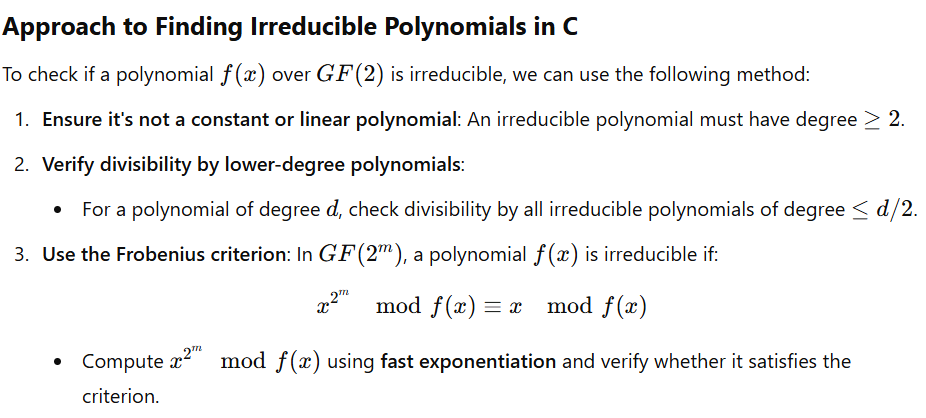
can find irreducible polynomials over finite fields in C, but it requires implementing field arithmetic manually. Unlike Sage, which provides built-in functions for algebraic operations in finite fields, C does not have direct support for such computations.



basic C program that finds an irreducible polynomial of a given degree over GF(2).

#include <stdio.h>

#include <stdint.h>

#define DEGREE 3 // Change this to find irreducible polynomials of different degrees

#define FIELD 2 // GF(2) field

// Function to count the number of 1s in a binary number

int popcount(uint32\_t x) {

int count = 0;

while (x) {

count += x & 1;

x >>= 1;

}

return count;

}

// Function to check if a polynomial is irreducible in GF(2)

int is\_irreducible(uint32\_t poly, int degree) {

if (popcount(poly) <= 1) return 0; // Ignore x^n and 1 (not irreducible)

uint32\_t mod = poly;

uint32\_t test = 2; // x in binary

// Compute x^(2^degree) mod poly(x) and check if it returns x

for (int i = 0; i < degree; i++) {

test = (test \* test) % mod; // Square in GF(2^m)

}

return (test == 2); // If x^(2^m) ≡ x (mod f(x)), f(x) is irreducible

}

int main() {

printf("Irreducible polynomials of degree %d over GF(2):\n", DEGREE);

for (uint32\_t poly = (1 << DEGREE) | 1; poly < (1 << (DEGREE + 1)); poly += 2) {

if (is\_irreducible(poly, DEGREE)) {

printf("Binary: %b, Polynomial: x^%d ", poly, DEGREE);

for (int i = DEGREE - 1; i >= 0; i--) {

if (poly & (1 << i)) printf("+ x^%d ", i);

}

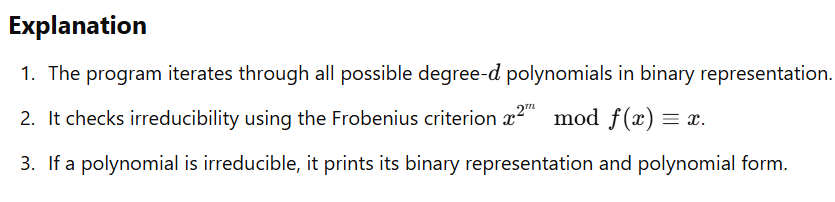
printf("\n");

}

}

return 0;

