Reed–Solomon (63,42) Decoder: Errors and Erasures with Detection

COM 5140 ECC Project 2

Spring 2025 – 111511015 Pin-Jing Li

1 Overview

This project implements a Reed–Solomon decoder over $\mathbb{F}_{64}$ for the (63,42) code used in Cinema Digital Sound. The decoder handles both errors and erasures, and enforces decoding bounds with explicit detection of uncorrectable cases.

We use a LSB-first polynomial representation:

$$[a_0, a_1, \ldots, a_n] \quad \text{represents} \quad a_0 + a_1 x + \cdots + a_n x^n$$

2 Decoder Architecture

The decoder performs the following major steps:

1. **Syndrome computation** $S(x)$
2. **Erasure locator** $\sigma_0(x)$
3. **Modified syndrome** $S_0(x) = \sigma_0(x)S(x) \bmod x^R$
4. **Key equation** $\sigma_1(x)S_0(x) \equiv \omega(x) \bmod x^R$ solved via the Extended Euclidean Algorithm
5. **Error location** using roots of $\sigma(x) = \sigma_0(x)\sigma_1(x)$
6. **Error evaluation** using Forney's formula
7. **Final validation** via post-correction syndrome check

The decoder checks as in the handouts,

- **Condition (A):** $\deg(\omega) \geq t_0 + \deg(\sigma_1)$
- **Condition (B):** $\sigma_1(0) \neq 0$
- **Condition (C):** $x^n - 1 \equiv 0 \mod \sigma(x)$

Some extra check trying to handle beyond-radius detections

- **Locator-Root Agreement:** $\deg(\sigma) = \#$ of roots found
  Validates Chien search: all claimed roots must be found. If not, decoding is beyond the radius.
- **Decoding Radius Budget:** $t_0 + 2\hat{t}_1 \leq R$
  If the detected error $\hat{t}_1 = \deg(\sigma_1)$ is exceeding the budget, decoding is beyond the radius. And also, if t_0 is directly exceeding the radius, the decoding will also be rejected here.

3 Alternative Key Equation Solver: Welch–Berlekamp Interpolation

As an extension, we investigated using Welch–Berlekamp interpolation instead of the Euclidean algorithm.

The key equation

$$\sigma_1(x)S_0(x) \equiv \omega(x) \bmod x^R$$

can be interpreted as a rational function interpolation problem: finding polynomials $\omega(x), \sigma_1(x)$ such that

$$\frac{\omega(x)}{\sigma_1(x)} = S_0(x) \quad \text{at known evaluation points}$$

This approach involves solving a linear system in the coefficients of ω and σ_1, under degree constraints:

$$\deg(\omega) < t_0 + \deg(\sigma_1), \quad \deg(\sigma_1) \leq \left\lfloor \frac{R - t_0}{2} \right\rfloor$$

Welch–Berlekamp interpolation may be preferable in some hardware or symbolic applications, where rational interpolation is more natural than iterative polynomial division.

```
pair<vector<int>, vector<int>> solve_key_wb(const vector<int>& s0, int
    e0) {
    int mu = (R - e0) / 2;
    int nu = (R + e0 + 1) / 2 - 1;

    int n_eqs = R;
    int n_unknowns = (mu + 1) + (nu + 1);

    vector<vector<int>> A(n_eqs, vector<int>(n_unknowns, 0));
    vector<int> b(n_eqs, 0);

    for (int j = 0; j < R; ++j) {
        int x = EXP_TABLE[j + 1];
        int Sj = poly_eval(s0, x);  // S0(alpha^j)
        b[j] = Sj;

        int power = 1;
        for (int i = 0; i <= mu; ++i) {
            A[j][i] = gf_mul(Sj, power); // S0 * x^i
            power = gf_mul(power, x);
        }

        power = 1;
        for (int i = 0; i <= nu; ++i) {
            A[j][mu + 1 + i] = gf_sub(0, power); // -x^i for omega(x)
            power = gf_mul(power, x);
        }
    }

    // Solve the system A * [sigma1_coeffs | omega_coeffs]^T = b
    vector<int> sol;
    bool ok = solve_linear_system(A, b, sol);

