Министерство науки и высшего образования Российской Федерации

Томский государственный университет
систем управления и радиоэлектроники

ПОМЕХОУСТОЙЧИВОЕ КОДИРОВАНИЕ

Учебно-методическое пособие
по практическим занятиям и самостоятельной работе

Томск 2024

УДК 621.396
БКК 32.811.4
П48

Рецензент:
Рогожников Е.В., заведующий кафедрой ТОР ТУСУР, к.т.н., доцент

Авторы:
Д.А. Покаместов, Я.В. Крюков, А.С. Шинкевич, Г.Н. Шалин

Покаместов, Дмитрий Алексеевич
П48 Помехоустойчивое кодирование: учеб.-метод. пособие по практическим занятиям и самостоятельной работе / Д.А. Покаместов [и др.]. – Томск : Томск. гос. ун-т систем упр. и радиоэлектроники, 2024. – 104 с.

Учебно-методическое пособие содержит указания для выполнения практических и самостоятельных работ. Описана теория и принципы реализации алгоритмов различных помехоустойчивых кодов. Рассмотрены базовые вопросы, роль и место помехоустойчивых кодов в системах связи. Приведены алгоритмы кодов, служащих для обнаружения ошибок (контрольная сумма и CRC). Рассмотрены алгоритмы кодирования и декодирования сверточных кодов. На их основе приводятся принципы построения и реализации турбокодов. Описаны вопросы построения блочных кодов включая простые коды Хэмминга. Рассмотрены наиболее эффективные в настоящий момент коды – LDPC и полярные. Описаны алгоритмы кодирования и декодирования, а также отдельные особенности их применения в системах связи пятого поколения 5G NR. Предназначено для студентов всех технических специальностей.

Одобрено на заседании ПИШ, протокол № 2 от 21.10.2023

УДК 621.396
БКК 32.811.4

Оглавление

Введение

Цифровые модуляции

В цифровых системах связи применяются цифровые виды модуляции

(манипуляции), такие, как FSK, PSK, ASK (frequency, phase, amplitude shift keying, частотная, фазовая, амплитудная манипуляция), QAM (Quadrature-Amplitude Manipulation, квадратурная амплитудная манипуляция), и др.

QAM модуляция основана на квадратурном представлении сигнала, которую можно получить из выражения (1) используя тригонометрическую формулу косинуса суммы двух аргументов:

$$S(t) = A(t)[\cos\omega_0 t \cdot \cos\varphi(t) - \sin\omega_0 t \cdot \sin\varphi(t)]$$

В этой записи можно сделать замену:

$$i(t) = A(t) \cdot \cos\varphi(t)$$

$$q(t) = A(t) \cdot \sin\varphi(t)$$

Тогда квадратурный сигнал может быть записан в виде:

$$S(t) = i(t) \cdot \cos\omega_0 t - q(t) \cdot \sin\omega_0 t,$$

где i(t) и q(t) (могут быть записаны как I и Q) – синфазная и квадратурные огибающие (квадратуры).

QAM модуляция основана на отображении группы (одного, или несколько) битов в значения I и Q. В этом случае QAM модулятор может иметь вид, приведенный на рисунке 1.

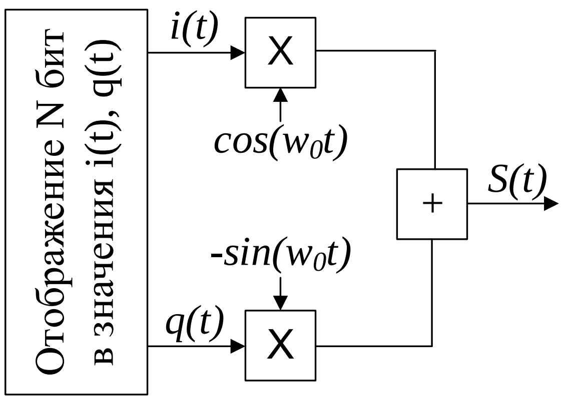


Рисунок 1 – QAM модулятор

Блок отображения битов в значения i(t) и q(t) может быть описан таблицей соответствия между реализацией группы битов и i(t) и q(t). Для модуляции QAM-4 (она же QPSK) это соответствие приведено в таблице 1.

Таблица 1 - QAM-4 (QPSK) модуляция

Входные биты	i(t)	q(t)
00	-1	-1
01	-1	1
10	1	-1
11	1	1

Также соответствие между битами и значениями i(t) и q(t) можно продемонстрировать с помощью диаграммы «Созвездие», оси которой соответствуют квадратурам, а точки соответствуют возможным символам модуляции (комбинациям i(t) и q(t)). Созвездие для модуляции QAM-4 (QPSK) приведен на рисунке 2.

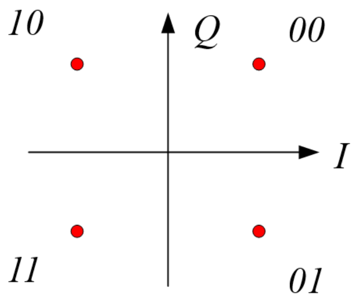


Рисунок 2 – Диаграмма «Созвездие» модуляции QAM-4

Физически процесс модуляции можно представить следующим образом (рисунок 3).

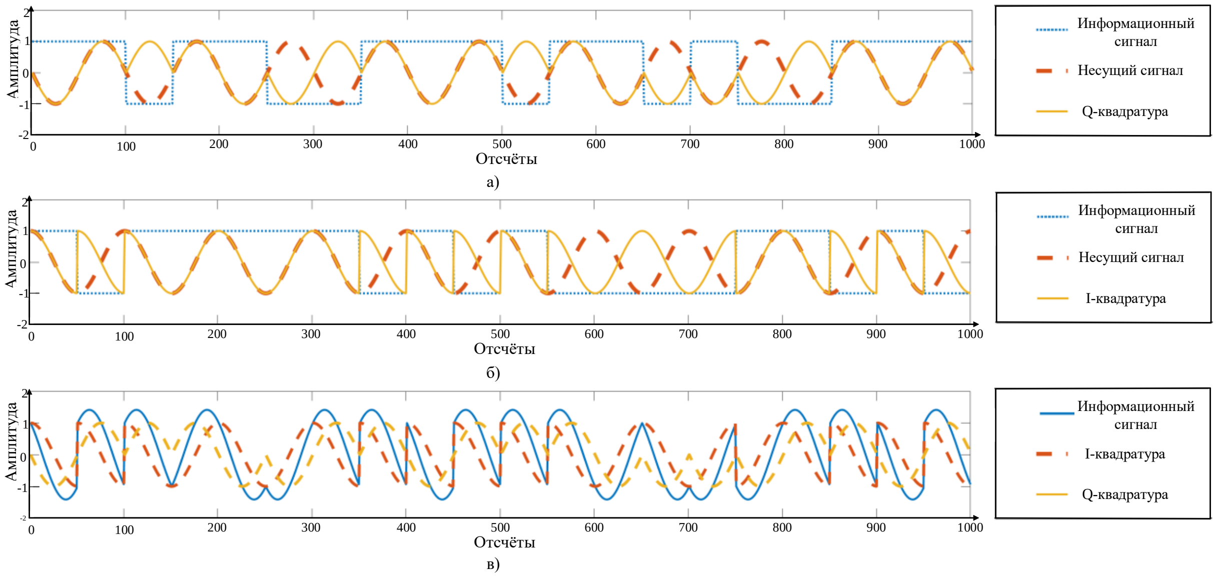


Рисунок 3 – квадратурная модуляция а) Модуляция I-квадратура, б) Модуляция Q-квадратуры, в) Результирующий сигнал

QAM демодуляция на рисунках 2 может быть построена на основании порогового устройства. Для модуляции QAM-4 с созвездием, приведенным на рисунке 2, этот алгоритм будет иметь следующую логику:

Если принятая точка входящая в вектор **Y** находится в левом верхнем секторе (I<0, Q>0), то принимается решение о том, что были переданы биты [0 1].

Иначе, если точка расположена в правом верхнем секторе, то были переданы биты [0 0].

Иначе, если точка расположена в правом нижнем секторе, то были переданы биты [0 1].

Иначе, если точка расположена в левом нижнем секторе, то были переданы биты [1 1].

Основы теории помехоустойчивого кодирования

На рисунке 4 изображена упрощенная структурная схема цифровой системы связи.

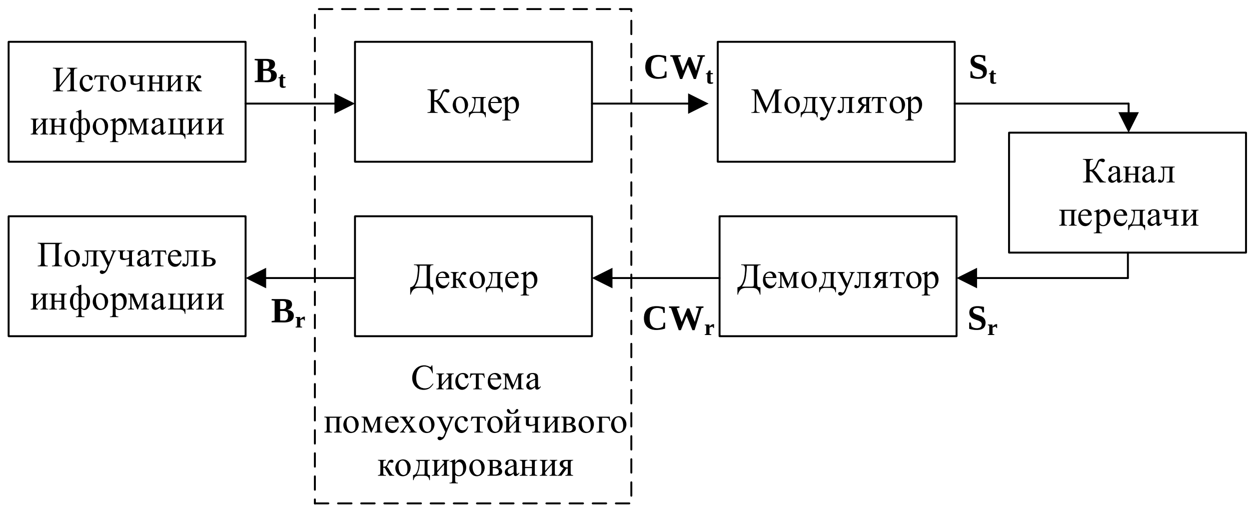


Рисунок 4 – Упрощенная модель канала цифровой связи

Рассмотрим подробнее данную схему. Вектор бит от источника информации $\mathbf{B_t}$ поступает на помехоустойчивый кодер, где к нему добавляется избыточность. Кодовое слово на выходе кодера $\mathbf{CW_t}$ поступает на модулятор, где биты преобразуются в физический сигнал $\mathbf{S_t}$. В канале передаче на сигнал накладываются различного рода шумы и помехи, соответственно на вход демодулятора приходит зашумленный сигнал $\mathbf{S_r}$. При демодуляции зашумленного сигнала возможно появление битовых ошибок. Вектор бит с ошибками $\mathbf{SW_r}$ поступает на декодер. В декодере, если количество битовых ошибок меньше порогового, то они успешно исправляются, а избыточность удаляется, соответственно вектор выходных бит $\mathbf{B_r}$ идентичен вектору входных бит $\mathbf{B_t}$. Данная схема обобщена, в разных системах помехоустойчивому кодированию могут предшествовать разные процедуры канального кодирования, такие как скремблирования (рандомизация). А после закодированные биты могут проходить операцию перемежения, для устранения пакетированных ошибок, также в реальных системах целостность пакета может контролироваться контрольной суммой.

Таким образом, помехоустойчивое кодирование напрямую влияет на вероятность появления битовых ошибок, а соответственно, на скорость передачи данных и дальность связи.