}

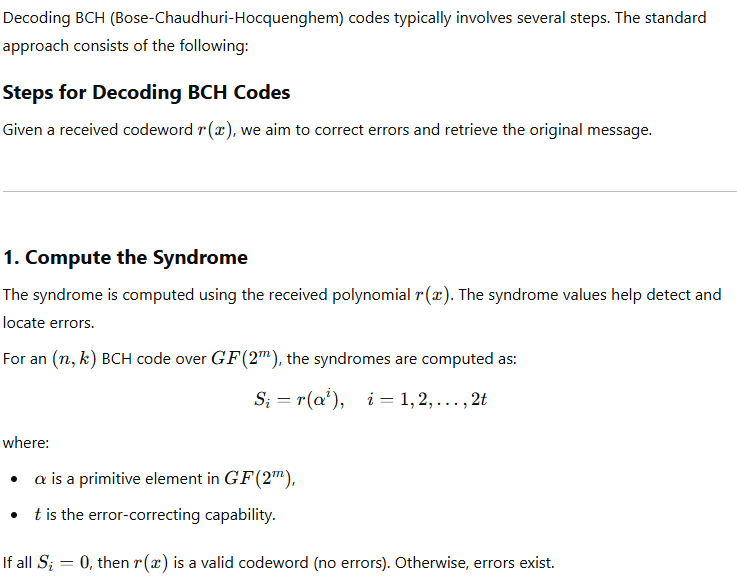


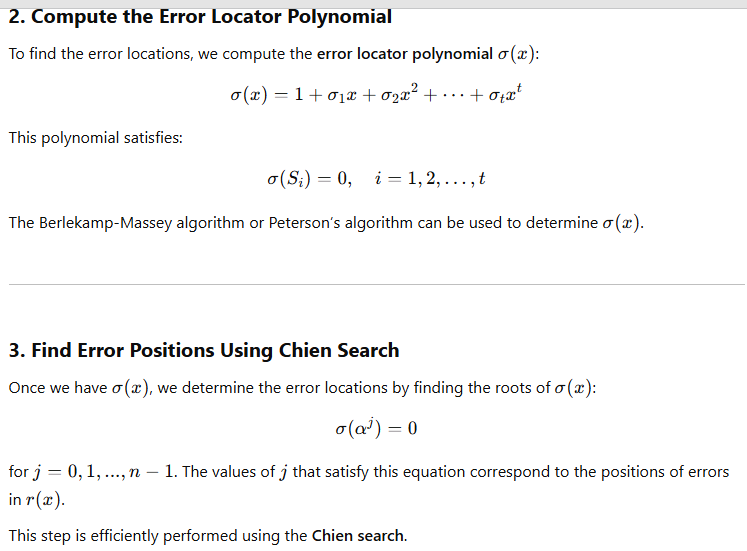
**Example Output for d=3**

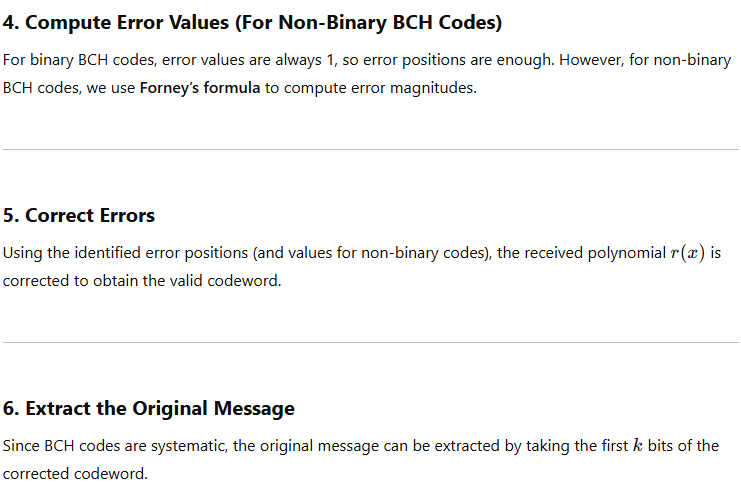
Irreducible polynomials of degree 3 over GF(2):

