<table>
<tr><td>Termin zajęć
DZIEŃ – GODZ.

Wtorek(parzysty)-9.15-11.15</td><td colspan="3">Niezawodność i diagnostyka układów cyfrowych</td></tr>
<tr><td colspan="3">Osoby wykonujące ćwiczenie:

Oleksandr Nedosiek, Błażej Michałek, Jakub Szot</td><td>Grupa nr:

3</td></tr>
<tr><td colspan="3">Tytuł projektu: Własna implementacja kodera i dekodera kodu RS (Reed-Solomona)</td><td>Projekt nr: 2</td></tr>
<tr><td colspan="2">Data oddania projektu:</td><td>DD-MM-RRRR

28.01.2025</td><td rowspan="2">Ocena:</td></tr>
<tr><td colspan="2">Data oddania sprawozdania</td><td>DD-MM-RRRR

31.01.2025</td></tr>
</table>

Spis treści

Podział pracy:

Oleksandr Nedosiek (275978) – implementacja ciała Galois, kodera, dekoderów prostego oraz pełnego, napisanie sprawozdania.
Błażej Michałek (281025) – implementacja ciała Galois, kodera, dekodera prostego oraz pełnego, napisanie sprawozdania.
Jakub Szot (281064) – implementacja ciała Galois, napisanie i przeprowadzenie testów, napisanie sprawozdania.

Wstęp teoretyczny:

Kod Reeda-Salomona (RS) to jeden z najważniejszych kodów korekcyjnych stosowanych w telekomunikacji i informatyce. Jest to kod nadmiarowy, który pozwala na wykrywanie i korygowanie błędów powstałych podczas transmisji lub przechowywania danych.

Kod RS należy do klasy kodów cyklicznych i jest oparty na algebrze ciał skończonych. Jego podstawową cechą jest możliwość korekcji wielu błędów – im więcej nadmiarowych symboli zostanie dodanych, tym większa odporność kodu na uszkodzenia. Stosuje się go m.in. w płytach CD, DVD, systemach satelitarnych oraz technologii RAID.

Kodowanie RS polega na dodaniu do danych tzw. symboli parzystości, które pozwalają na odtworzenie oryginalnych informacji nawet w przypadku utraty części przesyłanych symboli. Dekodowanie opiera się na metodach interpolacyjnych i algorytmach korekcji błędów, takich jak algorytm Berlekampa-Masseya czy Euklidesa nad ciałami skończonymi.

Parametry naszego kodu:

- zdolność korekcyjna t = 5;
- długość wektora kodowego n = 63;
- liczba pozycji kontrolnych $r = 2 * t$ = 10;
- liczba symboli informacyjnych $k = n - r$ = 53;
- potęga ciała Galois s = 6.

Kod o takich parametrach teoretycznie jest w stanie maksymalnie skorygować do 2*t błędów, czyli w naszym przypadku 10 błędów.

Implementacja klasy Galois

Pierwszym krokiem realizacji naszego projektu była implementacja klasy, odpowiadającej za podstawową funkcjonalność, umożliwiającą implementację kodera i dekodera kodu Reeda-Solomona w ciele GF(2^6). W podanej klasie zostały stworzone metody, dzięki którym realizowane są następujące funkcjonalności:

- tworzenie wielomianu generującego,
- dodawanie symboli,
- mnożenie symboli,
- generowanie wszystkich symboli dla ciała,
- dodawanie wielomianów,
- mnożenie wielomianów,
- dzielenie wielomianów z resztą.

Wielomiany, dzięki którym są definiowane nasze informacje, przekształciliśmy w tablicę potęg pierwiastków pierwotnych α, w której stopień danej potęgi oznaczał współczynnik kolejnego stopnia wielomianu. Na przykład, wielomian $\alpha^{33} * x^2 + \alpha^{12} * x^1 + \alpha^{28}$ zostanie zapisany jako [33, 12, 28].

Wartości potęg pierwiastków pierwotnych tego ciała należą do zbioru liczb całkowitych z przedziału [0;62] oraz (63;64]. W związku z brakiem domyślnego kodowania wartości zerowej, przyjęliśmy, że zamiast niej używamy α^{64}, ponieważ α^0 = 1.

