NiDUC2
Transmisja w systemie ARQ + FEC

Autorzy:
Agnieszka Jurijków
Filip Przygoński
Michał Skrok

Politechnika Wrocławska
Wydział Elektroniki
Prowadzący: mgr inż. Szymon Datko

1 Opis opracowywanego modelu systemu

1.1 Symulowany proces - opis oraz implementacja

Symulowane jest przesyłanie danych binarnych za pomocą techniki ARQ + FEC, w której przed wysłaniem pakietu jest on kodowany, wysyłany a następnie dekodowany po stronie odbiorcy.

- ARQ - Automatic Repeat Request - jeśli odbiorca wykryje błąd w pakiecie, powiadomi o tym nadawcę, aby powtórzono transmisję błędnego pakietu.
- FEC - Forward Error Correction - jeśli odbiorca wykryje błąd w pakiecie, może spróbować go naprawić. Jeśli to się uda, nie trzeba ponawiać transmisji.

ARQ + FEC oznacza, że jeśli wykryto błąd w pakiecie, dekoder próbuje naprawić wiadomość, dopiero po stwierdzeniu że błąd jest nienaprawialny, wysyła prośbę do nadawcy o retransmisję. Przesyłanie danych techniką ARQ + FEC przebiega w taki sposób:

- Mamy dane do wysłania.
- Dzielimy dane na pakiety.
- Kodujemy pakiety.

- Przesyłamy zakodowane pakiety przez kanał transmisyjny, w którym mogą one zostać zakłócone.
- Dekodujemy odebrane pakiety.

Podczas dekodowania pakietu, mogą wystąpić 3 przypadki:

- Pakiet jest poprawny.
- Pakiet jest z błędami naprawialnymi.
- Pakiet jest z błędami nienaprawialnymi - musi być przesłany ponownie.

Symulację przesyłania danych zaimplementowaliśmy w poniższy sposób:

- Ustalamy parametry eksperymentu (dokładny opis w sekcji *Dane wejściowe eksperymentu*).
- Generujemy ciąg losowych bitów - przesyłana wiadomość.
- Dzielimy dane na pakiety (zakładamy że długość wiadomości jest zawsze podzielna przez długość pakietu).
- Kodujemy pakiety.
- Przesyłamy zakodowane pakiety przez kanał transmisyjny - każdy bit może zostać zakłócony z tym samym prawdopodobieństwem.
- Dekodujemy odebrane pakiety.

Ponieważ działamy w symulatorze, wiemy jakie dane zostały wysłane, przez co mamy 4 możliwości:

- Pakiet przesłany poprawnie.
- Pakiet przesłany z błędami naprawialnymi.
- Pakiet przesłany z błędami nienaprawialnymi - musi być przesłany ponownie.
- Pakiet przesłany z błędami niewykrytymi - odebrany pakiet jest poprawny według dekodera, ale różni się on od faktycznie wysłanego pakietu.

Kod symulujący przesyłanie danych znajduje się na listingu *1*. Metoda *sending_data* przyjmuje zadane przez nas parametry, wykonuje symulację przesyłu danych i zwraca słownik o kluczach 'Correct', 'Fixed', 'Repeat' i 'Wrong', gdzie każdemu kluczowi jest przypisana liczba odpowiednio: poprawnych pakietów, naprawionych pakietów, powtórzeń wysłania pakietów oraz pakietów różniących się od oryginalnie wysłanych.

Listing 1: Metoda sending_data odpowiadająca za symulację przesyłania danych

```
def sending_data(bits: list, block_size: int,
                 code_type: str, probability: float) -> dict:
    """Simulates sending data, returns a dictionary of the results."""
    separated_data = separate_data(bits, block_size)
    data_size = len(separated_data)
    encoded_data = []
    for block in separated_data:
        encoded_data.append(encode_data(block, code_type))
    sent_data = list.copy(encoded_data)
    for i in range(data_size):
        sent_data[i] = distort_bits(sent_data[i], probability)
    decoded_data = []
    data_results = {correct: 0, fixed: 0, repeat: 0, wrong: 0}
    for i in range(data_size):
        while True:
            decoded_data.append(decode_data(sent_data[i], code_type))
            if (decoded_data[i])[-1] == repeat_message:
                data_results[repeat] += 1
                decoded_data.pop()
                sent_data[i] = distort_bits(encoded_data[i], probability)
            else:
                break
    for i in range(data_size):
        is_fixed = False
        if decoded_data[i][-1] == fixed_message:
            is_fixed = True
            decoded_data[i].pop()
        if decoded_data[i] == separated_data[i]:
            if is_fixed:
                data_results[fixed] += 1
            else:
                data_results[correct] += 1
        else:
            data_results[wrong] += 1
    return data_results
```

1.2 Zastosowane algorytmy i ich działanie

Do kodowania i dekodowania wykorzystaliśmy 3 algorytmy - kod Hamminga, kod powtórzeniowy oraz kod CRC. Każdy z nich posiada własne metody operujące na ciągu danych przed wysłaniem oraz po odbiorze, co pozwoli porównać ich sprawność dla zmiennych parametrów transmisji.

1.2.1 Kod Hamminga z dodatkowym bitem parzystości (SECDED)

Podstawowy kod Hamminga[2] wykrywa i koryguje błędy polegające na przekłamaniu jednego bitu oraz wykrywa (ale nie koryguje) błędy podwójne. Kod ten wykorzystuje bity parzystości, znajdujące się na określonych z

góry w algorytmie pozycjach oraz sprawdzające konkretne bity pakietu (w tym pozostałe bity parzystości).
Pozycja 2^n: opuszcza $2^n - 1$ bitów, sprawdza 2^n bit/y, opuszcza 2^n bit/y, sprawdza 2^n bity, opuszcza 2^n bity itd.
Liczba bitów parzystości zależy od liczby bitów w pakiecie.
Dodanie dodatkowego bitu parzystości pozwala kodowi korygować błędy pojedyncze i w tym samym czasie wykrywać błędy podwójne. Bez korekcji kod może wykrywać błędy potrójne.

