Kodowanie danych w systemach

Encoding PCM
Kod Hamminga
Kod Rice'a

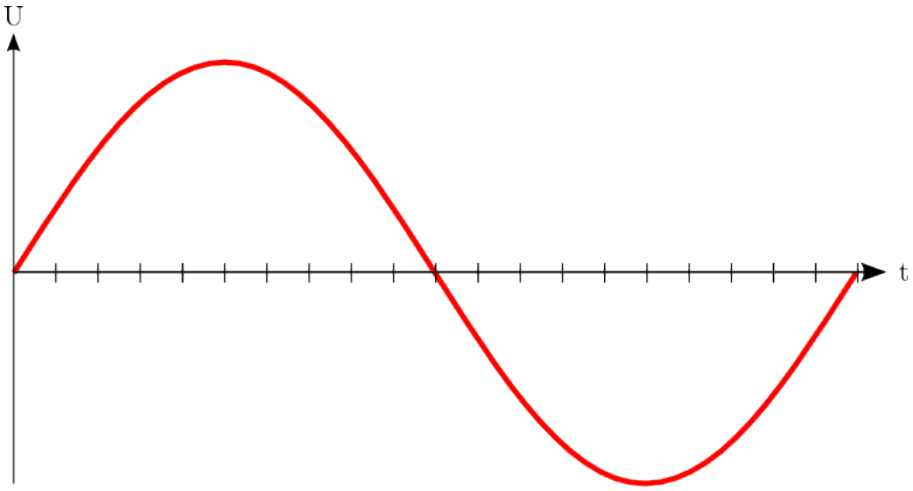
Sergiusz Urbaniak, IUz-12, 2008

Treść

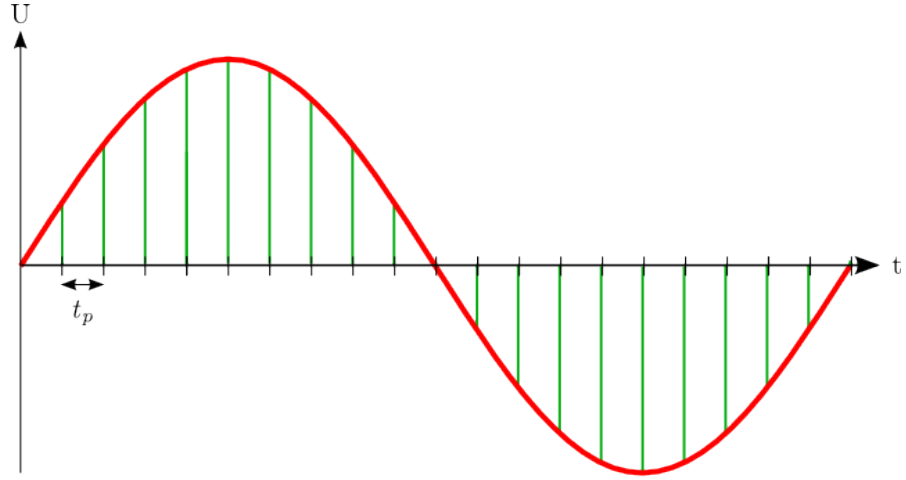
PCM

Opis ogólny

Kod PCM jest wykorzystany przy kwantyzacji sygnałów audio na sygnały cyfrowe. Analogowy sygnał jest zmiennym napięciem w czasie.