Wielomian generujący $g(x)$, opisany wzorem $g(x) = \prod_{i=1}^{r}(x + a^i)$, został stworzony w poniższy sposób:

```
def create_generating_polynomial(self, t) :  2 usages
    if 2*t < 1:
        raise ValueError("Potega musi byc wieksza lub rowna 1.")

    generating_polynomial = [0, 1]
    for i in range(2, 2*t+1):
        generating_help = [0, i]
        generating_polynomial = self.mul_polynomials(generating_polynomial, generating_help)
    return generating_polynomial
```

Obliczanie wielomianu generującego.

Przy **dodawaniu wielomianów** współczynniki x (liczby w ciele Galois) są sumowane za pomocą operacji bitowej XOR. Na przykład, procedura dodawania wielomianów $(a^3x^2 + a^5x) + (a^2x^2 + a^4x)$ da wynik $a^8x^2 + a^{10}x$, dlatego że w GF(2^6) $a^3 = 001000$, $a^5 = 100000$, $a^2 = 000100$, $a^4 = 010000$, a więc 001000 ⊕ 000100 = 001100 = a^8 oraz 100000 ⊕ 010000 = a^{10}. Poniżej znajduje się kod, odpowiadający za daną operację.

```python
def sum_two_polynomials(self, polynomial1, polynomial2):

    sum_polynomials = [polynomial1, polynomial2]

    for i in range(len(sum_polynomials)):
        missing_zero =  - len(sum_polynomials[i])
        sum_polynomials[i] = [64] * missing_zero + sum_polynomials[i]

    result = []
    for i in range(len(sum_polynomials[0])):
        suma = 0
        for j in range(len(sum_polynomials)):
            alfa = self.alfas[sum_polynomials[j][i]]
            suma ^= int(str(self.pol_to_number(alfa)), 2)

        result.append(self.find_alfa_power(bin(suma)[2:]))
    return result
```

Sumowanie wielomianów za pomocą operacji XOR

Działanie algorytmu:

1. **Wyrównanie długości** – wielomian którego długość jest mniejsza od 63 jest uzupełniany wartościami 64 (zerami).
2. **Dodawanie symboli** – wartości α są pobierane, konwertowane na binarną postać i sumowane operacją XOR.
3. **Konwersja wyniku** – suma jest przekształcana z powrotem na potęgi α i zwracana jako wynikowy wielomian.

W przypadku **mnożenia wielomianów** metoda jest nieco inna. Przy mnożeniu współczynników x (liczb w ciele Galois) stosujemy metodę, która polega na dodawaniu wykładników α, i jeśli wynik jest większy od $n-1$, to stosujemy $mod\ n$ do wyniku. Na przykład: $a^{35}x^0 * a^{44}x^0 = a^{(35+44)\ mod\ 63}\ x^0 = a^{16}\ x^0$.

Opis działania algorytmu dla mnożenia dwóch wielomianów jest taki:

1. **Mnożenie wielomianu –** każdy symbol pierwszego wielomianu mnoży cały drugi wielomian, przesuwając go o odpowiednią liczbę miejsc.
2. **Mnożenie współczynników –** współczynniki są mnożone poprzez dodanie ich wykładników w ciele Galois.
3. **Sumowanie wyników -** powstałe wielomiany sumujemy, stosując operację XOR na współczynnikach.

Poniżej znajdzie się kod, implementujący tą metodę.

```python
def mul_polynomials(self, polynomial1, polynomial2):  6 usages

    sum_polynomials = list()
    for i, symbol in enumerate(polynomial2):
        if symbol == 64:
            continue

        # przemnożenie przez x do danej potęgi == przesunięcie w prawo
        shift_amount = len(polynomial2) - 1 - i
        help1 = polynomial1.copy()
        for _ in range(shift_amount):
            help1.append(64)

        for j in range(len(help1)):
            if help1[j] != 64:
                help1[j] = self.add_alfa_powers(help1[j], symbol)

        sum_polynomials.append(help1)

    max_length = max(len(x) for x in sum_polynomials)

    # Dopisz zera do każdego wielomianu w sum_polynomials
    for i in range(len(sum_polynomials)):
        brakujace_zera = max_length - len(sum_polynomials[i])
        sum_polynomials[i] = [64] * brakujace_zera + sum_polynomials[i]

    result = []
    for i in range(len(sum_polynomials[0])):
        suma = 0
        for j in range(len(sum_polynomials)):
            alfa = self.alfas[sum_polynomials[j][i]]
            suma ^= int(str(self.pol_to_number(alfa)), 2)

        # tutaj suma to nie wykładnik alfy, ale wartość alfy
        # dla podanej wartości, chcemy znowu żeby współczynniki reprezentowały potęgi alfy
        result.append(self.find_alfa_power(bin(suma)[2:]))

    return result
```

