alpha)} bits")

    # Test byte
    data_byte = "Hello, World!"
    bits_byte = encode_data(data_byte, 1, MODE_BYTE)
    print(f"Byte '{data_byte}': {len(bits_byte)} bits")


if __name__ == "__main__":
    demo_encoding()
```

Listing 4: Data Encoding for QR Codes

5.7 References for This Section

- Thonky: Data Analysis (https://www.thonky.com/qr-code-tutorial/data-analysis)
- GeeksforGeeks: Introduction to Python qrcode Library (https://www.geeksforgeeks.org/introduction-to-python-qrcode-library/)

6 BCH Code for Format Information

The format information in a QR code is protected by a BCH (Bose-Chaudhuri-Hocquenghem) code, which is simpler than Reed-Solomon but also based on polynomial arithmetic.

6.1 Format Information Structure

Format information consists of 15 bits:

- 2 bits: Error correction level (L=01, M=00, Q=11, H=10)
- 3 bits: Mask pattern (0-7)
- 10 bits: BCH error correction

6.2 The (15, 5) BCH Code

The generator polynomial for the format BCH code is:

$$g(x) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1 \tag{24}$$

In binary: 10100110111_2.
This code:

- Has 5 data bits and 10 parity bits
- Minimum Hamming distance of 7
- Can correct up to 3 bit errors

6.3 Encoding Process

1. Take 5-bit format data (2 bits EC level + 3 bits mask)
2. Multiply by x^{10} (append 10 zeros)
3. Divide by generator polynomial, take remainder
4. Append remainder to original 5 bits
5. XOR with mask pattern: 101010000010010

6.4 Python Implementation: BCH Code

```
"""
BCH code for QR code format information.
"""

# BCH generator polynomial: x^10 + x^8 + x^5 + x^4 + x^2 + x + 1
BCH_GENERATOR = 0b10100110111

# Format mask pattern
FORMAT_MASK = 0b101010000010010

# Error correction level bits
EC_LEVEL_BITS = {
    'L': 0b01,
    'M': 0b00,
    'Q': 0b11,
    'H': 0b10
}


def bch_encode(data_5bits):
    """
    Encode 5 data bits using (15,5) BCH code.
```

```

    Args:
        data_5bits: 5-bit integer (EC level 2 bits + mask pattern 3 bits)

    Returns:
        15-bit encoded format information (before final XOR)
    """
    # Multiply by x^10 (shift left 10 positions)
    dividend = data_5bits << 10

    # Polynomial division to get remainder
    remainder = dividend
    for i in range(14, 9, -1):  # From bit 14 down to bit 10
        if remainder & (1 << i):  # If bit i is set
            remainder ^= BCH_GENERATOR << (i - 10)

    # Combine: original data + remainder
    return (data_5bits << 10) | remainder


def get_format_string(ec_level, mask_pattern):
    """
    Generate the complete 15-bit format string.

    Args:
        ec_level: Error correction level ('L', 'M', 'Q', 'H')
        mask_pattern: Mask pattern number (0-7)

    Returns:
        15-bit format string after XOR with mask
    """
    # Combine EC level and mask pattern
    data_5bits = (EC_LEVEL_BITS[ec_level] << 3) | mask_pattern

    # BCH encode
    encoded = bch_encode(data_5bits)

    # XOR with format mask
    return encoded ^ FORMAT_MASK


def format_bits_to_list(format_int):
    """Convert 15-bit integer to list of bits."""
    return [(format_int >> (14 - i)) & 1 for i in range(15)]


# Pre-computed format strings (for lookup)
FORMAT_STRINGS = {}
for ec in ['L', 'M', 'Q', 'H']:
    for mask in range(8):
        FORMAT_STRINGS[(ec, mask)] = get_format_string(ec, mask)


def demo_bch():
    """Demonstrate BCH encoding for format information."""
    print("=== BCH Format Information Demo ===\n")

    # Example: EC level M, mask pattern 0
    ec_level = 'M'
    mask = 0

    data_bits = (EC_LEVEL_BITS[ec_level] << 3) | mask
    print(f"EC Level: {ec_level}, Mask: {mask}")
```

```
    print(f"5-bit data: {bin(data_bits)[2:].zfill(5)}")

    encoded = bch_encode(data_bits)
    print(f"After BCH (15 bits): {bin(encoded)[2:].zfill(15)}")

    final = encoded ^ FORMAT_MASK
    print(f"After XOR mask: {bin(final)[2:].zfill(15)}")