```

```
    if (!ok) throw RSDecodeError("WB solve failed");

    vector<int> sigma1(sol.begin(), sol.begin() + mu + 1);
    vector<int> omega(sol.begin() + mu + 1, sol.end());

    if (sigma1[0] == 0)
        throw RSDecodeError("WB failure: sigma1(0) = 0");

    int inv = gf_inv(sigma1[0]);
    for (int& c : sigma1) c = gf_mul(c, inv);
    for (int& c : omega) c = gf_mul(c, inv);

    return {sigma1, omega};
}
```

Listing 1: an implementation of WB decoding

4 Extension: Guruswami–Sudan List Decoding

4.1 Motivation

In our current decoder, a failure occurs as soon as $t_0 + 2t_1 > R = 21$. But in many practical cases, even beyond that bound, the transmitted codeword may still be uniquely recoverable. The GS decoder leverages algebraic geometry to find *all* polynomials $f(x)$ of degree less than K that agree with a subset of the received points (x_i, y_i) with high enough multiplicity.

4.2 Key Idea

The GS decoder proceeds in two phases:

1. **Interpolation:** Construct a nonzero bivariate polynomial $Q(x, y) \in \mathbb{F}_q[x, y]$ such that:

$$Q(x_i, y_i) = Q^{(0,1)}(x_i, y_i) = \cdots = Q^{(0,m_i-1)}(x_i, y_i) = 0$$

   for selected interpolation multiplicities m_i, where (x_i, y_i) are the received (possibly corrupted) points.

2. **Factorization:** Find all univariate polynomials $f(x)$ such that $Q(x, f(x)) = 0$. These $f(x)$ correspond to potential codewords.

4.3 Decoding Radius

The GS algorithm can correct up to:

$$t < n - \sqrt{n(K-1)}$$

errors with high probability, where $n = 63$ and $K = 42$ in our case. This is significantly beyond the half-distance bound $\left\lfloor \frac{d_{\min}-1}{2} \right\rfloor = 10$.

4.4 Benefits and Challenges

GS decoding is deterministic, algebraic, and error-locating. It does not rely on soft information or probabilistic models. However, it is computationally more intensive due to bivariate interpolation and factorization, and may return multiple solutions.

In practical implementations, further ranking mechanisms (e.g., reliability scoring, CRC checks) are used to choose the most likely correct codeword from the list.

Did not have enough time to implement this before the deadline, hopefully can take a try throughout the summer.

5 Testing Design

We use a Python test generator that injects up to t_0 erasures and t_1 errors under the constraint $t_0 + 2t_1 \leq 21$, or beyond.

- Input vectors are saved as `input.txt`
- Ground-truth messages saved as `answer.txt`
- Decoder output saved as `output.txt`, with `*` lines for failure

We run through all the cases that are within the decoding radius $t_0 + 2t_1 \leq R$ and comfirmed the correct decoding implementation.

6 Detecting Behavior Far Beyond the Radius

We fix $t_0 + 2t_1 = 32$, vary error-to-erasure ratio, and for each case we run through 10k signals