Выбор оптимального алгоритма кодирования – сложная задача, т.к. помехоустойчивый код должен соответствовать противоречащим требованиям, а именно: высокая исправляющая способность, малая избыточность, простота кодирования/декодирования. На практике для разных задач применяются разные виды кода, а также кодеры с разной скоростью кода.

Существует много разных вариантов классификации помехоустойчивых кодов, они представлены на рисунке 5.

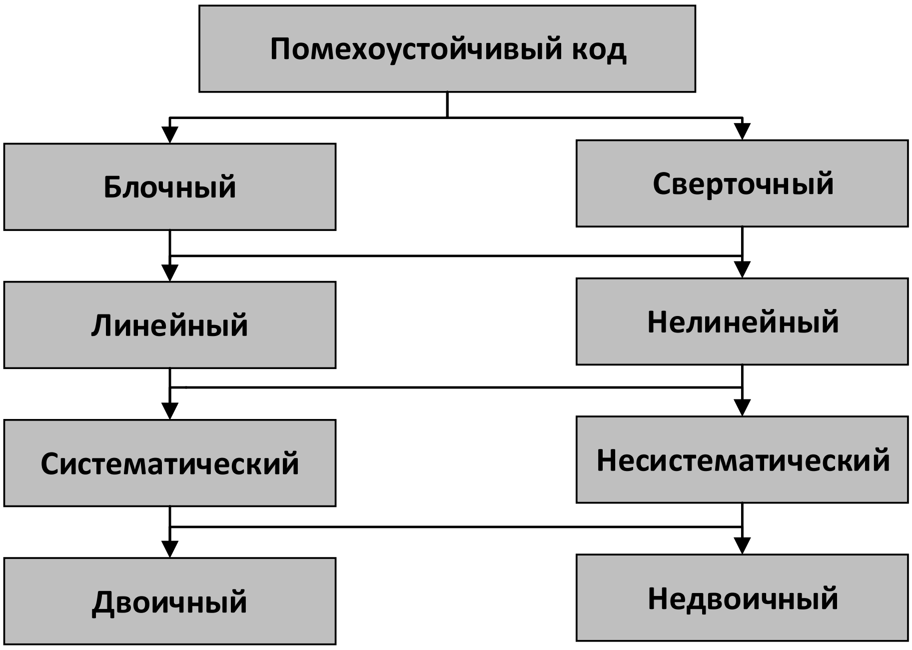


Рисунок 5 – Классификация помехоустойчивых кодов

На вход блочного кодера поступает блок бит, сверочные кодеры в свою очередь работают с непрерывной последовательностью бит.

В систематическом коде к информационным символам добавляются проверочные символы, т.е. кодовое слово в явном виде содержит информационные символы, в несистематическом коде информационных символов в явном виде нет.

Код является двоичным, если работает с двоичными данными, во всех остальных случаях код является недвоичным.

Основная теорема помехоустойчивого кодирования – теорема Шеннона, гласит следующее: существует такая система помехоустойчивого кодирования, при использовании которой можно получить сколь угодно малую вероятность битовой ошибки, если реальная скорость передачи R меньше, чем пропускная способность канала C, где C определяется по формуле.

$$C = \Delta F \log 2(1 + \frac{P_c}{P_ш})$$

где ΔF – полоса частот сигнала, P_c – мощность сигнала, $P_ш$ – мощность шума.

Данная теорема справедлива для широкого класса моделей каналов, но только при условии того, что размер кодового слова стремится к бесконечности. Однако в реальной жизни увеличение размера кодового слова приводит к увеличению вычислительной сложности алгоритмов, а также сложностям в аппаратной реализации кодеров и декодеров.

Рассмотрим обобщенный помехоустойчивый кодер, представленный на рисунке 6.

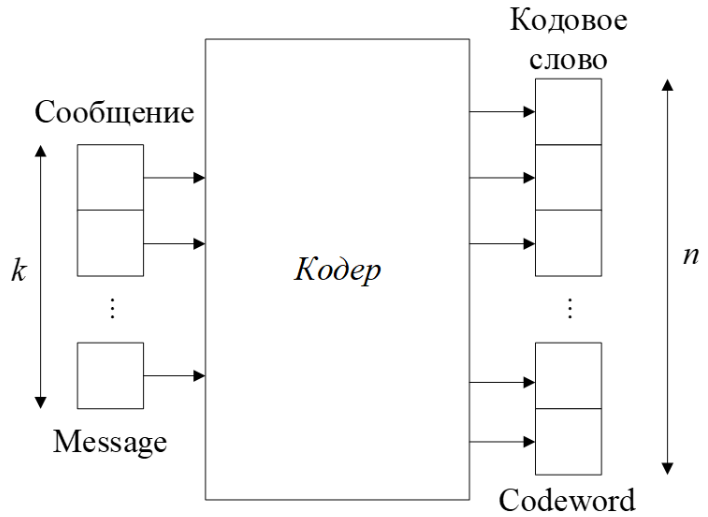


Рисунок 6 – Помехоустойчивый кодер

К основным параметрам кодера можно отнести скорость кода – отношения количества входных бит к количеству выходных бит R=(k/n) < 1.

Кодовое расстояние d_{min} – минимальное расстояние Хэмминга между двумя любыми кодовыми словами. Расстояние Хэмминга – количество несовпадающих битов, например, расстояние между 1101 и 1010 равно 3).

Количество гарантировано обнаруживаемых ошибок $v = d_{min} - 1$.

Количество гарантированно исправляемых ошибок $t = \lfloor (d_{min} - 1)/2 \rfloor$.

Помехи

В работах мы будем рассматривать канал с аддитивным белым гауссовским шумом (АБГШ). Вектор символов на выходе канала может быть записан как

$$\mathbf{Y}=\mathbf{X}+\mathbf{N},$$

где **X** – вектор символов модуляции, **N** – вектор аддитивной помехи (АБГШ).

Рассмотрим смысл понятия АБГШ. Слово аддитивный означает, что шум накладывается на полезный сигнал (в отличии от мультипликативной помехи). «Белый» характеризует спектральные свойства шума – он обладает одинаковой спектральной плотностью мощности на всех частотах. «Гауссов» означает, что его реализации имеют нормальное (гауссово) распределение вероятностей.

Основные метрики

Одной из основных метрик состояния канала является отношения сигнал-шум (ОСШ, signal-noise ratio, SNR). SNR можно выразить следующим образом:

$$SNR = P_с/P_ш = (A_с)^2/(A_ш)^2,$$

где $P_с$ и $P_ш$ – мощность сигнала и шума, а $A_с^2/A_ш^2$ – средняя амплитуда сигнала и шума соответственно.

В логарифмическом масштабе:

$$SNR(dB) = 10\log_{10}(P_с/P_ш) = 20\log_{10}(A_с/A_ш).$$

Другой метрикой, характеризующей канал, является величина отношения энергии, приходящейся на один бит сообщения E_b к спектральной плотности мощности шума N_0.

E_b/N_0 – отношение энергии, приходящейся на бит сообщения E_b к спектральной плотности мощности шума N_0;

$$E_b = P_c T_b,$$

P_c – мощность сигнала, T_b – интервал передачи бита, $T_b = 1/R$, R - скорость;

$$N_0 = P_ш/W,$$

$P_ш$ – мощность шума, W – полоса радиоканала;

$$E_b/N_0 = (P_c W)/ (P_ш R);$$

$R = WN_b$, N_b – количество бит, которое несет один символ модуляции;

Без кодирования: $R = W\log_2(M)$, M – индекс модуляции, $\log_2(M)$ - количество бит, которое несет один символ модуляции;

В разах: $E_b/N_0 = SNR/ \log_2(M)$.

В децибеллах: $E_b/N_0 = SNR - 10\log_{10}(\log_2(M))$

В случае с кодированием $E_b/N_0 = SNR - 10\log_{10}(\log_2(M)) - 10\log_{10}(R_{code})$, где R_{code} – скорость кодирования.

Важнейшей характеристикой системы связи, является ее помехоустойчивость, которую можно выразить через вероятность битовых ошибок BER (англ. Bit Error Rate). По определению

$$BER = N_{ош} / N,$$

где $N_{ош}$ – количество ошибок, N –число передаваемых бит. Поскольку это статистическая величина, BER определяется только при большом числе передаваемых бит. Очевидно, что количество ошибок (а соответственно и BER) зависит от отношения сигнал-шум на входе демодулятора. На рисунке 7 изображены зависимости BER от SNR в канале с АБГШ для разных видов модуляции.

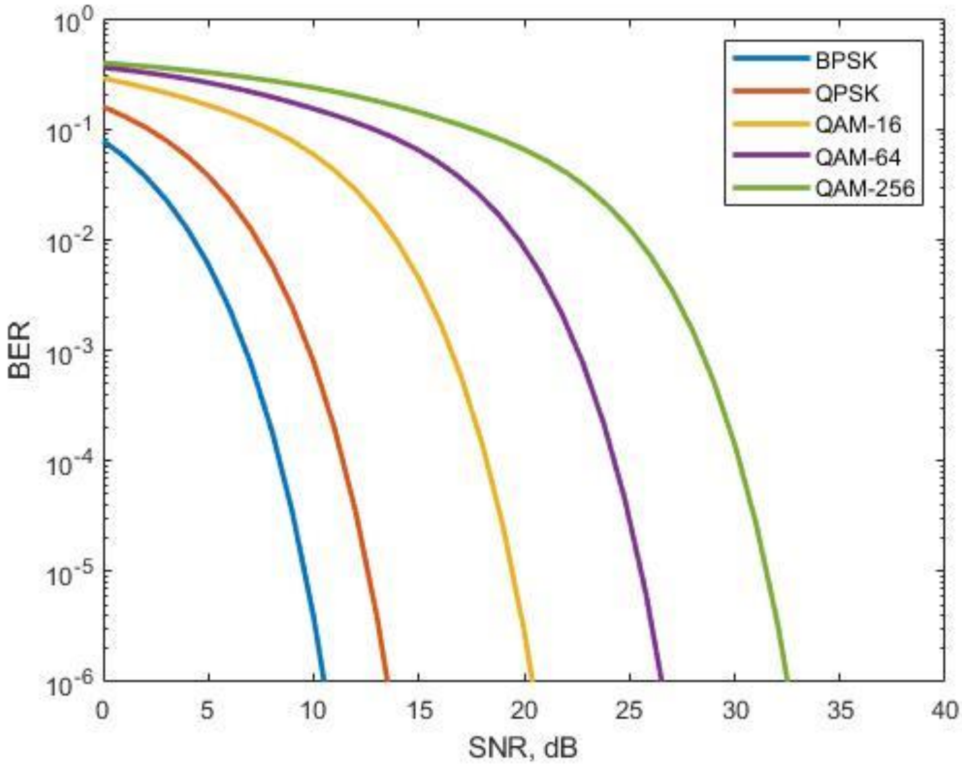


Рисунок 7 – зависимости BER от SNR

Но чтобы определить BER на приемной стороне надо точно знать какие биты передавались, что в реальных системах невозможно. Так что для оценки помехоустойчивости реальных систем используется другая характеристика – вероятность блоковой ошибки (BLER, Block Error Rate). Как было сказано ранее, наличие ошибок на приемной стороне определяется контрольной суммой. Таким образом, BLER показывает отношения количества пакетов, принятых с ошибками к общему количеству принятых пакетов.

$$\mathrm{BLER} = \frac{\mathrm{N}_{\text{пакетов с ошибками}}}{\mathrm{N}_{\text{общее число пакетов}}}$$

Практическая работа № 1

Основы языка Matlab. Реализация канала связи с цифровой модуляцией и аддитивным белым гауссовским шумом

Цель работы: Знакомство с языком Matlab. Реализация элементарного канала цифровой связи.

