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

Расчетно-графическая работа

по дисциплине «Основы систем мобильной связи»

«Реализация приема и передачи битовой последовательности в условиях помех»

Выполнил:

Шаповал Н.О.

Группа: ИА-232

Проверил: Дроздова В.Г.

GitHub: https://github.com/nikin1/osms

Новосибирск 2024

Содержание

Цель работы

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

Задачи работы:

- Сформировать битовую последовательность, состоящую из L битов, кодирующих имя и фамилию на латинице ASCII-символов;
- Вычислить CRC длиной M бит для данной последовательности;
- Добавить последовательность Голда длиной G-бит в начало битовой последовательности;
- Преобразовать биты с данными во временные отсчеты сигналов, так чтобы на каждый бит приходилось N-отсчетов;
- Создать нулевой массив длиной 2xNx(L+M+G) и в соответствии с введенным значением с клавиатуры вставить в него массив данных;
- Сформировать массив с шумом размером 2xNx(L+M+G), реализовав его с помощью нормального распределения;
- Поэлементно сложить информационный сигнал с полученным шумом;
- Реализовать функцию корреляционного приема и определить, начиная с какого отсчета (семпла) начинается синхросигнал в полученном массиве;
- Удалить все лишние биты до синхросигнала;
- Написать функцию, которая будет принимать решение по каждым N отсчетам – 0 передавался или 1, на выходе которой должно быть (L+M+G) битов данных;
- Удалить из полученного массива G-бит последовательности синхронизации;
- Проверить корректность приема бит, посчитав CRC;
- При отсутствии ошибок удалить биты CRC и оставшиеся данные подать на ASCII-декодер, чтобы восстановить посимвольно текст;
- Визуализировать спектр передаваемого и принимаемого (зашумленного) сигналов для разной длины символа;

Краткие теоретические сведения

Корреляция – это статистическая зависимость двух и более случайных величин. Корреляционная взаимосвязь в случае с сетями мобильной связи и используемыми в них радиосигналами позволяет обнаруживать сигналы синхронизации для того, чтобы с их помощью корректно разбивать ось времени на интервалы, предусматриваемые стандартами связи (например, слоты, кадры и пр.).

Корреляция бывает положительная, когда два процесса напрямую зависят друг от друга, то есть увеличение одной величины вызывает пропорциональный рост другой и наоборот. Например, можно проследить рост объемов продаж мороженного при повышении суточной температуры. Отрицательная корреляция свидетельствует об обратной взаимосвязи процессов – рост суточной температуры приводит к снижению объема продаж пуховиков. Бывает также нейтральная корреляция, когда явная взаимосвязь между процессами отсутствует (например, связь курса доллара и среднего балла за ЕГЭ у выпускников неочевидна).

Существуют различные подходы к измерению корреляции. Рассмотрим один из вариантов оценить ее значение (3.1)-(3.2):

$$Corr_{x,y} = \sum_{n=-\infty}^{\infty} x_n y_n \qquad (3.1)$$

или

$$Corr_{x,y} = \sum_{n=0}^{N-1} x_n y_n \qquad (3.2)$$

Для того, чтобы корректно определять корреляцию между функциями/процессами «энергия», которых столь различна, используется нормализованная функция корреляции (3.3).

$$Corr_{x,y} = \frac{\sum_{n=0}^{N-1} x_n y_n}{\sqrt{\sum_{n=0}^{N-1} x_n^2 \sum_{n=0}^{N-1} y_n^2}} \qquad (3.3)$$

Рассчитав нормализованную корреляцию для x и y, модно получить значение, равное 0.95, а для y и z - 0.38. Диапазон возможных значений для нормализованной корреляции от -1 до 1, где 1 и -1 – это максимальные значения положительной и отрицательной корреляции, 0 и близкие к нему значения – означает отсутствие корреляции.

Псевдослучайные двоичные последовательности

Псевдослучайные двоичные последовательности (PN-sequences – PseudoNoise) – это частный случай псевдослучайных последовательностей, элементами которой являются только 2 возможных значения (1 и 0 или -1 и +1). Такие последовательности очень часто используются в сетях мобильной связи. Возможные области применения:

- оценка вероятности битовой ошибки (BER – Bit Error Rate). В этом случае передатчик передает приемнику заранее известную PNпоследовательность бит, а приемник анализируя значения бит на конкретных позициях, вычисляет количество искаженных бит и вероятность битовой ошибки в текущих радиоусловиях, что затем может быть использовано для работы алгоритмов, обеспечивающих помехозащищенность системы;
- временная синхронизация между приемником и передатчиком.

Включаясь, абонентский терминал начинает записывать сигнал, дискретизируя его с требуемой частотой, в результате чего формируется массив временных отсчетов и требуется понять, начиная с какого элемента в этом массиве собственно содержатся какие-либо данные, как именно структурирована ось времени, где начинаются временные слоты. Используя заранее известную синхронизирующую PN-последовательность (синхросигнал), приемник сравнивает полученный сигнал с этой последовательностью на предмет «сходства» - корреляции. И если фиксируется корреляционный пик, то на стороне приема можно корректно разметить буфер с отсчетами на символы, слоты, кадры и пр.

- расширение спектра. Используется для повышения эффективности передачи информации с помощью модулированных сигналов через канал с сильными линейными искажениями (замираниями), делая систему устойчивой к узкополосным помехам (например, в 3G WCDMA).

Псевдослучайная битовая последовательность должна обладать следующими свойствами, чтобы казаться почти случайной:

1. Сбалансированность (balance), то есть число единиц и число нулей на любом интервале последовательности должно отличаться не более чем на одну.
2. Цикличность. Циклом в данном случае является последовательность бит с одинаковыми значениями. В каждом фрагменте псевдослучайной

битовой последовательности примерно половину составляли циклы длиной 1, одну четверть – длиной 2, одну восьмую – длиной 3 и т.д.

3. Корреляция. Корреляция оригинальной битовой последовательности с ее сдвинутой копией должна быть минимальной. Автокорреляция этих последовательностей – это практически дельта-функция во временной области, как для аддитивного белого гауссовский шума AWGN (Additive white Gaussian noise), а в частотной области – это константа.

Псевдослучайные двоичные последовательности

CRC — циклический избыточный код, иногда называемый также контрольным кодом или контрольной суммой. CRC – это добавочная порция избыточных бит, вычисляемых по заранее известному алгоритму на основе исходного передаваемого пакета данных (информационной битовой последовательности), которое передаѐтся вместе с самим пакетом по каналам связи (добавляется после информационных битов) и служит для контроля его безошибочной передачи.

Простыми словами, CRC – это остаток от двоичного деления оригинального пакета с данными на какое-то двоичное n-разрядное число (порождающий полином), и его длина будет равна n-1 бит. Рассмотрим пример, где имеется 7 бит данных: 100100 и 4-битный порождающий полином 1101. Требуется определить CRC. Для того, чтобы выполнить деление этих битовых последовательностей нужно в конце последовательности с данными добавить n-1 нулей, как показано ниже, где n=4, для нашего случая.

Делитель - 1 1 0 1 | 1 0 0 1 0 0 0 0 0 - Делимое (данные+n-1 нулей).

