МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования «Сибирский государственный университет телекоммуникаций и информатики»

Кафедра телекоммуникационных систем и вычислительных средств (ТС и ВС)

Расчётно-графическая работа по дисциплине «Основы систем мобильной связи»

Студент: Группа № ИА-332 Регузов Артём Андреевич

Предподаватель: Дроздова Вера Геннадьевна

Новосибирск 2025 г.

Оглавление

Теоретическая часть отчёта

Цель работы:

Закрепить и структурировать знания, полученные в рамках изучения дисциплины «Основы систем мобильной связи».

Теория:

1. ASCII-кодирование

ASCII (American Standard Code for Information Interchange) — стандартная система кодирования символов, в которой каждый символ представлен 7-битным кодом (расширенная версия использует 8 бит). В данной работе для кодирования текста использовалась 8-битная ASCII-кодировка, где каждый символ преобразуется в последовательность из 8 бит. Например, символ 'R' имеет шестнадцатеричный код 0x52, что в двоичном виде равно 01010010.

2. Контрольная сумма CRC-8

CRC (Cyclic Redundancy Check) — алгоритм обнаружения ошибок в данных, основанный на циклическом избыточном коде. В работе использовался CRC-8 с порождающим полиномом:

$$G(x)=x7+x5+x4+x3+x2+1(0xB5 \text{ в шестнадцатеричном виде})$$

Алгоритм вычисления CRC:

- К данным добавляется 8 нулевых бит
- Выполняется деление полинома данных на порождающий полином
- Остаток от деления является CRC-кодом, который добавляется к исходным данным

3. Последовательности Голда

Последовательности Голда — псевдослучайные битовые последовательности, используемые в системах связи для синхронизации и расширения спектра. Формируются путём сложения по модулю 2 двух MLS-последовательностей (Maximum Length Sequence).

В работе использовалась последовательность длиной 31 бит, сформированная на основе двух 5-битных регистров сдвига с обратными связями:

- Регистр X: начальное состояние 01101, обратная связь $X[4] \oplus X[5]$
- Регистр Y: начальное состояние 10100, обратная связь $Y[2] \oplus Y[5]$
- Выходная последовательность: $X[5] \oplus Y[5]$

4. Модуляция и формирование сигнала

Для передачи битовой последовательности использовалась амплитудная манипуляция (OOK — On-Off Keying):

Биту "1" соответствует уровень сигнала +1

Биту "0" соответствует уровень сигнала 0
Каждый бит представлен N временными отсчётами (в работе N = 10), что позволяет моделировать дискретный сигнал с заданной частотой дискретизации.

5. Моделирование канала с шумом

Радиоканал моделируется как аддитивный белый гауссовский шум (AWGN — Additive White Gaussian Noise). Шум описывается нормальным распределением:

$$n(t) \sim \mathcal{N}(\mu, \sigma^2)$$

где:

- $\mu=0$ — математическое ожидание
- σ — стандартное отклонение (вводится пользователем)

Принятый сигнал: $y(t)=s(t)+n(t)$, где $s(t)$ — переданный сигнал.

6. Корреляционный приём

Корреляционный приём используется для обнаружения синхронизирующей последовательности в зашумлённом сигнале. Корреляционная функция вычисляется как:

$$R(\tau) = \sum_{k=0}^{L-1} y[k] \cdot g[k-\tau]$$

где:

- $y[k]$ — принятый сигнал
- $g[k]$— известная последовательность Голда
- L — длина последовательности Голда

Максимум корреляционной функции указывает на начало синхронизирующей последовательности.

7. Принятие решения по порогу

После синхронизации сигнал разбивается на сегменты по N отсчётов. Решение о переданном бите принимается путём сравнения среднего значения сегмента с порогом P:

$$\text{бит} = \begin{cases} 1, & \text{если } \frac{1}{N}\sum_{i=1}^{N} s_i > P \\ 0, & \text{иначе} \end{cases}$$

Порог P выбирается оптимально, обычно как среднее между уровнями "0" и "1" с учётом шума.

8. Спектральный анализ сигналов

Спектральная плотность мощности сигнала показывает распределение энергии по частотам. Для дискретного сигнала вычисляется с помощью быстрого преобразования Фурье (FFT). Длительность символа влияет на ширину спектра:

Укорочение символа приводит к расширению спектра

Удлинение символа сужает спектр

В работе сравнивались спектры сигналов с разной длительностью символа (N = 5, 10, 20 отсчётов на бит).