    # Show all format strings
    print("\nAll format strings:")
    for ec in ['L', 'M', 'Q', 'H']:
        for mask in range(8):
            fs = FORMAT_STRINGS[(ec, mask)]
            print(f"  {ec}-{mask}: {bin(fs)[2:].zfill(15)}")


if __name__ == "__main__":
    demo_bch()
```

Listing 5: BCH Code for Format Information

6.5 References for This Section

- Wikipedia: BCH code (`https://en.wikipedia.org/wiki/BCH_code`)
- Thonky: Format and Version Information (`https://www.thonky.com/qr-code-tutorial/format-version-information`)

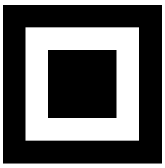
7 QR Code Matrix Construction

The QR code matrix contains both function patterns (required for detection and alignment) and data modules.

7.1 Function Patterns

7.1.1 Finder Patterns

Three 7×7 patterns in corners with ratio $1:1:3:1:1$:



Location: top-left at $(0, 0)$, top-right at $(size - 7, 0)$, bottom-left at $(0, size - 7)$.

7.1.2 Separators

One-module-wide white borders around finder patterns.

7.1.3 Timing Patterns

Alternating black-white modules in row 6 and column 6, connecting finder patterns.

7.1.4 Alignment Patterns

5×5 patterns for versions 2+. Positions defined in specification.

7.1.5 Dark Module

Single dark module at position $(8, 4V + 9)$ where V is version.

7.2 Data Placement Algorithm

Data is placed in a zigzag pattern:

1. Start at bottom-right corner
2. Move in 2-column-wide strips from right to left
3. Within each strip, alternate up and down
4. Skip function pattern areas

7.3 Python Implementation: Matrix Construction

```python
"""
QR Code matrix construction with function patterns and data placement.
"""

class QRMatrix:
    """QR Code matrix construction and manipulation."""

    def __init__(self, version):
        self.version = version
        self.size = 4 * version + 17

        # Matrix values: None=unassigned, 0=white, 1=black
        self.matrix = [[None] * self.size for _ in range(self.size)]

        # Track which modules are function patterns (cannot be masked)
        self.is_function = [[False] * self.size for _ in range(self.size)]

        self._place_function_patterns()

    def _place_function_patterns(self):
        """Place all function patterns."""
        self._place_finder_patterns()
        self._place_separators()
        self._place_timing_patterns()
        self._place_alignment_patterns()
        self._place_dark_module()
        self._reserve_format_area()
        if self.version >= 7:
            self._reserve_version_area()

    def _place_finder_patterns(self):
        """Place the three finder patterns."""
        positions = [
            (0, 0),                        # Top-left
            (self.size - 7, 0),            # Top-right
            (0, self.size - 7)             # Bottom-left
        ]

        for (x, y) in positions:
            self._place_finder_pattern(x, y)

    def _place_finder_pattern(self, x, y):
        """Place a single finder pattern at position (x, y)."""
        for dy in range(7):
```

```
            for dx in range(7):
                # Determine if this module should be black
                if (dy == 0 or dy == 6 or dx == 0 or dx == 6 or
                    (2 <= dx <= 4 and 2 <= dy <= 4)):
                    value = 1
                else:
                    value = 0

                self.matrix[y + dy][x + dx] = value
                self.is_function[y + dy][x + dx] = True

    def _place_separators(self):
        """Place white separators around finder patterns."""
        # Horizontal separators
        for x in range(8):
            if x < self.size:
                self._set_function(x, 7, 0)            # Top-left
                self._set_function(self.size-8+x, 7, 0)  # Top-right
                self._set_function(x, self.size-8, 0)    # Bottom-left

        # Vertical separators
        for y in range(8):
            if y < self.size:
                self._set_function(7, y, 0)            # Top-left
                self._set_function(self.size-8, y, 0) # Top-right
                self._set_function(7, self.size-8+y, 0)  # Bottom-left

    def _place_timing_patterns(self):
        """Place timing patterns (row 6 and column 6)."""
        for i in range(8, self.size - 8):
            value = (i + 1) % 2  # Alternating pattern
            self._set_function(i, 6, value)  # Horizontal
            self._set_function(6, i, value)  # Vertical

    def _place_alignment_patterns(self):
        """Place alignment patterns for version 2+."""
        if self.version < 2:
            return

        positions = self._get_alignment_positions()

        for row in positions:
            for col in positions:
                # Skip if overlapping with finder patterns
                if self._overlaps_finder(row, col):
                    continue
                self._place_alignment_pattern(col, row)

    def _get_alignment_positions(self):
        """Get alignment pattern center positions for this version."""
        # Simplified - full table should be used from specification
        if self.version == 1:
            return []