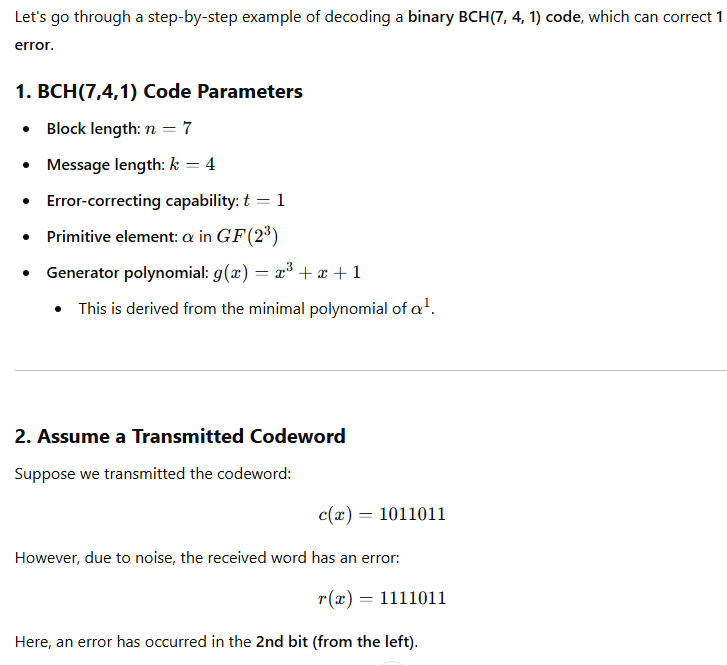
Binary: 1011, Polynomial: x^3 + x + 1

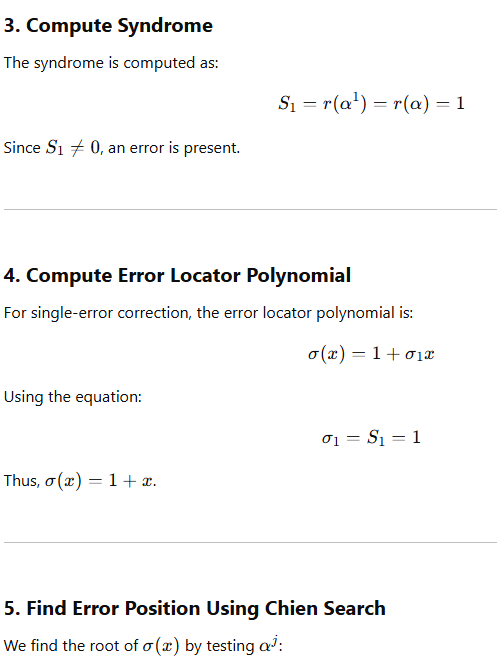
Binary: 1101, Polynomial: x^3 + x^2 + 1

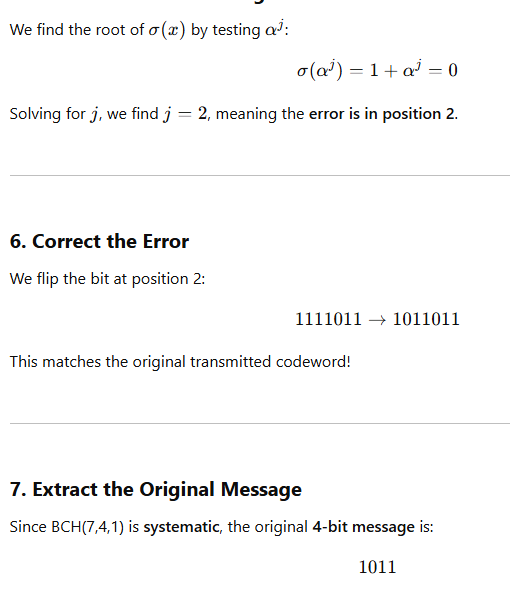


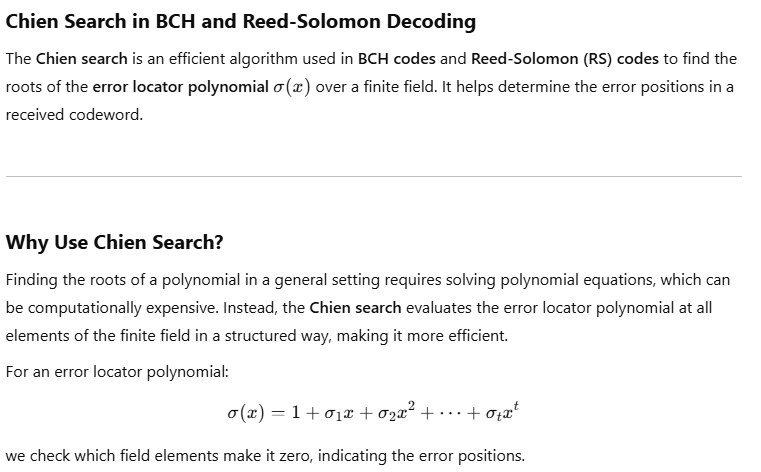


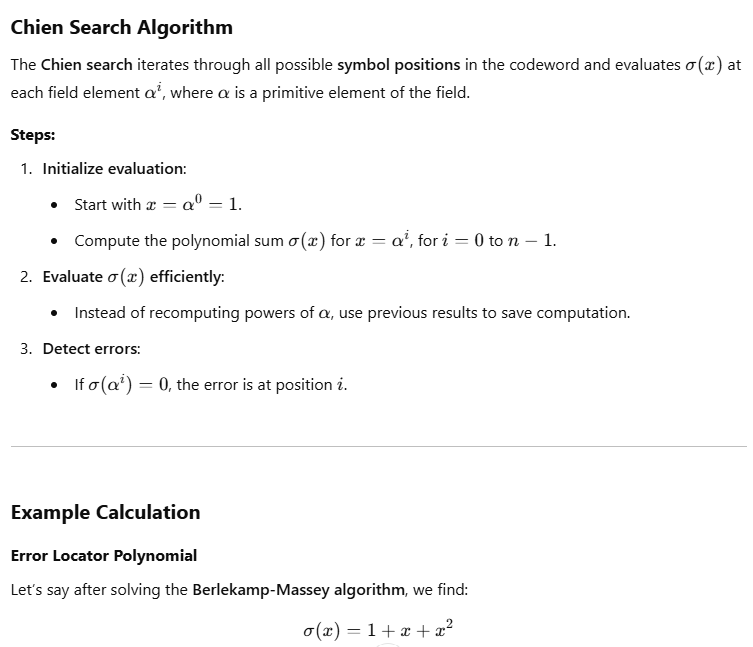


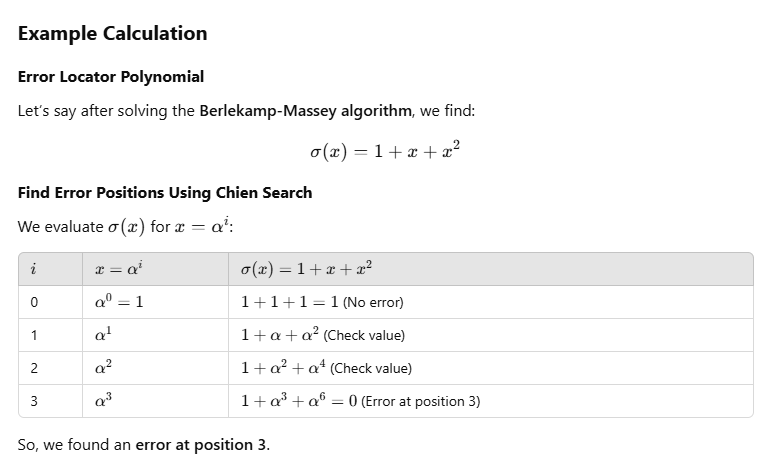












Here’s a basic **Chien search** implementation in C:

#include <stdio.h>

#include <stdint.h>

#define N 7 // Code length

#define T 2 // Number of errors

// Function to evaluate σ(x) at α^i

int chien\_search(uint16\_t sigma[], int degree) {

int error\_positions[N] = {0};

int num\_errors = 0;

// Primitive element α in GF(2^m)

for (int i = 0; i < N; i++) {

uint16\_t eval = 0;

uint16\_t x = 1; // α^0 = 1

// Compute σ(α^i)

for (int j = 0; j <= degree; j++) {

eval ^= sigma[j] \* x;

x <<= 1; // Move to next power of α

}

// If σ(α^i) == 0, error at position i

if (eval == 0) {

error\_positions[num\_errors++] = i;

}

}

// Print error locations

printf("Errors found at positions: ");

for (int i = 0; i < num\_errors; i++) {

printf("%d ", error\_positions[i]);

}

printf("\n");

return num\_errors;

}

int main() {

uint16\_t sigma[T + 1] = {1, 1, 1}; // Example σ(x) = 1 + x + x^2

printf("Performing Chien Search...\n");

chien\_search(sigma, T);

return 0;

}

**Complexity and Efficiency**

* **Naïve approach**: Evaluating σ(x)\sigma(x)σ(x) for each αi\alpha^iαi from scratch takes O(nt)O(n t)O(nt) multiplications.
* **Optimized approach (Chien Search)**: Computes the next σ(αi+1)\sigma(\alpha^{i+1})σ(αi+1) incrementally using shifts and additions, reducing overhead.