Основной операцией, используемой при делении бинарных чисел, является исключающее ИЛИ (XOR). Ниже показана таблица истинности для данной операции.

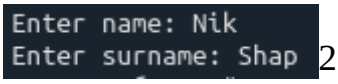
x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Этапы выполнения работы

1. Первым делом требуется реализовать ввод с клавиатуры имени и фамилии на латинице, чтобы в дальнейшем преобразовать их в битовую последовательность.

```
name = input("Введите имя на латинице: ")
surname = input("Введите фамилию на латинице: ")
```

В результате на экране появляется возможность ввода необходимых данных:



2. Теперь надо сформировать битовую последовательность, состоящую из L битов, кодирующих имя и фамилию на латинице ASCII-символов. Для этого сначала происходит объединение строк с именем и фамилией с добавлением пробела между ними, после чего новая строка подаётся в функцию разработанного ASCII-кодера:

```
name_surname = name + " " + surname
bit_sequence = ascii_coder(name_surname)
```

Функция ASCII-кодера:

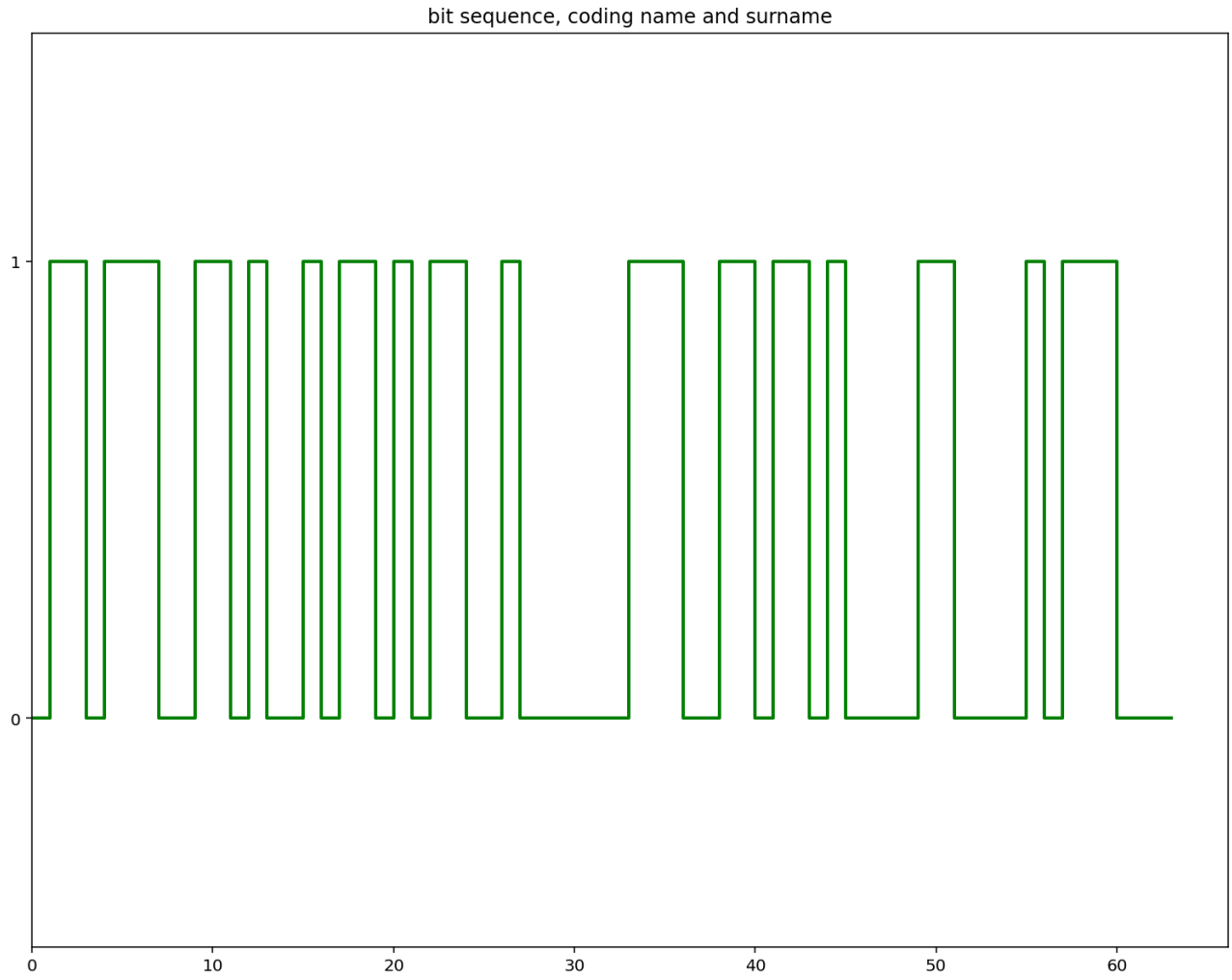
```
def ascii_coder(text):
    bit_sequence = []
    for symbol in text:
        ascii_symbol = ord(symbol)
        bits_symbol = bin(ascii_symbol)[2:].zfill(8)
        bit_sequence.extend(int(bit) for bit in bits_symbol)
    return bit_sequence
```

Каждый символ из строки сначала преобразовывается в ASCII-код, после чего этот код переводится из десятичной системы в двоичную систему. В итоге каждый символ состоит из 8 разрядов. В конце получается битовая последовательность, кодирующая имя и фамилию.

Визуализируем данную последовательность на графике:

```
plt.figure(figsize=(13, 10))
plt.step(range(len(bit_sequence)), bit_sequence, where='post', color='b', linewidth=2)
plt.yticks([0, 1], ['0', '1'])
```

```
plt.xlim(0)
plt.ylim(-0.5, 1.5)
plt.title('Битовая последовательность, кодирующая имя и фамилию')
```



3. Вычислим CRC длиной M бит для данной последовательности, используя входные данные варианта из работы №5, и выведем полученный результат в терминал:

```
G = [1, 1, 0, 1, 1, 1, 1, 0]
CRC = computeCRC(bit_sequence, G)
CRC_print = ''.join(map(str, CRC))
print(f"CRC для битовой последовательности с данными: {CRC_print}")
```

Функция computeCRC для вычисления CRC:

```
def computeCRC(packet, polynomial):
    packet_zeros = packet[:] + [0] * (len(polynomial) - 1)
    for i in range(len(packet)):
        if packet_zeros[i] == 1:
            for j in range(len(polynomial)):
                packet_zeros[i + j] ^= polynomial[j]

    CRC = packet_zeros[-(len(polynomial) - 1):]
    return CRC
```

Получаем результат:

```
CRC для битовой последовательности с данными: 1001010
```

Добавим полученное значение CRC в конец битовой последовательности:

```
bit_sequence_crc = bit_sequence + CRC
```

4. Для того чтобы приѐмник смог корректно принять наш сигнал, нужно реализовать синхронизацию. Для этого нужно добавить последовательность Голда из работы №4 в начало битовой последовательности:

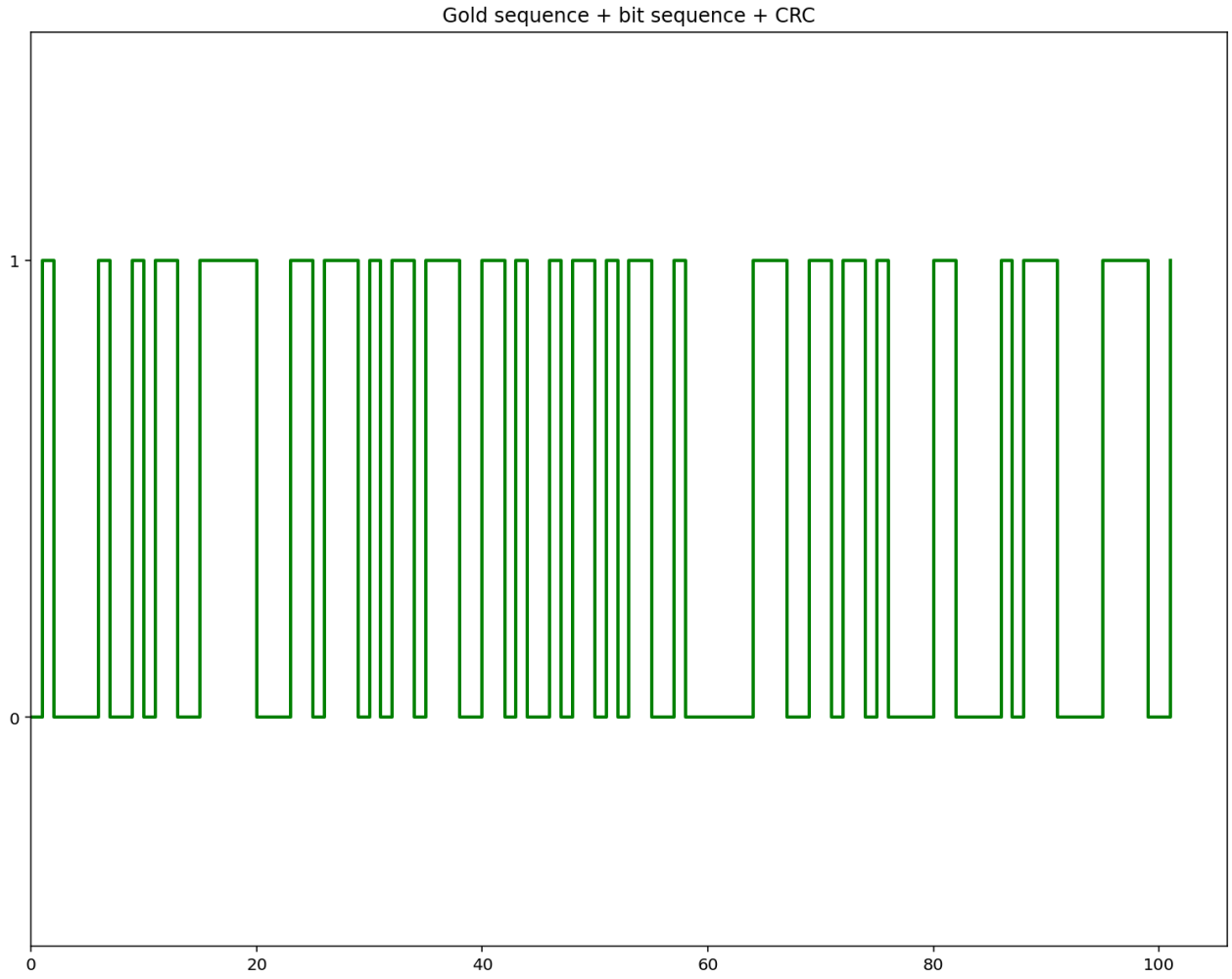
```
x = [0, 1, 1, 0, 0]
y = [1, 0, 0, 1, 1]
len_sequence = 31

gold_sequence = create_gold_sequence(x, y, len_sequence)
bit_sequence_crc_gold = gold_sequence + bit_sequence_crc
```

Функция create_gold_sequence для генерации последовательности Голда:

```
def create_gold_sequence(x, y, len_sequence):
    gold_sequence = []

    for i in range(len_sequence):
        xor_shift_x = x[2] ^ x[4]
        xor_shift_y = y[2] ^ y[4]

        gold_sequence.append(x[-1] ^ y[-1])

        x.pop()
        y.pop()

        x.insert(0, xor_shift_x)
        y.insert(0, xor_shift_y)

    return gold_sequence
```

Визуализируем нашу битовую последовательность вместе с синхронизацией и CRC на графике:

```
plt.figure(figsize=(13, 10))
plt.step(range(len(bit_sequence_crc_gold)), bit_sequence_crc_gold, where='post', color='b',
linewidth=2)
plt.yticks([0, 1], ['0', '1'])
plt.xlim(0)
plt.ylim(-0.5, 1.5)
plt.title('Последовательность Голда + биты данных + CRC')
```



4.1. Для того чтобы приѐмник мог успешно определить где находится конец наших передаваемых данных, в конец битовой последовательности было добавлено стоп-слово, которое будет заранее известно на приѐмной стороне.

```
stop_word = [0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0]
bit_sequence_crc_gold_stop = bit_sequence_crc_gold + stop_word
```

5. Теперь преобразуем биты с данными во временные отсчеты сигналов, так чтобы на каждый бит приходилось N-отсчетов. Как результат, должен получиться массив длиной Nx(L+M+G) нулей и единиц и это уже будут временные отсчеты сигнала. В качестве N возьмѐм значение в 10 отсчѐтов:

```
N = 10
signal_samples = bits_to_samples(bit_sequence_crc_gold_stop, N)
```