Задачи практической работы:

1) Знакомство с синтаксисом языка Matlab;
2) Знакомство с интерфейсом пакета Octave;
3) Разработка функции квадратурной амплитудной модуляции;
4) Разработка функции квадратурной амплитудной демодуляции;
5) Разработка модели канала цифровой связи с аддитивным белым гауссовским шумом.

Оборудование и программное обеспечение: Octave, version 6.4.0.

Теоретический материал

Элементарная модель канала цифровой связи может быть построена по схеме, приведенной на рисунке 1.

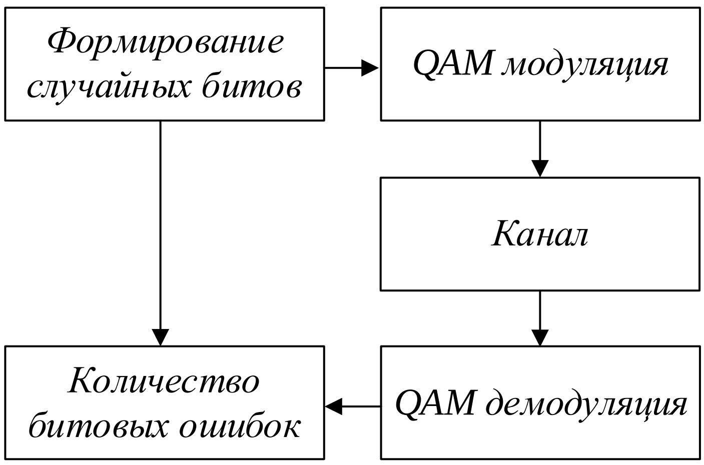


Рисунок 1 – Упрощенная модель канала цифровой связи

На стороне передатчика формируется вектор случайных бит, который поступает в блок квадратурной амплитудной QAM (Quadrature Amplitude Modulation) модуляции. На выходе этого блока образуется вектор QAM-символов, поступающий в канал связи. В этой и следующих работах в качестве модели канала мы будем использовать канал с аддитивным белым гауссовским шумом АБГШ. Символы с наложенным на них шумом поступают в блок QAM демодуляции, где принимается решение о том, какие биты были переданы. На последнем этапе вектор битов в приемнике сравнивается с вектором битов в передатчике и вычисляется количество ошибок, возникших при передаче.

Ход работы

В рамках первой работы вам необходимо в программном обеспечении Octave реализовать модель канала связи, приведенную на рисунке 1.

Для работы в Octave с телекоммуникационными функциями нужно загрузить специальную библиотеку, для этого нужно написать в командной строке команду:

pkg load communications

и нажать Enter. При этом выполнится соответствующая команда и загрузится библиотека.

Программы в Octave пишутся в двух типах файлов – скриптах (сценариях) и функциях. Скрипты являются исполняемыми файлами (файлами верхнего уровня), а функции подключаются в скрипты, или другие функции.

Для того чтобы создать скрипт нужно выбрать меню Файл – Создать – Создать сценарий.

Создайте отдельную папку (рекомендуем каждый новый проект создавать в отдельной папке). Создайте скрипт и назовите его main. В начале файла напишите следующие команды (каждую в отдельной строке):

clc

(стирает все сообщения из адресной строки);

clear all

(стирает все переменные из рабочей области);

close all

(закрывает все окна графиков).

Далее объявите переменные

M = 2;

(индекс модуляции, возможные значения 2, 4, 16, …);

N = 10000;

(количество передаваемых символов);

SNR = 20;

(отношение сигнал-шум).

Далее объявите команду создания вектора символов (битов):

bits = randi([0 1],1,N);

Эта команда формирует вектор случайных целых чисел со значениями от 0 до 1 (вектор случайных бит).

Сформированный вектор символов должен подаваться на модулятор. В Octave есть встроенная функция QAM – модуляции *qammod*, которая поддерживает индексы *M*>4, BPSK модуляцию необходимо реализовать самостоятельно.

Создайте функцию модулятора: Файл – Создать – Создать функцию. Назовите функцию modulator. Функция должна иметь входы bits, M и выход symbols. Это можно реализовать написав в первой строке

function symbols = modulator (bits, M)

Далее, если (условие реализуется с помощью оператора *if*) индекс *M* равен 2, должна выполняться BPSK модуляция, которую можно математически реализовать следующим образом:

*symbols = bits*2-1;*

Иначе (*else*) вызывается встроенная (библиотечная) функция, но на ее вход должны подаваться группы бит, преобразованные в десятичный вид, поэтому сначала сгруппируем биты, а потом переведем группы бит в десятичный вид.

sym = bi2de(reshape(bits,[],log2(M)));

symbols = qammod(sym,M)

После этого закрывается оператор *else-if* с помощью *end*

и завершается работа функции

endfunction.

Важно понимать, что размер сообщения должен быть кратен log2(*M*), для корректной работы модулятора.

Сохраните функцию в отдельный файл и назовите его modulator.

Далее в файле main следует вызвать функцию модулятора:

qam_sym = modulator(bits, M);

После этого моделируется АБГШ канал связи с помощью функции

chan_sym = awgn(qam_sym,SNR,'measured');

где *'measured'*- обозначает измерение мощности сигнала при наложении на него шума. Если оставить функцию без этого аргумента, мощность сигнала будет считаться равной 1 Вт.

Алгоритм демодуляции следует реализовать в отдельной функции *demodulator* с входами *r_sym, M* и выходом *r_bits* по аналогии с модуляцией. Если *M*=2 (BPSK) демодуляция может быть реализована как простое пороговое устройство:

r_bits = r_sym>0;

В случае, если элемент вектора r_sym больше нуля, в соответствующий ему элемент вектора *r_bits* будет записана логическая единица, в противном случае, логический ноль.

В случае если *M*>2 мы вызываем функцию *qamdemod,*

sym = qamdemod(r_sym,M);

r_bits = reshape(de2bi(sym),1,[])

Вызовите функцию demodulator

r_bits = demodulator(chan_sym, M);

Следующим этапом работы скрипта является расчет количества ошибок, который можно реализовать следующим образом

length(find(bits - r_bits)),

функция *de2bi* переводит символы в двоичный вид (биты), *find* ищет ненулевые элементы (различающиеся биты при вычитании не будут равны нулю), а *length* считает их количество. Если после вызываемой функции не ставить символ «;», то результат выполнения функции будет выводиться в командную строку.

Последним этапом работы скрипта будет вывод сигнального созвездия на входе приемника (после канала)

scatterplot(chan_sym).

Запустить скрипт можно нажав на кнопку «Сохранить и выполнить/продолжить» на панели инструментов выше текстового редактора (рисунок 2), либо нажав F5 на клавиатуре.

Рисунок 2 – Запуск скрипта

Количество ошибок будет выведено в командную строку. Изменяйте индекс модуляции, отношение сигнал-шум, чтобы пронаблюдать сигнальное созвездие для разных модуляций и количество битовых ошибок при различных сценариях работы (добейтесь появления большого числе ошибок). Определите связь между разбросом точек на сигнальном созвездием и появлением ошибок.

Для отчета по проделанной работе сделайте скриншоты сигнальных созвездий для модуляции QPSK при *SNR*=15дБ и 5дБ.

Контрольные вопросы:

1. Что такое индекс модуляции?
2. Каково сигнальное созвездие модуляции QAM-16?
3. Как найти количество битовых ошибок?

Практическая работа № 2

Системные характеристики телекоммуникационных систем. Связь между помехоустойчивостью и скоростью передачи

Цель работы: Реализация кодирования повторением. Определение зависимостей помехоустойчивости связи от отношения сигнал-шум.

Задачи практической работы:

1) Реализация кодера повторением;
2) Реализация декодера повторением;
3) Оценка зависимостей BER от отношения сигнал-шум;

Теоретический материал

Текущая работа является логическим продолжением и расширением предыдущей работы, рисунок 1.

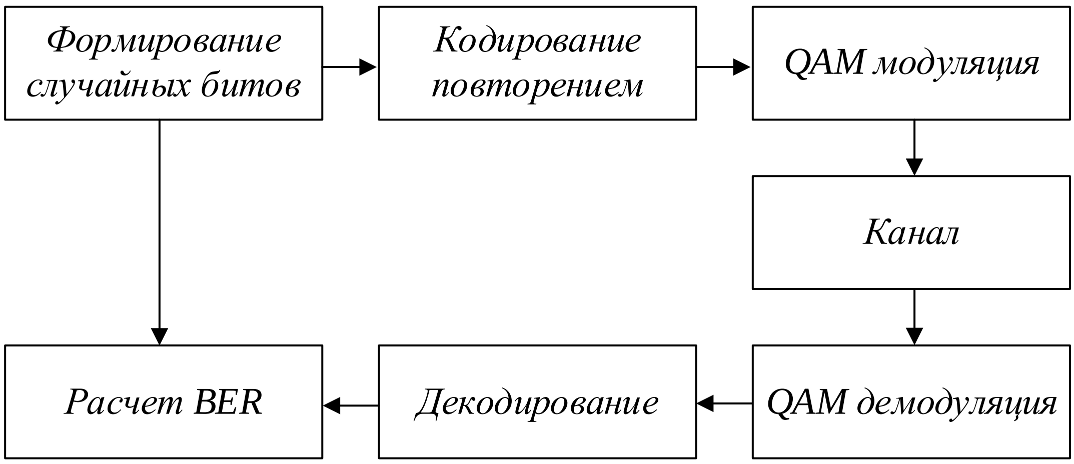


Рисунок 1 – Схема реализуемой в текущей работе модели

Вам предстоит добавить блоки помехоустойчивого кодирования и декодирования и получить зависимости вероятности битовых ошибок от отношения сигнал-шум (ОСШ, SNR).

В качестве помехоустойчивого кодирования в этой мы будем применять его простейший способ – кодирование повторением. Этот способ обладает низкой эффективностью, однако не требует реализации сложных алгоритмов и применяется в низкоскоростных системах связи.

Суть кодирования повторением сводится к повторению каждого бита D раз. Пример такого кодирования для D = 3 приведен на рисунке 2.

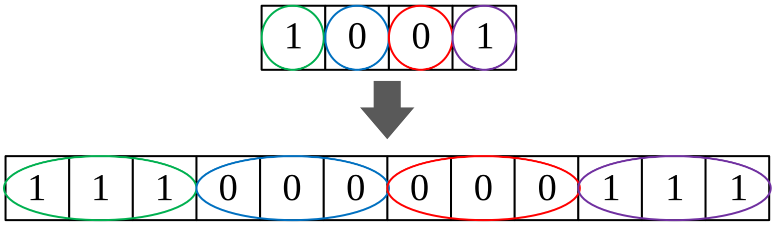


Рисунок 2 – Пример кодирования повторением

Нетрудно определить, что кодовое расстояние будет равно D. Для D=3 можно обнаружить до двух ошибок и исправить одну.

Простейший декодер можно построить, рассчитав среднее значение группы битов и округлив его. Так, если передавался бит «1», кодовое слово которого будет «1 1 1» и появилась ошибка во втором переданном бите, то на входе декодера будет «1 0 1». Среднее значение этого вектора составляет 0.66, округлив которое мы получим 1.

Ход работы

Целью работы является создание имитационной модели системы связи с кодированием повторением, работающей в заданном диапазоне ОСШ и рассчитывающей зависимости BER(E_b/N_0) и BER(*SNR*). Далее приведена возможная последовательность действий по реализации этой модели, однако, вы можете реализовать ее самостоятельно.