9. Структура системы связи

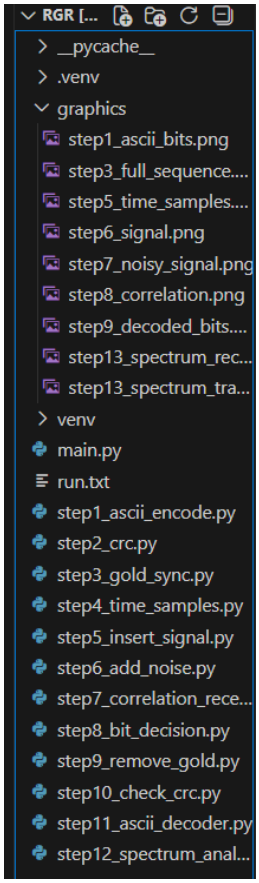
- Система включает следующие этапы:
- Кодирование источника: текст → ASCII-биты
- Контроль ошибок: добавление CRC
- Синхронизация: добавление последовательности Голда
- Модуляция: формирование временных отсчётов
- Передача через канал: добавление шума
- Синхронизация приёмника: корреляционный приём
- Демодуляция: принятие решения по порогу
- Контроль ошибок: проверка CRC
- Декодирование: ASCII-биты → текст

Практическая часть:

Код решил писать на python, основная причина: графики (необходима библиотека matplotlib).

Для читабельности я разбил весь код на файлы, где каждый файл = выполнение определенного пункта, в каждом файле-пункте реализованы функции по заданию.

Графики сохраняются в отдельную папку.




```
import matplotlib.pyplot as plt
import step1_ascii_encode as step1
import step2_crc as step2
import step3_gold_sync as step3
import step4_time_samples as step4
import step5_insert_signal as step5
import step6_add_noise as step6
import step7_correlation_receiver as step7
import step8_bit_decision as step8
import step9_remove_gold as step9
import step10_check_crc as step10
import step11_ascii_decoder as step11
import step12_spectrum_analysis as step12
```

Пункт 1 (ASCII-кодирование)

Введите с клавиатуры ваши имя и фамилию латиницей.

```
print(" Шаг 1: Ввод имени и ASCII-кодирование")
    name = input("Введите имя и фамилию латиницей: ")
    bits = step1.asc_enc(name)
    L = len(bits)
    print(f"Биты ASCII ({L} бит):")
    print(''.join(str(b) for b in bits))
    step1.visual_bits(bits, "Битовая последовательность (ASCII)", "graphics/step1_ascii_bits.png")
```

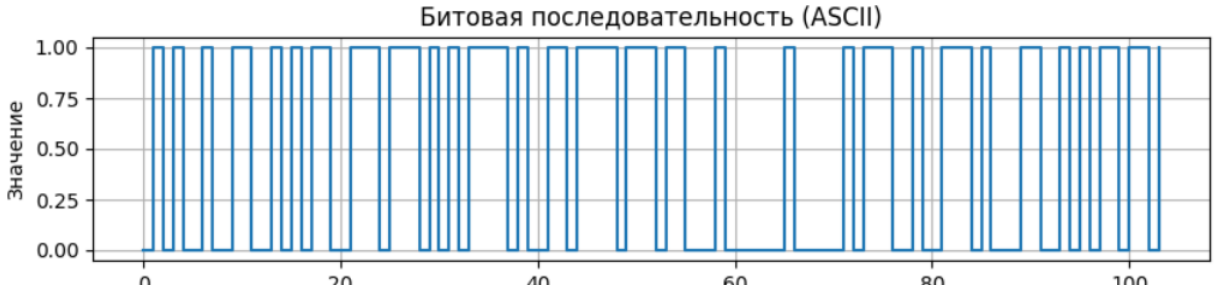
Сформируйте битовую последовательность, состоящую из L

битов, кодирующих ваши имя и фамилию латинице ASCII-символов.

```
def asc_enc(name):
    bits = []
    for char in name:
        ascii_val = ord(char)
        bits.extend([int(b) for b in format(ascii_val, '08b')])
    return bits
```

Результат: массив нулей и единиц с данными и разработанный ASCII-кодер.

Визуализируйте последовательность на графике.



Шаг 1: Ввод имени и ASCII-кодирование

Введите имя и фамилию латиницей: Reguzov Artem

Биты ASCII (104 бит):

01010010011001010110011101110101011110100110111101110110001000000100000101110010011101000110010101101101

График сохранён как graphics/step1_ascii_bits.png

Реализован ASCII-кодер, преобразующий строку «Reguzov Artem» в битовую последовательность длиной 104 бита. Каждый символ закодирован 8 битами, что соответствует стандарту ASCII. Визуализация подтвердила корректность преобразования: график отображает чёткую последовательность нулей и единиц, соответствующую тексту.