1.2.2 Kod powtórzeniowy - każdy bit jest powtórzony ’n’ razy

Najprostszy z wykorzystywanych kodów. Każdy bit w wysyłanym pakiecie jest powielany ‘n’ razy. Na przykład dla $n = 3$, pakiet ’*abc*’ zostałby wysłany jako ’*aaabbbccc*’. Odbiorca odczytując odebrany pakiet, patrzy na bloki ’n’ bitowe i stwierdza, która wartość występuje najczęściej i uznaje że to była zamierzona wartość. Na przykład, jeśli odbiorca odbierze ’111 011 010’, to dekodując taką wiadomość otrzyma ’1 1 0’. Dla naszego symulatora wybraliśmy $n = 4$, aby występował dodatkowy przypadek - tyle samo ’0’ i ’1’ w danym bloku, co sprawia że nie wiadomo jak zdekodować taki blok - trzeba ponowić transmisję całego pakietu.

1.2.3 Cykliczny kod nadmiarowy

Cykliczny kod nadmiarowy n-bitowy[6] jest resztą z dzielenia ciągu danych przez $n+1$-bitowy dzielnik CRC, nazywany wielomianem CRC. Reszta ta dopisywana jest do wysyłanego ciągu danych, umożliwiając sprawdzenie jego poprawności po transmisji. Po stronie odbiorczej wykonywane jest dzielenie otrzymanych bitów przez ten sam wielomian CRC - dla poprawnej transmisji reszta z tego dzielenia będzie równa 0.
Zaimplementowany algorytm pozwala również na korekcję błędów występujących w części kontrolnej (reszty CRC) kodu - sytuacja występuje, jeżeli po podzieleniu przesłanych danych uzyskamy "1" tylko na jednym bicie. Korzystając z tabel odległości Hamminga[8], wybraliśmy wielomian CRC 100101 pozwalający na poprawne stosowanie metody dla ciągu danych do 26 bitów (+5 bitów CRC). W przypadku więcej niż jednego błędu, czyli jedynki w reszcie dzielenia otrzymanych danych, wiadomość oznaczana jest jako błędnie przesłana.

2 Omówienie poznanych metod analizy właściwości modelu

2.1 Dane wejściowe eksperymentu

- Długość wiadomości.
- Długość jednego pakietu.
- Liczba pakietów = $\frac{\text{długość wiadomości}}{\text{długość pakietu}}$
- Prawdopodobieństwo przekłamania pojedynczego bitu.
- Algorytm kodowania (Hamminga, powtórzeniowy lub CRC).

2.2 Dane wyjściowe eksperymentu

Eksperyment generuje plik .csv o odpowiedniej nazwie, w którym każdy wiersz zawiera wyniki pojedynczej symulacji:

- liczbę poprawnych pakietów,
- liczbę naprawionych pakietów,
- liczbę powtórek przesłania pakietów,
- liczbę pakietów z błędami niewykrytymi.

2.3 Metody badania wyjścia systemu

Przykład kompletnej analizy znajduje się w sekcji *Długość pakietu*.

2.3.1 Moduł statistics[1]

Z modułu statistics wykorzystaliśmy funkcje:

- *mode* do obliczenia mody (najczęstszej wartości),
- *mean* do obliczenia średniej,
- *stdev* do obliczenia odchylenia standardowego,
- *quantiles* do wyznaczenia kwartyli.

Za pomocą powyższych wartości wyznaczyliśmy również przedział międzykwartylowy (IQR $= Q_3 - Q_1$) oraz skośność rozkładu Pearsona mody i mediany.[3] Na listingu *2* przedstawiono wykorzystanie modułu w naszym programie.

Listing 2: Obliczanie mody, średniej etc. za pomocą modułu statistics (desc - klucz słownika: correct, fixed, repeat, wrong)

```
mode = statistics.mode(results_list)
print(f"{desc}: Mode = {mode}")
average = statistics.mean(results_list)
print(f"{desc}: Average = {average}")
standard_deviation = statistics.stdev(results_list)
print(f"{desc}: Standard deviation = {standard_deviation}")
quartiles = statistics.quantiles(results_list, method='inclusive')
print(f"{desc}: Quartiles = {quartiles}")
iqr = quartiles[2] - quartiles[0]
if standard_deviation:
    skewness_mode = (average - mode) / standard_deviation
    skewness_median = 3 * (average - quartiles[1]) / standard_deviation
    print(f"{desc}: Pearson skewness (mode) = {skewness_mode}")
    print(f"{desc}: Pearson skewness (median) = {skewness_median}")
else:
    print("Skewness = 0")
```

2.3.2 Biblioteka Matplotlib[4]

Matplotlib posiada:

- funkcję *pyplot.boxplot* - tworzy wykres pudełkowy,
- funkcję *pyplot.hist* - tworzy histogram,
- i wiele innych funkcji dopasowujących wykresy do naszych potrzeb tj. nazwanie osi czy włączenie siatki.

Do stworzenia wykresu pudełkowego potrzebowaliśmy wyznaczonych wcześniej kwartyli oraz 'wąsów': min $= Q_1 - 1,5$ IQR i max $= Q_3 + 1,5$ IQR. Natomiast aby narysować histogram wystarczy przekazać funkcji listę wyników (i opcjonalnie liczbę przedziałów albo krańce przedziałów). Do wyznaczenia liczby przedziałów histogramu wymyśliliśmy taką zasadę: jeśli różnica między największą a najmniejszą wartością jest mniejsza od 20, to przedziałami są (min, min+1), (min+1, min+2), ... (max-1, max). Do stworzenia takich przedziałów wykorzystaliśmy bibliotekę NumPy[5] i funkcję *arange*. W przeciwnym wypadku, ograniczamy się do 20 przedziałów. Ta reguła pozwala jednocześnie na dokładność przy małej rozbieżności wyników i uniemożliwa stworzenie histogramu z np. tysiącem przedziałów. Fragment kodu odpowiadający za narysowanie tychże wykresów znajduje się na listingu *3*.