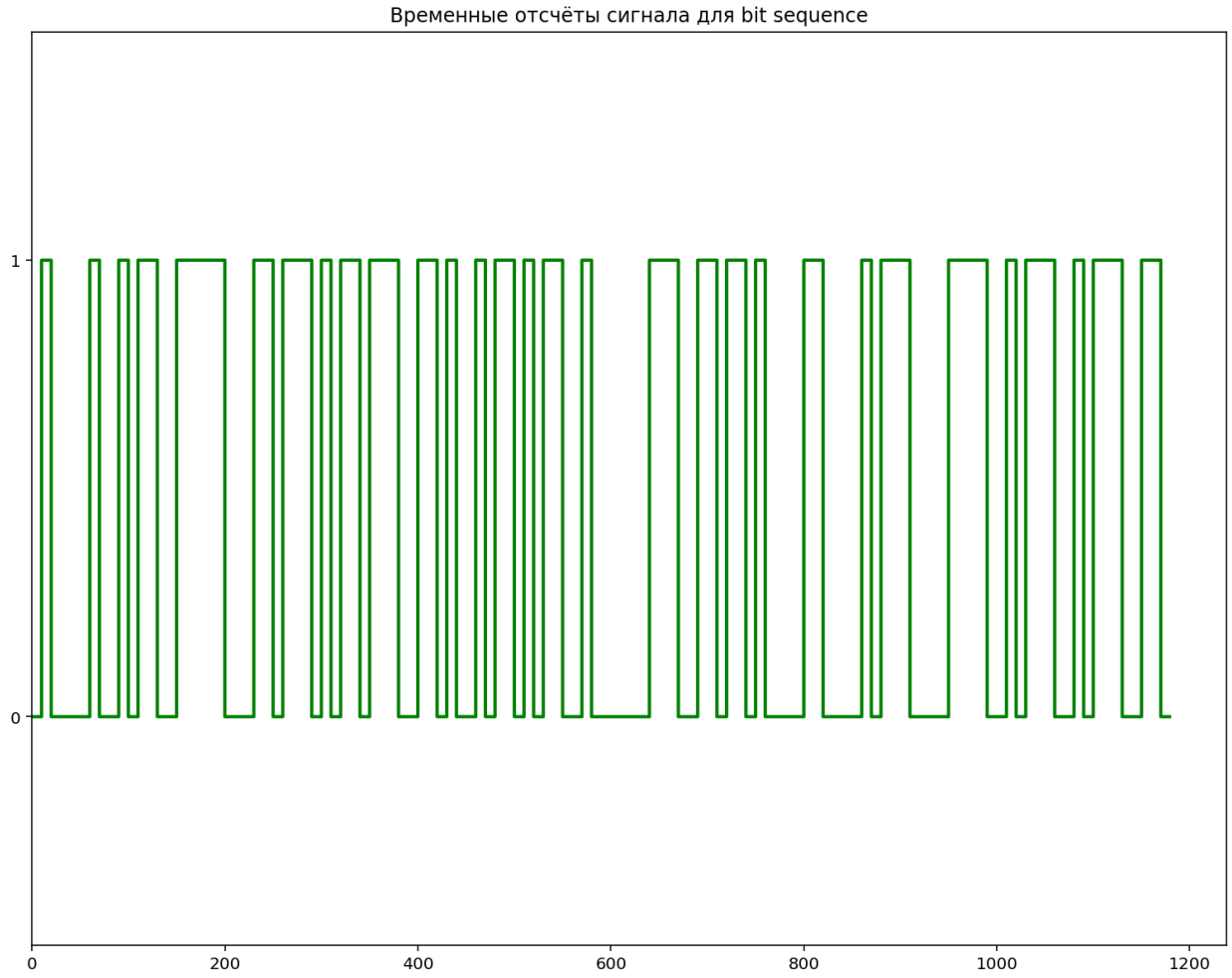
Функция bits_to_samples для преобразования битов во временные отсчѐты сигналов:

```
def bits_to_samples(bit_sequence, N):
    signal_samples = []
    for bit in bit_sequence:
        signal_samples.extend([bit] * N)
    return signal_samples
```

Визуализируем полученную последовательность на графике:

```
plt.figure(figsize=(13, 10))
plt.step(range(len(signal_samples)), signal_samples, where='post', color='b', linewidth=2)
plt.yticks([0, 1], ['0', '1'])
```

```
plt.xlim(0)
plt.ylim(-0.5, 1.5)
plt.title('Временные отсчѐты сигнала для битовой последовательности')
```



6. Создадим нулевой массив длиной 2xNx(L+M+G), а также реализуем ввод с клавиатуры числа от 0 до Nx(L+M+G). В соответствии с введѐнным значением в нулевой массив будет вставляться битовая последовательность с данными. То есть с клавиатуры вводится индекс массива, куда мы ходим вставить данные.

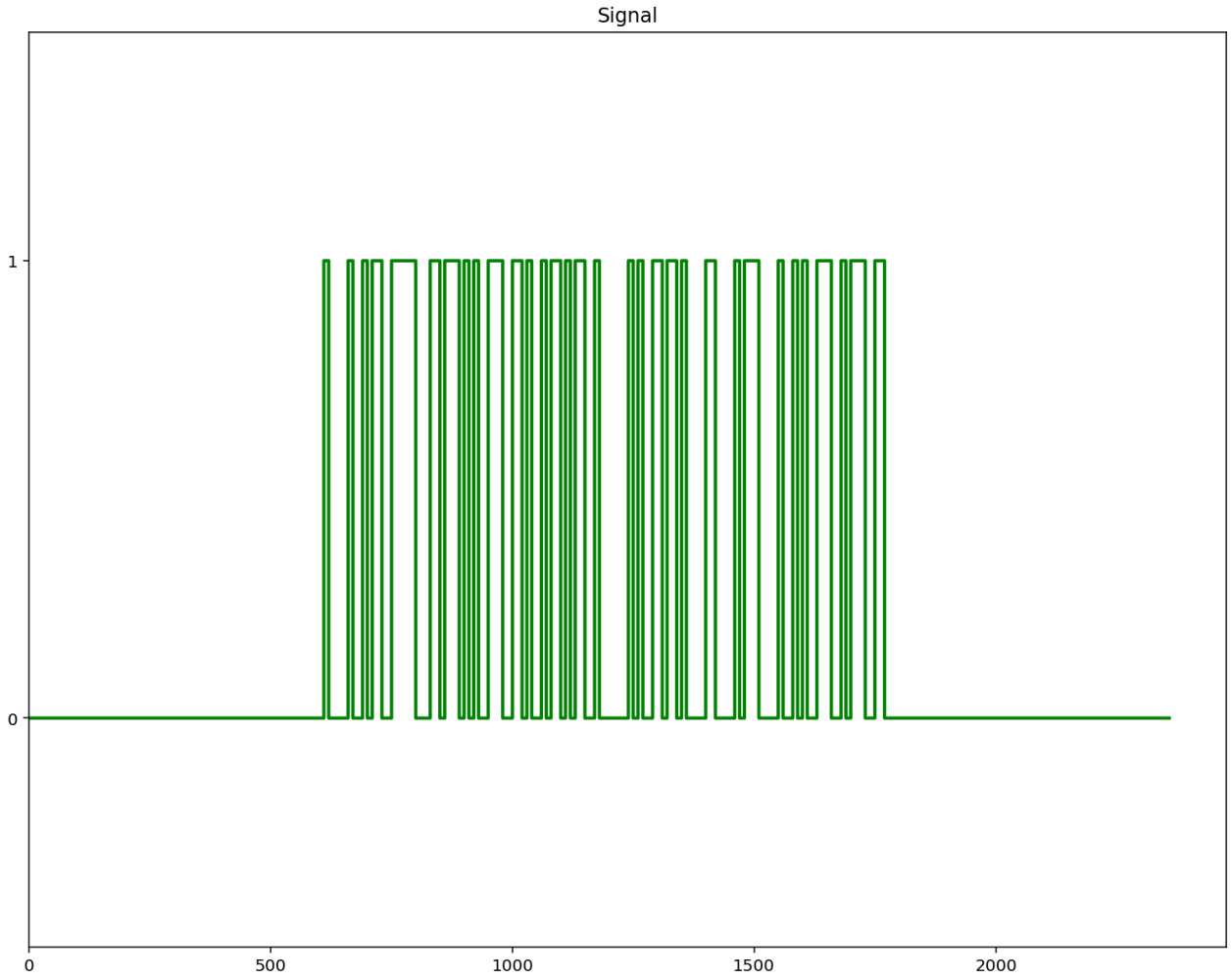
```
signal = [0] * (2 * len(signal_samples))
position = int(input(f'Введите номер позиции для вставки битовой последовательности (от 0 до {len(signal_samples)}): '))
insert_length = min(len(signal_samples), (2 * len(signal_samples)) - position)
signal[position:position + insert_length] = signal_samples[:insert_length]
```