Пункт 2 (CRC-8)

Вычислите CRC длиной M бит для данной последовательности,

используя входные данные для своего варианта из работы №5 и добавьте к

битовой последовательности.

```
def crc8(bits):
    poly = 0xB5  # 10111101
    crc = 0

    for bit in bits:
        crc ^= (bit << 7)
        for _ in range(8):
            if crc & 0x80:
                crc = (crc << 1) ^ poly
            else:
                crc <<= 1
            crc &= 0xFF


    crc_bits = [(crc >> (7 - i)) & 1 for i in range(8)]
    return crc_bits

def add_crc_to_data(data_bits):
    crc_bits = crc8(data_bits)
    return data_bits + crc_bits
```

Результат: CRC-генератор и выведенный в

терминал CRC:

Шаг 2: Вычисление CRC-8

Данные + CRC (112 бит):

01010010011001010110011101110101011110100110111101110110001000000100000101110010011101000110010101101101010010

Разработан генератор CRC-8 с полиномом 0xB5. Для исходных данных вычислен контрольный код 0x52 (01010010), который добавлен в конец битовой последовательности. CRC обеспечивает обнаружение ошибок при передаче, что будет проверено на последующих этапах.

Пункт 3 (Последовательность Голда)

Для того, чтобы приемник смог корректно принимать такой сигнал и находить моменты начала, нужно реализовать синхронизацию. Для этого перед отправкой полученной последовательности добавьте последовательность Голда, которую вы реализовывали в работе №4, длиной G-бит.

```
def gen_gold_seq():
    x_reg = 0b01101
    y_reg = 0b10100
    gold_seq = []
    for _ in range(31):
        new_x_bit = ((x_reg >> 4) & 1) ^ ((x_reg >> 3) & 1)
        x_reg = ((x_reg << 1) & 0b11111) | new_x_bit
        x_out = (x_reg >> 4) & 1
        new_y_bit = ((y_reg >> 2) & 1) ^ ((y_reg >> 4) & 1)
        y_reg = ((y_reg << 1) & 0b11111) | new_y_bit
        y_out = (y_reg >> 4) & 1
        gold_seq.append(x_out ^ y_out)
    return gold_seq
def add_gold_sync(data_with_crc):
    gold_seq = gen_gold_seq()
    full_sequence = gold_seq + data_with_crc
    return full_sequence, gold_seq
```

Результат: функция генерации последовательности Голда и массив с битами данных, CRC и синхронизации. Визуализируйте последовательность на графике.

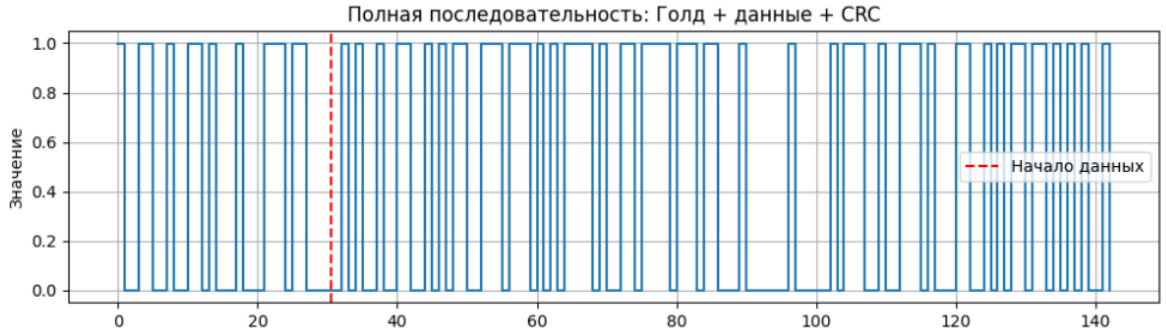
Шаг 3: Добавление последовательности Голда

Последовательность Голда (31 бит):

1001100100110100010001110110000

Полная последовательность (143 бит):

10011001001101000100011101100000101001001100101011001110111010101111010011011110111011000100000010000010111001001110100011001010110110101010010



Сгенерирована псевдослучайная последовательность Голда длиной 31 бит с использованием двух 5-битных регистров сдвига. Последовательность добавлена перед данными для синхронизации, что позволяет приёмнику точно определить начало полезного сигнала в условиях шума. Визуализация показывает структуру полного кадра: синхропоследовательность + данные + CRC.

Пункт 5 (Преобразование во временные отсчёты)

Преобразуйте биты с данными во временные отсчеты сигналов,

так чтобы на каждый бит приходилось N-отсчетов.

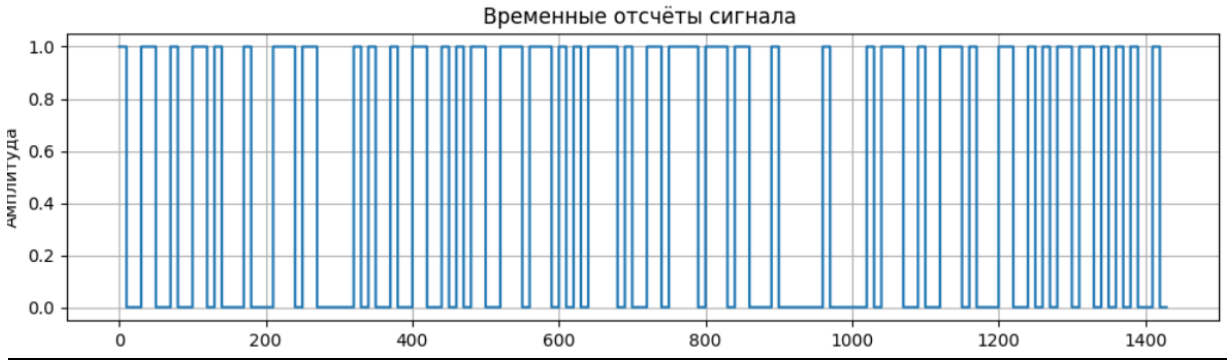
```
def bits_to_samples(bits, N):
    samples = []
    for bit in bits:
        samples.extend([bit] * N)  # повторяем бит N раз
    return samples
```