Добавьте к параметрам разработанной в ходе прошлого занятия модели коэффициент повторения D, а одно значение *SNR* замените на два числа: минимальное и максимальное значение *SNR*, для которых вы планируете проводить моделирование:

SNR = [0 20];

Далее необходимо объявить цикл с помощью оператора *for*, в котором значения *snr_i* будут меняться от *SNR(1)* до *SNR(2)*. Кроме того, необходимо объявить индекс итерации, например *k*. В каждой следующей итерации *k* будет увеличиваться на 1.

k = 0;

for snr_i = SNR(1):SNR(2)

k = k+1

Создайте функцию (в отдельном файле) кодирования повторением *repeat_code*. Аргументы функции *bits*, *D*, выход *coded_bits*. Саму функцию можно реализовать используя следующий алгоритм. Создается пустой вектор *coded_bits*, который в дальнейшем мы наполним закодированными битами.

coded_bits = [];

Создаем цикл от 1 до длины вектора входных битов, в каждой итерации объявляем пустой вектор *bits_D* и далее в цикле от 1 до *D* наполняем его копиями *i*-го бита (вставляем *D* раз) и записываем в вектор *coded_bits*:

for i = 1:length(bits)

bits_D = [];

for d = 1:D

bits_D = [bits_D,bits(i)];

end

coded_bits = [coded_bits,bits_D];

end

На этом код функции заканчивается.

Вставьте функцию в скрипт *main* и расположите ее в соответствии с рисунком 1 (на вход подаются биты с генератора случайных чисел, выход функции подключен к модулятору):

bits   = randi([0 1],N,1);

coded_bits = repeat_code (bits, D);

qam_sym = modulator(coded_bits, M);

Измените аргументы функции *awgn*: вместо *SNR* подайте текущее значение *snr_i*. Далее добавьте реализованный раннее демодулятор:

r_bits = demodulator(chan_sym, M);

Создайте функцию декодирования со входами *r_bits*, *D* и выходом *decoded_bits*. Она может иметь следующую логику. Создается пустой вектор *decoded_bits*, который в дальнейшем мы наполним закодированными битами.

decoded_bits = [];

В цикле с *i* от 1 до длины вектора входных битов с шагом *D* необходимо в каждой итерации вырезать из вектора соответствующую группу битов, сложить значения всех битов, разделить на *D* и округлить. Результат добавить к *decoded_bits*:

```
for i = 1:D:length(bits)
bits_group    = bits(i:i+D-1);
decoded_bit   = round(sum(bits_group)/D);
decoded_bits  = [decoded_bits, decoded_bit];
end
```

В каждой итерации (для каждого значения *snr_i*) необходимо рассчитывать *BER(k)*:

```
ber(k) = (length(find(de2bi(bits) - de2bi(decoded_bits))))/(N*log2(M));
```

На этом итерация глобального цикла *snr_i* заканчивается. Последним этапом является построение в логарифмическом масштабе зависимостей $BER(E_b/N_0)$ и *BER(SNR)*:

```
semilogy(SNR(1):SNR(2), ber)
```

Для того, чтобы следующий график построился в новом графическом окне, можно воспользоваться функций *figure*, которая его открывает.

```
semilogy([SNR(1):SNR(2)] - 10*log10(1/D), ber)
```

Для построения двух графиков в одном графическом окне нужно между соответствующими функциями построения записать команду *hold on*. Оси графиков можно подписать следующим образом.

```
xlabel(' SNR');
ylabel(' BER ');
```

Для отчета по проделанной работе сделайте скриншоты зависимостей *BER* от *SNR* и *BER* от E_b/N_0 для *D* равном 2 и 3.

Дополнительное задание для самостоятельной работы

1. Разработанная модель будет корректно работать только для BPSK модуляции, поскольку генератор случайных чисел формирует символы, а не биты. Для значений $M>2$ будут повторяться именно символы. Модифицируйте модель для корректной работы с $M>2$.
2. Разработанный помехоустойчивый код обладает относительно низкой эффективностью. Более продуктивным будет объединение процедур демодуляции и декодирования (мягкое декодирование). В этом случай на вход декодера должны поступать не биты с выхода демодулятора, а непосредственно значения отсчетов на входе приемника. Модифицируйте модель для мягкого декодирования и постройте зависимости $BER(E_b/N_0)$ и $BER(SNR)$.

Контрольные вопросы:

1. Как рассчитать значение *BER*
2. Каково может быть максимальное значение *BER*?
3. Каким образом можно декодировать повторенные биты?

Практическая работа № 3

Коды, обнаруживающие ошибки.

Цель работы: Реализация циклического избыточного кода CRC. Проверка целостности пакетов.

Задачи практической работы:

1) Создание функции расчета CRC;
2) Оценка целостности пакетов;
3) Оценка зависимостей BLER от отношения сигнал-шум;

Оборудование и программное обеспечение: Octave, version 6.4.0.

Теоретический материал

Текущая работа является логическим продолжением и расширением предыдущей работы, рисунок 1.

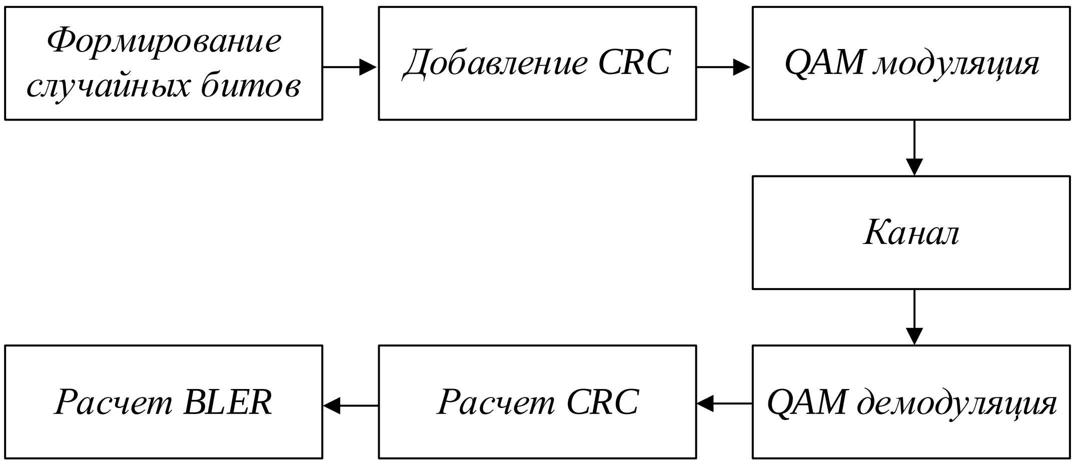


Рисунок 1 – Схема системы, реализуемой в текущей работе

Помехоустойчивые коды можно разделить на две большие группы – коды, которые используются для обнаружения ошибок и коды, которые используются для исправления ошибок. Эта работа посвящена первой группе. Коды, исправляющие ошибки также могут быть использованы для обнаружения ошибок, причем они могут обнаружить большее число ошибок, чем могут исправить.

Простейшей реализацией кодов, обнаруживающих ошибки, является расчет контрольной суммы. Все биты передаваемого пакета складываются, а получившийся результат переводится в двоичный вид. Младшие биты результата и являются рассчитанным значением контрольной суммы. В интерфейсе UART используется этот алгоритм при длине пакета 7 и одном бите контрольной суммы. Например, при значении битов в пакете 0 0 1 1 0 1 1 контрольная сумма будет иметь значение 4, или в двоичном виде 1 0 0, младший бит равен 0, таким образом, передаваемый пакет будет 0 0 1 1 0 1 1 0.

Этот алгоритм также называется расчетом бита четности, поскольку при нечетном количестве единиц в пакете контрольная сумма будет равна 1, а при четном 0.

Контрольная сумма имеет ряд недостатков – минимальное кодовое расстояние равно двум, а значения битов пакета в большей степени влияют на младшие биты контрольной суммы.

Другой подход для контроля целостности (наличия ошибок) передаваемых пакетов – использование циклического избыточного кода CRC (Cyclic redundancy check). Передаваемый пакет с CRC, как и в случае с контрольной суммой состоит из исходного пакета (так называемая систематическая часть) и проверочных битов, расположенных справа. При расчете CRC используется полиномиальная форма записи векторов бит. Правый бит вектора соответствует коэффициент полинома $x^0=1$, а каждый следующий бит на i-ой позиции справа обозначается как x^i. Например, вектору 1 1 1 0 1 1 соответствует полином

$$P(x) = 1x^5+1x^4+1x^3+0x^2+1x^1+1x^0 = x^5+x^4+x^3+x^1+1.$$

Умножение полинома на x^N эквивалентно добавлению N нулей справа исходного вектора, например:

$$P(x)x^2 = x^7+x^6+x^5+x^3+x^2,$$

что соответствует вектору 1 1 1 0 1 1 0 0.

Для расчета CRC необходимо рассчитать значение

$$R(x) = P(x)x^N \bmod G(x),$$

где R(x) – остаток от деления – значение CRC, добавляется к передаваемому сообщению;

P(x) – полином, соответствующий сообщению;

G(x) – порождающий полином;

N – степень порождающего полинома.

Наглядный способ выполнения операции деления – деление полиномов в столбик. В этом случае полиномы представляются в виде 0 и 1, а вычисления выполняются как при делении обычных чисел. Сложение и вычитание 1 выполняется по модулю 2 без переноса, пример такого расчета приведен на рисунке 2, сообщение 1 1 1 0 1 1, $P(x) = x^5+x^4+x^3+x^1+1$, $G(x) = x^5+x^3+1$. Порождающий полином имеет степень N = 5, тогда $P(x)x^5 = x^{10}+x^9+x^8+x^6+x^5$. В результате вычислений остаток от деления R(x) = 0 1 1 1 1. Передаваемый пакет с CRC в этом случае будет равен 1 1 1 0 1 1 0 1 1 1 1.

```
1 1 1 0 1 1 0 0 0 0 0 | 1 0 1 0 0 1
1 0 1 0 0 1           | 1 1 0 1 1 1
----------------------
  1 0 0 1 0 0
  1 0 1 0 0 1
----------------------
    0 1 1 0 1 0
    0 0 0 0 0 0
----------------------
      1 1 0 1 0 0
      1 0 1 0 0 1
----------------------
        1 1 1 0 1 0
        1 0 1 0 0 1
----------------------
          1 0 0 1 1 0
          1 0 1 0 0 1
----------------------
            0 1 1 1 1
```

Рисунок 2 – Пример вычисления CRC при делении в столбик

При реализации алгоритма операцию деления в столбик можно заменить вычислением на сдвиговом регистре, для порождающего полинома $G(x) = x^5+x^3+1$, схема кодера приведена на рисунке 3.

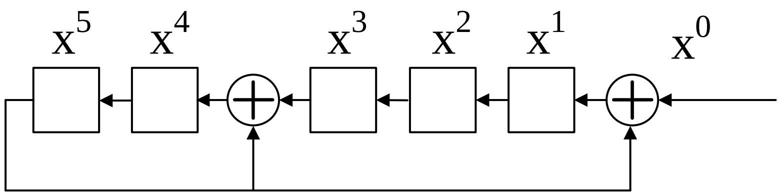


Рисунок 3 – Схема кодера CRC с $G(x) = x^5+x^3+1$

В первые пять тактов биты сообщения просто заполняют разряды сдвигового регистра, рисунок 4.