Введѐм любое доступное значение. Допустим, это будет 600:

```
Введите номер позиции для вставки битовой последовательности (от 0 до 1180): 600
```

Визуализируем полученный новый массив:

```
plt.figure(figsize=(13, 10))
plt.step(range(len(signal)), signal, where='post', color='b', linewidth=2)
plt.yticks([0, 1], ['0', '1'])
plt.xlim(0)
plt.ylim(-0.5, 1.5)
plt.title('Битовая последовательность, вставленная в массив Signal')
```



7. Реализуем прохождение нашей последовательности через радиоканал. Проходя через канал, отсчеты сигнала исказились – к ним добавились значения шумов, присутствовавших в канале, которые можно получить, используя нормальный закон распределения с μ = 0 и σ – вводится с клавиатуры (float). Сформируем массив размером 2xNx(L+M+G), реализовав его с помощью нормального распределения. Для этого используем функцию np.random.normal(), после чего поэлементно сложим информационный сигнал с полученным шумом.

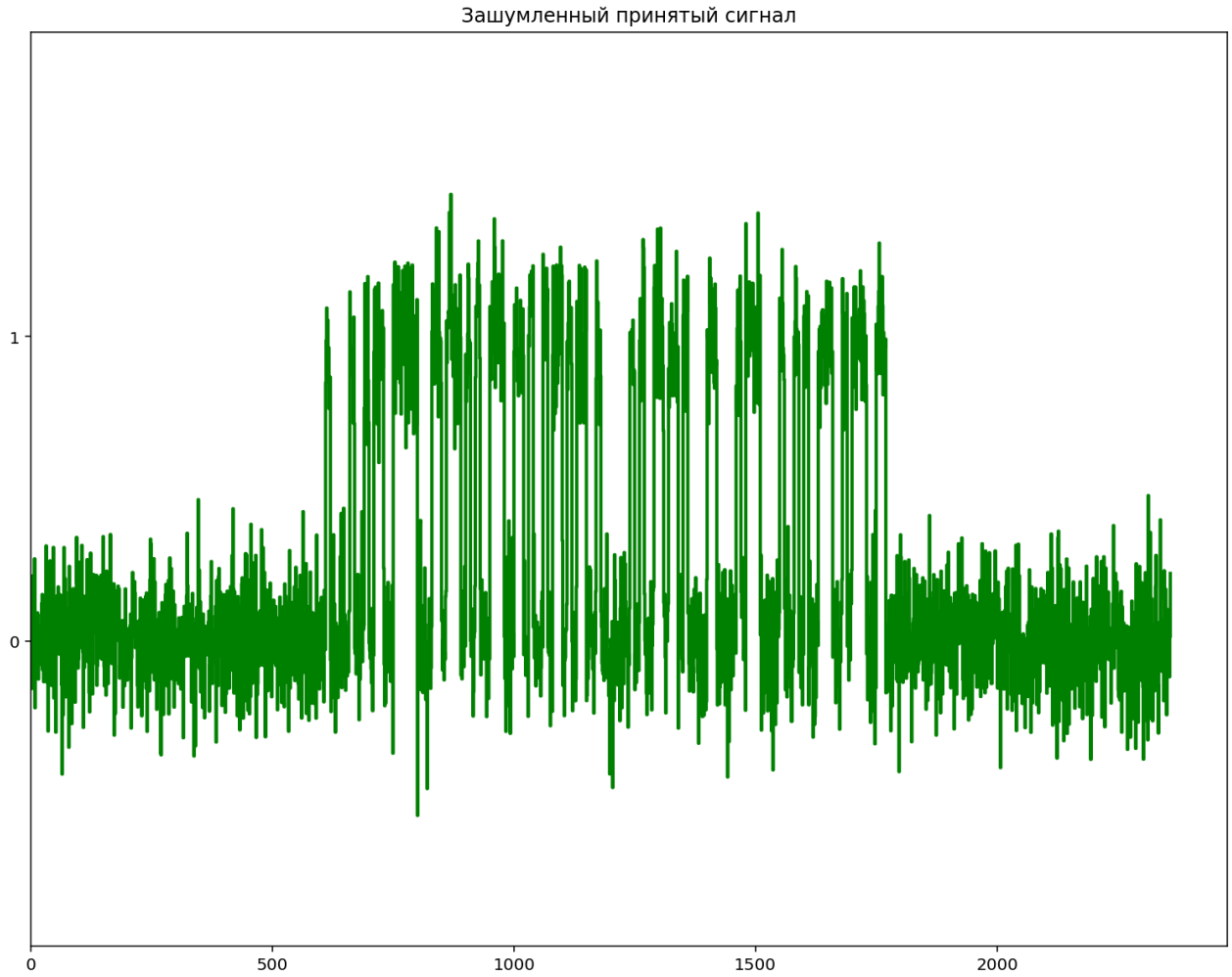
```
sigma = float(input("Введите значение отклонения (sigma): "))
noise = np.random.normal(0, sigma, 2 * len(signal_samples))
noisy_signal = [s + n for s, n in zip(signal, noise)]
```

Введѐм с клавиатуры значение отклонения (например, 0.15):

```
Введите значение отклонения (sigma): 0.15
```

Визуализируем массив отсчетов зашумленного принятого сигнала:

```
plt.figure(figsize=(13, 10))
plt.step(range(len(noisy_signal)), noisy_signal, where='post', color='b', linewidth=2)
plt.yticks([0, 1], ['0', '1'])
plt.xlim(0)
plt.ylim(-1, 2)
plt.title('Зашумленный принятый сигнал')
```



8. Теперь реализуем функцию корреляционного приема и определим, начиная с какого отсчета (семпла) начинается синхросигнал в полученном массиве. Также в этой же функции определим конец нашего передаваемого пакета благодаря корреляционной функции и стоп-слова:

```
def correlation_receiver(x, y, stop_word):
    corr_array = NormalizedCorrelation(x, y)
    start_useful_bits = np.argmax(corr_array)
    print(f'Индекс начала полезного сигнала: {start_useful_bits}')
    corrected_signal = x[start_useful_bits:]

    corr_array_stop = NormalizedCorrelation(corrected_signal, stop_word)
    start_stop_word = np.argmax(corr_array_stop)
    corrected_signal = corrected_signal[:start_stop_word]

    return corrected_signal
```

Сначала функция находит при помощи корреляции последовательность Голда. Это означает начало нашего пакета. Индекс начала полезного сигнала выводится в терминал. После чего все лишние биты, которые идут до нашего пакета, отсекаются. В новом массиве без лишних бит в начале проводится поиск стоп-слова всѐ также при помощи корреляции. Как только стоп-слово