        # Calculate positions (simplified algorithm)
        first = 6
        last = self.size - 7

        if self.version == 2:
            return [6, 18]

        # For larger versions, calculate intermediate positions
        step = (last - first) // ((self.version // 7) + 1)
```

```
        step = ((step + 1) // 2) * 2  # Round to even

        positions = [first]
        pos = last
        while pos > first + step:
            positions.insert(1, pos)
            pos -= step
        positions.append(last)

        return positions

    def _overlaps_finder(self, row, col):
        """Check if alignment pattern would overlap finder patterns."""
        # Top-left finder: (0,0) to (8,8)
        if row <= 8 and col <= 8:
            return True
        # Top-right finder
        if row <= 8 and col >= self.size - 9:
            return True
        # Bottom-left finder
        if row >= self.size - 9 and col <= 8:
            return True
        return False

    def _place_alignment_pattern(self, x, y):
        """Place a single alignment pattern centered at (x, y)."""
        for dy in range(-2, 3):
            for dx in range(-2, 3):
                if abs(dy) == 2 or abs(dx) == 2 or (dy == 0 and dx == 0):
                    value = 1
                else:
                    value = 0
                self._set_function(x + dx, y + dy, value)

    def _place_dark_module(self):
        """Place the dark module (always black)."""
        x, y = 8, 4 * self.version + 9
        self._set_function(x, y, 1)

    def _reserve_format_area(self):
        """Reserve space for format information."""
        # Around top-left finder
        for i in range(9):
            self.is_function[8][i] = True
            self.is_function[i][8] = True

        # Below top-right finder and right of bottom-left finder
        for i in range(8):
            self.is_function[8][self.size - 1 - i] = True
            self.is_function[self.size - 1 - i][8] = True

    def _reserve_version_area(self):
        """Reserve space for version information (version 7+)."""
        for i in range(6):
            for j in range(3):
                self.is_function[i][self.size - 11 + j] = True
                self.is_function[self.size - 11 + j][i] = True

    def _set_function(self, x, y, value):
        """Set a function pattern module."""
        if 0 <= x < self.size and 0 <= y < self.size:
            self.matrix[y][x] = value
            self.is_function[y][x] = True
```

```python

    def place_data(self, data_bits):
        """Place data bits in zigzag pattern."""
        bit_index = 0

        # Start from bottom-right, moving left in 2-column strips
        x = self.size - 1
        upward = True

        while x >= 0:
            # Skip column 6 (timing pattern)
            if x == 6:
                x -= 1

            for y in range(self.size - 1, -1, -1) if upward else range(self.size):
                for dx in [0, -1]:
                    col = x + dx
                    if col < 0:
                        continue

                    # Skip function pattern modules
                    if self.is_function[y][col]:
                        continue

                    # Place data bit (or 0 if we've run out)
                    if bit_index < len(data_bits):
                        self.matrix[y][col] = data_bits[bit_index]
                        bit_index += 1
                    else:
                        self.matrix[y][col] = 0

            x -= 2
            upward = not upward

        return bit_index

    def to_string(self):
        """Convert matrix to string representation."""
        result = []
        for row in self.matrix:
            line = ""
            for cell in row:
                if cell == 1:
                    line += "##"
                elif cell == 0:
                    line += "  "
                else:
                    line += ".."
            result.append(line)
        return "\n".join(result)


def demo_matrix():
    """Demonstrate matrix construction."""
    print("=== QR Matrix Construction Demo ===\n")

    matrix = QRMatrix(1)  # Version 1 (21x21)
    print(f"Version 1 matrix ({matrix.size}x{matrix.size}):")
    print(matrix.to_string())


