Niezawodność i diagnostyka układów cyfrowych –

Projekt Sprawozdanie

Jan Napieralski – 273036

Anastasiya Martynenka - 276720

26.05.2024r

Zadanie projektowe	
Koder i dekoder kodu Reed-Salomon	
Prowadzący kurs	Dr. Inż. Maciej Nikodem
Termin zajęć	Poniedziałki Nieparzyste 7:30-9:15
Termin oddania	

Podział pracy

Anastasiya Martynenka – implementacja Dekodera prostego i złożonego, Enkodera, pisanie testów jednostkowych, poprawki implementacji ciała Gallois, sprawozdanie.

Jan Napieralski – implementacja ciała Gallois, Enkodera i Transmitera symulującego kanał szumu. Zrobienie pomiarów i sprawozdania.

Wprowadzenie

Kod Reeda-Salomona, opracowany w 1960 roku przez Irvinga S. Reeda i Gustava Solomona, jest jednym z najefektywniejszych i najczęściej stosowanych kodów korekcyjnych w systemach cyfrowych. Jest często stosowany w dyskach CD, DVD, Blue-ray, pamięci flash oraz w systemach komunikacji satelitarnej i standardach telekomunikacyjnych DVB i ATSC.

Jest to kod blokowy, operujący na symbolach zamiast poszczególnych bitów. Arytmetyka takich symboli opiera się na binarnych ciałach skończonych Gallois.

Parametry naszego kodu:

$zdolność\ korekcyjna\ t - 3$;

$liczba\ pozycji\ kontrolnych\ r - 6$ ($r = 2 \cdot t$);

$liczba\ symboli\ informacyjnych\ k - 57$ ($k = n - r = 63 - 6$);

$długość\ wektora\ kodowego\ n - 63$ ($n = 2^s - 1$);

$potęga\ binarnego\ ciała\ Gallois\ symboli\ s - 6$.

Kod o takich parametrach może skorygować do $t = \frac{n-k}{2}$ błędnych symboli, czyli w naszym przypadku 3. Zdolność do wykrycia błędów jest na poziomie $2t$ czyli 6 symboli.

Implementacja ciała skończonego

Cały projekt został zaimplementowany w języku Python. Realizację projektową zaczęliśmy od stworzenia klasy w której zrobiliśmy wszystkie potrzebne metody do pracy na zadanym ciele $GF(2^6)$:

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

Wielomiany w koderze jak i w dekoderze traktowaliśmy jako tablice potęg pierwiastków pierwotnych (α), w których położenie danej potęgi definiowało stopień wielomianu. Dla przykładu kodowany przez nas wielomian $\alpha^{61} * x^3 + \alpha^2 * x^2 + \alpha^{13} * x^1 + \alpha^{34} * x^0$ miał postać [61,2,13,34].

Wartościami potęgi pierwiastków pierwotnych tego ciała są liczby całkowite z przedziału <0;62> i (63;64>. Ze względu na brak domyślnego kodowania wartości zero, ponieważ jako α^0 = 1, zdecydowaliśmy się je zakodować jako α^{64}.

Dla obliczenia wielomianu generującego został zastosowany wzór:
$g(x) = \prod_{i=1}^{r}(x + \alpha^i)$.

```python
# funkcja obliczania wielomianu generającego g(x)
def generate_generating_polynomial(self, t: int) -> list[int]:
    if 2*t < 1:
        raise ValueError("Power must be greater than or equal to 1")

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

    return generating_polynomial
```