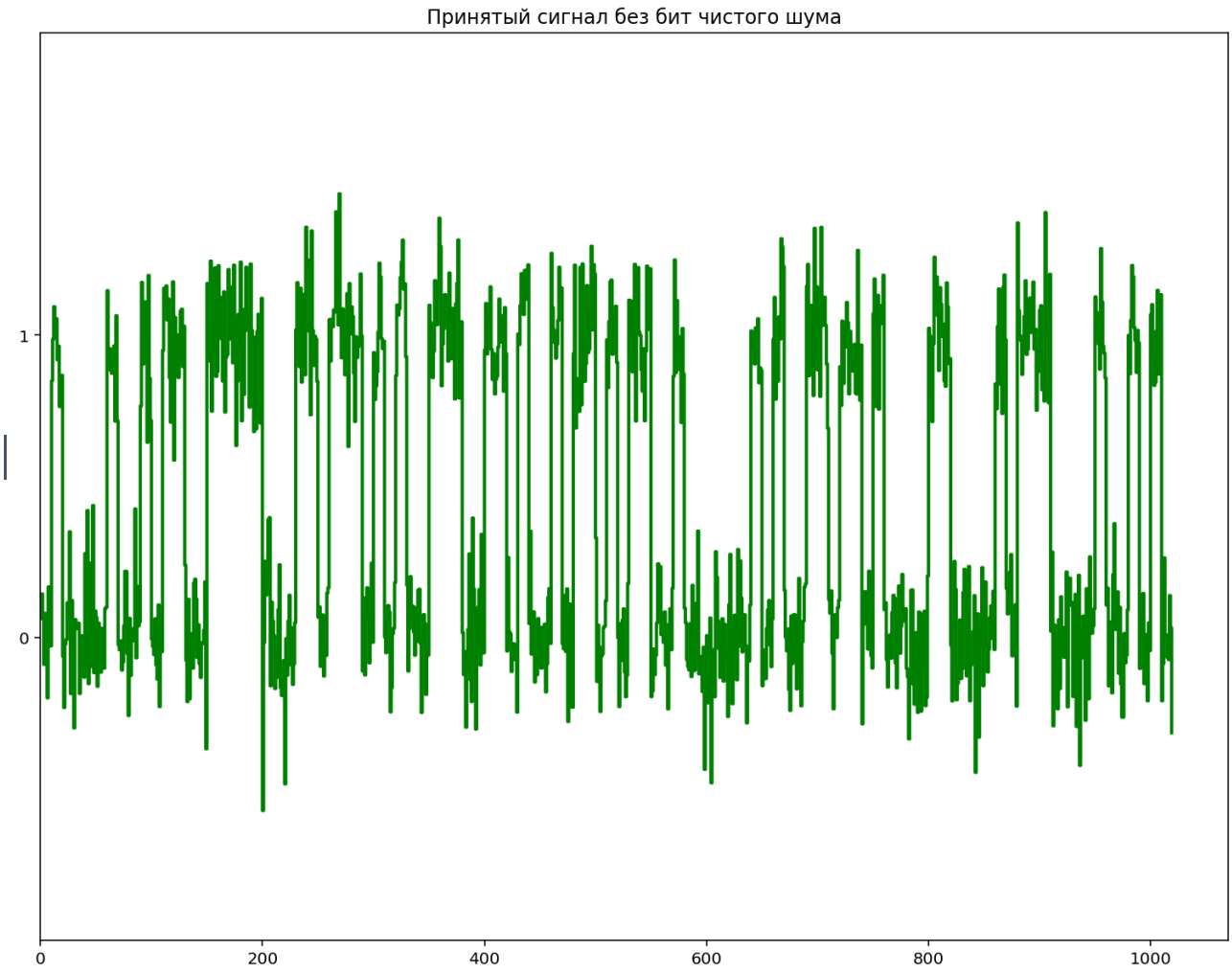
было найдено, функция отсекает лишние биты после пакета (включая стоп-слово). На выход функции подаѐтся принятый сигнал, содержащий только полезные биты.

Функция NormalizedCorrelation:

```
def NormalizedCorrelation(x, y):
    corr_array = []
    for i in range(len(x) - len(y) + 1):
        sumXY = 0.0
        sumX2 = 0.0
        sumY2 = 0.0
        corr = 0.0
        shifted_sequence = x[i:i + len(y)]
        for j in range(len(shifted_sequence)):
            sumXY += shifted_sequence[j] * y[j]
            sumX2 += shifted_sequence[j] * shifted_sequence[j]
            sumY2 += y[j] * y[j]
        corr = sumXY / np.sqrt(sumX2 * sumY2)
        corr_array.append(corr)
    return corr_array
```

Подадим принятый сигнал на функцию приѐма и визуализируем принятый сигнал без лишнего шума:

```
corrected_signal = correlation_receiver(noisy_signal, bits_to_samples(gold_sequence, N),
bits_to_samples(stop_word, N))

plt.figure(figsize=(13, 10))
plt.step(range(len(corrected_signal)), corrected_signal, where='post', color='b', linewidth=2)
plt.yticks([0, 1], ['0', '1'])
plt.xlim(0)
plt.ylim(-1, 2)
plt.title('Принятый сигнал без бит чистого шума')
```



9. Зная длительность в отсчетах N каждого символа, разберѐм оставшиеся символы. Будем накапливать по N отсчетов и сравнивайте их с пороговым значением P для интерпретации полученных семплов нулями или единицами.

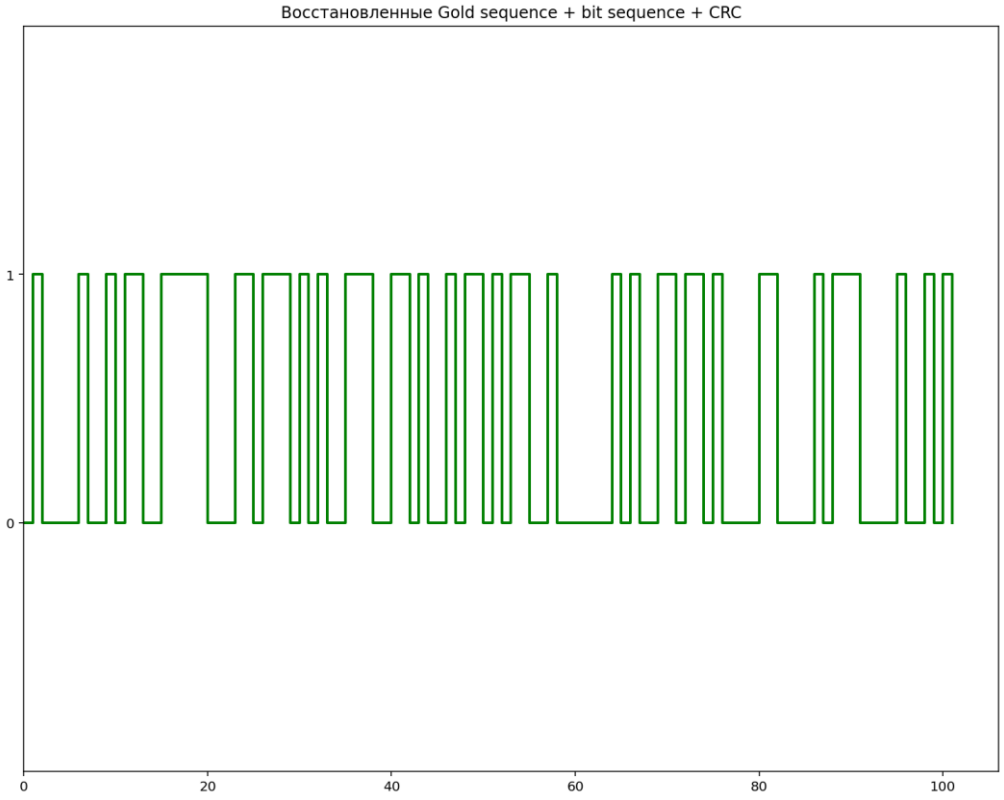
```
def samples_to_bits(signal_samples, N):
    bit_sequence = []
    P = 0.5
    num_blocks = len(signal_samples) // N
    for i in range(num_blocks):
        block = signal_samples[i * N:(i + 1) * N]
        mean = np.mean(block)
        if mean >= P:
            bit_sequence.append(1)
        else:
            bit_sequence.append(0)

    return bit_sequence
```