Listing 3: Rysowanie wykresu pudełkowego i histogramu
(desc - klucz słownika: correct, fixed, repeat, wrong)

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec

# [...]

fig = plt.figure()
grid = GridSpec(5, 1, figure=fig)
ax_box = fig.add_subplot(grid[0, 0])
ax_hist = fig.add_subplot(grid[1:, 0])
ax_box.axis('off')
ax_hist.grid()
ax_box.get_shared_x_axes().join(ax_box, ax_hist)
fig.suptitle(f"{desc}: transmissions")
ax_hist.set_xlabel(f"Number of packets sent: {desc}")
ax_hist.set_ylabel("Occurences")
q0 = quartiles[0] - 1.5 * iqr
q4 = quartiles[2] + 1.5 * iqr

ax_box.boxplot([q0,quartiles[0],quartiles[1],quartiles[2],q4],vert=False)
hist_bins = np.arange(min(results_list), max(results_list) + 1, 1)
if len(hist_bins) > 20:
    hist_bins = 20
counts, bins, bars = ax_hist.hist(results_list, bins=hist_bins)
```

2.3.3 Biblioteka SciPy[7]

Z biblioteki SciPy wykorzystaliśmy tylko jedną funkcję, *curve_fit*, do dopasowania krzywej Gaussa do narysowanego histogramu. Dane zwracane przez stworzenie histogramu to krańce przedziałów i liczba wystąpień dla każdego przedziału. Z krańców każdego przedziału liczymy średnią aby otrzymać jego 'środek'. Wtedy przekazujemy funkcji *curve_fit* jako 'x' i 'y' odpowiednio 'środki' przedziałów oraz liczbę wystąpień. Przekazujemy również wstępne przypuszczenie co do parametrów rozkładu Gaussa $A \cdot e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$.

- A = największa liczba wystąpień
- μ = mediana
- $\sigma = \frac{\text{IQR}}{1{,}349}$

Fragment kodu odpowiadający za wyznaczenie estymacji funkcji Gaussa oraz narysowanie jej na histogramie znajduje się na listingu *4*.

Listing 4: Estymacja funkcji Gaussa i narysowanie jej na histogramie (desc - klucz słownika: correct, fixed, repeat, wrong)

```
import scipy.optimize as opt

# [...]

def gauss_function(x, a, mu, sigma):
    """Gauss function used in curve fitting."""
    return a * math.e ** ((-1 / 2) * (((x - mu) / sigma) ** 2))