Mnożenie dwóch wielomianów w ciele Galois

W przypadku **dzielenia wielomianów** metoda jest nieco inna. Przy dzieleniu współczynników x (liczb w ciele Galois) stosujemy operację **odejmowania** wykładników α. Jeśli wynik jest ujemny, dodajemy n, aby uzyskać poprawny indeks.
Na przykład:

$$\alpha^{35}x^0 \div \alpha^{44}x^0 = \alpha^{(35-44) mod\ 63}x^0 = \alpha^{54}x^0$$

Podobnie jak przy mnożeniu, działanie to odbywa się na wykładnikach α, ale zamiast dodawania stosujemy odejmowanie i redukcję modulo n.

```python
# obliczanie reszty r(x) z dzielenia przez g(x)
def div_polynomials(self, polynomial1, polynomial2):  2 usages

    for i in range(len(polynomial2)):
        if polynomial2[0] == 64:
            polynomial2.pop(0)
        else:
            break

    # cała część z dzielenia
    div_whole_part = [64] * (len(polynomial1) - len(polynomial2) + 1)

    # Liczba cyfr w cała części odpowiedzi zależy od rozmiarów wielomianów
    for i in range(len(polynomial1) - len(polynomial2) + 1):
        poly_multi = [64] * len(div_whole_part)
        next_coeff = polynomial1[i]
        if next_coeff == 64:
            continue

        if next_coeff < polynomial2[0]:
            if polynomial2[0] == 64:
                continue
            else:
                next_coeff = (63 + polynomial1[i] - polynomial2[0]) % 63
        else:
            next_coeff -= polynomial2[0]

        div_whole_part[i] = next_coeff
        if i != 0:
            poly_multi[i] = div_whole_part[i]
            multi_result = self.mul_polynomials(poly_multi, polynomial2)
            polynomial1 = self.sum_two_polynomials(multi_result, polynomial1)
        else:
            multi_result = self.mul_polynomials(div_whole_part, polynomial2)
            polynomial1 = self.sum_two_polynomials(multi_result, polynomial1)
    remainder = polynomial1

    return remainder
```

Dzielenie dwóch wielomianów

Działanie algorytmu:

1. **Usuwanie początkowych zer** z polynomial2.
2. **Inicjalizacja ilorazu** jako lista zer (64).
3. **Iteracyjne dzielenie**:
   - Obliczanie współczynnika ilorazu,
   - Mnożenie przez polynomial2,
   - Odejmowanie wyniku od polynomial1.
4. **Zwrócenie reszty** – wynik to remainder.

Implementacja kodera

Koder został zaimplementowany jako klasa, której główną metodą jest kodowanie informacji podanej w postaci liczby stałoprzecinkowej. Proces ten przebiega w kilku etapach:

1. **Konwersja do postaci binarnej**
   Liczba wejściowa jest zamieniana na zapis binarny, a następnie uzupełniana wiodącymi zerami, aby jej długość była wielokrotnością długości pojedynczego symbolu.
2. **Reprezentacja wielomianowa**
   Każdy blok bitów odpowiada określonej potędze pierwiastka pierwotnego. Na tej podstawie tworzona jest tablica wartości reprezentujących wielomian.
3. **Generowanie wektora kodowego**
   Wektor kodowy c(x) obliczany jest w trzech krokach:
   - **Mnożenie wielomianu informacyjnego**
     Wielomian reprezentujący informację m(x) jest mnożony przez x^{n-k}, co odpowiada dodaniu 2t zer kontrolnych na końcu.
     $$x^{n-k} * m(x)$$
   - **Dzielenie przez wielomian generujący**
     Wynik mnożenia dzielony jest przez wielomian generujący g(x), a reszta z dzielenia r(x) zostaje wyznaczona:
     $$x^{n-k} * m(x) = q(x) * g(x) + r(x)$$
   - **Obliczenie końcowego wektora kodowego**
     Wektor kodowy otrzymujemy poprzez dodanie reszty r(x) do iloczynu $x^{n-k} * m(x)$:
     $$c(x) = x^{n-k} * m(x) + r(x)$$

```
def code_vector(self, m, t):  1 usage
    if t < 1:
        raise ValueError("Power must be greater than or equal to 1")
    g = self.generate_generating_polynomial(t)
    m += [64] * (2 * t)
    r = self.div_polynomials(m, g)

    code_vec = self.sum_two_polynomials(m, r)

    return code_vec
```

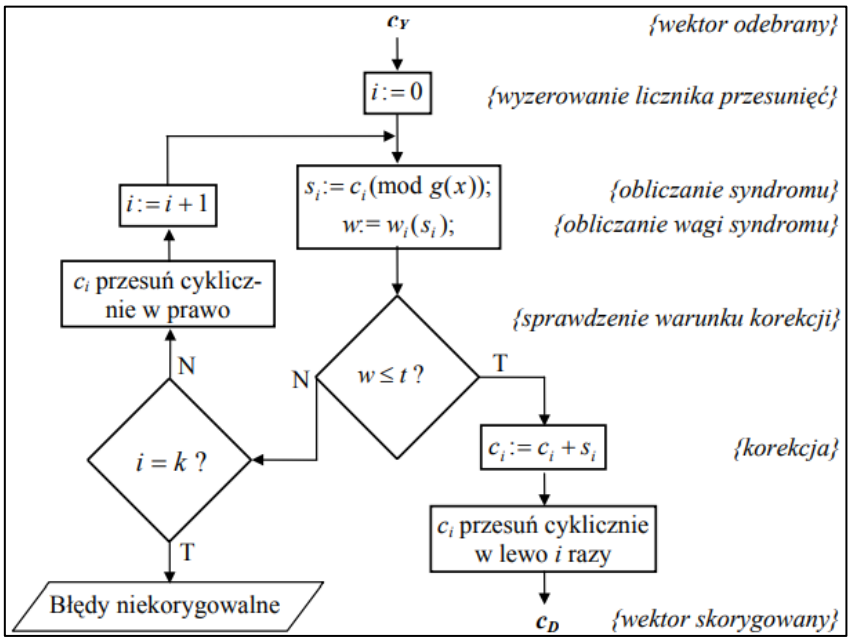
Kodowanie wielomianu informacyjnego za pomocą metody code_vector

Implementacja dekodera prostego

Dekoder prosty został w pełni zaimplementowany w funkcji *simple* klasy *Decoder*. Działa on poprzez przesuwanie symboli informacyjnych na miejsca kontrolne i obliczanie syndromów przez dzielenie przesuniętego wielomianu przez wielomian generujący. Jeśli waga syndromu mieści się w granicach zdolności korekcyjnej, sumujemy go z wielomianem odebranym i przesuwamy symbole na oryginalne pozycje.

Kroki dekodowania:

1. **Inicjalizacja**
   Przyjmujemy wektor c_y i zerujemy licznik przesunięć i.
2. **Obliczanie syndromu**
   Dzielimy c_y przez wielomian generujący $g(x)$ i obliczamy wagę syndromu w_i, czyli liczbę niezerowych symboli.
3. **Sprawdzenie korekcji**
   Jeśli $w_i \leq t$, przechodzimy do kroku 7. W przeciwnym razie – do kroku 4.
4. **Weryfikacja przesunięć**
   Jeśli $i = k$, błędy są niekorygowalne; w przeciwnym razie przechodzimy do kroku 5.
5. **Przesunięcie cykliczne**
   Przesuwamy wektor c_y i zwiększamy i o 1.
6. **Powtórzenie procedury**
   Kroki 2–5 powtarzamy, aż $w_i \leq t$.
7. **Korekcja błędów**
   Wykonujemy operację XOR na $c_{y(i)}$ i syndromie s_i, a następnie przesuwamy wektor i razy cyklicznie w przeciwną stronę.