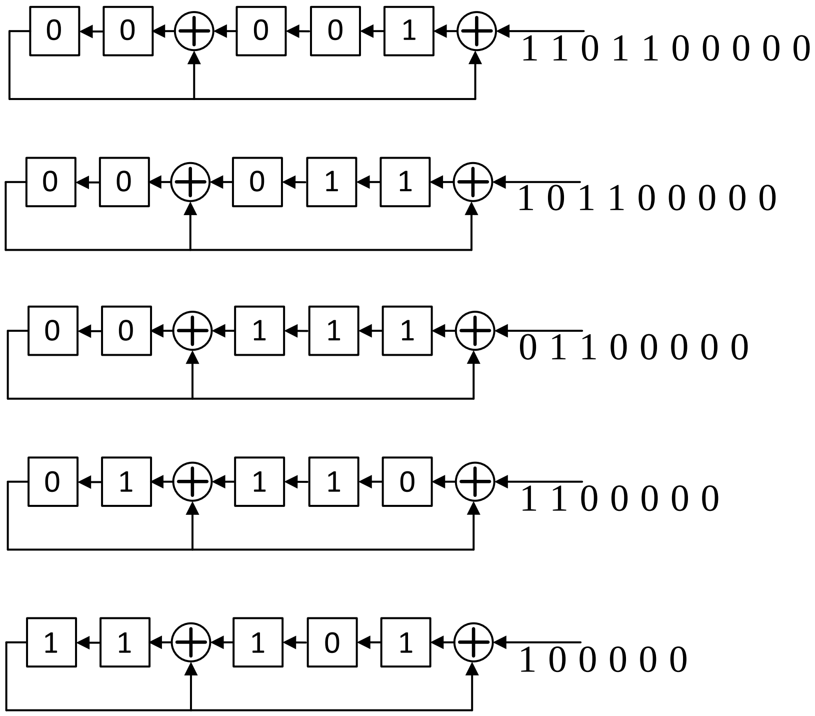


Рисунок 4 – Первые пять тактов работы генератора

После чего начинается последовательный сдвиг и расчет значений, при этом обратите внимание, что проводимые операции эквивалентны делению полиномов, рисунок 5.

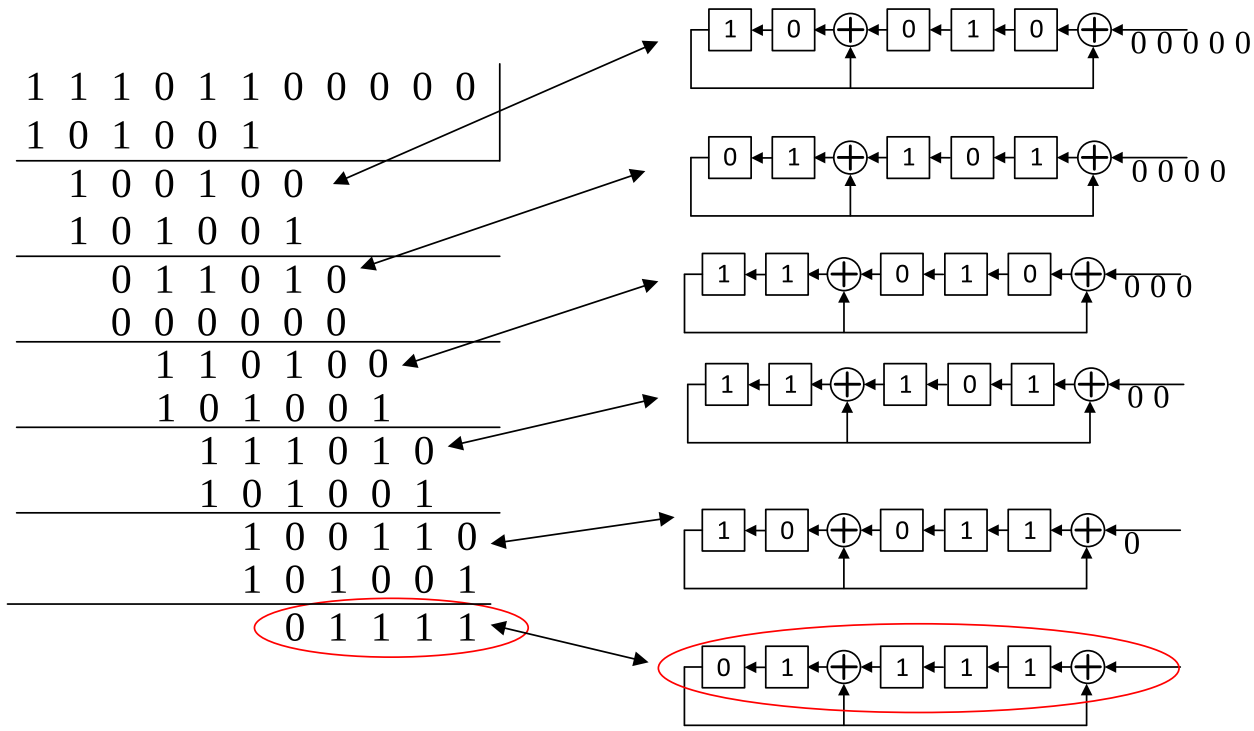


Рисунок 5 – Расчет значения CRC в течении оставшихся тактов

Как контрольная сумма, так и CRC позволяют рассчитывать значение BLER.

Ход работы

В ходе этой работы вам необходимо реализовать модель, схема которой приведена на рисунке 1. Алгоритм CRC должен работать с полиномом $G(x) = x^5+x^3+1$ на основе схемы, приведенной на рисунке 3. По CRC на стороне приемника необходимо вести расчет *BLER*.

Основой работы будет созданная в течении прошлых занятий модель. Добавьте к параметрам значение количества битов в пакете, например:

L   = 100;

Параметр *N* в этой модели будет означать количество пакетов.

Основные алгоритмы также как и в прошлой работе выполняются в глобальном цикле, с изменением переменной *snr_i* в диапазоне *SNR(1):SNR(2)*. В начале каждой итерации необходимо обнулять счетчик ошибок

err = 0;

И далее, объявить цикл от 1 до *N*, в каждой итерации которого будет создаваться, кодироваться, приниматься и проверяться новый пакет битов:

for n = 1:N

На первом этапе создается вектор битов длиной *L*

bits   = randi([0 1],L,1);

После чего выполняется расчет контрольной суммы. Для этого создайте функцию *crc_gen*. На первом этапе объявите степень порождающего полинома

N_G   = 5;

Добавьте к вектору битов *N_G* нулей справа (аналог операции $P(x)x^5$).

bits   = [bits; zeros(N_G,1)];

Объявите сдвиговый регистр *sh_reg* (в виде вектора) и заполните его первыми битами сообщения:

sh_reg = zeros (N_G);

$$sh\_reg \ = bits(N\_G{:}{-1}{:}1);$$

Обратите внимание, что биты записываются в обратном порядке, поскольку первый бит в первый такт поступает в первый разряд, а к пятому такту поступает в пятый разряд, в то время как в первом будет пятый бит.

После этого необходимо объявить цикл, в котором индекс текущего входного бита *i* изменяется от *N_G+1* до последнего бита:

for i = N_G+1:length(bits)

Каждый такт вычисления выполняются в соответствии со схемой на рисунке 3:

```
x_bit           =   sh_reg(5);
sh_reg(5)       =   sh_reg(4);
sh_reg(4)       =   xor(x_bit,sh_reg(3));
sh_reg(3:-1:2)  =   sh_reg(2:-1:1);
sh_reg(1)       =   xor(x_bit,bits(i));
```

На этом одна итерация заканчивается. Выход функции – значение сдвигового регистра в последней итерации, считанное в обратном порядке.

crc = sh_reg(end:-1:1);

Подключите функцию в модель и рассчитайте контрольную сумму:

crc_t = crc_gen (bits);

Добавьте рассчитанное значение в конец пакета:

bits_crc = [bits, crc_t];

После этого выполняются процедуры модуляции, прохождения через АБГШ (awgn) канал и демодуляции. Биты на выходе демодулятора поступают на блок расчета контрольной суммы в приемнике (для этого используется тот же алгоритм и та же функция, как и в передатчике):

crc_r = crc_gen (r_bits_crc);

После этого проверяется, равна ли контрольная сумма нулю, если нет, то к счетчику ошибок пакетов добавляется 1:

if sum(crc_r) ~= 0

err = err+1;

end

На этом итерация по передаче одного пакета заканчивается.

После выполнения *N* итераций необходимо рассчитать значение *BLER(k)*:

$$BLER(k) \quad = err/N;$$

Для отчета по проделанной работе сделайте скриншоты зависимости *BLER* от *SNR* для QPSK модуляции.

Дополнительное задание для самостоятельной работы

1. Сделайте функцию универсального генератора CRC, на входы которой будут подаваться биты сообщения и значение полинома G(x) в виде вектора из 0 и 1.

Контрольные вопросы:

1. Как рассчитать значение BLER?
2. Каково может быть максимальное значение BLER?
3. Чем CRC отличается от классической контрольной суммы?
4. Как приемник может проверить наличие ошибок в пакете.

Практическая работа № 4

Сверточный кодер.

Цель работы: Реализация алгоритма сверточного кодирования.

Задачи практической работы:

1) Изучение теории сверточного кодирования;
2) Реализация сверточного кодера;

Оборудование и программное обеспечение: Octave, version 6.4.0.

Теоретический материал

В рамках текущей работы необходимо реализовать алгоритм сверточного кодирования и изучить основные теоретические вопросы.

Сверточные коды – один из основных классов помехоустойчивого кодирования (наряду с блочными коды). Как правило, алгоритмы кодирования сверточных кодов строятся на основе сдвигового регистра, рисунок 1.

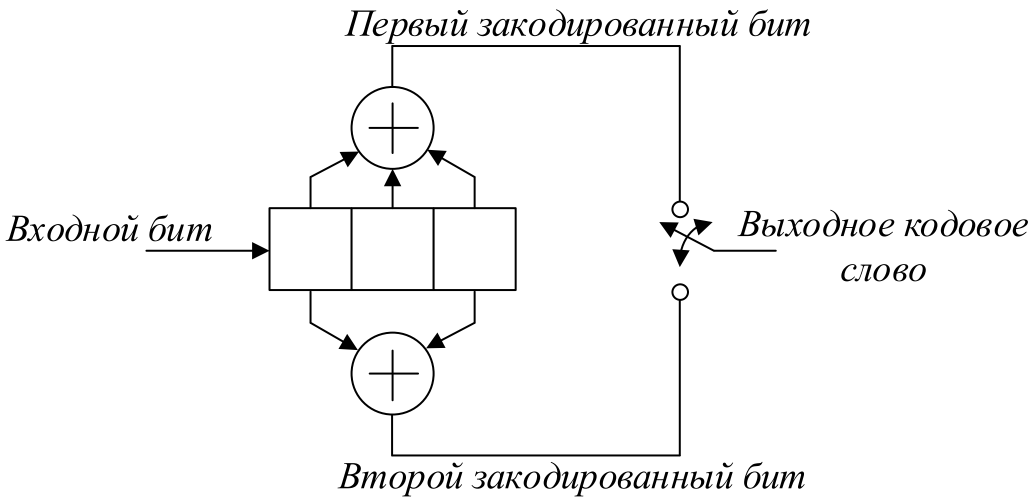


Рисунок 1 – Пример реализации сверточного кодера

Основными параметрами сверточных кодов являются k – количество битов на входе (обычно, k равно 1, реже до 3)., n – количество битов на выходе, R = k/n – скорость кодирования, L – длина кодового ограничения (количество разрядов сдвигового регистра). Соединение сумматоров с разрядами сдвигового регистра задаются векторами связи (порождающими полиномами). В стандартах они, как правило, записаны в виде чисел в восьмеричной системе счисления, перевод которых в двоичную даст вектор из 0 и 1 длиной L.

Важно запомнить, что сверточный кодер является цифровым устройством с памятью, т.е. значение на выходе зависит не только от текущего значения на входе, но и от L-1 предыдущих входных значений. За каждый такт из k бит на входе формируется n бит на выходе.