Obliczanie wielomianu generującego

Przy dodawaniu wielomianów, do dodawania współczynników x (liczb w ciele Gallois) jest stosowana operacja bitowa XOR. Na przykład, przy dodawaniu wielomianów $(a^2x^2 + a^3x) + (a^7x^2 + a^5x)$ otrzymujemy jako wynik $ax^2 + a^{15}x$, ponieważ w naszym kodzie $a^2 = 000100$, $a^3 = 001000$, $a^7 = 000110$, $a^5 = 100000$.

```python
def add_polynomials(
    self, polynomial1: list[int], polynomial2: list[int]
) -> list[int]:

    sum_polynomials = [polynomial1, polynomial2]
    max_dlugosc = max(len(x) for x in sum_polynomials)

    # wyrównujemy oba wielomiany żeby miały taką samą długość
    # przez dopisanie reprezentacji 0 z lewej strony do mniejszego
    for i in range(len(sum_polynomials)):
        brakujace_zera = max_dlugosc - len(sum_polynomials[i])
        sum_polynomials[i] = [64] * brakujace_zera + sum_polynomials[i]

    wynik = []
    # dla każdego symbolu (czyli potęgi alfy) odnajdujemy jej wartość w ciele
    # i dodajemy modulo dwie reprezentacje alf przy tych samych potęgach do siebie
    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)

        # ponieważ dodaliśmy wartości alf, musimy dopisać do wielomianu wynikowego
        # odnalezioną potęgę sumowanej wartości
        wynik.append(self.find_alfa_power(bin(suma)[2:]))
    return wynik
```

Dodawanie dwóch wielomianów w ciele

Dla mnożenia wielomianów przy mnożeniu współczynników x (liczb w ciele Gallois) zastosujemy inną metodę – dodajemy wykładniki i jeśli wynik jest większy od $n-1$, to stosujemy $mod\ n$ do tego wyniku. Przykład: $a^{17}x^2 \cdot a^{50}x^2 = a^{(17+50)\ mod\ 63}x^2 = a^4x^2$

```python
def mul_polynomials(
    self, polynomial1: list[int], polynomial2: list[int]
) -> list[int]:

    sum_polynomials = list()

    # na początku będziemy brali poszczególne symbole z drugiego wielomianu
    # i przemnażając je przez wszystkie symbole pierwszego, otrzymamy listę
    # wielomianów częściowych, któe potem trzeba będzie dodać do jednego
    for i, symbol in enumerate(polynomial2):

        # mnożenie przez 0 nie ma sensu
        if symbol == 64:
            pass

        # 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)

        # po pomnożeniu przez x do danej potęgi, mnożymy alfy, czyli dodajemy ich potęgi
        for j in range(len(help1)):
            if help1[j] != 64:
                help1[j] = self.add_alfa_powers(help1[j], symbol)

        sum_polynomials.append(help1)

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

    # Wyrównaj długości wielomianów częściowych przed dodawaniem
    for i in range(len(sum_polynomials)):
        brakujace_zera = max_dlugosc - len(sum_polynomials[i])
        sum_polynomials[i] = [64] * brakujace_zera + sum_polynomials[i]

    wynik = []
    # analogicznie jak dla sum_polynomials, dodajemy wielomiany częściowe do siebie
    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)
        wynik.append(self.find_alfa_power(bin(suma)[2:]))

    return wynik
```

Mnożenie dwóch wielomianów w ciele

Implementacja kodera

Koder został stworzony jako klasa z metodą przyjmującą informację jako liczbę stałoprzecinkową. Taka liczba jest zamieniana na jej wartość binarną, wypełniana z lewej strony zerami, żeby była podzielna przez długość pojedynczego symbolu. Następnie dla podzielonych bitów wyszukuje się jakiej potędze pierwiastka pierwotnego dana wartość odpowiada i wstawia się ją do tabeli reprezentującej wielomian. Gdy mamy wielomian generujący, to obliczenie wektora kodowego $c(x)$ jest wykonywane w trzech etapach:

1) Mnożymy wielomian odpowiadający informacji przez x^{n-k}:
   $x^{n-k} \cdot m(x)$
   Dla naszej implementacji to znaczy, że musimy dodać do wielomianu 2t reprezentacji 0 na końcu jako sumę kontrolną.
2) Otrzymany iloczyn dzielimy przez wielomian generujący kod g(x) i wyznaczamy resztę r(x) z tego dzielenia:
   $x^{n-k} \cdot m(x) = q(x) \cdot g(x) + r(x)$
3) Dodajemy iloczyn $x^{n-k} \cdot m(x)$ i otrzymaną resztę r(x):
   $c(x) = x^{n-k} \cdot m(x) + r(x)$

```python
def code_vector(self, m: list[int], t: int):
    if t < 1:
        raise ValueError("Power must be greater than or equal to 1")

    # mnożymy m(x) i x^power
    # dodajemy 64 na koniec (64 reprezentacja 0 w ciele Galua)

    g = self.generate_generating_polynomial(t)
    m += [64] * (2*t)
    r = self.div_polynomials(m, g)

    code_vec = self.add_polynomials(m, r)

    return code_vec
```

Kodowanie wielomianu informacyjnego m

Implementacja dekodera prostego

Dekoder prosty udało nam się w całości pokryć w funkcji simple klasy Decoder. Ten typ dekodera polega na przesuwaniu symboli informacyjnych na miejsca kontrolne i liczenie syndromów poprzez dzielenie tak przesuniętego wielomianu przez wielomian generujący. Jeśli waga syndromu jest w granicach zdolności korekcyjnej, to dodajemy do siebie wielomian odebrany z syndromem i przesuwamy symbole na oryginalne pozycje.

Kroki wykonania dekodowania poprzez dekoder uproszczony:

1) Przyjmujemy wektor c_y i zerujemy licznik przesunięć i;
2) Obliczamy syndrom s_i jako wynik dzielenia wektora $c_{y(i)}$ przez wielomian generujący $g(x)$. Dalej obliczamy wagę syndromu w_i czyli sumę wszystkich symboli które nie są równe zero (czyli 64 w naszej reprezentacji).
   $s_i = c_i(mod\ g(x)), w_i = w_i(s_i)$
3) Sprawdzamy, czy waga syndromu mniejsza lub równa zdolności korekcyjnej t: jeśli tak – przechodzimy do kroku 7), jeśli nie – do kroku 4).
4) Jeśli waga syndromu jest większa od t, sprawdzamy, czy licznik przesunięć i jest równy liczbie bitów informacyjnych k: jeśli tak – błędy nie są korygowalne, jeśli nie – przechodzimy do kroku 5).
5) Jeśli $i < k$, robimy przesunięcie cykliczne wektora odebranego i zwiększamy i o jeden.
6) Powtarzamy powyższe kroki, dopóki waga syndromy w_i nie będzie mniejsza lub równa zdolności korekcyjnej.
7) Ostatnim krokiem wykonujemy korekcje błędów – robimy operacje XOR między wektorem $c_{y(i)}$ a syndromem s_i i przesuwamy otrzymany wektor i razy cyklicznie w przeciwną stronę.

```python
def simple(self, wektor_odebrany: list[int]):
    wektor_skorygowany = list()
    g = self.gf.generate_generating_polynomial(self.t)

    # dla każdego symbolu  w części informacyjnej
    for i in range(0, self.k + 1):
        syndrom = self.gf.div_polynomials(wektor_odebrany, g)
        waga_syndromu = 0

        # liczymy wagę syndromu czyli ile niezerowych symboli ma w sobie
        for el in syndrom:
            if el != 64:
                waga_syndromu += 1


        if waga_syndromu <= self.t:
            # da się naprawić błąd, dodajemy syndrom do wektora i obracamy tak długo
            # żeby przywrócić oryginalną kolejność symboli
            wektor_skorygowany = self.gf.add_polynomials(wektor_odebrany, syndrom)
            for _ in range(0, i):
                wektor_skorygowany.insert(63, wektor_skorygowany.pop(0))
            break
        else:
            # jesteśmy na końcu obracania wielomianu i nadal nie można naprawić błędu
            if i == self.k:
                return "Błędy niekorygowalne"
            else:
                # obracamy wielomian o jeden w prawo
                wektor_odebrany.insert(0, wektor_odebrany.pop())

    return wektor_skorygowany
```