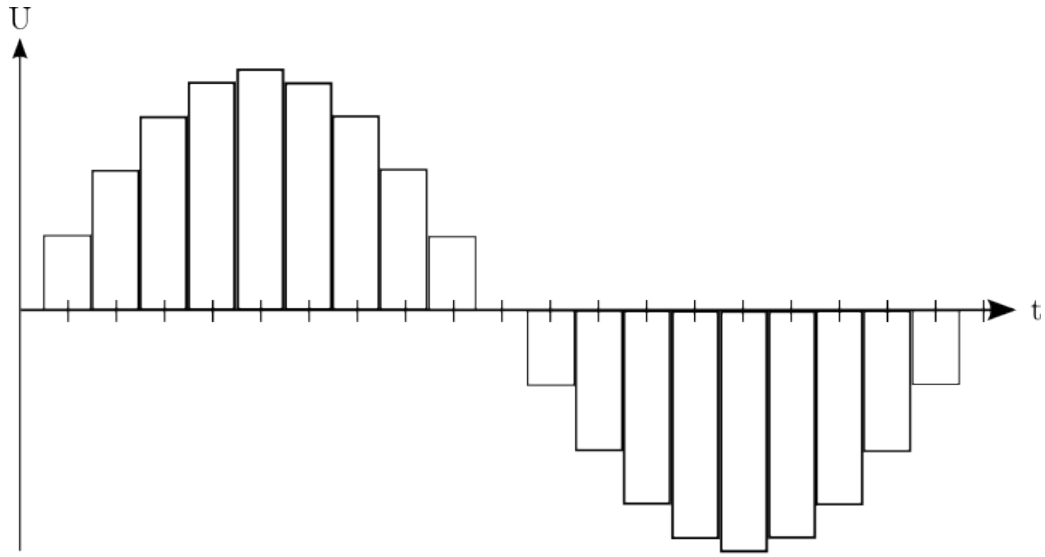
Za pomocą próbkowania wysokość napięcia sygnału analogowy jest czytany co jakiś czas określony przez czas próbkowania $t_p[s]$.



Czas próbkowania określa częstotliwość próbkowania:

$$f_p[Hz]=\frac{1}{t_p}$$

Na końcu są wysokości napięć przekształcane w ciąg sampli zapisanych na płycie w formie digitalnej.



Jeden sample zawiera informację o wysokości napięcia sygnału analogowego w czasie t_p . Ponieważ sample zawierają tylko średnią wysokość napięcia w czasie próbkowania, istnieje pewien błąd odczytu nazwanym błędem kwantyzacji.

Wysokość napięcia jest zapisana w postaci liczb całkowitych. Maksymalna wartość liczby całkowitej określa dokładność. W danym przykładzie można zapisać 2047 różnych napięć określanych zmienną WE:

$$WE = [0..2047]$$

Zakodowana powyższa zmienna jako liczba binarna potrzebuje 11 bitów ($2^{11} = 2048$). Aby zapisać również negatywne napięcia, potrzebny jest dodatkowy bit (sign bit) a więc w sumie 12 bitów. Przedstawiany w tym dokumencie kod PCM jest wstanie skomprymować 12 bitów na 8 bitów, z czego bit ósmy kodu PCM również oznacza pozytywne lub negatywne napięcie. Pozostałe 7 bitów kodu PCM mogą przejąc maksymalnie 128 wartości ($۲^{۷} = ۱۲۸$) określany zmienną WY:

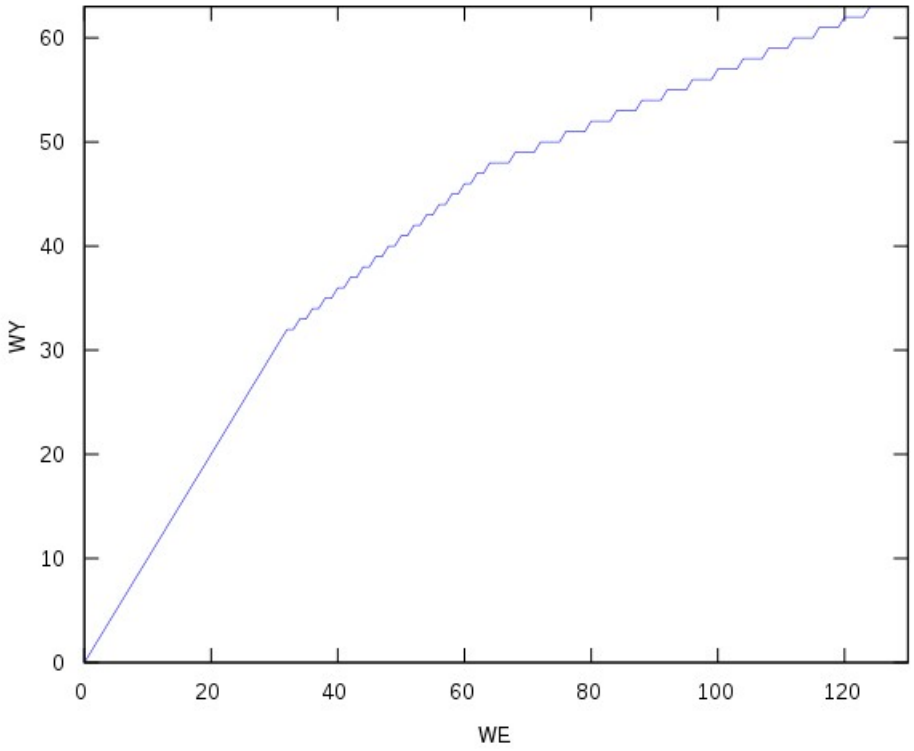
$$WY = [0..127]$$

Koder

Zaimplementowany algorytm działa na przypisaniu wartości zmiennej WE na wartości WY. Za pomocą indeksu i wartości WE są podzielone na grupy. Pierwsza grupa ($i=0$) przepisuję wartości WE bezpośrednio na wyjście WY:

$$WY = WE, \; przy \; WE < ٣٢$$

Związek ten widać na wykresie zaimplementowanego algorytmu PCM. Pierwsze 32 wartości na wykresie WE/WY są linearnie przypisane. Do tej wartości kod nie traci informacji.



Dalsze grupy wartości zmiennych WE są wyliczane za pomocą indeksu i . Grupy wartości WE są określone następującymi początkowymi wartościami:

$$WE_i = 2^{i+4}$$

Odpowiednie początkowe wartości na wyjściu obliczają się następująco:

$$WY_i = 16(i+1)$$

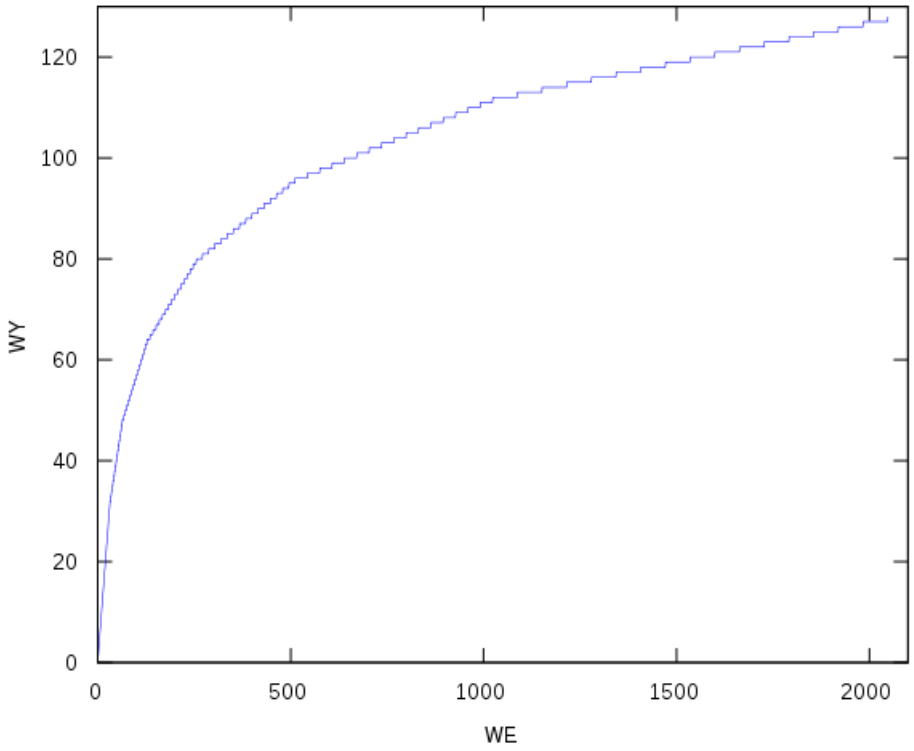
Ilość przepisanych wartości oblicza się następująco:

$$n_i = \frac{WE_{i+1} - WE_i}{16}$$

Na podstawie powyższych wzorów można zapisać następującą tabelę wartości:

i	WE_i	WY_i	n_i
1	32	32	2
2	64	48	4
3	128	64	8
4	256	80	16
5	512	96	32
6	1024	112	64
7	2048	128	

Jak widać, nim wyższa zmienna WE tym więcej wartości musi być przypisanych aby się zmieścić na wyjściu WY. Za tym można interpretować nim większa wartość wejścia (większa amplituda) tym większa strata przy kodowaniu. Następujący wykres pokazuję powyżej opisane straty.