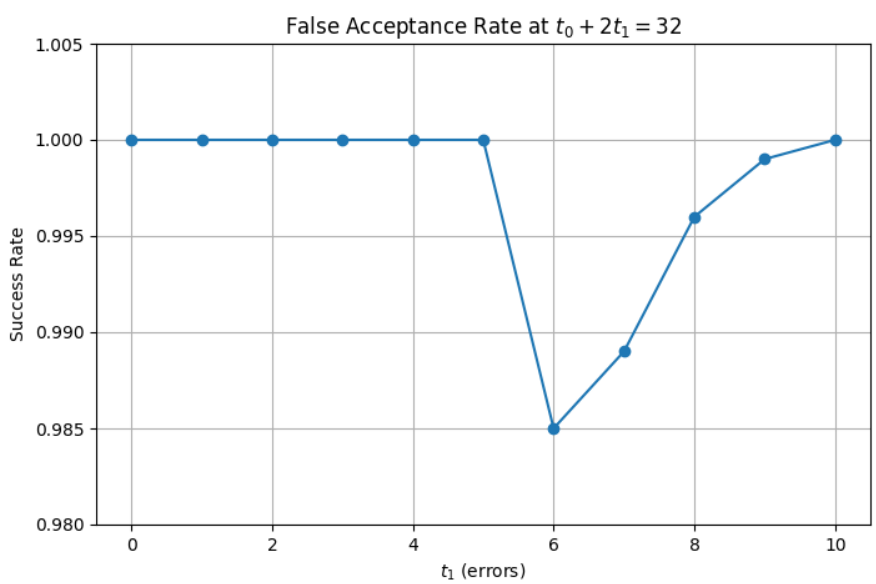

Figure 1: Testing the failure detection ability

- test 1 has some false detection when $2t_1$ and t_0 has a closer value
- Perfect rejection near the boundary (all erasure / all error cases)
- But rare false acceptances at extreme configurations, where the error pattern coincidentally satisfied all algebraic conditions.

7 Decoder Behavior Just Beyond the Radius

While the classical decoding radius of a Reed–Solomon code is given by the bound $t_0 + 2t_1 \leq R = N - K$, the behavior of algebraic decoders just beyond this radius exhibits subtle and non-monotonic phenomena. In this section, we examine this edge behavior both empirically and algebraically.

7.1 Experimental Observations

When only errors are presented, the decoder can almost always detect if the error was beyond radius. The following discussed the case when both erasure and errors presents. We fixed the number of injected errors to $t_1 = 3$, and varied the number of erasures $t_0 \in \{16, \ldots, 21\}$, such that $t_0 + 2t_1 > R$. Although all of these test cases technically exceed the decoder's guaranteed radius, we observed that the success or failure of the decoder varies dramatically with t_0. For instance, with $t_1 = 3$, the decoder's ability to reject corrupted codewords showed a striking pattern:

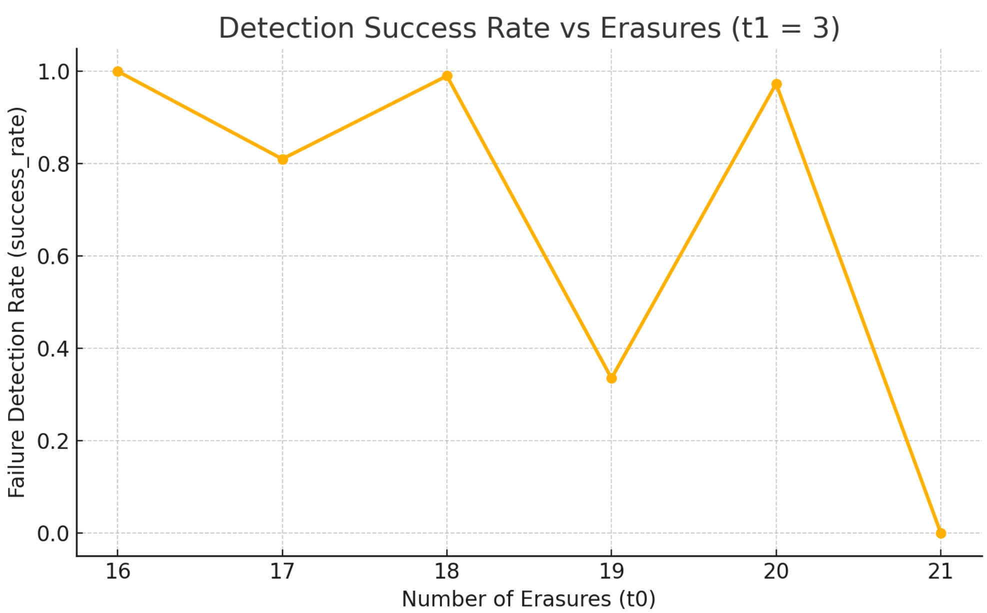

Figure 2: Testing the failure detection ability just beyond the edge

Despite all cases exceeding the decoding bound, detection success is not monotonic in t_0. In particular, detection performance degrades drastically at $t_0 = 19$ and fails completely at $t_0 = 21$, even though the total decoding load is higher at $t_0 = 20$ and 21.