Dekoder prosty

Implementacja dekodera pełnego

Dekoder pełny jest zaimplementowany w funkcji full klasy Decoder. Składa się on z obliczania $2t$ syndromów wielomianu, przygotowaniem wielomianu lokalizującego błędy, wyliczaniem pierwiastków tego wielomianu i w oparciu o nie, naprawie błędnych symboli. Jako algorytm generowania wielomianu lokalizującego błędy wybraliśmy algorytm Euklidesa zamiast Berlekampa. Jako algorytm wyszukiwania pierwiastków wielomianu lokalizującego zastosowaliśmy wyszukiwanie Chiena, czyli po prostu podstawianie kolejno $\alpha^0, \alpha^1, \alpha^2$ pod x i sprawdzenie czy wynik jest równy 0.

Stosowane wzory dla różnych etapów implementacji:

1) Obliczenie $2t$ syndromów:
   W otrzymany wielomian z błędami $R(x)$ zamiast x po kolei podstawiamy a^i dla $i = 1, 2 \dots 2t$.
2) Wielomian lokalizujący błędy perz metodę Euklidesa:
   a) $S(x) = \sum_{i=1}^{2t} S_i X^{i-1}$;
   b) Dzielimy x^{2t} przez $S(x)$: $\frac{x^{2t}}{S(x)} = q_1 \cdot S(x) + \frac{r_1}{S(x)}$ – jeśli r_1 jest mniejszy od t znaczy to 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:
   $S(x)\sigma(x) = A(x) + x^2 B(x)$, gdzie $\sigma(x)$ jest szukanym wielomianem
3) Szukanie pierwiastków wielomianu za pomocą metodę Chiena:
   Metoda jest podobna do obliczania syndromów, ale dla $i = 0, 1 \dots 2t - 1$ potęg a.
4) 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 i różną ilość równań nam potrzebnych dla kolejnych obliczeń. Na przykład, dla dwóch błędów potrzebujemy:
   $y_1(z_1) + y_2(z_2) = S_1$
   $y_1(z_1)^2 + y_2(z_2)^2 = S_2$
   W takim przypadku kolejne obliczenia będą wyglądali następująco:
   $DET\,|Y| = DET\,|\begin{matrix} z_1 & z_2 \\ (z_1)^2 & (z_2)^2 \end{matrix}|$
   $y_1 = |\begin{matrix} z_2 & S_1 \\ (z_2)^2 & S_2 \end{matrix}|$
   $y_2 = |\begin{matrix} z_1 & S_1 \\ (z_1)^2 & S_2 \end{matrix}|$
5) Ostatnim krokiem jest operacja XOR pomiędzy wartościami błędnymi i y_i:
   $E(x)' = \sum_{i=1}^{T} y_i x_i$
   $C(x)' = R(x) + E(x)'$

```python
def full(self, received: list[int]):
    syndromes = self.__calculate_syndromes(received)
    errors_location = self.__euclidean_algorythm(syndromes)
    roots = self.__chein_search(errors_location)
    y_tab = self.__error_values(syndromes, roots)

    if len(roots) == 0:
        return []

    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.add_polynomials(received, e_x)

    return c_x
```