Dekoder

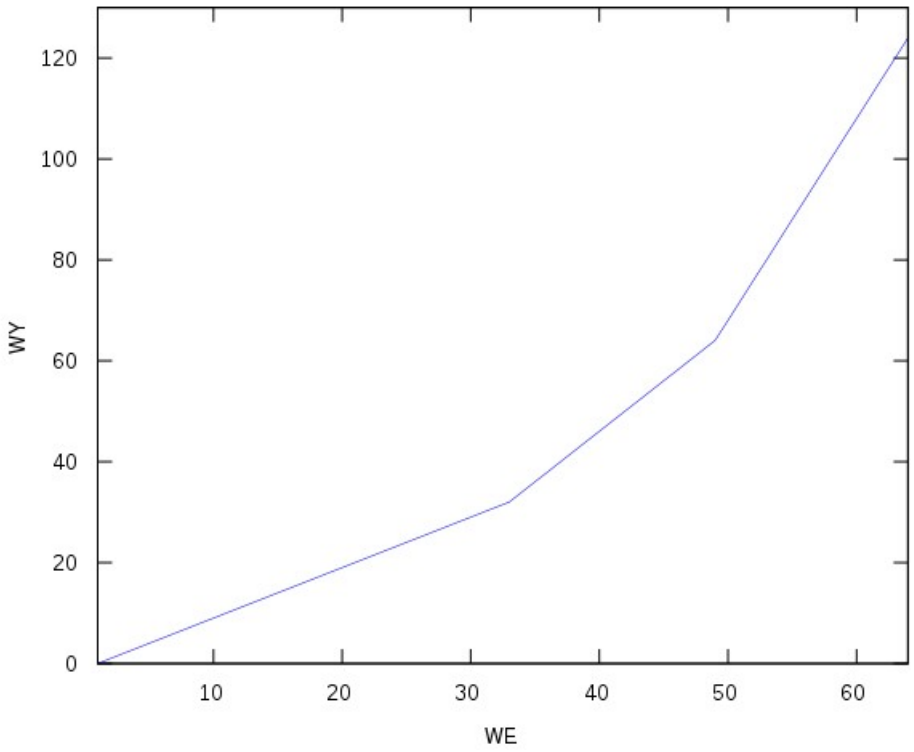
Strona dekodująca jest w implementowanym algorytmie realizowana za pomocą pomocniczej tabeli budowanej w trakcie kodowania. Pamięta ona najbliższe wartości parzyste:

$$WY_{dekoder} = WE_{koder} \, mod \, 2$$

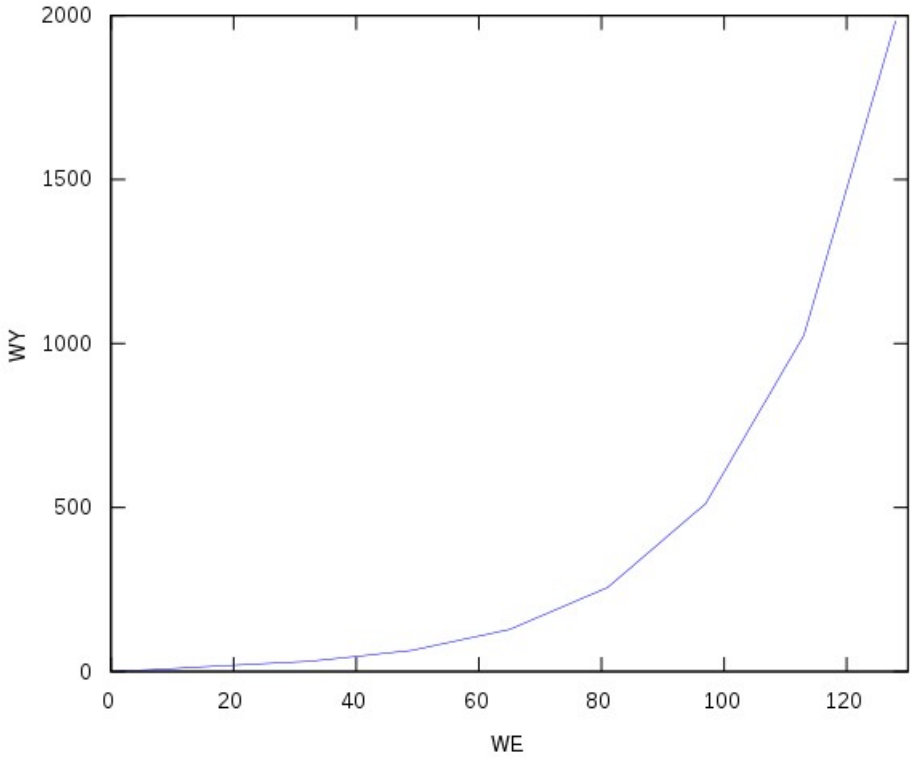
Podobnie jak w kodowaniu wartości WE są na początku bezpośrednio na wyjście WY przypisane:

$$WY_{dekoder} = WE_{dekoder}, \, przy \, WE < 32$$

Związek ten podobnie widać na wykresie WE/WY:

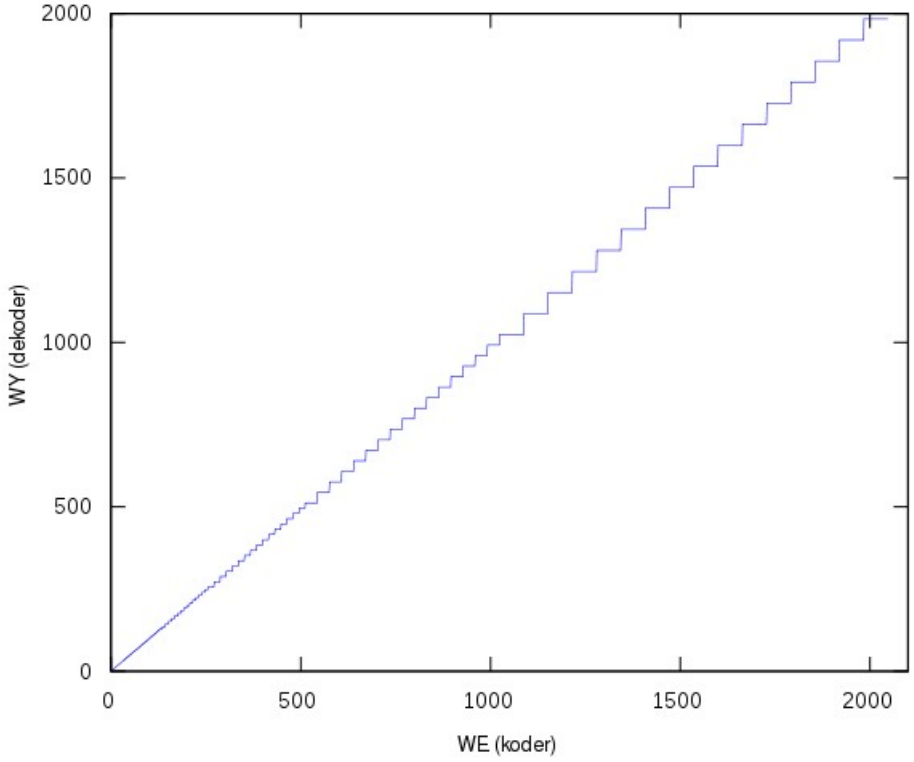


Dalsze wartości są przepisane najbliższą parzystą na wyjściu:



Wnioski

Jak widać kod PCM jest kodem ze stratą. Przy małych amplitudach, a więc „cichych” sygnałach można zauważyć bezstratną przesyłkę danych. Jednak przy większych amplitudach sygnału straty się zwiększają. Dobrze to można zauważyć przy bezpośrednim porównaniu wejścia do kodera i wyjścia z dekodera:



Kod Hamminga

Opis

Kod hamminga jest kodem korekcyjnym. Dodaje on do strumienia bitów bity korekcyjne zawierające informacje o parzystości strumienia. Bit parzystości wskazuję, czy istnieje błąd w ciągu danych i oblicza się operacją XOR.