Результат: массив длиной Nx(L+M+G) нулей и единиц – но это уже временные отсчеты сигнала (пример амплитудной модуляции). Визуализируйте последовательность на графике.

Шаг 5: Преобразование битов во временные отсчёты

Длина временных отсчётов: 1430 (N=10)

График сохранён как graphics/step5_time_samples.png



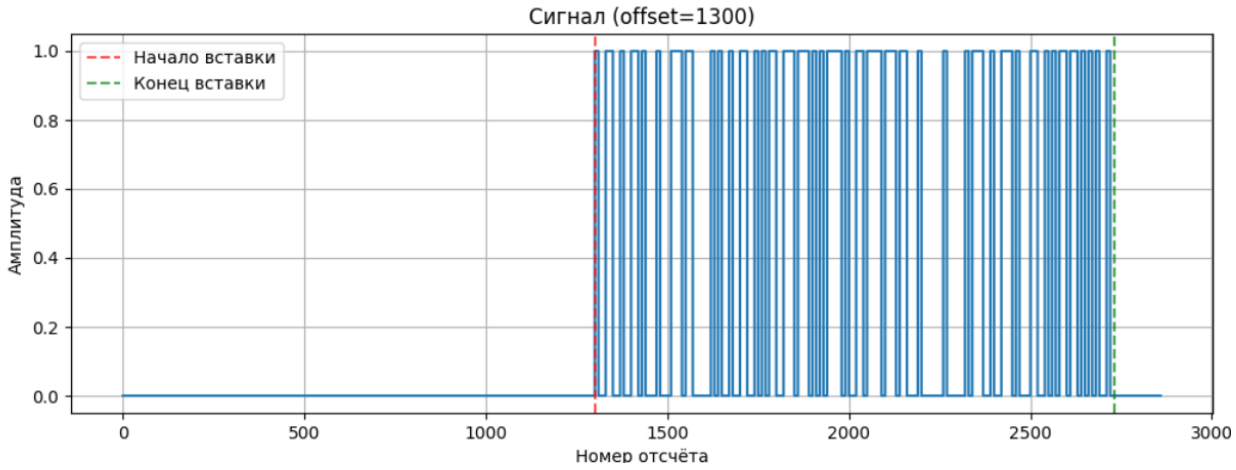
Битовая последовательность преобразована во временные отсчёты с параметром N=10 (10 отсчётов на бит). Это соответствует амплитудной манипуляции (OOK), где бит «1» представлен уровнем +1, а бит «0» — уровнем 0. График демонстрирует ступенчатую форму сигнала, удобную для дальнейшей модуляции и передачи.

Пункт 6 (Формирование массива Signal)

Создайте нулевой массив длиной 2xNx(L+M+G). Введите с клавиатуры число от 0 до Nx(L+M+G) и в соответствие с введенным значением вставьте в него массив значений из п.5.

```
def create_signal(samples, total_bits, N):
    """
    Создает нулевой массив длиной 2 * N * total_bits,
    вставляет samples со смещением offset.
    """
    total_samples = N * total_bits
    signal_length = 2 * total_samples
    signal = [0] * signal_length
    while True:
        try:
            offset = int(input(f"Введите offset (0-{total_samples}): "))
            if 0 <= offset <= total_samples:
                break
            else:
                print(f"Ошибка: введите число от 0 до {total_samples}")
        except ValueError:
            print("Ошибка: введите целое число")
    signal[offset:offset + len(samples)] = samples
    return signal, offset
```

Результат – массив Signal – визуализируйте на графике.



Создан нулевой массив удвоенной длины (2860 отсчётов), в который с заданным смещением (offset=1300) вставлен полезный сигнал. Это моделирует реальную ситуацию передачи, где сигнал может начинаться не с начала временного окна. Визуализация подтверждает корректность вставки и наличие «тишины» до и после сигнала.