# [...]

counts, bins, bars = ax_hist.hist(results_list, bins=hist_bins)
x_data = []
for i in range(len(bins) - 1):
    x_data.append((bins[i] + bins[i + 1]) / 2)
y_data = counts
try:
    params, params_cov = opt.curve_fit(gauss_function, x_data, y_data,
                          p0=[max(y_data), quartiles[1], iqr / 1.349])
    print(f"{desc}: Gauss parameters (a, mu, sigma): {params}")
    ax_hist.plot(x_data, gauss_function(x_data,
                                        params[0],params[1],params[2]))
except Exception as e:
    print("Couldn't estimate Gauss function")
```

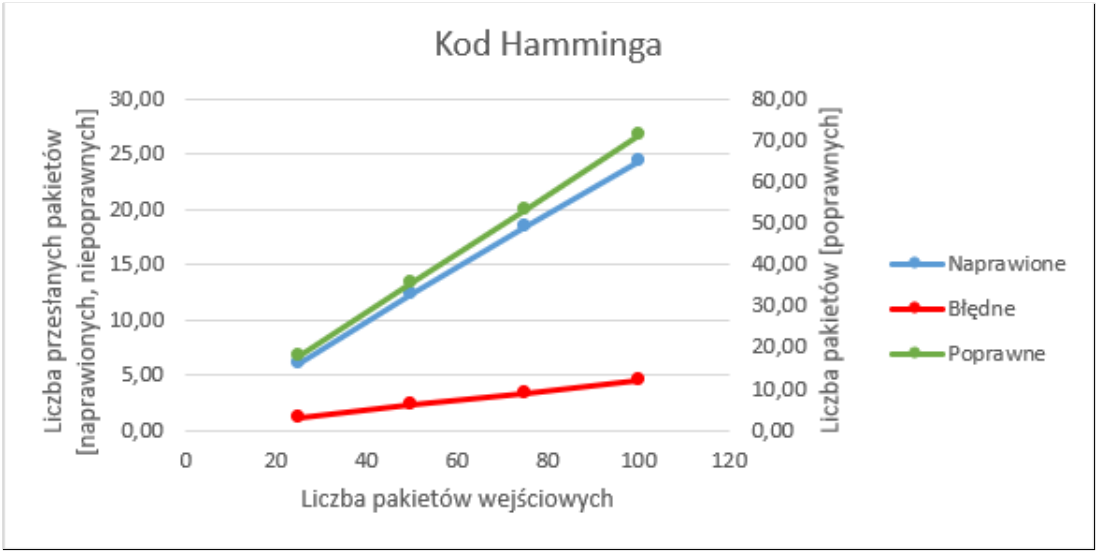
3 Wpływ parametrów na wyjście systemu

Eksperyment dla każdego zbioru danych wejściowych był przeprowadzony 10000 razy.

3.1 Liczba pakietów

Parametry symulacji:

- Liczba pakietów: 25, 50, 75, 100
- Długość pakietu: 5
- Prawdopodobieństwo przekłamania bitu: 5%



Rysunek 1: Zależność liczby pakietów poprawnych, naprawionych i niepoprawnych od liczby wysyłanych pakietów dla kodu Hamminga (2 osie)

Rysunek 2: Zależność liczby powtórzeń przesłań od liczby wysyłanych pakietów dla kodu Hamminga

Liczba pakietów nie wpływa na efektywność kodów, tzn. jeśli kod Hamminga dla pewnych parametrów i liczby pakietów przesyła średnio konkretną liczbę dobrych pakietów, to dla tych samych parametrów i dwa razy większej liczby pakietów prześle średnio dwa razy więcej dobrych pakietów. Analogiczna sytuacja występuje dla liczby ponownie przesłanych, błędnych oraz naprawionych pakietów.

W przypadku kodów powtórzeniowego oraz CRC zaobserwowano to samo co dla kodu Hamminga.

3.2 Długość pakietu

Parametry symulacji:

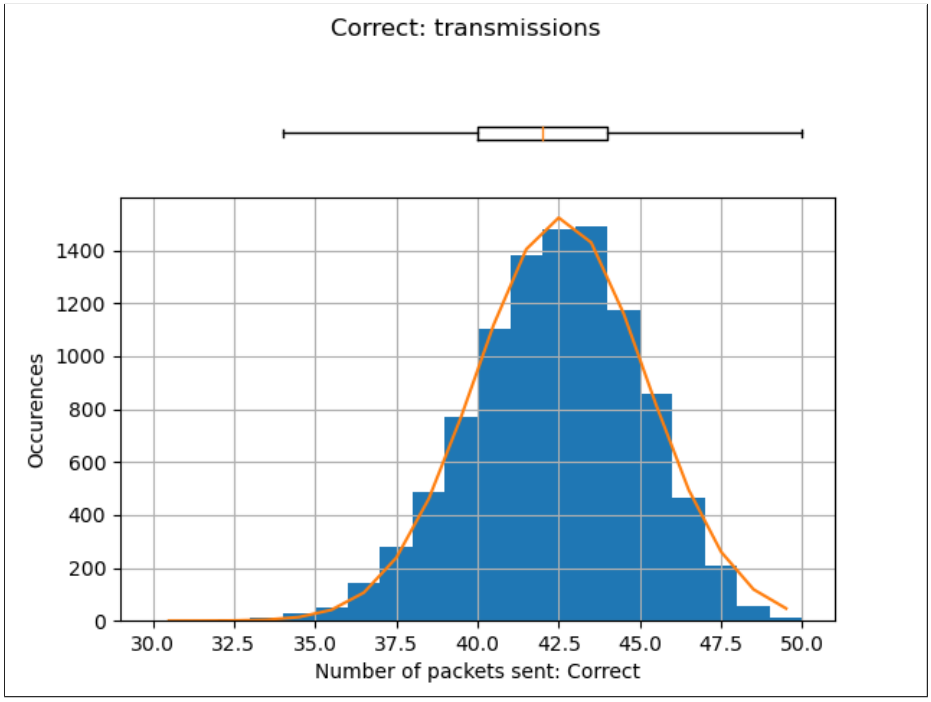
- Liczba pakietów: 50
- Długość pakietu: 5, 10, 20
- Prawdopodobieństwo przekłamania bitu: 2%