В примере кодера, приведенном на рисунке 1 k = 1, n = 2, R = 1/2, L = 3, векторы связи G1 = 7_8 (1 1 1), G2 = 5_8 (1 0 1). На рисунке 2 приведен пример работы кодера при подаче на его вход вектора [1 0 1 1].

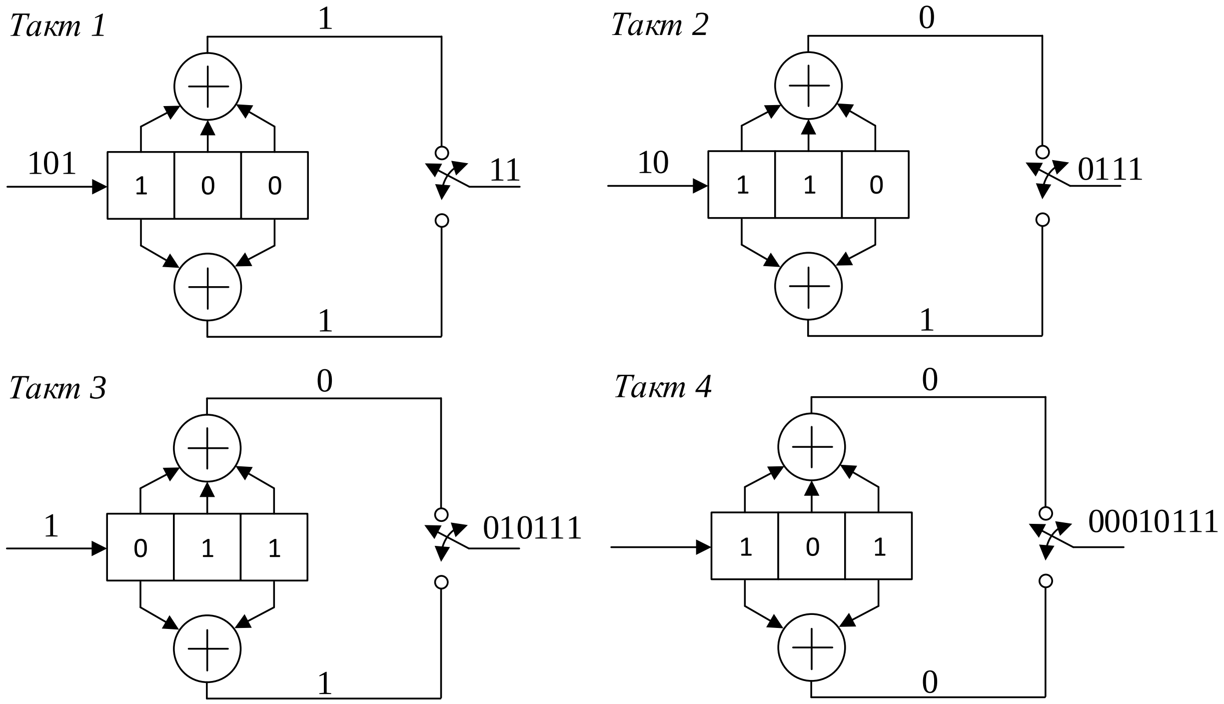


Рисунок 2 – Пример работы сверточного кодера на протяжении первых четырех тактов

Работу кодера можно также описать с помощью древовидно диаграммы, рисунок 3.

Работу кодера можно описать другими способами, например, с помощью древовидной диаграммы (рисунок 3).

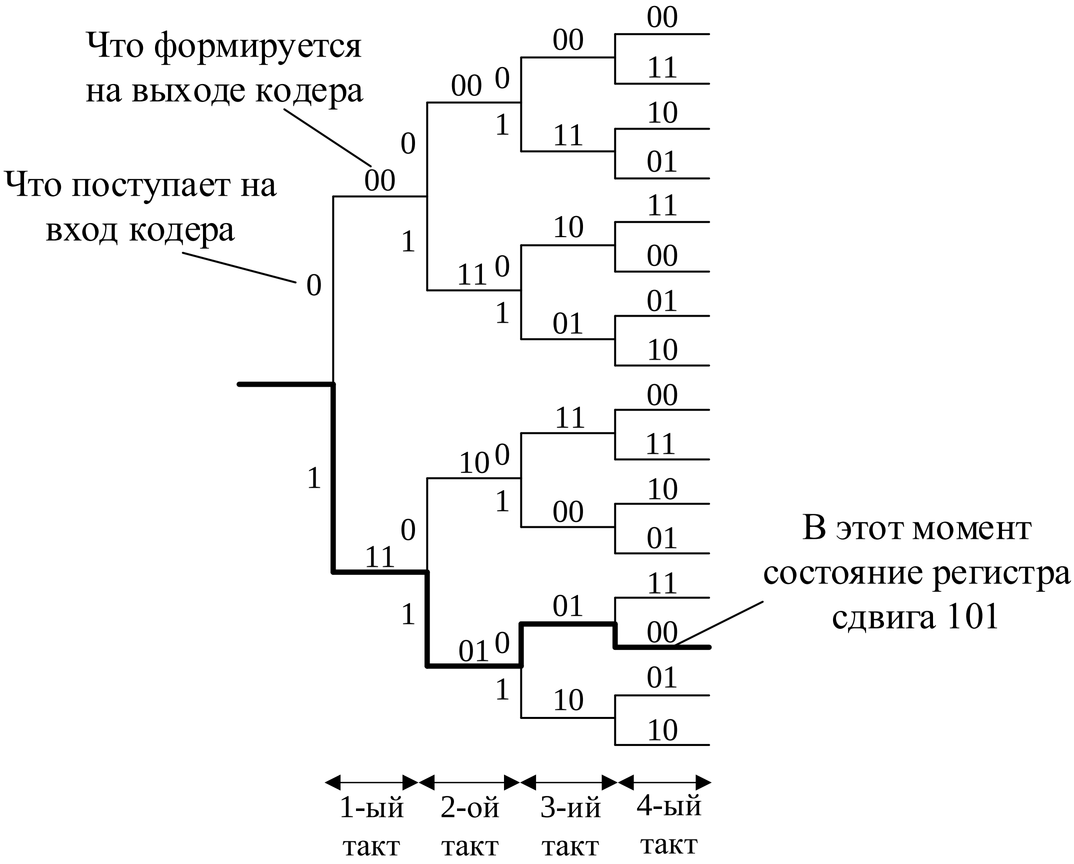


Рисунок 3 – Древовидная диаграмма сверточного кодера

Древовидная диаграмма характеризует всевозможные выходные значения битов кодера, при условии поступления на его вход всевозможных входных битов. Состояние кодера в первый момент времени (до подачи на него битов) соответствует левой точке диаграммы. При подаче на его вход первого бита возможны два варианта выходных значений: 0 0, если значение бита 0 (значение бита 0 всегда соответствует верхней ветви дерева) и 1 1, если значение бита 1 (значение бита 1 всегда соответствует нижней ветви дерева). Далее, во втором такте, в зависимости от значения следующего бита возможны уже четыре варианта комбинации выходных битов (поскольку значения на выходе кодера зависят не только от текущего входного бита, но и от предыдущих).

На рисунке 3 жирными линиями выделен путь кодирования, соответствующей битовой комбинации 1 0 1 1 на входе (при кодировании начиная с конца вектора). Следует отметить, что зная путь, построенный на этом дереве можно однозначно определить последовательность на входе кодера, а значит построить декодер.

С каждым тактов количество ветвей древовидной диаграммы растет в два раза, что является основным недостатком такого представления процесса кодирования. Вместе с тем, можно заметить, что определенные участки дерева совпадают друг с другом, рисунок 4.

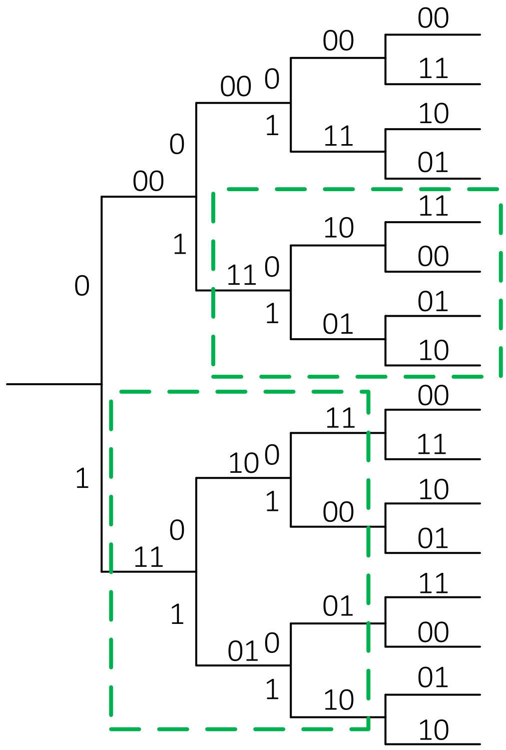


Рисунок 4 – Повторяющиеся участки древовидной диаграммы

Благодаря такой структуре древовидную диаграмму можно сократить и представить в виде решетчатой диаграммы, приведенной на рисунке 5.

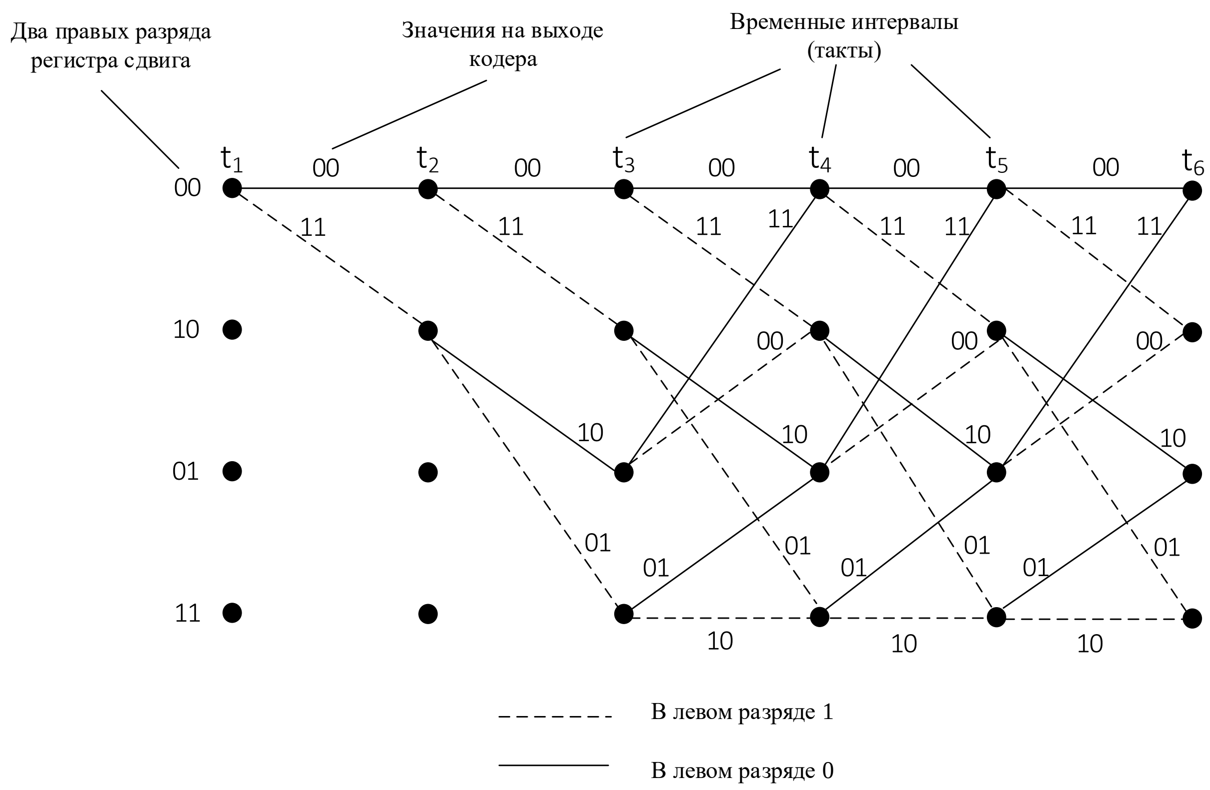


Рисунок 5 – Решетчатая диаграмма сверточного кода

Поскольку выходные значения сверточного кодера полностью определяются состоянием сдвигового регистра, для кодера с L=3 может быть 8 возможных выходных значений (существует 8 комбинаций его состояний от 0 0 0 до 1 1 1). Следующее состояние зависит от того, какой бит поступает на вход кодера. Идея решетчатой диаграммы состоит в том, что вместо дерева строится $2^{(L-1)}$ возможных состояний L-1 его правых разрядов (горизонтальные ряды точек диаграммы). Для L=3 это 4 состояния: 00, 10, 01, 11. Между этими состояниями установлено однозначное соответствие с помощью ветвей диаграммы. Они показывают, в какое состояние можно перейти из каждого предыдущего при поступлении в левый разряд кодера значения 1 (непрерывные линии) и 0 (пунктирные линии). Над каждой ветвью написан вектор из двух бит – это выходные биты кодера, формируемые при данном переходе. Например, в точку первой линии в момент t_4 (четвертый такт) можно перейти либо из состояния 00, либо 01, в которых мог находиться сдвиговый регистр в момент t_3. Преимуществом решетчатой диаграммы является постоянное количество ветвей, не зависящее от длины кодируемого сообщения.