Функция берѐт блоки по N отсчѐтов и считает их среднее значение, после чего происходит сравнение с переменной P, которая равно 0.5. Если среднее N отсчѐтов больше или равно P, то функция считает что была передана 1, в противном же случае — 0. На выходе функции получается массив длиной

L+M+G битов данных.

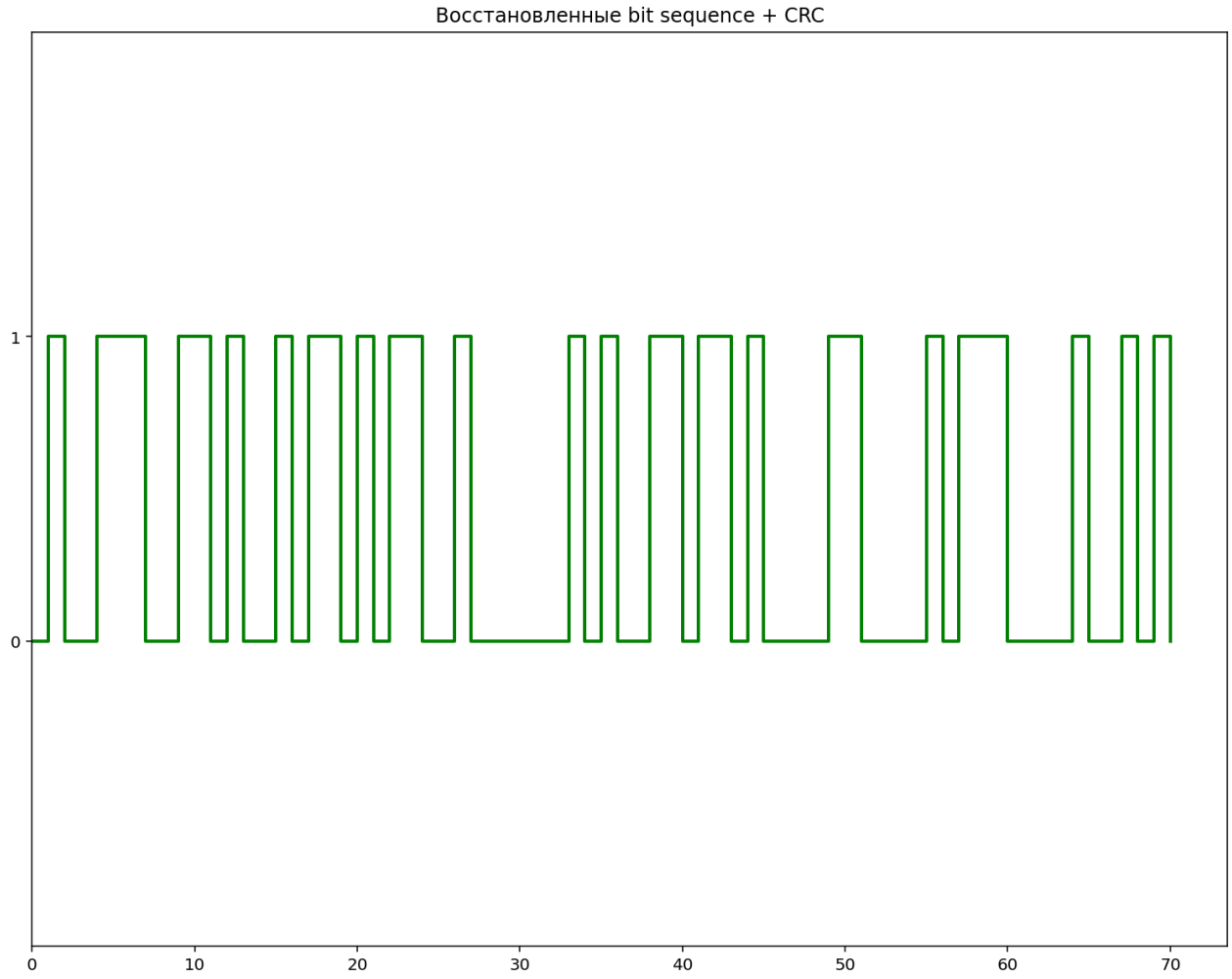
Подадим наш принятый сигнал на эту функцию и визуализируем результат:

```
bit_sequence_crc_gold_restored = samples_to_bits(corrected_signal, N)

plt.figure(figsize=(13, 10))
plt.step(range(len(bit_sequence_crc_gold_restored)), bit_sequence_crc_gold_restored,
where='post', color='b', linewidth=2)
plt.yticks([0, 1], ['0', '1'])
plt.xlim(0)
plt.ylim(-1, 2)
plt.title('Восстановленные последовательность Голда + биты данных + CRC')
```



10. Удалим из полученного массива G-бит последовательности синхронизации:

```
bit_sequence_crc_restored = bit_sequence_crc_gold_restored[len_sequence:]

plt.figure(figsize=(13, 10))
plt.step(range(len(bit_sequence_crc_restored)), bit_sequence_crc_restored, where='post',
color='b', linewidth=2)
plt.yticks([0, 1], ['0', '1'])
plt.xlim(0)
plt.ylim(-1, 2)
plt.title('Восстановленные биты данных + CRC')
```



11-12. Проверим корректность приема бит, посчитав CRC, и выведем в терминал информацию о факте наличия или отсутствия ошибки. Если ошибок в данных нет, то удалим биты CRC и оставшиеся данные подадим на ASCII-декодер, чтобы восстановить посимвольно текст, и выведем его в терминал:

```
if check_packet(bit_sequence_crc_restored, G) == True:
    print("Ошибок в принятом пакете не обнаружено.")

    bit_sequence_restored = bit_sequence_crc_restored[:-(len(G) - 1)]
    restored_text = ascii_decoder(bit_sequence_restored)
    print(f"Востановленный текст: {restored_text}")
else:
    print("Обнаружена ошибка в принятом пакете.")
```

Функция проверки ошибок в пакете check_packet:

```
def check_packet(received_packet, polynomial):
    result = computeCRC(received_packet, polynomial)

    return all(bit == 0 for bit in result)
```

Функция ASCII-декодера:

```
def ascii_decoder(bit_sequence):
    text = ''
    for i in range(0, len(bit_sequence), 8):
        bits_symbol = bit_sequence[i:i + 8]
        if len(bits_symbol) < 8:
```

```
            break
        ascii_symbol = int(''.join(map(str, bits_symbol)), 2)
        text += chr(ascii_symbol)
    return text
```

Декодер переводит каждые 8 бит в десятичный формат, после чего заменяет число на символ согласно ASCII.