Schemat blokowy uproszczonego algorytmu dekodowania

Powyżej znajduję się schemat blokowy uproszczonego algorytmu dekodera, który jest zawarty w książce Władysława Mochnackiego „*Kody korekcyjne i kryptografia*”.

Kod, implementujący metodę, odpowiedzialną za dekoder prosty:

```
def simple(self, vector_received):  1 usage
    messenger = "Uncorrectable errors"
    vector_corrected = list()
    g = self.gf.create_generating_polynomial(self.t)

    for i in range(0, self.n): #k=53
        syndrome = self.gf.div_polynomials(vector_received, g)
        #print(syndrome)
        syndrome_weight = 0
        for el in syndrome:
            if el != 64:
                syndrome_weight += 1


        if syndrome_weight <= self.t:
            vector_corrected = self.gf.sum_two_polynomials(vector_received, syndrome)
            if i == 0:
                break
            else:
                for j in range(0, i):
                    first_element = vector_corrected.pop(0)
                    vector_corrected.insert(63, first_element)
                return vector_corrected # jezeli uda sie poprawic wektor
        else:
            if i == self.n:
                return messenger
            else:
                last_element = vector_received.pop()
                vector_received.insert(0, last_element)
                #print(vector_received)
                #print(last_element)

    # z tego sie bierze blad - nasz program tutaj wywala pusta liste
    return messenger # blad - nie udalo sie poprawic wektora
```

Kod do dekodera prostego

Implementacja dekodera pełnego

Składa się on z kilku etapów:

- Obliczania 2t syndromów wielomianu
- Przygotowania wielomianu lokalizującego błędy
- Wyznaczania pierwiastków tego wielomianu
- Korekcji błędnych symboli na podstawie znalezionych pozycji błędów

Zamiast algorytmu Berlekampa do generowania wielomianu lokalizującego błędy wybraliśmy algorytm Euklidesa. Do wyszukiwania pierwiastków wielomianu lokalizującego zastosowaliśmy metodę Chiena, polegającą na podstawianiu kolejnych potęg α (α0,α1,α2…) pod x i sprawdzaniu, czy wynik jest równy 0.

Wzory użyte w implementacji:

1. **Obliczenie 2t syndromów:**
   W otrzymanym wielomianie z błędami R(x) zamiast x po kolei podstawiamy a^i dla i = 1, 2, 3… 2t.

2. **Wielomian lokalizujący błędy perz metodę Euklidesa:**
   a) $S(x) = \sum_{i=1}^{2t} S_i X^{i-1}\, S(x)\, S(x) = \sum_{i=1}^{2t} S_i X^{i-1}$;

   b) Dzielimy x^{2t} przez S(x): $\frac{x^{2t}}{S(x)} = q_1 * S(x) + \frac{r_1}{S(x)}$ – jeśli r1 jest mniejszy od t to oznacza ostatni krok;

   c) Dalej kontynuujemy wykonanie dzielenia dzielnika z poprzedniego kroku na resztę z tego kroku, dopóki r_i nie będzie mniejszy od zdolności korekcyjnej;

   d) Umieszczamy poprzednie wyniki w poniższą formę:

   S(x)σ(x) = A(x) + x^2B(x), gdzie σ(x) jest szukanym wielomianem

3. **Szukanie pierwiastków wielomianu za pomocą metodę Chiena:**
   Metoda jest podobna do obliczania syndromów, ale dla i = 0, 1 ... 2t − 1 potęg α .