На рисунке 6 приведен пример построения решетчатой диаграммы для кодирования вектора 1 0 1 1.

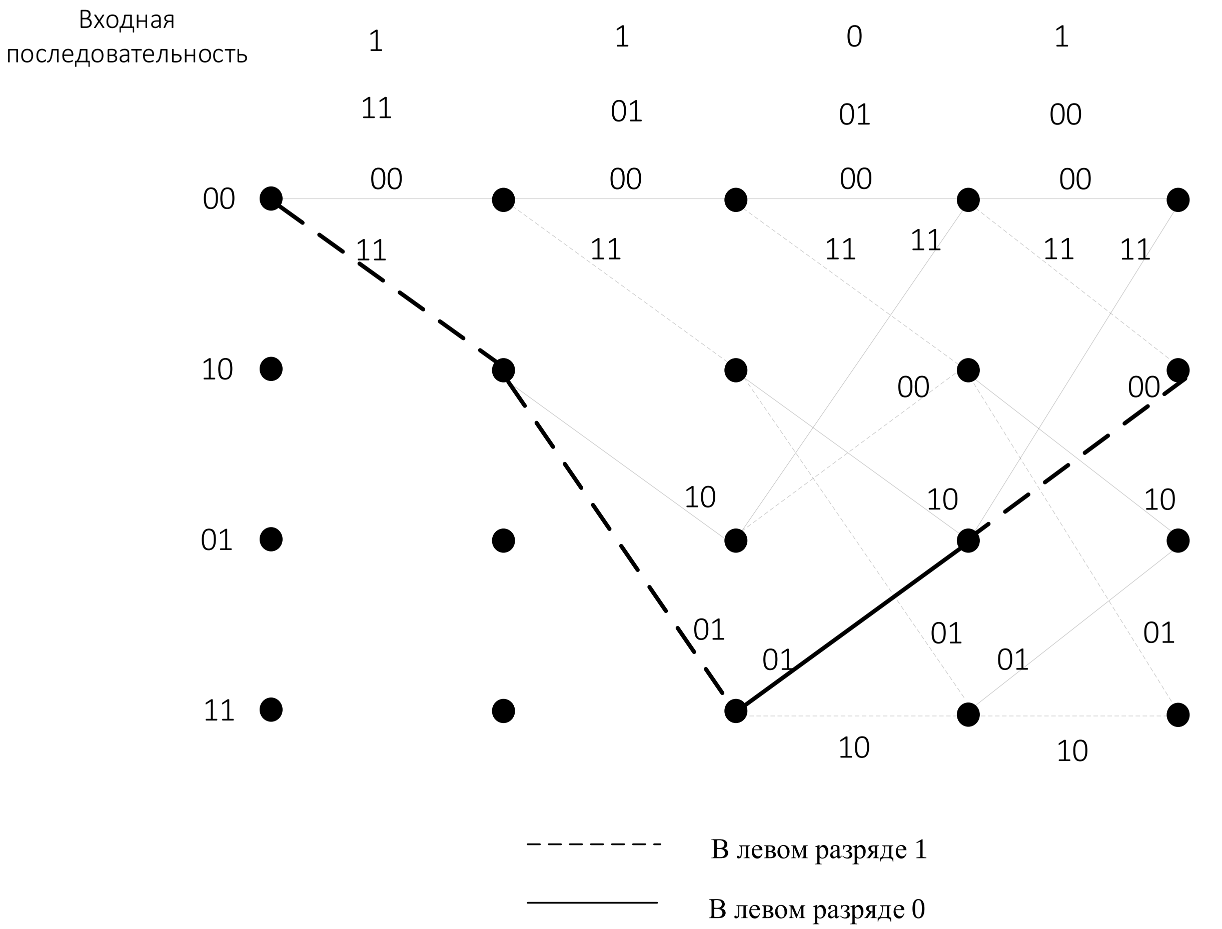


Рисунок 6 – Построение решетчатой диаграммы

По сколько сверточный кодер по своей сути является автоматом конечных состояний, то его работу можно проиллюстрировать в виде диаграммы состояний (рисунок 7).

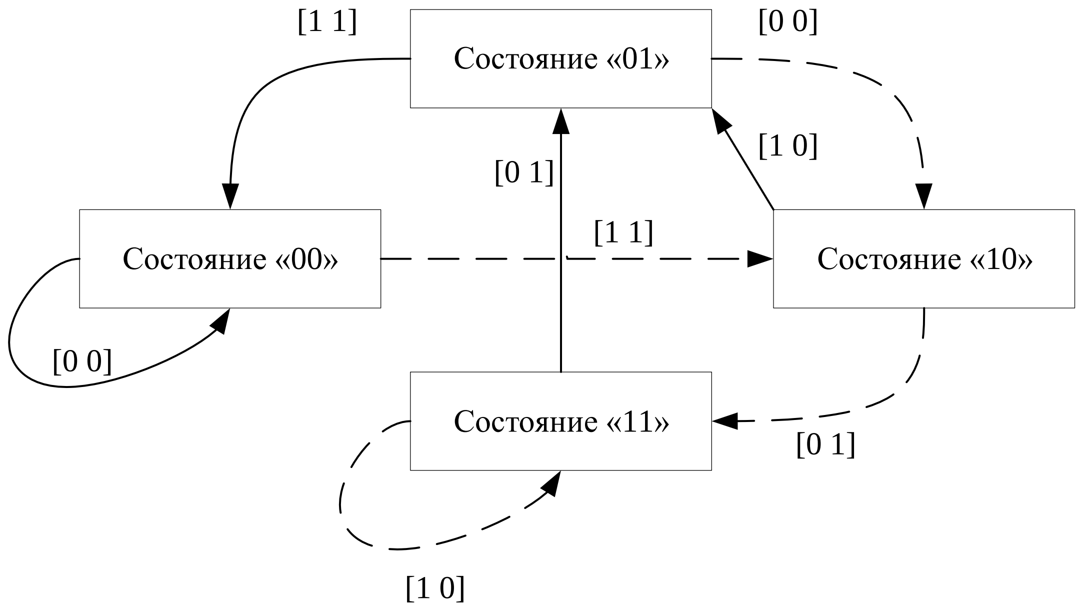


Рисунок 7 – Диаграмма состояний кодера

На рисунке 7 представлена диаграмма состояний для кодера, рассмотренного ранее. На данном рисунке пунктирными линиями показаны входные биты, равные «1», сплошными линиями «0». В блоках под «состоянием» подразумеваются два правых разряда сдвигового регистра. В квадратных скобках показаны выходные биты кодера.

Ход работы

В рамках этой работы вам предстоит создать сверточный кодер, схема которого приведена на рисунке 1.

Создайте функцию *conv_coder* со входным аргументом *input_bits* и возвращаемым значением *coded_bits*.

Создайте вектор сдвигового регистра длиной 3

shift_reg = zeros(3,1);

Создайте пустой вектор выходных битов

coded_bits = [];

Далее, в цикле от 1 до длины вектора входных битов

for i = 1:length(input_bits)

Необходимо реализовать последовательный сдвиг значений

shift_reg(3:-1:2) = shift_reg(2:-1:1);

shift_reg(1) = input_bits(i);

И добавление к выходному вектору значений, поступающий с сумматоров. Эти значения могут быть получены с помощью операции *XOR* – сложению по модулю 2, либо, путем простого сложений и взятия остатка от деления суммы на 2.

dibit=[xor(xor(shift_reg(3),shift_reg(2)),shift_reg(1))
xor(shift_reg(3),shift_reg(1))];

coded_bits = [coded_bits, dibit];

На этом итерация заканчивается.

Создайте сценарий main, в котором сгенерируйте *N* бит и подайте их на вход сверточного кодера. Убедитесь, что кодер работает, а на его выходе формируется вектор длиной 2*N*.

Для отчета по проделанной работе выведите в командную строку закодированную последовательность битов, при подаче на кодер сообщения [0 1 1 0 1 0 0 1] и сделайте скриншот.

Дополнительное задание для самостоятельной работы

1. Сделайте универсальный сверточный кодер со входными аргументами *bits*, *L* (длина кодового ограничения) и *G1*, *G2* (векторы связи).

Контрольные вопросы:

1. Найдите все повторяющиеся участки дерева на рисунке 4.
2. В чем отличие древовидной и решетчатой диаграмм?
3. Как построить древовидную диаграмму кодера?
4. Можно ли считать количество битов на выходе кодера длиной кодового слова?

Практическая работа № 5
Декодер Витерби.

Цель работы: Реализация алгоритма сверточного декодирования.

Задачи практической работы:

1) Изучение теории декодера Витерби;
2) Реализация декодера Витерби;
3) Построение имитационной модели системы связи со сверточным кодеком;
4) Изучение помехоустойчивости системы связи со сверточным кодеком.

Оборудование и программное обеспечение: Octave, version 6.4.0.

Теоретический материал

В рамках текущей работы необходимо реализовать алгоритм сверточного декодирования и встроить его в разработанную ранее имитационную модель, приведенную на рисунке 1.

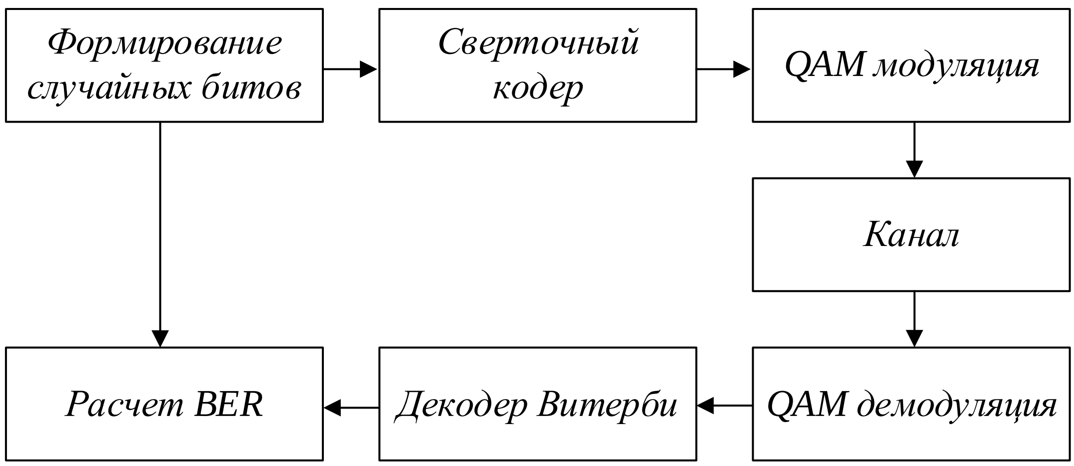


Рисунок 1 – Схема реализуемой в текущей работе модели

Логика построения модели совпадает с моделью системы с кодированием повторением. На входе формируется вектор случайных битов, который кодируется и отображается в QAM символы. После канала связи и демодуляции биты поступают на декодер Витерби. Декодированные биты сравниваются с переданными, рассчитывается BER и строится его зависимость от SNR.