Główna funkcja dekodera pełnego

Pomiary

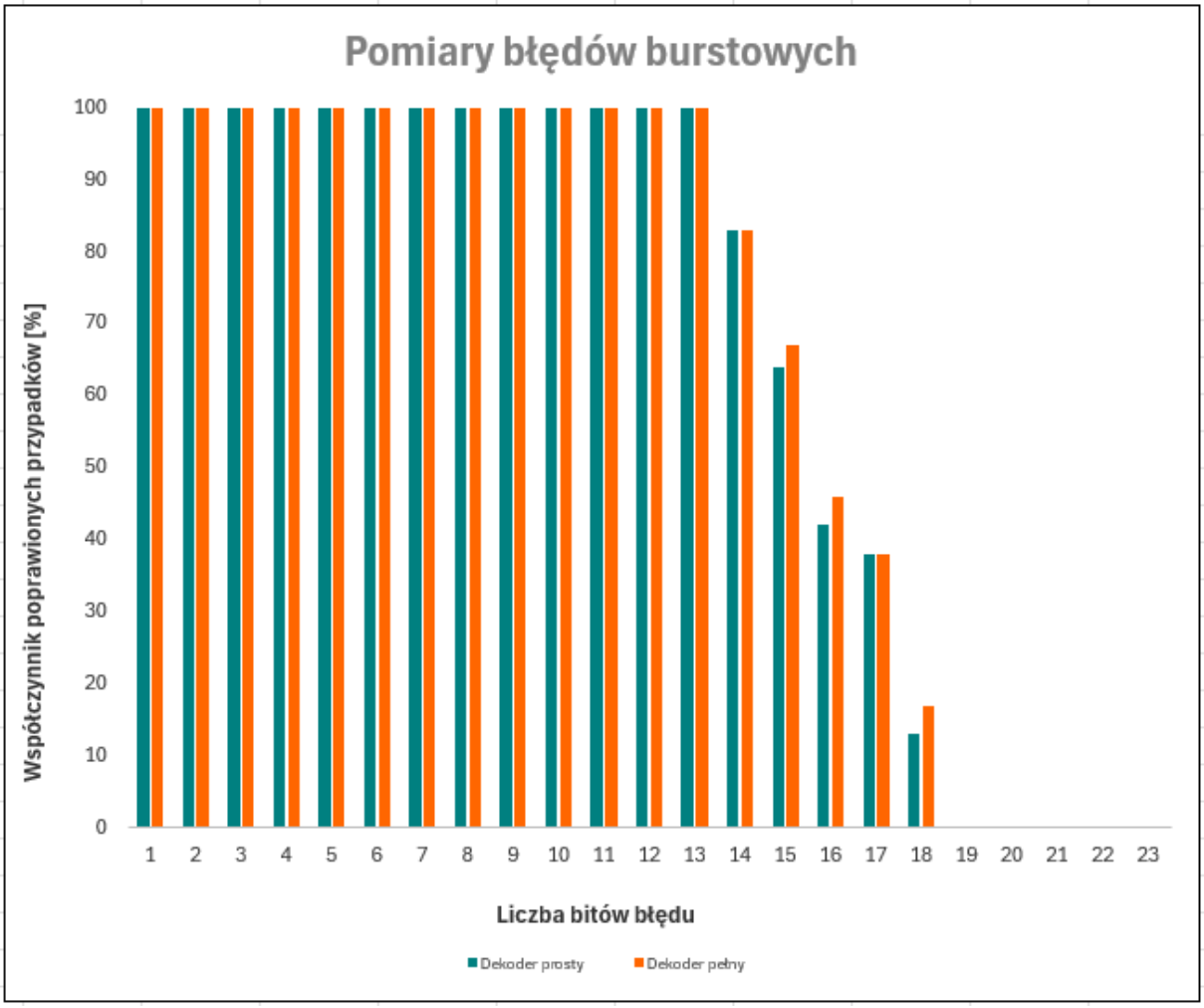
Zrobiliśmy testy dla dwóch przypadków błędów – burstowych oraz na symbolach. Błędy burstowe powstają, gdy w sąsiadujących ze sobą bitach powstaną przekłamania łącząc się w ciągłe grupy. Błędy na symbolach zostały zrealizowane jako zastąpienie na losowych indeksach symboli w wielomianie innymi wartościami. Mogą być dowolnie od siebie oddalone. Wszystkie pomiary przeprowadziliśmy na 600 próbkach losowych.