7.2 Algebraic Interpretation

This non-monotonicity arises from how the erasure locator polynomial $\sigma_0(x)$ and the resulting modified syndrome polynomial $S_0(x) = \sigma_0(x) \cdot S(x)$ affect the key equation:

$$\sigma_1(x) \cdot S_0(x) \equiv \omega(x) \pmod{x^R}.$$

As t_0 increases, the degree bound $\mu = \left\lfloor \frac{R-t_0}{2} \right\rfloor$ imposed on the error locator $\sigma_1(x)$ shrinks. When $\mu = 0$, the decoder is restricted to constant error locators, typically forcing $\sigma_1(x) = 1$. In this degenerate regime (e.g., $t_0 = 21$), the decoder cannot model any error positions, and blindly trusts the message portion of the received word. Thus, even a single corrupted message symbol becomes undetectable.

7.3 Implications

These findings emphasize that the reliability of algebraic decoding just beyond the nominal radius is highly sensitive to the degree bounds in the key equation solver. While such decoders are mathematically correct, they may be semantically untrustworthy near the edge of their radius, particularly under full parity erasure. Implementations should therefore include structural checks on the syndrome and error evaluator polynomial, or impose additional constraints when decoding near $t_0 \approx R$.

8 Discussion

These results confirm both the correctness and robustness of the decoder. The detection mechanism ensures that uncorrectable cases are rejected — even under randomized noise — unless they fall into pathological configurations.

Such false acceptances are theoretically known to occur when the erroneous received vector happens to lie within the decoding "ball" of some other codeword, even though it exceeds the designed radius. This behavior is consistent with the limitations of minimum distance decoding.