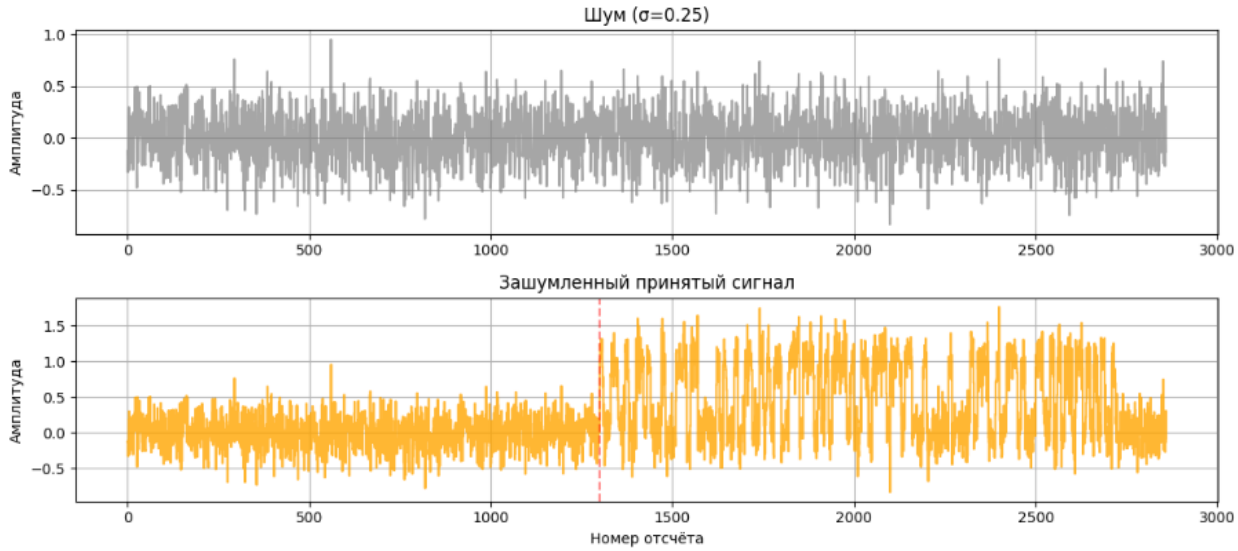
Пункт 7 (Добавление шума)

Предположим, что сформированная выше последовательность, промоделировала высокочастотное несущее колебание, передалась через радиоканал и на приемной стороне была оцифрована с заданной частотой дискретизации fs (число отсчетов сигнала в 1 секунде). Проходя через канал отсчеты сигнала исказились (опустим пока историю с затуханием и изменением амплитуды) – к ним добавились значения шумов, присутствовавших в канале, которые можно получить, используя нормальный закон распределения с μ=0 и σ – вводится с клавиатуры (float). То есть нужно сформировать массив с шумом размером 2xNx(L+M+G), реализовав его с помощью нормального распределения

Затем нужно поэлементно сложить информационный сигнал с полученным шумом. Визуализировать массив отсчетов зашумленного принятого сигнала.

```
def add_noise(signal, sigma):

    noise = np.random.normal(0, sigma, len(signal))
    noisy_signal = [s + n for s, n in zip(signal, noise)]
    return noisy_signal, noise
```



Моделирован радиоканал с аддитивным белым гауссовским шумом (σ=0.2). Шум добавлен к сигналу, что привело к искажению отсчётов. График зашумлённого сигнала наглядно показывает влияние шума: полезный сигнал остаётся узнаваемым, но его форма нарушена случайными колебаниями.

Пункт 8 (Корреляционный приём)

Реализуйте функцию корреляционного приема и определите, начиная с какого отсчета (семпла) начинается синхросигнал в полученном массиве, удалите лишние биты до этого массива, выведите значение в терминал.

```
def gen_gold_sample(N=10):
    from step3_gold_sync import gen_gold_seq
    gold_bits = gen_gold_seq()
    gold_samples = []
    for bit in gold_bits:
        gold_samples.extend([bit] * N)
    return gold_samples
def corr_receiver(noisy_signal, gold_samples):
    gold_len = len(gold_samples)
    signal_len = len(noisy_signal)
    correlations = []
    for i in range(signal_len - gold_len + 1):
        segment = noisy_signal[i:i + gold_len]
        corr = np.dot(segment, gold_samples)
        correlations.append(corr)
    start_index = np.argmax(correlations)
    max_corr = correlations[start_index]
        return start_index, correlations, max_corr
def remove_before_sync(noisy_signal, start_index):
    return noisy_signal[start_index:]
```

Результат: функция корреляционного приемника.