Kompletna analiza liczby poprawnych pakietów w zależności od długości pakietu dla kodu Hamminga:

3.2.1 Długość pakietu = 5

- Najczęstsza wartość = 43
- Średnia = 41,889
- Odchylenie standardowe = 2,609
- Kwartyle: $Q_1 = 40$, $Q_2 = 42$, $Q_3 = 44$
- Skośność Pearsona (moda) = -0,426
- Skośność Pearsona (mediana) = -0,128
- Rozkład lewostronnie skośny
- Estymacja funkcji Gaussa: $1523,5 \cdot e^{-\frac{1}{2}\left(\frac{x-42,5}{2,6}\right)^2}$

Na rysunku *3* przedstawiony jest wykres pudełkowy oraz histogram.

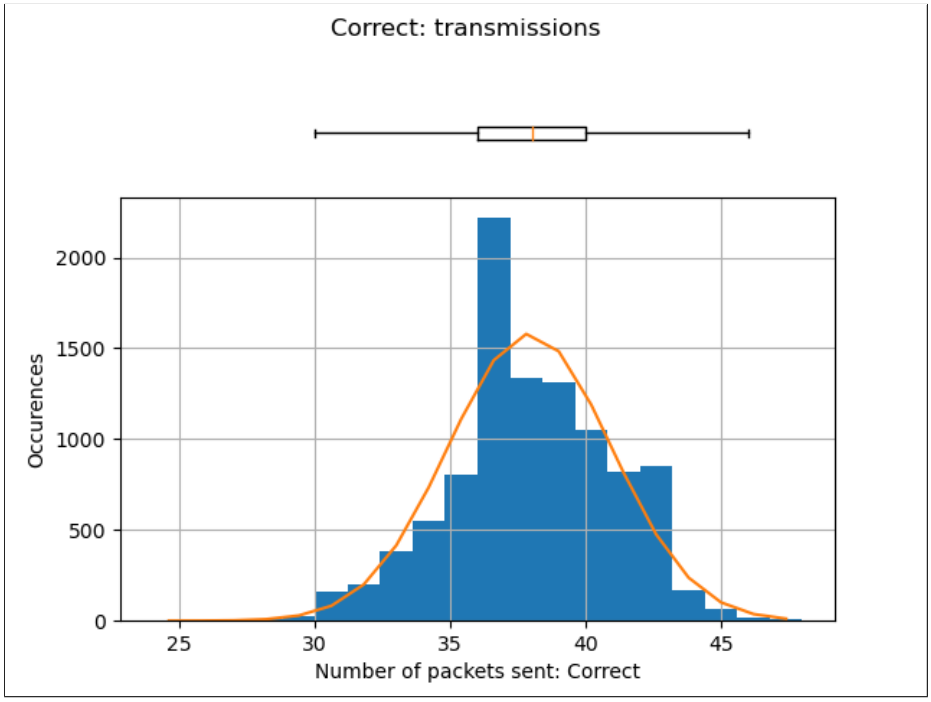


Rysunek 3: Wykres pudełkowy i histogram - liczba poprawnych pakietów dla kodu Hamminga i długości pakietu = 5

3.2.2 Długość pakietu = 10

- Najczęstsza wartość = 38
- Średnia = 37,872
- Odchylenie standardowe = 3,027
- Kwartyle: $Q_1 = 36$, $Q_2 = 38$, $Q_3 = 40$
- Skośność Pearsona (moda) = -0,042
- Skośność Pearsona (mediana) = -0,127
- Rozkład lewostronnie skośny
- Estymacja funkcji Gaussa: $1580,5 \cdot e^{-\frac{1}{2}\left(\frac{x-38}{3}\right)^2}$

Na rysunku *4* przedstawiony jest wykres pudełkowy oraz histogram.

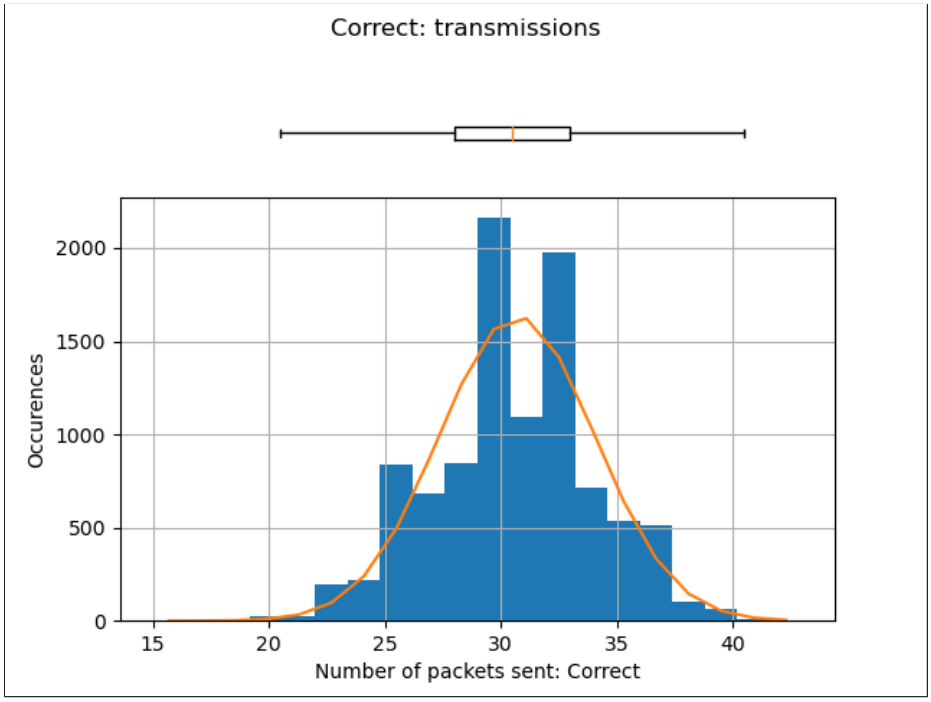


Rysunek 4: Wykres pudełkowy i histogram - liczba poprawnych pakietów dla kodu Hamminga i długości pakietu = 10

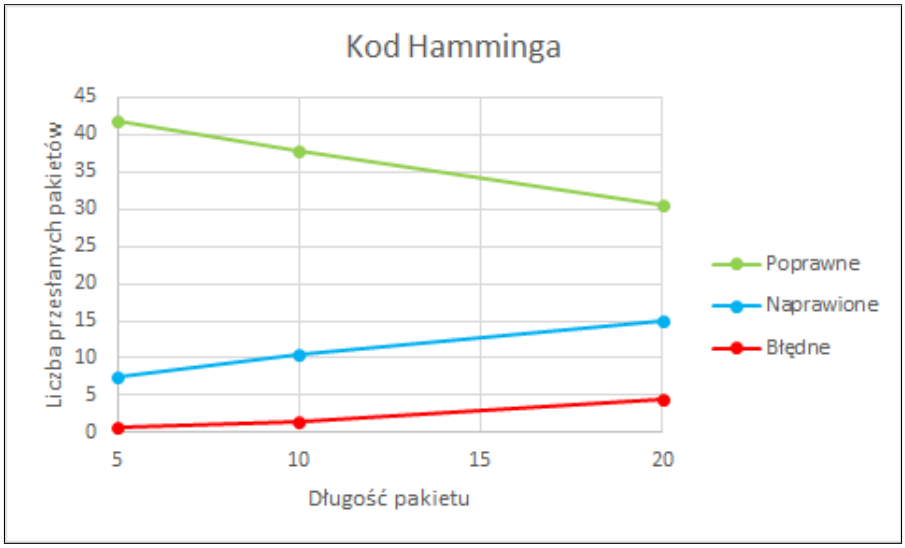
3.2.3 Długość pakietu = 20

- Najczęstsza wartość = 30
- Średnia = 30,452
- Odchylenie standardowe = 3,485
- Kwartyle: Q_1 = 28, Q_2 = 30,5, Q_3 = 33
- Skośność Pearsona (moda) = 0,130
- Skośność Pearsona (mediana) = -0,041
- Estymacja funkcji Gaussa: $1634 \cdot e^{-\frac{1}{2}\left(\frac{x-30,7}{3,4}\right)^2}$

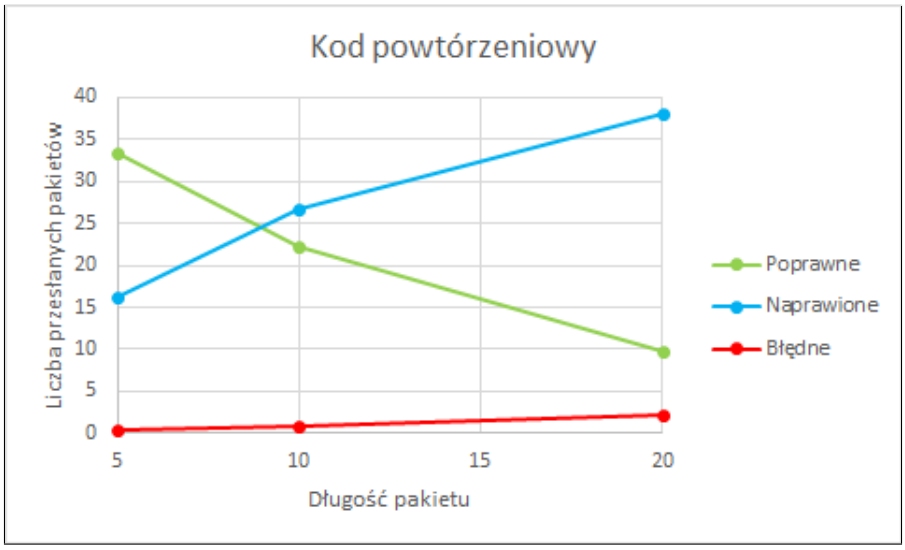
Na rysunku *5* przedstawiony jest wykres pudełkowy oraz histogram.



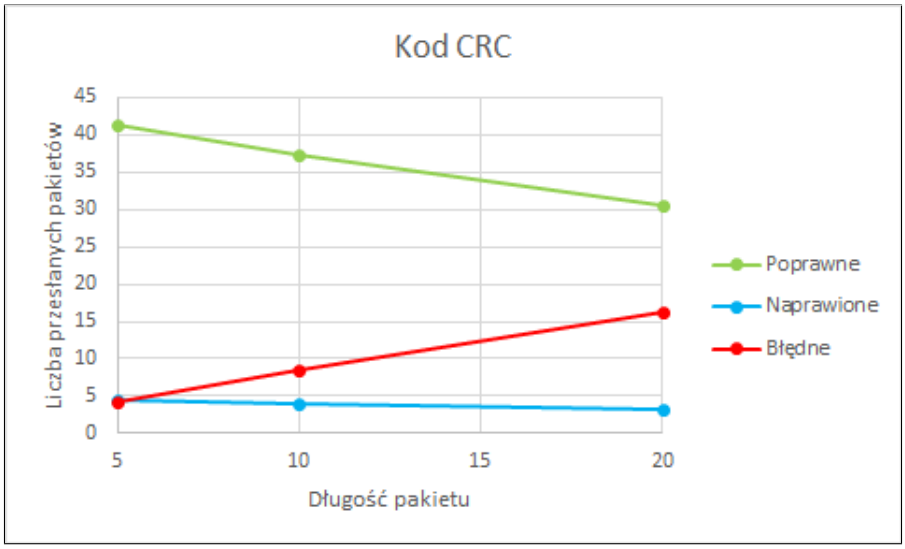