Pomiary błędów burstowych 600 prób				
Liczba bitów błędu	Liczba poprawionych przypadków Dekoder prosty	Współczynnik poprawionych przypadków [%] Dekoder prosty	Liczba poprawionych przypadków Dekoder pełny	Współczynnik poprawionych przypadków [%] Dekoder pełny
1	600	100	600	100
2	600	100	600	100
3	600	100	600	100
4	600	100	600	100
5	600	100	600	100
6	600	100	600	100
7	600	100	600	100
8	600	100	600	100
9	600	100	600	100
10	600	100	600	100
11	600	100	600	100
12	600	100	600	100
13	600	100	600	100
14	498	83	498	83
15	384	64	402	67
16	252	42	276	46
17	228	38	228	38
18	78	13	102	17
19	0	0	0	0
20	0	0	0	0
21	0	0	0	0
22	0	0	0	0
23	0	0	0	0

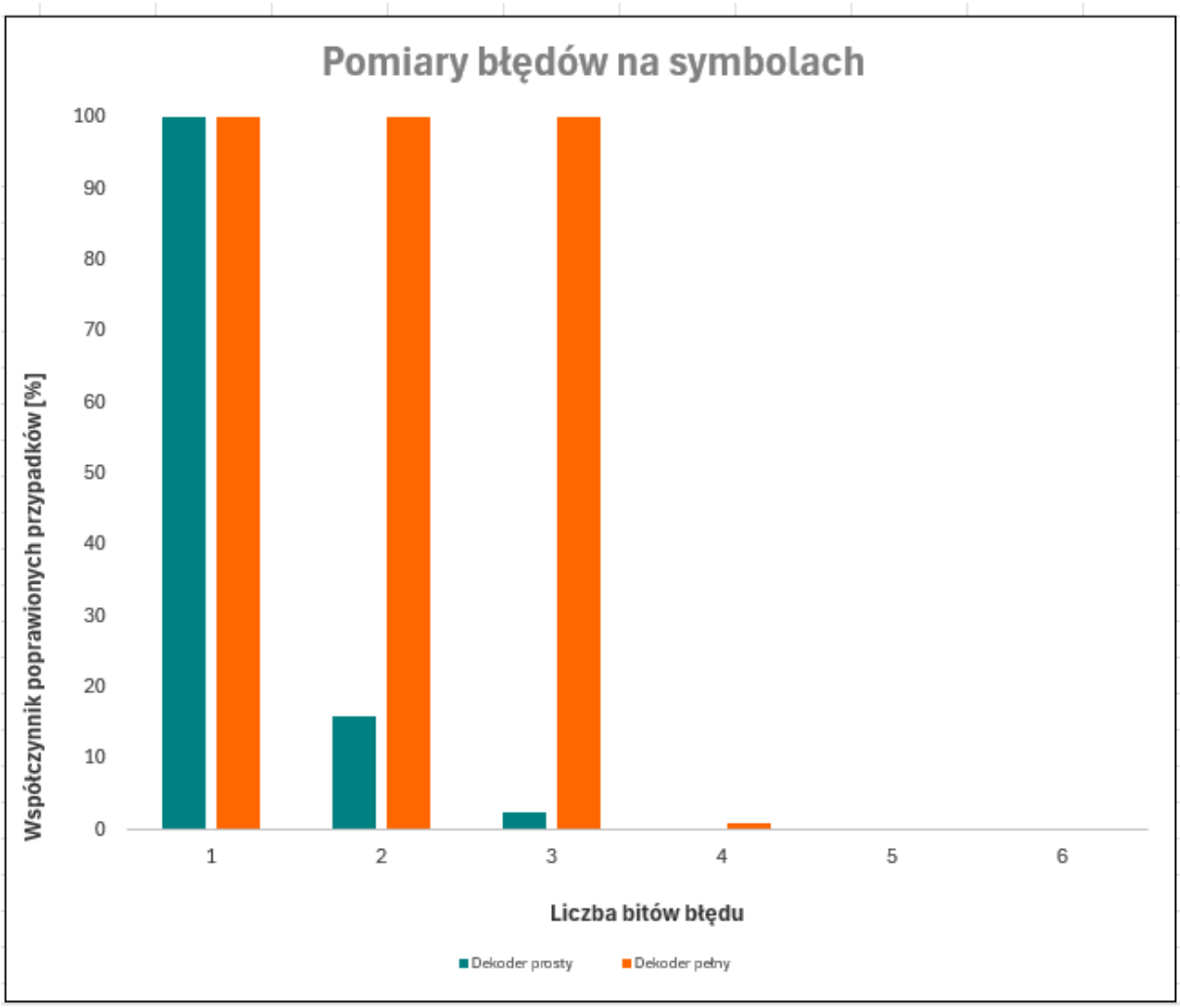
Tabela 1. Pomiary błędów burstowych

Pomiary błędów na symbolach 600 prób				
Liczba błędnych symboli	Liczba poprawionych przypadków Dekoder prosty	Współczynnik poprawionych przypadków [%] Dekoder prosty	Liczba poprawionych przypadków Dekoder pełny	Współczynnik poprawionych przypadków [%] Dekoder pełny
1	600	100	600	100
2	96	16	600	100
3	15	2,5	600	100
4	0	0	6	1
5	0	0	0	0
6	0	0	0	0

Tabela 2. Pomiary błędów symbolicznych na losowej próbie 100 przypadków



Wykres 1. Porównanie dekoderów prostego i pełnego na błędach burstowych



Wykres 2. Porównanie dekoderów prostego i pełnego na błędach symbolicznych

Wnioski

Ze względu na blokowy charakter, kod Reeda-Salomona bardzo dobrze sprawdza się w przypadku naprawiania błędów wiązkowych, co widać na pomiarach. Aż do błędów obejmujących przylegające 13 bitów, oba dekodery miały 100% skuteczność w naprawie informacji. Dzieje się tak, ponieważ takie bity przekładają się bezpośrednio na przekłamane symbole koło siebie, a nawet najprostszy kod Reeda-Salomona jest w stanie naprawić 1 błędny symbol, co w naszym przypadku ciała Gallois oznacza aż 6 bitów.