Otrzymane wartości w poprzednim kroku to miejsca błędów w wielomianie R(x). Aby naprawić te błędy, musimy znaleźć wartości, których dodanie (dodawanie w ciele Galois to XOR) zmieni odpowiednie współczynniki x. Te wartości oznaczymy jako y_i. W zależności od ilości błędów mamy różną ilość y_i, z_i różną ilość równań nam potrzebnych dla kolejnych obliczeń. Na przykład, dla dwóch błędów potrzebujemy:

$$y_1(z_1) + y_2(z_2) = S_1$$
$$y_1(z_1)^2 + y_2(z_2)^2 = S_2$$

$$DET\,|Y| = DET \begin{vmatrix} z_1 & z_2 \\ (z_1)^2 & (z_2)^2 \end{vmatrix}$$

$$y_1 = \begin{vmatrix} z_2 & S_1 \\ (z_1)^2 & (z_2)^2 \end{vmatrix}$$

$$y_2 = \begin{vmatrix} z_1 & S_1 \\ (z_1)^2 & S_2 \end{vmatrix}$$

4. **Ostatnim krokiem jest operacja XOR pomiędzy wartościami błędnymi i y_i:**
   E(x)' = $\sum_{i=1}^{T} y_i x_i$
   C(x)' = R(x) + E(x)

```
def decoder(self, received: list[int]):  2 usages
    syndromes = self.calculate_syndromes(received)
    errors_location = self.algorithm_Euclidean(syndromes)
    roots = self.chein_Search(errors_location)
    y_tab = self.error_Values(syndromes, roots)
    e_x = [64] * (roots[len(roots) - 1] + 1)
    for i in range(1, len(roots) + 1):
        e_x[roots[len(roots) - i]] = y_tab[len(y_tab) - i]

    e_x = list(reversed(e_x))
    c_x = self.gf.sum_two_polynomials(received, e_x)

    return c_x
```