Niech strumień S_1 bitów składa się z czterech bitów a, b, c, d a bit parzystości p dodany jest na końcu strumienia:

$$\begin{gathered} S_1 = 1101\mathrm{p} \\ \Rightarrow a=1, b=1, c=0, d=1 \\ p = a \oplus b \oplus c \oplus d = 1 \\ \Rightarrow S_1 = 11011 \end{gathered}$$

Bit parzystości pozwala na sprawdzanie poprawności strumienia. Jednak mogą powstać przy tym błędy. Starczy wprowadzić dwa błędy w powyższy strumień a bit parzystości będzie ten sam. Następujący przykład przekłamuje zmienne a i b :

$$\begin{gathered} S_2 = 0001\mathrm{p} \\ \Rightarrow a=0, b=0, c=0, d=1 \\ p = a \oplus b \oplus c \oplus d = 1 \\ \Rightarrow S_2 = 00011 \end{gathered}$$

Strumienia S_1 , S_2 różnią się dwoma bitami. Ilość różniących bitów dwóch strumieni danych określana jest długością hamminga d_H .

$$\begin{array}{r} 1101 \\ \underline{0001} \\ \oplus 1100 \end{array}$$
$$\Rightarrow d_H = 2$$

Można zauważyć że potrzebna jest różnica o długości $d_H = 2$, aby oszukać jeden prosty bit parzystości. Niestety jeden bit parzystości nie jest w stanie skorygować strumienia danych. W ramach na zajęciach zaimplementowanego kodu hamminga wiele bitów parzystości były wprowadzone do strumienia. Pozwala to znaleźć i skorygować jeden przekłamany bit.

Ilość r bitów nadmiarowych stanowiące bity parzystości w przypadku kodu hamminga oblicza się następująco:

$$r = \log_2 n + 1$$

Gdzie n jest ilością kodowanych bitów. Bity parzystości są wkładane w miejsca k_i , gdzie

$$\begin{gathered} k_i = 2^i \\ i = [0..\mathrm{r} - 1] \end{gathered}$$

Bity parzystości obliczają różne pola strumienia. W przypadku

$$n=4$$
$$\Rightarrow r=\log_2 4+1=3$$
$$\Rightarrow k_0=1, k_1=2, k_2=4$$

całkowita długość strumienia wynosi 7 bitów. W implementacji stworzono matryce pomocniczą:

	1	**2**	**3**	**4**	**5**	**6**	**7**
k0	**1**	0	1	0	1	0	1
k1	0	**1**	1	0	0	1	1
k2	0	0	0	**1**	1	1	1

Ta tablica została budowana za pomocą prostego obliczenia wartości binarnych i w pisanych w kolumnach pomocniczej matrycy. Wiersze tej matrycy pomocniczej zawierają miejsca przy których są obliczane parzystości a dokładnie tam gdzie jedynki.

Algorytm został w taki sposób zaimplementowany że może uwzględnić też inne długośći danych:

$$n=16$$
$$\Rightarrow r=\log_2 16+1=5$$
$$\Rightarrow k_0=1, k_1=2, k_2=4, k_3=8, k_4=16$$

Pomocnicza matryca w tym przypadku jest:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**
k0	**1**	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
k1	0	**1**	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
k2	0	0	0	**1**	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1
k3	0	0	0	0	0	0	0	**1**	1	1	1	1	1	1	1	0	0	0	0	0	0
k4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	1	1	1	1	1

Dekoder działa podobnie jak koder. Najpierw obliczane są bity parzystości i porównane z bitami parzystości które zostały przesyłane. Następnie różniące się miejsca indykują błędne miejsca transmisji.