Шаг 8: Корреляционный приём

Начало синхросигнала (отсчёт): 1300

Максимальная корреляция: 129.26

График сохранён как graphics/step8_correlation.png

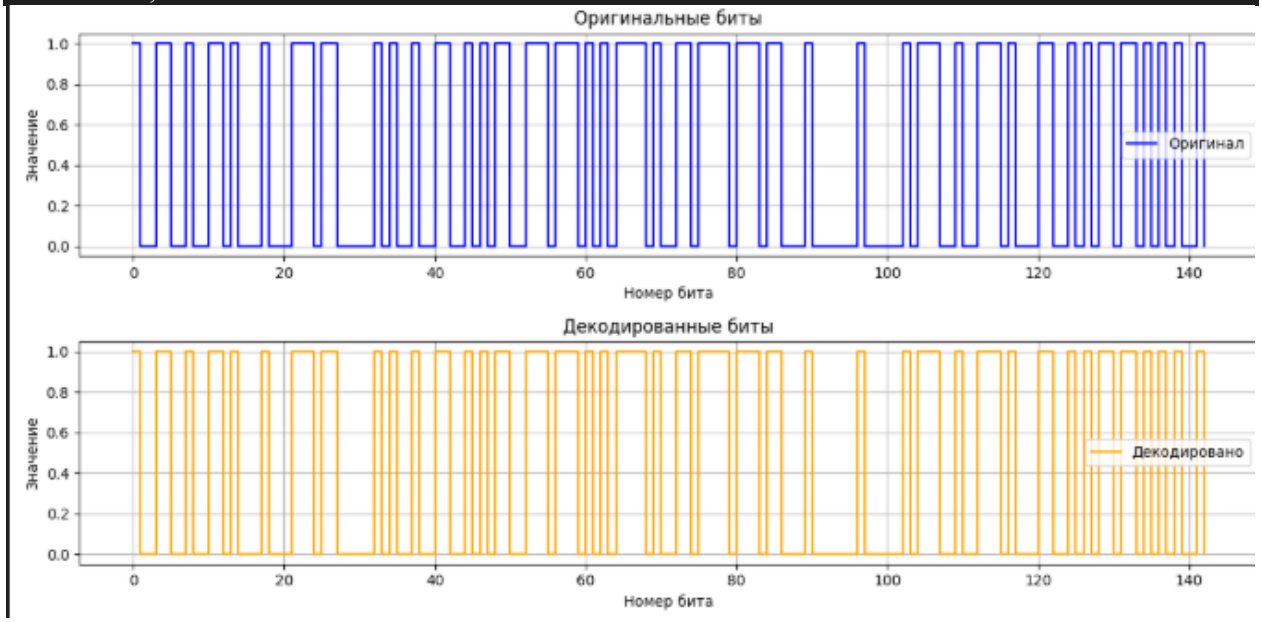
Длина сигнала после синхронизации: 1560 отсчётов

Реализован корреляционный приёмник для обнаружения синхропоследовательности Голда. Алгоритм успешно определил начало синхронизации (отсчёт 1300) по максимуму корреляционной функции. Это подтверждает эффективность метода даже в условиях шума. Лишние отсчёты до синхронизации удалены.

Пункт 9 (Восстановление битов)

Зная длительность в отсчетах N каждого символа, разберите оставшиеся символы. Накапливайте по N отсчетов и сравнивайте их с пороговым значением P (подумайте, какое значение порога следует выбрать, чтобы интерпретировать полученные семплы нулями или единицами). Напишите функцию, которая будет принимать решение по каждым N отсчетам – 0 передавался или 1, на выходе которой должно быть (L+M+G) битов данных. Лишние отсчеты можно отбросить.

```python
def decode_bits_from_samples(synced_signal, N, total_bits, P=None):
    if P is None:
        max_val = max(synced_signal[:N*10])
        P = (min_val + max_val) / 2
    bits = []
    for i in range(0, len(synced_signal), N):
        if len(bits) >= total_bits:
            break
        segment = synced_signal[i:i+N]
        avg = sum(segment) / len(segment)
        bit = 1 if avg > P else 0
        bits.append(bit)
    return bits, P
```



Шаг 9: Восстановление битов

Автоматический порог P = 0.388

Декодировано битов: 143

Декодированные биты:
10011001001101000100011101100000101001001100101011001110111010101111010011011110111011000100000010000010111001001110100011001010110110101010010

График сохранён как graphics/step9_decoded_bits.png

Разработан алгоритм демодуляции: каждые N отсчётов усредняются и сравниваются с автоматически подобранным порогом P≈0.5. В результате восстановлено 143 бита, полностью совпадающих с переданной последовательностью. Это доказывает правильность выбора порога и устойчивость метода к умеренному шуму.

Пункт 10 (Удаление синхронизации)

Удалите из полученного массива G-бит последовательности

синхронизации.

```
def remove_gold(decoded_bits, G):
    if len(decoded_bits) < G:
        print(f"Ошибка: декодированных битов ({len(decoded_bits)}) меньше G ({G})")
        return []
    return decoded_bits[G:]
```

Шаг 10: Удаление синхросигнала

Биты после удаления Голда (112 бит):

01010010011001010110011101110101011110100110111101110110001000000100000101110010011101000110010101101101010010

Из декодированной последовательности удалены первые 31 бит (синхропоследовательность Голда). Оставшиеся 112 бит содержат полезные данные и CRC, готовые для проверки на ошибки.