if __name__ == "__main__":
```

```
    demo_matrix()
```

Listing 6: QR Code Matrix Construction

7.4 References for This Section

- Thonky: Module Placement in Matrix (https://www.thonky.com/qr-code-tutorial/module-placement-matrix)
- Nayuki: Creating a QR Code step by step (https://www.nayuki.io/page/creating-a-qr-code-step-b

8 Data Masking

Masking XORs data modules with a pattern to avoid problematic patterns.

8.1 Mask Patterns

There are 8 mask patterns, each defined by a formula:

Table 6: Mask Pattern Formulas (dark if formula is true)

Pattern	Condition (row r, column c)
0	$(r+c) \mod 2 = 0$
1	$r \mod 2 = 0$
2	$c \mod 3 = 0$
3	$(r+c) \mod 3 = 0$
4	$(\lfloor r/2 \rfloor + \lfloor c/3 \rfloor) \mod 2 = 0$
5	$(r \cdot c) \mod 2 + (r \cdot c) \mod 3 = 0$
6	$((r \cdot c) \mod 2 + (r \cdot c) \mod 3) \mod 2 = 0$
7	$((r+c) \mod 2 + (r \cdot c) \mod 3) \mod 2 = 0$

8.2 Penalty Calculation

The optimal mask is chosen by minimizing penalty score:

1. **Runs**: 3 points + (N-5) for each run of $N \geq 5$ same-color modules
2. **Boxes**: 3 points for each 2×2 block of same color
3. **Finder-like**: 40 points for patterns resembling finder patterns
4. **Balance**: Points based on proportion of dark modules

8.3 Python Implementation: Masking

```python
"""
Data masking for QR codes.
"""

MASK_PATTERNS = [
    lambda r, c: (r + c) % 2 == 0,
    lambda r, c: r % 2 == 0,
    lambda r, c: c % 3 == 0,
    lambda r, c: (r + c) % 3 == 0,
```

```python
    lambda r, c: (r // 2 + c // 3) % 2 == 0,
    lambda r, c: (r * c) % 2 + (r * c) % 3 == 0,
    lambda r, c: ((r * c) % 2 + (r * c) % 3) % 2 == 0,
    lambda r, c: ((r + c) % 2 + (r * c) % 3) % 2 == 0,
]


def apply_mask(matrix, is_function, mask_num):
    """Apply mask pattern to data modules only."""
    size = len(matrix)
    result = [row[:] for row in matrix]  # Copy
    mask_func = MASK_PATTERNS[mask_num]

    for r in range(size):
        for c in range(size):
            if not is_function[r][c] and mask_func(r, c):
                result[r][c] ^= 1

    return result


def calculate_penalty(matrix):
    """Calculate total penalty score for a masked matrix."""
    size = len(matrix)
    penalty = 0

    # Penalty 1: Runs of same color
    penalty += _penalty_runs(matrix, size)

    # Penalty 2: 2x2 boxes
    penalty += _penalty_boxes(matrix, size)

    # Penalty 3: Finder-like patterns
    penalty += _penalty_finder_like(matrix, size)

    # Penalty 4: Dark/light balance
    penalty += _penalty_balance(matrix, size)

    return penalty


def _penalty_runs(matrix, size):
    """Penalty for runs of 5+ same-color modules."""
    penalty = 0

    for r in range(size):
        # Horizontal
        run_length = 1
        for c in range(1, size):
            if matrix[r][c] == matrix[r][c-1]:
                run_length += 1
            else:
                if run_length >= 5:
                    penalty += 3 + (run_length - 5)
                run_length = 1
        if run_length >= 5:
            penalty += 3 + (run_length - 5)

    for c in range(size):
        # Vertical
        run_length = 1
        for r in range(1, size):
            if matrix[r][c] == matrix[r-1][c]:
```

```
                run_length += 1
            else:
                if run_length >= 5:
                    penalty += 3 + (run_length - 5)
                run_length = 1
        if run_length >= 5:
            penalty += 3 + (run_length - 5)

    return penalty


def _penalty_boxes(matrix, size):
    """Penalty for 2x2 same-color boxes."""
    penalty = 0
    for r in range(size - 1):
        for c in range(size - 1):
            color = matrix[r][c]
            if (matrix[r][c+1] == color and
                matrix[r+1][c] == color and
                matrix[r+1][c+1] == color):
                penalty += 3
    return penalty


def _penalty_finder_like(matrix, size):
    """Penalty for patterns similar to finder patterns."""
    penalty = 0
    pattern1 = [1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0]
    pattern2 = [0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1]

    for r in range(size):
        for c in range(size - 10):
            # Check horizontal
            if [matrix[r][c+i] for i in range(11)] in [pattern1, pattern2]:
                penalty += 40

    for c in range(size):
        for r in range(size - 10):
            # Check vertical
            if [matrix[r+i][c] for i in range(11)] in [pattern1, pattern2]:
                penalty += 40

    return penalty


def _penalty_balance(matrix, size):
    """Penalty based on dark/light module ratio."""
    dark_count = sum(sum(row) for row in matrix)
    total = size * size
    percent = (dark_count * 100) // total