Appendix: Decoder Source Code

```cpp
#include <iostream>
#include <vector>
#include <stdexcept>
#include <fstream>
#include <sstream>
#include <map>
#include <algorithm>


using namespace std;

const int N = 63;
const int K = 42;
const int R = N - K;

struct RSDecodeError : public std::exception {
    std::string message;
```

```
    RSDecodeError(const std::string& msg) : message(msg) {}
    const char* what() const noexcept override { return message.c_str(); }
};

// EXP_TABLE[i] = alpha^i
const int EXP_TABLE[63] = {
    1, 2, 4, 8, 16, 32, 3, 6, 12, 24,
    48, 35, 5, 10, 20, 40, 19, 38, 15, 30,
    60, 59, 53, 41, 17, 34, 7, 14, 28, 56,
    51, 37, 9, 18, 36, 11, 22, 44, 27, 54,
    47, 29, 58, 55, 45, 25, 50, 39, 13, 26,
    52, 43, 21, 42, 23, 46, 31, 62, 63, 61,
    57, 49, 33
};

int LOG_TABLE[64];

const vector<int> GENERATOR_POLY_COEFFS = {
    58, 62, 59, 7, 35, 58, 63, 47, 51, 6, 33,
    43, 44, 27, 7, 53, 39, 62, 52, 41, 44, 1
};

const int G_EVAL_LIST[63] = {
    34, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 16, 13, 43, 41, 48, 15, 11, 52, 33,
    1, 22, 20, 28, 17, 13, 19, 14, 56, 25,
    45, 10, 46, 45, 8, 12, 14, 44, 14, 17,
    26, 31, 22, 59, 29, 52, 31, 57, 48, 45,
    51, 13
};

// Initialization of LOG_TABLE
void init_log_table() {
    for (int i = 0; i < 64; ++i) LOG_TABLE[i] = -1;
    for (int i = 0; i < 63; ++i)
        LOG_TABLE[EXP_TABLE[i]] = i;
}

// Field operations
inline int gf_add(int a, int b) {
    return a ^ b;
}

inline int gf_sub(int a, int b) {
    return a ^ b;
}

int gf_mul(int a, int b) {
    if (a == 0 || b == 0) return 0;
    return EXP_TABLE[(LOG_TABLE[a] + LOG_TABLE[b]) % 63];
}

int gf_div(int a, int b) {
```

```
    if (b == 0) throw runtime_error("Division by zero in GF(2^6)");
    if (a == 0) return 0;
    return EXP_TABLE[(LOG_TABLE[a] - LOG_TABLE[b] + 63) % 63];
}

int gf_inv(int a) {
    if (a == 0) throw runtime_error("Inverse of 0 in GF(2^6)");
    return EXP_TABLE[(63 - LOG_TABLE[a]) % 63];
}

int gf_pow(int a, int n) {
    if (a == 0) return 0;
    return EXP_TABLE[(LOG_TABLE[a] * n) % 63];
}

// Polynomial evaluation at x
int poly_eval(const vector<int>& poly_coeffs, int x) {
    int result = 0, power = 1;
    for (int coeff : poly_coeffs) {
        result = gf_add(gf_mul(coeff, power), result);
        power = gf_mul(power, x);
    }
    return result;
}

// Verification routine
void verify_generator_polynomial() {
    bool passed = true;
    for (int i = 0; i < 63; ++i) {
        int alpha_i = EXP_TABLE[i];
        int expected = G_EVAL_LIST[i];
        int actual = poly_eval(GENERATOR_POLY_COEFFS, alpha_i);
        if (actual != expected) {
            cout << "Mismatch at alpha^" << i << ": expected " << expected
                 << ", got " << actual << endl;
            passed = false;
        }
    }
    if (passed)
        cout << "All g(alpha^i) values match the official table." << endl;
}

// polynomial helper functions

vector<int> poly_add(const vector<int>& f, const vector<int>& g) {
    size_t max_len = max(f.size(), g.size());
    vector<int> f_pad = f, g_pad = g;
    f_pad.resize(max_len, 0);
    g_pad.resize(max_len, 0);
    vector<int> result(max_len);
    for (size_t i = 0; i < max_len; ++i)
        result[i] = gf_add(f_pad[i], g_pad[i]);
    return result;
}
```

```cpp

vector<int> poly_mul(const vector<int>& f, const vector<int>& g) {
    vector<int> result(f.size() + g.size() - 1, 0);
    for (size_t i = 0; i < f.size(); ++i)
        for (size_t j = 0; j < g.size(); ++j)
            result[i + j] = gf_add(result[i + j], gf_mul(f[i], g[j]));
    return result;
}

vector<int> poly_scale(const vector<int>& p, int scalar) {
    vector<int> result;
    for (int c : p)
        result.push_back(gf_mul(c, scalar));
    return result;
}

vector<int> poly_shift(const vector<int>& p, int n) {
    vector<int> result(n, 0);
    result.insert(result.end(), p.begin(), p.end());
    return result;
}

int poly_deg(const vector<int>& p) {
    for (int i = static_cast<int>(p.size()) - 1; i >= 0; --i)
        if (p[i] != 0) return i;
    return -1;
}

vector<int> poly_trim(const vector<int>& p) {
    int i = static_cast<int>(p.size()) - 1;
    while (i >= 0 && p[i] == 0) --i;
    return vector<int>(p.begin(), p.begin() + i + 1);
}

vector<int> poly_make_monic(const vector<int>& p) {
    vector<int> p_trimmed = poly_trim(p);
    if (p_trimmed.empty()) return {};
    int inv = gf_inv(p_trimmed.back());
    vector<int> result;
    for (int c : p_trimmed)
        result.push_back(gf_mul(c, inv));
    return result;
}

pair<vector<int>, vector<int>> poly_divmod(vector<int> f, vector<int> g) {
    f = poly_trim(f);
    g = poly_trim(g);
    int deg_f = poly_deg(f);
    int deg_g = poly_deg(g);

    if (deg_g < 0)