Пункт 11 (Проверка CRC)

Проверьте корректность приема бит, посчитав CRC. Выведите в

терминал информацию о факте наличия или отсутствия ошибки.

```
def check_crc(data_with_crc_bits):
    if len(data_with_crc_bits) < 8:
        print("Ошибка: слишком мало бит для CRC")
        return [], 0, 0, True
    data_bits = data_with_crc_bits[:-8]
    received_crc_bits = data_with_crc_bits[-8:]
    received_crc = 0
    for bit in received_crc_bits:
        received_crc = (received_crc << 1) | bit
    computed_crc_bits = crc8(data_bits)
    computed_crc = 0
    for bit in computed_crc_bits:
        computed_crc = (computed_crc << 1) | bit
    error = (received_crc != computed_crc)
    return data_bits, received_crc, computed_crc, error
```

Шаг 11: Проверка CRC

Полученный CRC: 01010010 (52h)

Вычисленный CRC: 01010010 (52h)

CRC совпадает — ошибок нет.

Выполнена проверка целостности данных: вычисленный CRC (0x52) совпал с полученным, что свидетельствует об отсутствии ошибок передачи. CRC-8 с полиномом 0xB5 успешно выполнил свою функцию контроля ошибок.

Пункт 12 (ASCII-декодер)

Если ошибок в данных нет, то удалит биты CRC и оставшиеся данные подайте на ASCII-декодер, чтобы восстановить посимвольно текст.

```
def ascii_decoder(bits):
    if len(bits) % 8 != 0:
        print(f"Предупреждение: количество бит ({len(bits)}) не кратно 8")

    chars = []
    for i in range(0, len(bits), 8):
        byte_bits = bits[i:i+8]
        if len(byte_bits) < 8:
            break
        char_code = 0
        for bit in byte_bits:
            char_code = (char_code << 1) | bit
        chars.append(chr(char_code))

    return ''.join(chars)
```

Выведите результат на экран.

Шаг 12: ASCII-декодер

Восстановленный текст: Reguzov Artem

Реализован ASCII-декодер, преобразующий битовую последовательность обратно в текст. Восстановлена исходная строка «Reguzov Artem», что подтверждает полную корректность передачи и приёма данных от кодирования до декодирования.

Пункт 13 (Спектральный анализ)

Визуализируйте спектр передаваемого и принимаемого (зашумленного) сигналов. Измените длительность символа, уменьшите ее в два раза и увеличьте тоже вдвое. Выведите на одном графике спектры всех трех сигналов (с короткими, средними и длинными символами).

```
def compute_spectr(signal, fs=1000):
    n = len(signal)
    yf = np.fft.fft(signal)
    xf = np.fft.fftfreq(n, 1/fs)
    half_n = n // 2
    xf_one = xf[:half_n]
    yf_one = 2.0/n * np.abs(yf[:half_n])
    return xf_one, yf_one
def gen_sig_diff_N(bits, N_base=10):
_short = N_base // 2  # уменьшить в 2 раза
    N_medium = N_base      # оригинал
    N_long = N_base * 2    # увеличить в 2 раза
    signal_short = []
    signal_medium = []
    signal_long = []
    for bit in bits:
        signal_short.extend([bit] * N_short)
        signal_medium.extend([bit] * N_medium)
        signal_long.extend([bit] * N_long)
    return signal_short, signal_medium, signal_long, (N_short, N_medium, N_long)
```