Rysunek 5: Wykres pudełkowy i histogram - liczba poprawnych pakietów dla kodu Hamminga i długości pakietu = 20



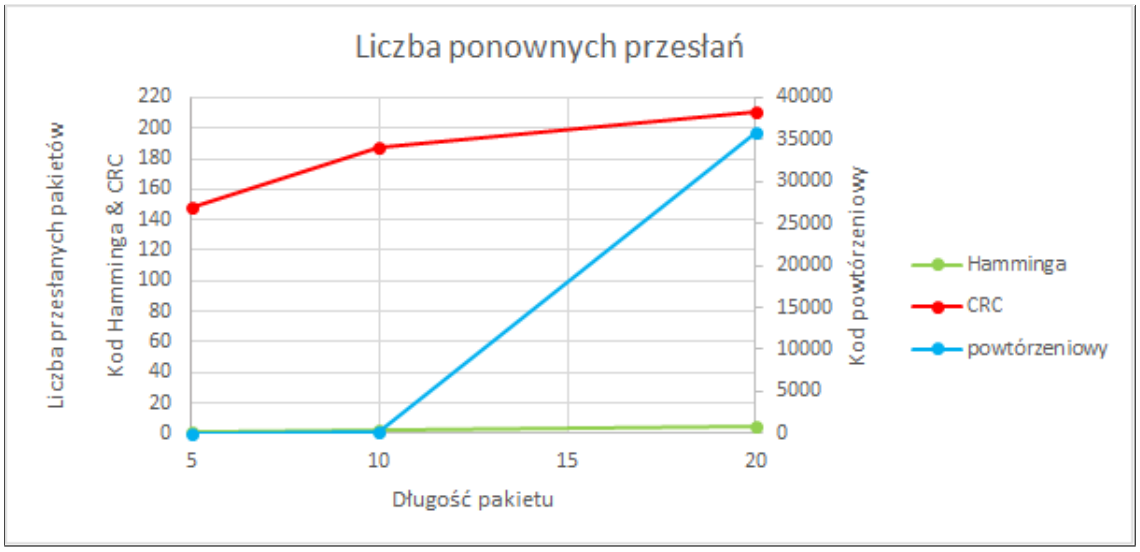
Rysunek 6: Zależność liczby poprawnych, naprawionych i niepoprawnych pakietów od długości pakietu dla kodu Hamminga



Rysunek 7: Zależność liczby poprawnych, naprawionych i niepoprawnych pakietów od długości pakietu dla kodu powtórzeniowego



Rysunek 8: Zależność liczby poprawnych, naprawionych i niepoprawnych pakietów od długości pakietu dla kodu CRC

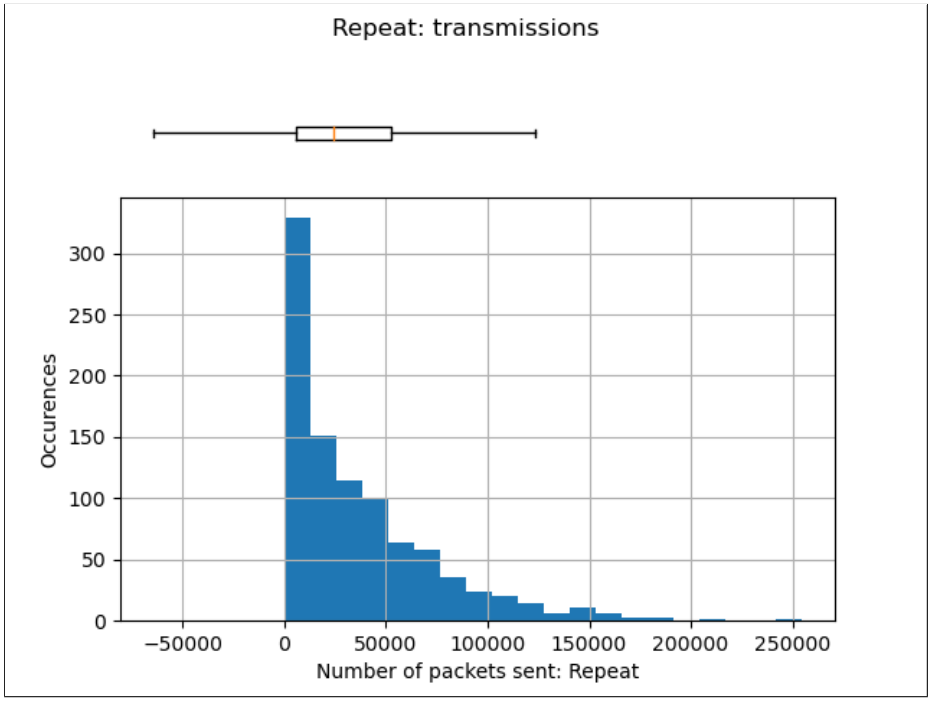


Rysunek 9: Zależność liczby ponownie przesłanych pakietów od długości pakietu dla testowanych kodów (2 osie)

Dla każdego kodu zwiększanie długości pakietu spowodowało zmniejszenie się liczby pakietów poprawnych oraz zwiększenie się liczby pakietów błędnych. W przypadku kodu CRC zwiększenie długości pakietów powodowało spadek skuteczności naprawy pakietów, dla pozostałych kodów tendencja była odwrotna.

Liczba ponownych przesłań pakietów dla każdego kodu rosła razem z ich długością, jednak można zauważyć dużą rozbieżność pomiędzy tymi liczbami. Kod Hamminga najrzadziej wymagał ponownego przesłania pakietu (co nie sprawiło, że liczba błędnych pakietów była większa niż dla innych kodów). Kod powtórzeniowy natomiast, mimo że dla małych długości pakietu nie odstawał od pozostałych kodów, to przy większych długościach dla niektórych wiadomości liczba powtórzeń sięgała setek tysięcy.

Przyczyną tak dużej liczby jest fakt, że dla pakietu o długości np. 20, przez kanał transmisyjny przesyłane jest 80 bitów. Wystarczy, że w jednym z dwudziestu 4-bitowych bloków zostaną zakłamane 2 bity, aby zaszła potrzeba przesłania **całego** pakietu od nowa. Sprawiło to że musieliśmy przeprowadzić mniejszą liczbę testów (ok. 1000) ponieważ testy