    # Penalty based on deviation from 50%
    prev_multiple = percent - (percent % 5)
    next_multiple = prev_multiple + 5

    penalty = min(
        abs(prev_multiple - 50) // 5,
        abs(next_multiple - 50) // 5
    ) * 10

    return penalty

```

```python
def choose_best_mask(matrix, is_function):
    """Choose the mask pattern with lowest penalty."""
    best_mask = 0
    best_penalty = float('inf')

    for mask_num in range(8):
        masked = apply_mask(matrix, is_function, mask_num)
        penalty = calculate_penalty(masked)

        if penalty < best_penalty:
            best_penalty = penalty
            best_mask = mask_num

    return best_mask, best_penalty


def demo_masking():
    """Demonstrate masking patterns."""
    print("=== Masking Demo ===\n")

    # Create a simple test matrix
    size = 7
    matrix = [[0] * size for _ in range(size)]
    is_function = [[False] * size for _ in range(size)]

    for mask_num in range(8):
        print(f"Mask Pattern {mask_num}:")
        masked = apply_mask(matrix, is_function, mask_num)
        for row in masked:
            print("".join("##" if c else "  " for c in row))
        print()


if __name__ == "__main__":
    demo_masking()
```

Listing 7: Mask Pattern Application and Penalty Calculation

9 Complete QR Code Generator

Now we combine all components into a complete QR code generator.

9.1 Complete Implementation

```python
"""
Complete QR Code Generator from Scratch

This module combines all components to generate valid QR codes.
"""

# [Previous class definitions would be included here]


class QRCodeGenerator:
    """Complete QR code generator."""

    # Error correction codewords per version and level
    EC_CODEWORDS = {
        1: {'L': 7, 'M': 10, 'Q': 13, 'H': 17},
        2: {'L': 10, 'M': 16, 'Q': 22, 'H': 28},
        # ... extend for all versions
```

```
    }

    # Data capacity (bytes) per version and level
    DATA_CAPACITY = {
        1: {'L': 19, 'M': 16, 'Q': 13, 'H': 9},
        2: {'L': 34, 'M': 28, 'Q': 22, 'H': 16},
        # ... extend for all versions
    }

    def __init__(self, ec_level='M'):
        self.ec_level = ec_level
        self.gf = GF256()
        self.rs = ReedSolomonEncoder(self.gf)

    def generate(self, data, version=None):
        """
        Generate a QR code for the given data.

        Returns: 2D list of 0s and 1s representing the QR code
        """
        # Step 1: Determine version if not specified
        if version is None:
            version = self._determine_version(data)

        # Step 2: Encode data
        data_bits = encode_data(data, version)

        # Step 3: Add terminator and padding
        data_codewords = self._pad_data(data_bits, version)

        # Step 4: Generate error correction
        num_ec = self.EC_CODEWORDS[version][self.ec_level]
        ec_codewords = self.rs.encode(data_codewords, num_ec)

        # Step 5: Interleave (for version 1, just concatenate)
        final_message = data_codewords + ec_codewords

        # Step 6: Convert to bits
        final_bits = []
        for byte in final_message:
            final_bits.extend(int_to_bits(byte, 8))

        # Step 7: Create matrix and place patterns
        qr = QRMatrix(version)

        # Step 8: Place data
        qr.place_data(final_bits)

        # Step 9: Apply best mask
        best_mask, _ = choose_best_mask(qr.matrix, qr.is_function)
        final_matrix = apply_mask(qr.matrix, qr.is_function, best_mask)

        # Step 10: Add format information
        self._add_format_info(final_matrix, qr.is_function, best_mask)

        return final_matrix

    def _determine_version(self, data):
        """Determine minimum version for the data."""
        mode = detect_mode(data)
        data_len = len(data)

        for version in range(1, 41):
```

```
            capacity = self.DATA_CAPACITY.get(version, {}).get(self.ec_level,
    0)
            if capacity >= data_len:
                return version

        raise ValueError("Data too long for any QR version")

    def _pad_data(self, data_bits, version):
        """Pad data to required length."""
        # Add terminator
        data_bits = list(data_bits)
        capacity = self.DATA_CAPACITY[version][self.ec_level] * 8

        # Add up to 4 terminator bits
        term_bits = min(4, capacity - len(data_bits))
        data_bits.extend([0] * term_bits)