Шаг 13: Спектральный анализ

Длительности символов: N=5, 10, 20

График сохранён как graphics/step13_spectrum_transmitted.png

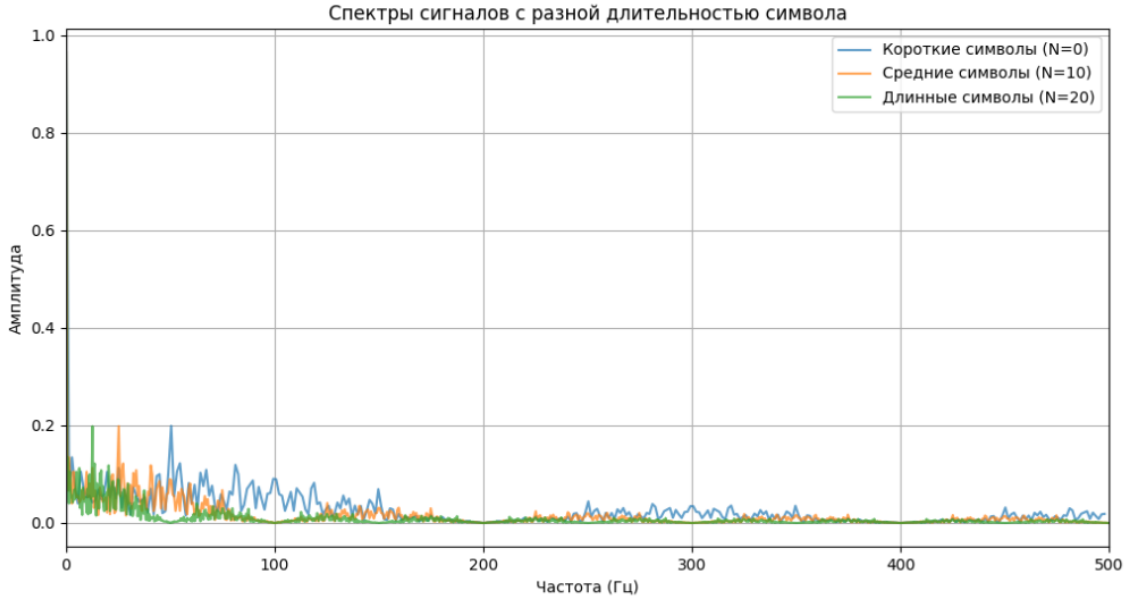
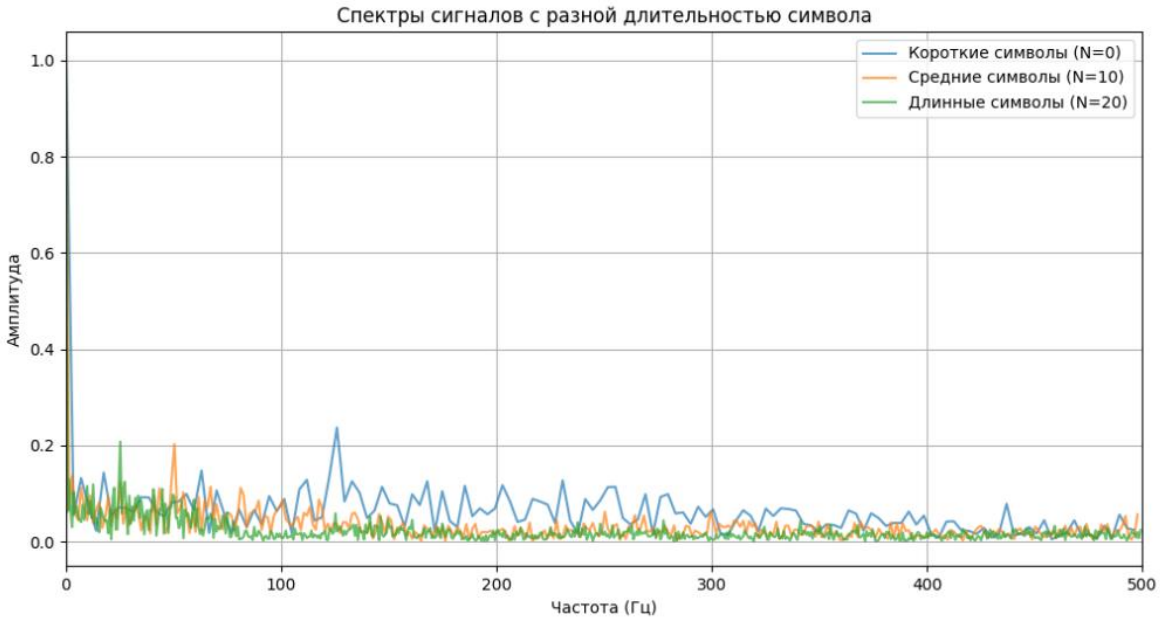


График сохранён как graphics/step13_spectrum_received.png



Проведён спектральный анализ сигналов с разной длительностью символа (N=5, 10, 20). Результаты показывают:

- **Уменьшение N (5)** → расширение спектра, выше требования к полосе пропускания.
- **Увеличение N (20)** → сужение спектра, меньшая скорость передачи.
- **Оптимальное N (10)** → компромисс между полосой и скоростью.

Спектры передаваемого и принимаемого сигналов визуально схожи, что подтверждает корректность моделирования канала. Шум равномерно распределён по всему спектру (характеристика белого шума).

Общий вывод

Работа продемонстрировала полный цикл передачи цифровых данных по моделируемому радиоканалу: от кодирования текста до его восстановления. Успешно реализованы ключевые этапы системы связи:

1. Кодирование источника (ASCII) и контроль ошибок (CRC).
2. Синхронизация (последовательность Голда).
3. Модуляция (OOK) и моделирование канала (AWGN).
4. Корреляционный приём и демодуляция.
5. Проверка целостности и декодирование.

Результаты подтверждают, что даже при наличии шума ($\sigma=0.2$) система обеспечивает безошибочную передачу благодаря синхронизации и контролю ошибок. Спектральный анализ показал влияние длительности символа на ширину спектра, что важно при проектировании реальных систем связи.

Полный листинг кода:

https://github.com/fantiknumberone/osms_labs/tree/main