        throw runtime_error("Division by zero polynomial");

    vector<int> quotient(deg_f - deg_g + 1, 0);
```

```
    vector<int> remainder = f;

    while (poly_deg(remainder) >= deg_g) {
        int shift = poly_deg(remainder) - deg_g;
        int lead_coeff = gf_div(remainder.back(), g.back());

        vector<int> scaled_g = poly_scale(g, lead_coeff);
        vector<int> aligned_g(shift, 0);
        aligned_g.insert(aligned_g.end(), scaled_g.begin(), scaled_g.end()
            );

        quotient[shift] = lead_coeff;
        remainder = poly_trim(poly_add(remainder, aligned_g));
    }

    return {quotient, remainder};
}

vector<int> poly_gcd(vector<int> a, vector<int> b) {
    a = poly_trim(a);
    b = poly_trim(b);
    if (b.empty()) return poly_make_monic(a);
    while (!b.empty()) {
        auto [q, r] = poly_divmod(a, b);
        a = b;
        b = poly_trim(r);
    }
    return poly_make_monic(a);
}

pair<vector<int>, vector<int>> extended_euclidean(vector<int> a, vector<
    int> b, int mu, int nu) {
    a = poly_trim(a);
    b = poly_trim(b);

    vector<int> r_prev = a, r_curr = b;
    vector<int> u_prev = {1}, u_curr = {0};
    vector<int> v_prev = {0}, v_curr = {1};

    while (poly_deg(r_curr) > nu) {
        // r_prev = q_next*r_curr + r_next
        auto [q, r_next] = poly_divmod(r_prev, r_curr);
        r_prev = r_curr;
        r_curr = r_next;

        // u_i = u_{i-2} + q_i * u_{i-1}
        vector<int> u_next = poly_trim(poly_add(u_prev, poly_mul(q, u_curr
            )));
        u_prev = u_curr;
        u_curr = u_next;

        // v_i = v_{i-2} + q_i * v_{i-1}
        vector<int> v_next = poly_trim(poly_add(v_prev, poly_mul(q, v_curr
            )));
```

```
        v_prev = v_curr;
        v_curr = v_next;
    }

    return {v_curr, r_curr}; // (sigma1, omega)
}

vector<int> build_erasure_locator(const vector<int>& erasures) {
    vector<int> sigma0 = {1};
    for (int i : erasures) {
        vector<int> term = {1, gf_sub(0, EXP_TABLE[i])}; // 1 - \alpha^i x
        sigma0 = poly_mul(sigma0, term);
    }
    return sigma0;
}

vector<int> erase_positions(vector<int> received, const vector<int>&
    erasures) {
    for (int i : erasures)
        received[i] = 0;
    return received;
}

vector<int> compute_syndrome(const vector<int>& received) {
    // S_j = \sum R_i * alpha^{ij} \forall j = 1 \ldots R
    vector<int> S;
    for (int j = 1; j <= R; ++j)
        S.push_back(poly_eval(received, EXP_TABLE[j]));
    return S;
}

vector<int> modified_syndrome(const vector<int>& syndrome, const vector<
    int>& sigma0) {
    // S0(x) = sigma0(x) * S(x) mod x^r
    vector<int> S0 = poly_mul(sigma0, syndrome);
    S0.resize(R);
    return S0;
}

pair<vector<int>, vector<int>> solve_key_equation(const vector<int>& s0,
    int e0) {
    vector<int> r_poly(R + 1, 0);
    r_poly[R] = 1;
    int mu = (R - e0) / 2;
    int nu = (R + e0 - 1) / 2;

    auto [sigma1, omega] = extended_euclidean(r_poly, s0, mu, nu);

    int sigma1_0 = sigma1[0];
    if (sigma1_0 == 0)
        throw RSDecodeError("Condition (B) violated, sigma1(0) = 0");

    for (int& c : sigma1) c = gf_div(c, sigma1_0);
    for (int& c : omega) c = gf_div(c, sigma1_0);
```

```cpp

    return {sigma1, omega};
}

vector<int> combine_locators(const vector<int>& sigma0, const vector<int>& sigma1) {
    return poly_mul(sigma0, sigma1);
}

vector<int> find_error_positions(const vector<int>& sigma) {
    // time-domain implementation
    vector<int> positions;
    for (int i = 0; i < N; ++i) {
        int x_inv = EXP_TABLE[(63 - i) % 63];
        if (poly_eval(sigma, x_inv) == 0)
            positions.push_back(i);
    }
    return positions;
}

vector<int> poly_derivative(const vector<int>& p) {
    if (p.size() < 2) return {0};
    vector<int> deriv(p.size() - 1, 0);
    for (size_t i = 1; i < p.size(); ++i)
        if (i & 1) deriv[i - 1] = p[i];
    while (deriv.size() > 1 && deriv.back() == 0)
        deriv.pop_back();
    return deriv;
}

map<int, int> evaluate_error_magnitudes(const vector<int>& sigma, const vector<int>& omega, const vector<int>& positions) {
    map<int, int> error_vector;
    vector<int> sigma_prime = poly_derivative(sigma);
    for (int i : positions) {
        int x_inv = EXP_TABLE[(63 - i) % 63];
        int num = poly_eval(omega, x_inv);
        int denom = poly_eval(sigma_prime, x_inv);
        if (denom == 0)
            throw RSDecodeError("sigma'(alpha^-" + to_string(i) + ") = 0 ");
        error_vector[i] = gf_sub(0, gf_div(num, denom));
    }
    return error_vector;
}

vector<int> apply_error_correction(vector<int> received, const map<int, int>& error_vector) {
    for (const auto& [i, mag] : error_vector)
        received[i] = gf_sub(received[i], mag);
    return received;
}