Im więcej pojawiało się błędów ponad 13 bitów, tym skuteczność dekodowania malała. Teoretycznie oba dekodery powinny być w stanie poradzić sobie z 18 przekłamanymi bitami koło siebie, co równałoby się idealnie 3 symbolom, jednak początek wiązki błędów mógł pojawić się w połowie symbolu, przez co taka wiązka mogła równać się większej ilości błędnych symbolów niż wynikałoby to tylko z przeliczenie na ciało Gallois. Dekoder pełny naprawiał nieco więcej przypadków w tych testach, jednak do oceny czy jest on skuteczniejszy należałoby przeprowadzić więcej testów, żeby wyeliminować błędy pomiarowe.

Jeśli chodzi o błędy na symbolach, to występowała duża dysproporcja pomiędzy dekoderem prostym i złożonym. Pomimo takiej samej teoretycznej zdolności korekcyjnej, dekoder prosty nie radził sobie już z 2 błędnymi symbolami, podczas gdy dekoder pełny zachował 1000% skuteczności naprawy dla 3 symboli. Podczas badania przyczyny takiej rozbieżności zauważyliśmy, że dekoder prosty był w stanie naprawić tylko te symbole, które były od siebie oddalone o maksymalnie 2t symboli, czyli o zdolność korekcyjną. Ponieważ błędy na symbolach były generowane na losowych indeksach, czyli współczynnikach wielomianu, to występowało stosunkowo mało przypadków tak bliskiego oddalenia błędnych symboli. Wraz ze zwiększeniem błędów do 3, ta sama zależność miała miejsce. Dekoder prosty naprawiał tylko te 3 symbole które sumarycznie były oddalone o $2t$ symboli, czyli w naszym przypadku o 6.

Podsumowując, dekoder pełny jest lepszy od dekodera prostego. Zachowuje 100% zdolności korekcyjnej wynikającej z właściwości kodu niezależnie od ustawienia symboli. Może też wykryć istnienie większej ilości błędów nienaprawialnych niż dekoder prosty i dać o tym informację. Czasami udaje mu się podać lokalizację błędów, jeśli ich liczba nie przekracza znacząco zdolności korekcyjnej.