Przykład

Niech następujący przykład zawiera:

$$n = 16$$
$$\Rightarrow r = 5$$

Zaimplementowana funkcja `generate_random_vector` generuję przypadkowy ciąg wejściowych danych:

$$WE = 1101010001100011$$

Następna funkcja `insert_empty_parities` generuję puste bity parzystości:

$$WE_{puste} = \mathbf{00}1\mathbf{0}101\mathbf{0}0100011\mathbf{0}00011$$

W końcu funkcja `calculate_parities` generuję własciwe bity parzystości:

$$WY = \mathbf{1}\mathbf{1}1\mathbf{0}101\mathbf{1}0100011\mathbf{0}00011$$

Niech zostanie przekłamany przedostatni bit wejścia na miejscu dwudziestym:

$$WY_{error} = 111\cdot1\cdot11\cdot1\cdots1\cdot\cdots\square1$$
$$\Rightarrow i_{error} = ٢٠$$

Jeżeli wyliczymy bity parzystości K_{error} dla WY_{error} :

$$k_{0,error} = 1$$
$$k_{1,error} = 1$$
$$k_{2,error} = 1$$
$$k_{3,error} = 1$$
$$k_{3,error} = 1$$

i porównamy z bitami parzystości K_{WY} przesłanymi:

$$k_{0,WY} = 1$$
$$k_{1,WY} = 1$$
$$k_{2,WY} = 0$$
$$k_{3,WY} = 1$$
$$k_{4,WY} = 0$$

to można za operacją XOR wyliczyć błędne pozycję bitów parzystości:

$$K_{\Delta} = K_{error} \oplus K_{WY} = 00101$$

Odwracając K_{Δ} można wyliczyć pozycję błedy:

$$i_{error} = 0b10100 = 20$$

Badania

Badania zostały przeprowadzone na wektorze o długości $n=16$. Ilość bitów nadmiarowych wynosi $r=5$. Zostało generowane 48.000 przypadkowych wektorów, a więc całkowita długość ciągu danych wynosi 1.008.000 bitów.

$$n=16$$
$$\Rightarrow r=5$$
$$n_{vectors}=48.000$$
$$\Rightarrow l_{vectos}=1.008.000\,bits$$

Został wprowadzone przypadkowe błędne bity o prawdopodobieństwu 1%:

$$p_{errors}=1\ \%$$
$$\Rightarrow l_{errors}=l_{vectors}\,p_{errors}=10.080\,bits$$

W następującej tabeli zostały wyliczone wyniki pięciu badań gdzie algorytm próbował korygować powyżej opisane błędne bity. Ilość błędnie korygowanych bitów zostały liczone zmienną n_{false} i prawdopodobieństwo p_{false} .

n_{false}	p_{false}
1838	0.239323%
1750	0.227865%
1849	0.240755%
1739	0.226432%
1837	0.239193%

Widać że średnie prawdopodobieństwo błędnie dekodowanych bitów $p_{false,avg}$ jest prawie czterokrotnie niższe niż p_{false} :

$$p_{false,avg}=0.2347\ \%$$
$$p_{false}=1\ \%$$

Oznacza to że kod hamminga nadaję się do transmisji i korekcji danych, jednak nie bezbłędnie.

Kod Rice'a

Opis

Kod Rice'a koduję liczby naturalne. Aby efektywnie kodować ciąg danych binarnych należy najpierw przekształcić ciąg danych w kod unarny:

$$WE = 0000010001$$
$$darrow\, WE_{unary} = \{5, 3\}$$

W danym przykładzie ilość zer można określić prawdopodobieństwiem:

$$n_{zeros} = 8$$
$$n_{total} = 10$$
$$\Rightarrow p = \frac{n_{zeros}}{n_{total}} = 0.8$$

Pomocnicza zmienna k określa ilość grup kodowych:

$$k = \left\lceil \log_2 \frac{\log_2 \frac{\sqrt{5}-1}{2}}{\log_2(p)} \right\rceil$$

W danym przypadku:

$$k = \left\lceil \log_2 \frac{\log_2 \frac{\sqrt{5}-1}{2}}{\log_2(0.8)} \right\rceil = \lceil 1.1087 \rceil = 2$$

Kod dla liczby x składa się z części unarnej u i binarnej v, gdzie:

$$u = \left\lfloor \frac{x}{2^k} \right\rfloor$$
$$v = x \bmod 2^k$$

W danym przykładzie $x = 5$:

$$u = \left\lfloor \frac{5}{2^2} \right\rfloor = 1$$
$$v = 5 \bmod 2^2 = 1$$

Przetworzone zmienne na odpowiednie kody unarne i binarne:

$$u_{unary} = 01$$
$$v_{binary} = 01$$

Kod przesłany składa się z:

$$WY_5 = u_{unary} : v_{binary} = 0101$$

Następująca tabela zawiera kodowane wartości od 0 do 9 dla $k=2$:

WE_i	$u_{unary,i}$	$v_{binary,i}$	WY_i
0	1	00	100
1	1	01	101
2	1	10	110
3	1	11	111
4	01	00	0100
5	01	01	0101
6	01	10	0110
7	01	11	0111
8	001	00	00100
9	001	01	00101

Dekoder musi znać wartość k aby dekodować dane. W następnym przypadku jest $k=2$.

$$WE_{dekoder}=00101$$
$$k=2$$
$$\Rightarrow u=2$$
$$\Rightarrow v=1$$
$$WY_{dekoder}=u\,2^k+v$$
$$\Rightarrow WY_{dekoder,unary}=2\,2^2+1=9$$
$$\Rightarrow WY_{dekoder}=0000000001$$

Przybliżona wartość kompresji kodu Rice'a można wyliczyć następującym wzorem:

$$L_R=(1-p)\left(k+\frac{1}{1-p^{2^k}}\right)$$

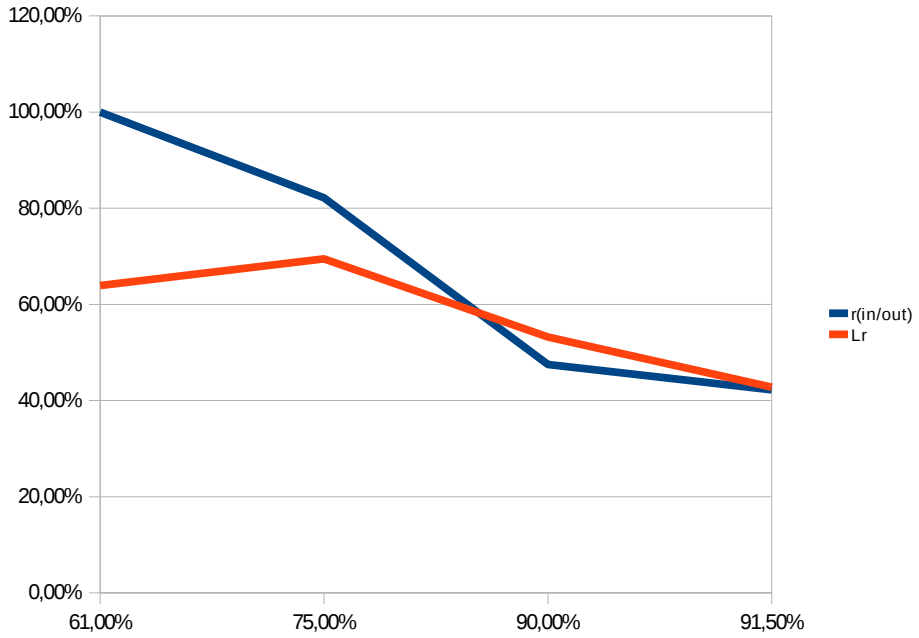
Badania

Cztery pliki zostały sprawdzone z kodem Rice'a. Podane w następującej tabeli jest wejściowa długość danych l_{in}, wyjśćowa długość danych kodowanych l_{out}, wyliczona wartość zmiennej k, ilość wystąpień zer na wejściu dekodera p_0 i miara kompresji $r_{\text{in/out}}$, gdzie:

$$r_{\text{in/out}} = \frac{l_{\text{out}}}{l_{\text{in}}}$$

File	l_{in}	l_{out}	k	p_0	$r_{\text{in/out}}$	L_r
61	100000	100000	0	61%	100%	63.93%
75	100000	82185	1	75%	82.18%	69.44%
90	100000	47503	3	90.00%	47.50%	53.23%
915	10000	4222	3	91.50%	42.22%	42.80%

W końcu dodano przybliżoną wartość kompresji L_r. Widać że algorytm nie działa dobrze przy małych wartościach k i p_0. Dopiero przy większej liczbie zer algorytm efektywnie komprymuję. Związek ten widać dobrze na następującym wykresie:



Miara kompresji spada przy większej ilości zer w strumieniu. Przy małej (61%) ilości zer kompresja nie istnieje.