```

```
vector<int> rs_decode(const vector<int>& received, const vector<int>&
    erasures) {
    vector<int> sigma0 = build_erasure_locator(erasures);
    vector<int> R_erased = erase_positions(received, erasures);
    vector<int> S = compute_syndrome(R_erased);
    vector<int> S0 = modified_syndrome(S, sigma0);

    auto [sigma1, omega] = solve_key_equation(S0, erasures.size());
    if (poly_deg(omega) >= erasures.size() + poly_deg(sigma1)) {
        throw RSDecodeError("Condition (A) violated: deg(omega) geq t_1 +
            deg(sigma_1)");
    }


    int t0 = erasures.size();
    int t1_budget = (R - t0) / 2;
    if (poly_deg(sigma1) > t1_budget) {
        throw RSDecodeError("Decoding failure: t0=" + to_string(t0)
            + ", 2deg(sigma_1)=" + to_string(2 * poly_deg(sigma1))
            + ", R=" + to_string(R));
    }
    int t1_est = poly_deg(sigma1);
    if (t0 + 2 * t1_est > R) {
        throw RSDecodeError("Radius exceeded: t0 + 2t_1 = " +
            to_string(t0 + 2 * t1_est) + " > R = " + to_string(R));
    }


    vector<int> sigma = combine_locators(sigma0, sigma1);
    vector<int> error_pos = find_error_positions(sigma);
    if ((int)error_pos.size() != poly_deg(sigma)) {
        throw RSDecodeError("Locator degree != number of roots found (
            beyond radius)");
    }

    vector<int> xn_minus_1(64, 0);
    xn_minus_1[0] = 1;
    xn_minus_1[63] = gf_sub(0, 1);

    auto [qqq, rrr] = poly_divmod(xn_minus_1, sigma);
    if (poly_deg(rem) != -1) {
        throw RSDecodeError("Condition (C) violated");
    }

    auto error_mag = evaluate_error_magnitudes(sigma, omega, error_pos);
    vector<int> corrected = apply_error_correction(received, error_mag);

    if (any_of(compute_syndrome(corrected).begin(), compute_syndrome(
        corrected).end(), [](int s){ return s != 0; })) {
        throw RSDecodeError("Syndrome non-zero after correction (beyond
            radius)");
    }

    return vector<int>(corrected.end() - K, corrected.end());
```

```cpp
}

int main() {
    init_log_table();
    verify_generator_polynomial();

    ifstream infile("input.txt");
    ofstream outfile("output.txt");

    string line;
    int line_num = 0;
    while (getline(infile, line)) {
        ++line_num;
        vector<int> received;
        vector<int> erasures;
        istringstream iss(line);
        string token;

        int pos = 0;
        while (iss >> token) {
            if (token == "*") {
                received.push_back(0);
                erasures.push_back(pos);
            } else {
                received.push_back(stoi(token));
            }
            ++pos;
        }

        try {
            vector<int> decoded = rs_decode(received, erasures);
            for (int i = 0; i < K; ++i) {
                outfile << decoded[i]; if (i < K - 1) outfile << " ";
            }

            outfile << "\n";
        } catch (const RSDecodeError& e) {
            cerr << "Line " << line_num << ": " << e.what() << "\n";
            for (int i = 0; i < K; ++i) outfile << "* ";
            outfile << "\n";
        } catch (const exception& e) {
            cerr << "Line " << line_num << ": Unexpected error: " << e.what() << "\n";
            for (int i = 0; i < K; ++i) outfile << "* ";
            outfile << "\n";
        }
    }

    infile.close();
    outfile.close();

    cout << "Decoding complete. Results saved to output.txt\n";
    return 0;
}
```