**What is uint16\_t in C?**

uint16\_t is an **unsigned 16-bit integer type** defined in the **stdint.h** header file. It ensures a fixed width of **16 bits (2 bytes)** regardless of the platform.

**Why Use uint16\_t?**

1. **Fixed size across platforms**: Unlike int, which may vary in size depending on the system (e.g., 16-bit, 32-bit, or 64-bit), uint16\_t always takes **exactly 16 bits**.
2. **Efficient memory usage**: When dealing with binary polynomials, error-locator polynomials, or finite fields, we often need small fixed-size integers to minimize memory overhead.
3. **Avoids unnecessary sign handling**: Since it’s **unsigned**, it can store values from 0 to 65535 (2^16 - 1).
4. **Where is uint16\_t Defined?**
5. It is defined in **stdint.h**, which provides fixed-width integer types:

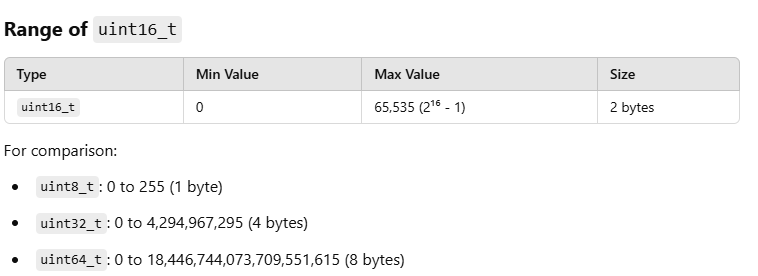
c

#include <stdint.h>

uint16\_t a = 65535; // Maximum value

uint16\_t b = 1000;

uint16\_t sum = a + b; // Overflow risk if > 65535



**Example Usage**

**1. Storing Binary Polynomial Coefficients**

#include <stdio.h>

#include <stdint.h>

int main() {

uint16\_t poly = 0b1011; // Represents x^3 + x + 1 in binary

printf("Polynomial in binary: %u\n", poly);

return 0;

}

Here, 0b1011 (binary) is stored as uint16\_t.

2. **Bitwise Operations in Finite Fields**

uint16\_t a = 0b1101; // Polynomial representation of x^3 + x^2 + 1

uint16\_t b = 0b1011; // Polynomial representation of x^3 + x + 1

uint16\_t result = a ^ b; // XOR for addition in GF(2)

printf("Result: %u\n", result); // Expected: 0110 (x^2 + x)

**Conclusion**

* uint16\_t is a **portable**, **memory-efficient** way to store 16-bit values.
* Useful in **finite field arithmetic**, **polynomial representation**, and **error correction codes**.
* Always include <stdint.h> to use it.

there is a clear reason why implementations move from **SageMath** to **C** in Post-Quantum Cryptography (PQC):

**1. Performance & Efficiency**

* **SageMath is an interpreted language (Python-based)**, which makes it slower than compiled languages like **C**.
* **C is much faster** because it directly compiles to machine code, making it suitable for high-speed cryptographic operations.

**2. Practical Deployment**

* SageMath is mainly used for **prototyping and research**.
* Real-world cryptographic libraries (e.g., OpenSSL, libsodium, NIST PQC candidates) need implementations in **C**, **Rust**, or **assembly** to integrate with security systems.

**3. Memory & Hardware Constraints**

* PQC algorithms often need to run on **embedded devices (IoT, smart cards, etc.)**.
* C allows **manual memory control**, making it suitable for optimizing cryptographic implementations for constrained environments.

**4. NIST PQC Standardization Requires C Implementations**

* NIST’s Post-Quantum Cryptography competition requires **optimized C implementations** for fair benchmarking.
* Many submissions (e.g., **Kyber, Dilithium, Falcon**) started in SageMath for theoretical analysis but were later optimized in C.

**5. Parallelization & Hardware Acceleration**

* C allows **SIMD (Single Instruction, Multiple Data)** and **GPU acceleration** for cryptographic computations.
* PQC schemes often use **polynomial multiplications**, **FFT**, or **lattice operations**, which are much faster in C.

**Conclusion**

SageMath is great for **theoretical development and testing**, but C is necessary for **real-world, efficient, and secure implementations**.