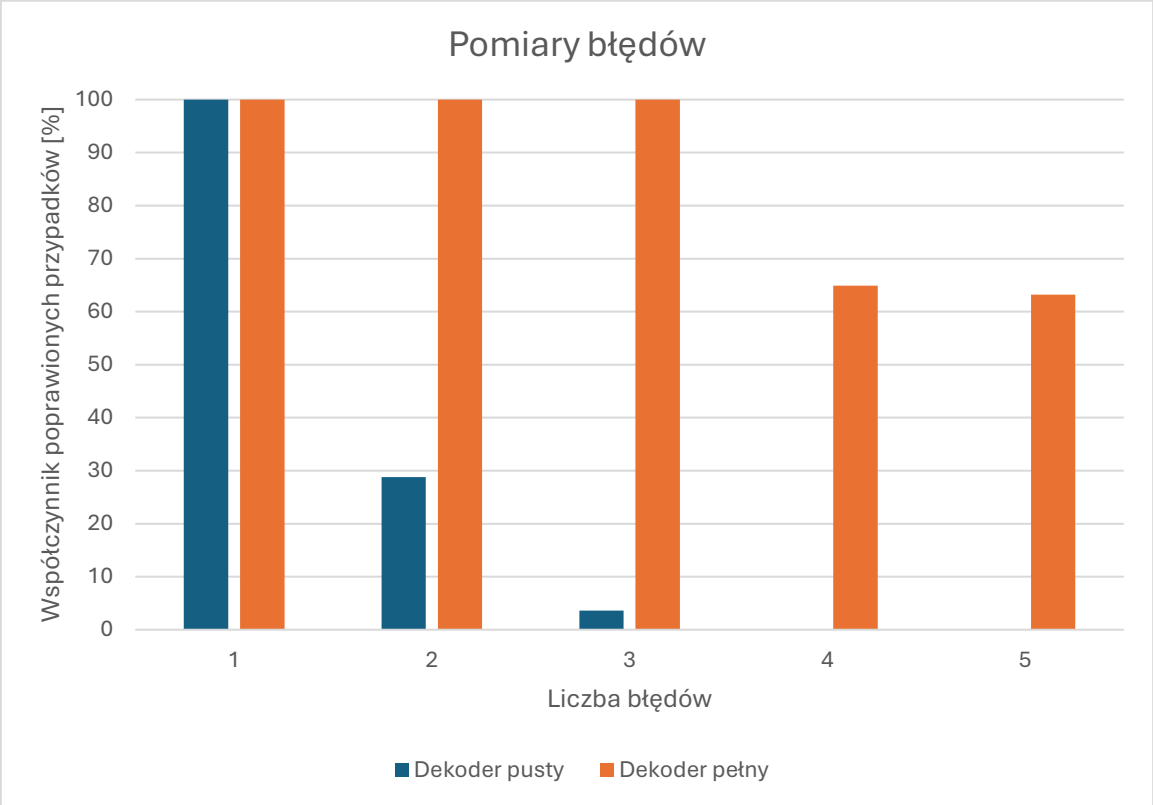
Główna funkcja dekodera pełnego

Pomiary

Nasze testy zostały zrealizowane jako błędy losowe, czyli zastąpienie symboli na losowych indeksach w wielomianie innymi wartościami. Są oddalone od siebie dowolnie, ponieważ przy odległości mniej niż 2t będą naprawiane wszystkie błędy. Pomiary dla dekodera prostego przeprowadziliśmy na 250 próbkach losowych, a dla dekodera pełnego na 1000 próbkach losowych.

Pomiary błędów na symbolach				
Liczba błędnych symboli	Liczba poprawionych przypadków Dekoder prosty	Współczynnik poprawionych przypadków [%] Dekoder prosty	Liczba poprawionych przypadków Dekoder prosty	Współczynnik poprawionych przypadków [%] Dekoder prosty
1	250	100	1000	100
2	72	28,8	1000	100
3	9	3,6	1000	100
4	0	0	649	64,9
5	0	0	632	63,2

Tabela 1. Pomiary błędów na symbolach na losowych próbach.



Wykres 1. Porównanie dekoderów prostego i pełnego na błędach.

Wnioski

W ramach projektu zaimplementowano pełny system kodowania i dekodowania kodu Reed-Solomona o parametrach n=63, k=53, t=5 w ciele skończonym GF(2^6). Opracowane algorytmy zostały przetestowane pod kątem poprawności i skuteczności w korekcji błędów.

Implementacja

- Stworzono klasę do operacji w ciele Galois, obejmującą podstawowe operacje arytmetyczne w tym ciele i manipulację wielomianami.
- Zaimplementowano koder, który poprawnie generuje wektory kodowe, dodając symbole parzystości.
- Opracowano dwa warianty dekodera: prosty oraz pełny, wykorzystujące syndromy (dotyczy obu dekoderów) oraz algorytm Euklidesa oraz metodę Chiena (dotyczy dekodera pełnego) do lokalizacji i korekcji błędów.

Skuteczność dekodowania

- Dekoder prosty działa poprawnie w określonej zdolności korekcyjnej t=5, jednak im większa ilość błędów, skuteczność znacznie maleje, czego należało oczekiwać ze względu na ograniczenia dekodera tego typu.
- Nasz dekoder pełny umożliwia korekcję do 3 błędnych symboli. Wyniki eksperymentów pokazały, że implementacja nie koryguje skutecznie większej ilości błędów, co wynika z niepełnej realizacji tego dekodera.

Testowanie

- Dekoder prosty:
    - Skuteczność dla 1 błędu: 100%, dla 2 błędów: 28,8%, dla 3 błędów: 3,6%.
    - Przy ≥4 błędach dekoder nie koryguje wiadomości (0% skuteczności), ponieważ testy odbywały się na losowych próbach i odległościach pomiędzy poszczególnymi błędami. Prawdopodobieństwo wystąpienia przypadku możliwego do skorygowania przy t≥4 zmierza do 0.

- Dekoder pełny:
    - 100% skuteczności dla ≤3 błędów.
    - Dla 4 błędów: 64,9%, dla 5 błędów: 63,2% - nasz dekoder pełny charakteryzuje się niepełną skutecznością teoretycznej detekcji błędów tj. dekoder ten nie koryguje błędnych symboli gdy ich liczba przekracza 3. Problemem prawdopodobnie jest nieobsługiwanie macierzy wymiaru większego niż 3x3

Podsumowanie

Udało się zaimplementować podstawowe funkcjonalności zadane przez Prowadzącego. Przy implementacji dekodera pełnego wystąpiły pewne problemy, których nie udało się naprawić. Co za tym idzie – dekoder pełny nie realizuje w pełni swojego zadania.