        # Pad to byte boundary
        while len(data_bits) % 8 != 0:
            data_bits.append(0)

        # Convert to bytes
        codewords = bits_to_bytes(data_bits)

        # Add pad codewords (alternating 236, 17)
        pad_bytes = [236, 17]
        i = 0
        while len(codewords) < self.DATA_CAPACITY[version][self.ec_level]:
            codewords.append(pad_bytes[i % 2])
            i += 1

        return codewords

    def _add_format_info(self, matrix, is_function, mask):
        """Add format information to the matrix."""
        format_bits = format_bits_to_list(
            get_format_string(self.ec_level, mask)
        )

        size = len(matrix)

        # Place format bits around top-left finder
        positions_primary = [
            # Horizontal (row 8, left side)
            (8, 0), (8, 1), (8, 2), (8, 3), (8, 4), (8, 5),
            (8, 7), (8, 8),
            # Vertical (column 8, top)
            (7, 8), (5, 8), (4, 8), (3, 8), (2, 8), (1, 8), (0, 8)
        ]

        positions_secondary = [
            # Vertical (column 8, bottom)
            (size-1, 8), (size-2, 8), (size-3, 8), (size-4, 8),
            (size-5, 8), (size-6, 8), (size-7, 8),
            # Horizontal (row 8, right side)
            (8, size-8), (8, size-7), (8, size-6), (8, size-5),
            (8, size-4), (8, size-3), (8, size-2), (8, size-1)
        ]

        for i, bit in enumerate(format_bits):
            if i < len(positions_primary):
                r, c = positions_primary[i]
                matrix[r][c] = bit
```

```python

        for i, bit in enumerate(format_bits):
            if i < len(positions_secondary):
                r, c = positions_secondary[i]
                matrix[r][c] = bit


def demo_complete():
    """Generate a complete QR code."""
    print("=== Complete QR Code Generator Demo ===\n")

    generator = QRCodeGenerator(ec_level='M')

    data = "HELLO"
    print(f"Encoding: '{data}'")

    try:
        qr = generator.generate(data, version=1)

        print(f"\nGenerated QR Code ({len(qr)}x{len(qr)}):")
        for row in qr:
            print("".join("##" if c else "  " for c in row))
    except Exception as e:
        print(f"Error: {e}")


if __name__ == "__main__":
    demo_complete()
```

Listing 8: Complete QR Code Generator

10 Conclusion and Further Reading

This tutorial covered the complete process of QR code generation from mathematical foundations through implementation.

10.1 Summary of Key Concepts

1. **Galois Field Arithmetic**: All QR code math operates in $GF(256)$ using the primitive polynomial $x^8 + x^4 + x^3 + x^2 + 1$
2. **Reed-Solomon Codes**: Error correction uses polynomial division over $GF(256)$ to generate redundant codewords
3. **BCH Codes**: Format information uses a simpler $(15, 5)$ BCH code for error protection
4. **Data Encoding**: Four modes (numeric, alphanumeric, byte, kanji) optimize storage for different data types
5. **Masking**: Eight mask patterns with penalty scoring ensure readability

10.2 References and Further Reading

10.2.1 Official Standards

- ISO/IEC 18004:2015 - QR Code bar code symbology specification

10.2.2 Tutorials and Guides

- Thonky's QR Code Tutorial: `https://www.thonky.com/qr-code-tutorial/`
- Wikiversity - Reed-Solomon codes for coders: `https://en.wikiversity.org/wiki/Reed-Solomon_codes_for_coders`
- Nayuki - Creating a QR Code step by step: `https://www.nayuki.io/page/creating-a-qr-code-step-b`

10.2.3 Python Libraries (for Comparison)

- GeeksforGeeks - Generate QR Code using qrcode in Python: `https://www.geeksforgeeks.org/generate-qr-code-using-qrcode-in-python/`
- GeeksforGeeks - Python qrcode Library: `https://www.geeksforgeeks.org/introduction-to-python-q`

10.2.4 Mathematical Background

- Wikipedia - Reed-Solomon error correction: `https://en.wikipedia.org/wiki/Reed-Solomon_error_correction`
- Wikipedia - BCH code: `https://en.wikipedia.org/wiki/BCH_code`
- Wikipedia - Finite field arithmetic: `https://en.wikipedia.org/wiki/Finite_field_arithmetic`
- Research.swtch.com - Finite Field Arithmetic: `https://research.swtch.com/field`