В итоге получаем результат анализа ошибок в пакете и восстановленный текст с именем и фамилией:

```
Ошибок в принятом пакете не обнаружено.
Востановленный текст: Nik Shap
```

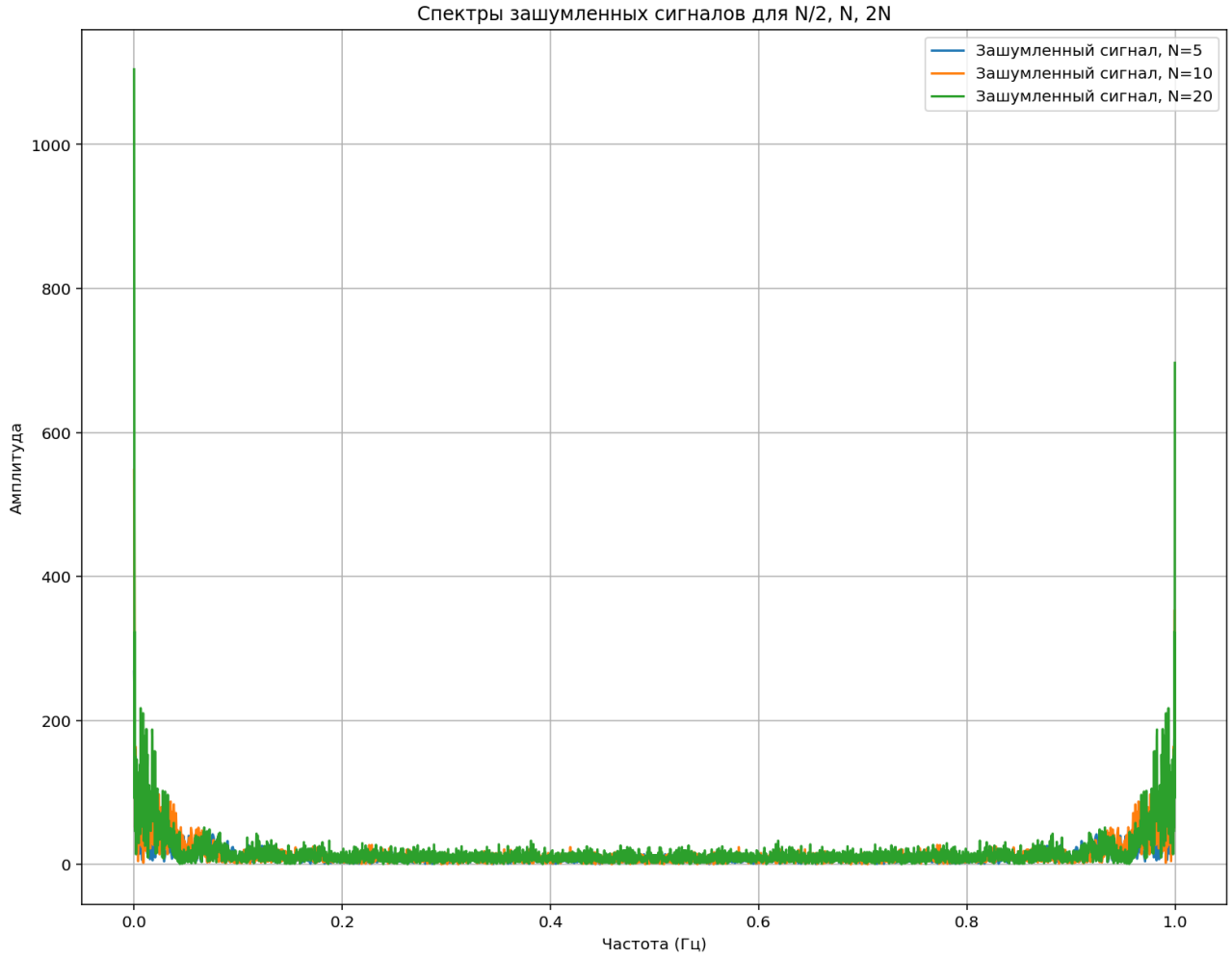
13. Визуализируем спектр передаваемого и принимаемого (зашумленного) сигналов для N = 5, 10, 20:

```
N_array = [5, 10, 20]

plt.figure(figsize=(13, 10))
for N in N_array:
    signal_samples = bits_to_samples(bit_sequence_crc_gold_stop, N)
    plot_spectrum(signal_samples, f'Передаваемый сигнал, N={N}')

plt.xlabel('Частота (Гц)')
plt.ylabel('Амплитуда')
plt.title('Спектры передаваемого сигнала для N/2, N, 2N')
plt.legend()
plt.grid()

plt.figure(figsize=(13, 10))
for N in N_array:
    signal_samples = bits_to_samples(bit_sequence_crc_gold_stop, N)
    signal = [0] * (2 * len(signal_samples))
    insert_length = min(len(signal_samples), (2 * len(signal_samples)) - position)
    signal[position:position + insert_length] = signal_samples[:insert_length]
    noise = np.random.normal(0, sigma, 2 * len(signal_samples))
    noisy_signal = [s + n for s, n in zip(signal, noise)]

    plot_spectrum(noisy_signal, f'Зашумленный сигнал, N={N}')

plt.xlabel('Частота (Гц)')
plt.ylabel('Амплитуда')
plt.title('Спектры зашумленных сигналов для N/2, N, 2N')
plt.legend()
plt.grid()
```

Функция для расчѐта и построения спектра:

```
def plot_spectrum(signal, name):
    spectrum = fftpack.fft(signal)
    freqs = np.arange(0, 1, 1/len(signal))

    plt.plot(freqs, np.abs(spectrum), label=name)
```

В результате получаем:



Заключение

В процессе выполнения расчѐтно-графической работы мы систематизировали и углубили знания, полученные в рамках курса «Основы систем мобильной связи».

Первым шагом мы создали битовую последовательность, которая кодирует имя и фамилию. Для этой последовательности был рассчитан CRC-код, позволяющий на приѐмной стороне проверить наличие ошибок. Также мы сгенерировали и добавили к полезным данным последовательность Голда, которая служит для определения начала передаваемого пакета. Кроме того, к данным было добавлено стоп-слово, определяющее конец пакета.

Далее полезные данные были переданы через радиоканал. Чтобы смоделировать возможные искажения, мы сгенерировали шум с помощью нормального распределения и добавили его к передаваемым данным.

На приѐмной стороне была реализована функция корреляционного приѐма, которая с помощью корреляции определяла начало и конец полезных данных.

Следующим шагом мы преобразовали полученный сигнал обратно в последовательность бит. В конце мы проверили корректность приѐма бит, посчитав CRC, и декодировали битовую последовательность в текст.

Наконец, мы построили спектры сигналов до и после приѐма с различным количеством N.