trwały by zdecydowanie za długo. Jednakże, najczęściej wiadomość była przesyłana bez powtórzeń. Jest możliwe że wpływ na to miał generator liczb pseudolosowych z modułu ’random’ w Pythonie. Na rysunku *10* przedstawiono histogram powtórzeń pakietów dla kodu powtórzeniowego i długości pakietu = 20 (brak nałożonej funkcji Gaussa, ponieważ bibliotece SciPy nie udało się z dostarczonych danych wyestymować tejże funkcji).

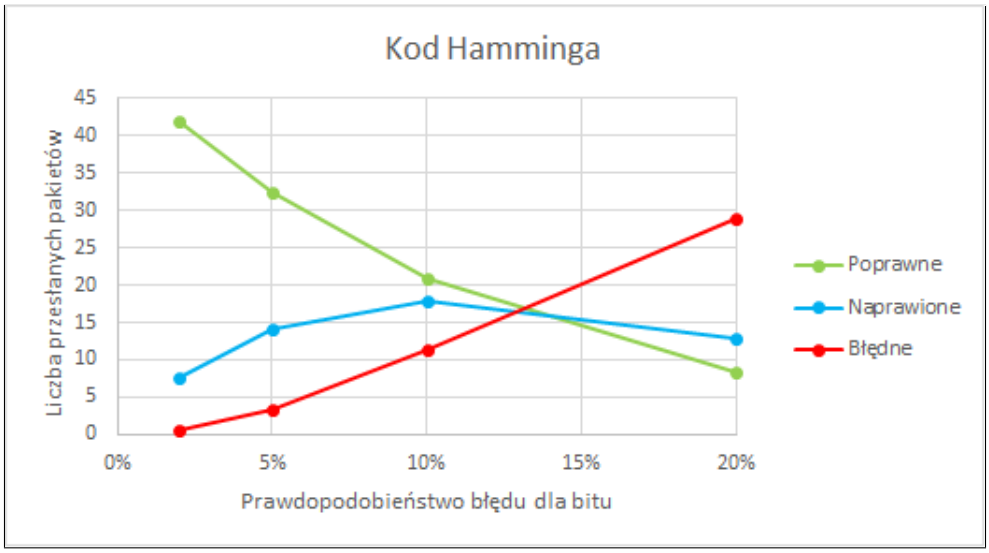


Rysunek 10: Wykres pudełkowy i histogram - liczba powtórzeń pakietów dla kodu powtórzeniowego i długości pakietu = 20

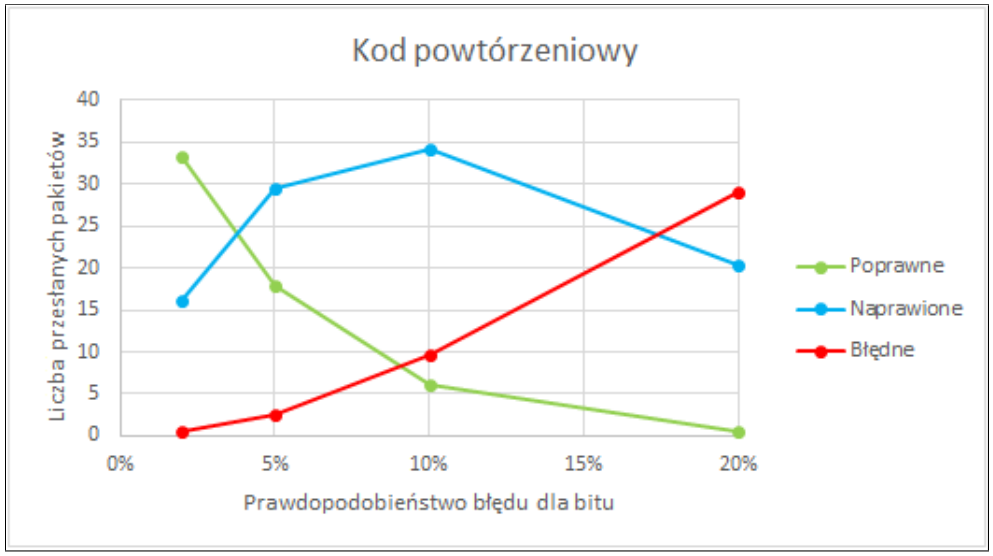
3.3 Prawdopodobieństwo przekłamania bitu

Parametry symulacji:

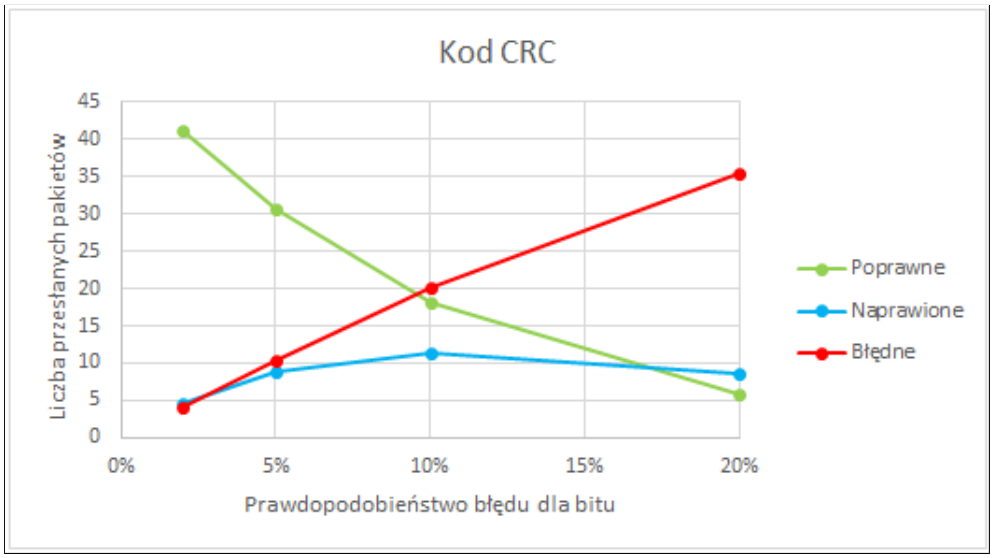
- Liczba pakietów: 50
- Długość pakietu: 5
- Prawdopodobieństwo przekłamania bitu: 2%, 5%, 10%, 20%



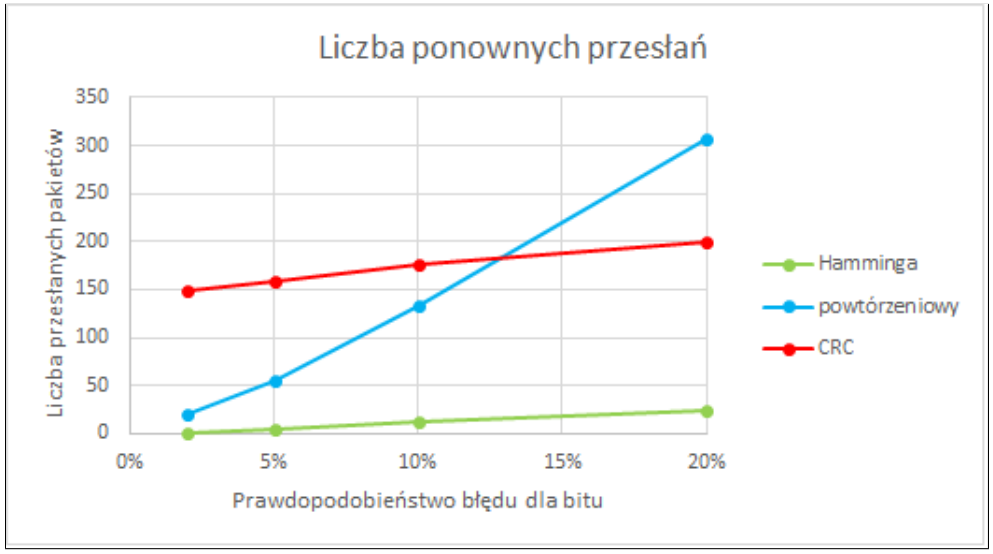
Rysunek 11: Zależność liczby poprawnych, naprawionych i niepoprawnych pakietów od prawdopodobieństwa przekłamania bitu dla kodu Hamminga



Rysunek 12: Zależność liczby poprawnych, naprawionych i niepoprawnych pakietów od prawdopodobieństwa przekłamania bitu dla kodu powtórzeniowego



Rysunek 13: Zależność liczby poprawnych, naprawionych i niepoprawnych pakietów od prawdopodobieństwa przekłamania bitu dla kodu CRC



Rysunek 14: Zależność liczby ponownie przesłanych pakietów od prawdopodobieństwa przekłamania bitu dla badanych kodów

Dla wszystkich kodów wraz ze wzrostem prawdopodobieństwa przekłamania bitu rośnie liczba wiadomości błędnie przesłanych, a spada przesłanych poprawnie. Również we wszystkich algorytmach widoczny jest początkowy wzrost liczebności pakietów naprawionych, która osiąga maksimum przy prawdopodobieństwie około 10%, przy dalszym zwiększaniu prawdopodobieństwa "zaszumienia" ich liczba maleje. Powodem jest zwiększanie się liczby pakietów przesyłanych z błędami, których rosnąca ilość doprowadza do błędów niewykrywalnych dla algorytmów dekodujących.

Liczba powtarzanych pakietów również rośnie w przybliżeniu liniowo dla każdego z kodów. Największy jej wzrost widoczny jest przy zastosowaniu kodu powtórzeniowego. Dla algorytmów kodujących CRC oraz Hamminga liczba powtórzeń rośnie w podobnym tempie, aczkolwiek jest zauważalnie wyższa dla pierwszego z nich.

Kod powtórzeniowy oraz Hamminga, dla największego prawdopodobieństwa, są podobne pod względem liczby pakietów odebranych bez błędu. Osiągają tę wartość jednak w odmienne sposoby - kod powtórzeniowy praktycznie wszystkie wartości poprawne otrzymuje poprzez naprawę, w kodzie Hamminga liczba pakietów poprawnych oraz naprawionych jest znacznie bardziej zbliżona. CRC, choć przesyła więcej poprawnych pakietów niż powtórzeniowy, to ze wzrostem szansy przekłamania bitu wykrywa znacznie mniej błędów, co skutkuje największą ze wszystkich trzech kodów ilością odebranych błędnych danych.

Łącząc liczbę niewykrytych błędów oraz potrzebne retransmisje można zauważyć, że najsprawniejszy jest algorytm Hamminga, zapewniający największą poprawność oraz prędkość (ze względu na liczbę powtórzeń) transmisji, na całej rozpiętości sprawdzanej przez nas szansy błędnego przesłania bitu.

Literatura

[1] https://docs.python.org/3/library/statistics.html#module-statistics.

[2] https://en.wikipedia.org/wiki/Hamming_code.

[3] https://en.wikipedia.org/wiki/Skewness#Other_measures_of_skewness.

[4] https://matplotlib.org/.

[5] https://numpy.org/.

[6] https://pl.wikipedia.org/wiki/Cykliczny_kod_nadmiarowy.

[7] https://scipy.org/.

[8] http://users.ece.cmu.edu/~koopman/crc/index.html.