Классический декодер основан на принципе максимального правдоподобия, которое сводится к выражению

$$\mathrm{decoded} = \max\left(P(Z|U^m)\right),\ m = 1 \dots 2^N,$$

где $P(Z|U^m)$ – вероятность того, что последовательность битов после демодулятора Z совпадает со всевозможными закодированными последовательностями U^m. Нетрудно заметить, что прямая реализация этого принципа будет сводиться к сравнению Z с 2^N всевозможными закодированными последовательностями, где N – их длина. Разумеется, такой декодер имеет колоссальную вычислительную сложность и не может быть реализован на практике при большой длине пакетов.

Одним из возможных путей снижения вычислительной сложности является декодер Витерби. Этот декодер основан на анализе решетчатой диаграммы. Схема кодера приведена на рисунке 2, а соответствующая ей решетчатая диаграмма на рисунке 3.

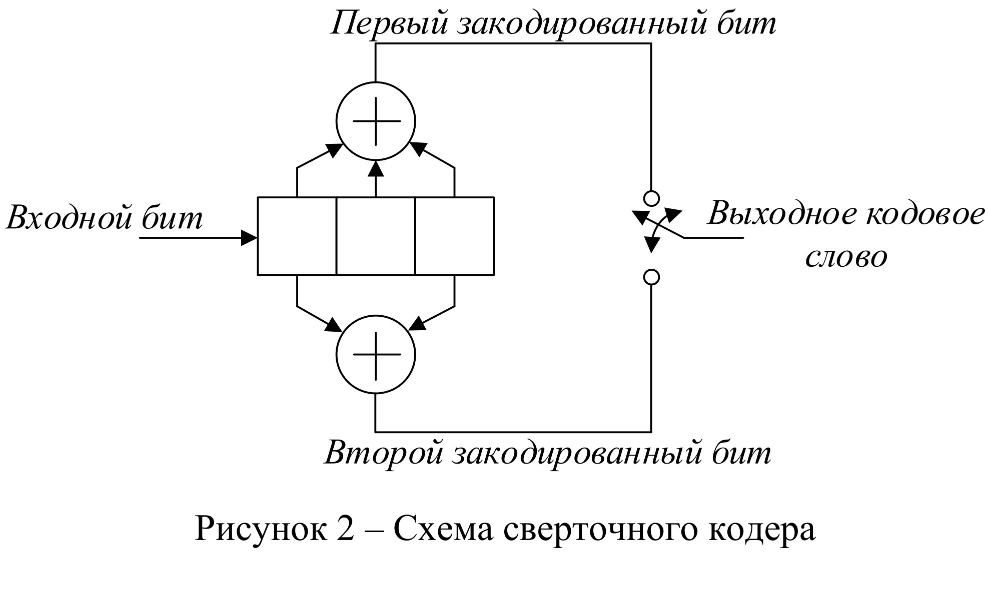


Рисунок 2 – Схема сверточного кодера

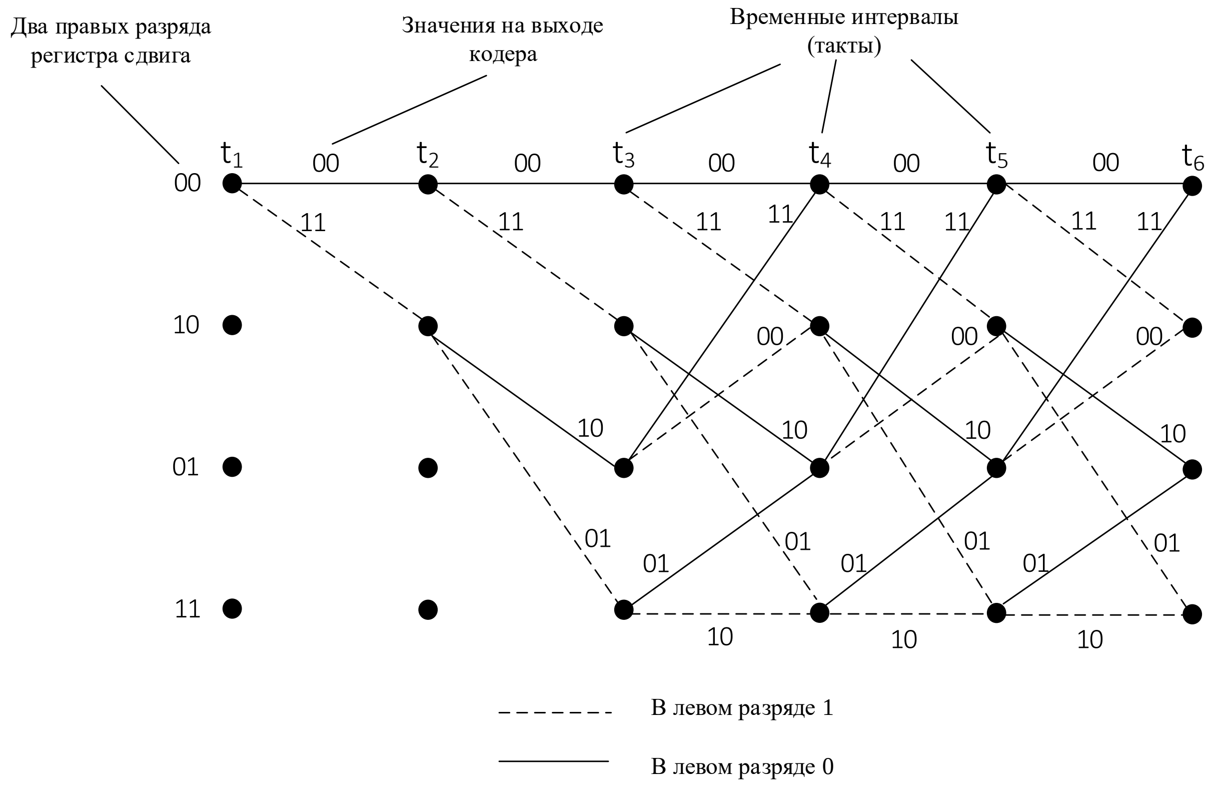


Рисунок 3 – Решетчатая диаграмма сверточного кода

Выходные значения кодера зависят от значения битов в сдвиговом регистре, при длине сдвигового регистра L = 3 возможны $2^L = 2^3 = 8$ состояний. На решетчатой диаграмме в виде точек отображаются возможные состояния двух правых разрядов сдвигового регистра, точки соединены друг с другом ветвями (возможными переходами). Пунктирная линия означает, что в левом разряде находится ноль, а непрерывная соответствует единицей. Так, после первого такта, переход из состояния 00 может быть либо в 00, в этом случае на выходе кодера будет пара бит 00, либо в 10 (если в левом разряде была 1), в этом случае на выходе будет 11. Начиная с третьего такта возможны 8 состояний, каждое из которых отображается на диаграмме.

В процессе декодирования сообщение разбивается на пары бит (так называемые дибиты) соответствующие временным тактом. В каждом такте пара бит сравнивается с всевозможными значениями на выходе декодера, и в хэмминговом пространстве рассчитываются веса (количество несовпадающих битов), рисунок 4.

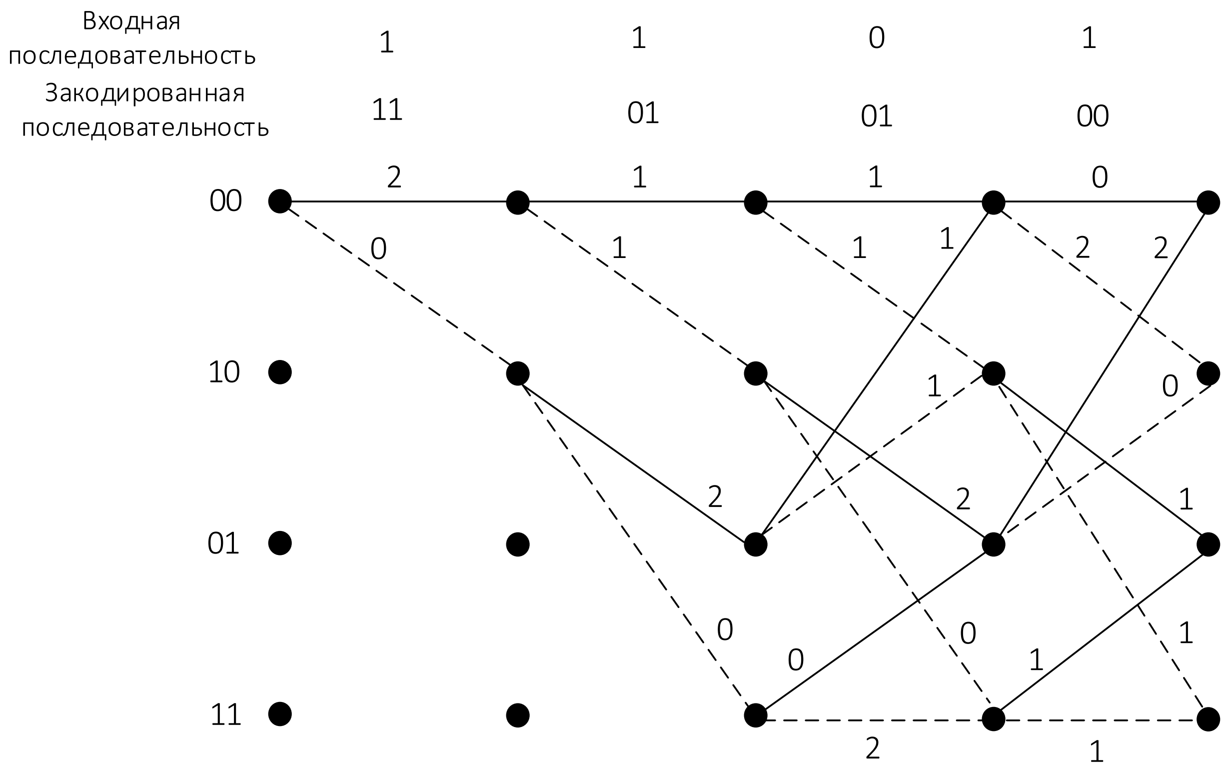


Рисунок 4 – Расчет весов на решетчатой диаграмме

После прохождения всех тактов, для каждого финального состояния сдвигового регистра рассчитывается метрика – минимальная сумма всех весов во всевозможных путях от начала до конца. На рисунке 5 приведены метрики для каждой результирующей точки. Путь с минимальной метрикой считается правильным и по нему восстанавливаются переданные биты (они соответствуют значениям битов в левом разряде, или типу ветвей на диаграмме). В случае, если получилось несколько путей с одинаковыми метриками может быть выбран любой из них, т.к. с точки зрения вероятности они будут равновероятны. Данный момент является особенностью декодирования по алгоритму Витерби.

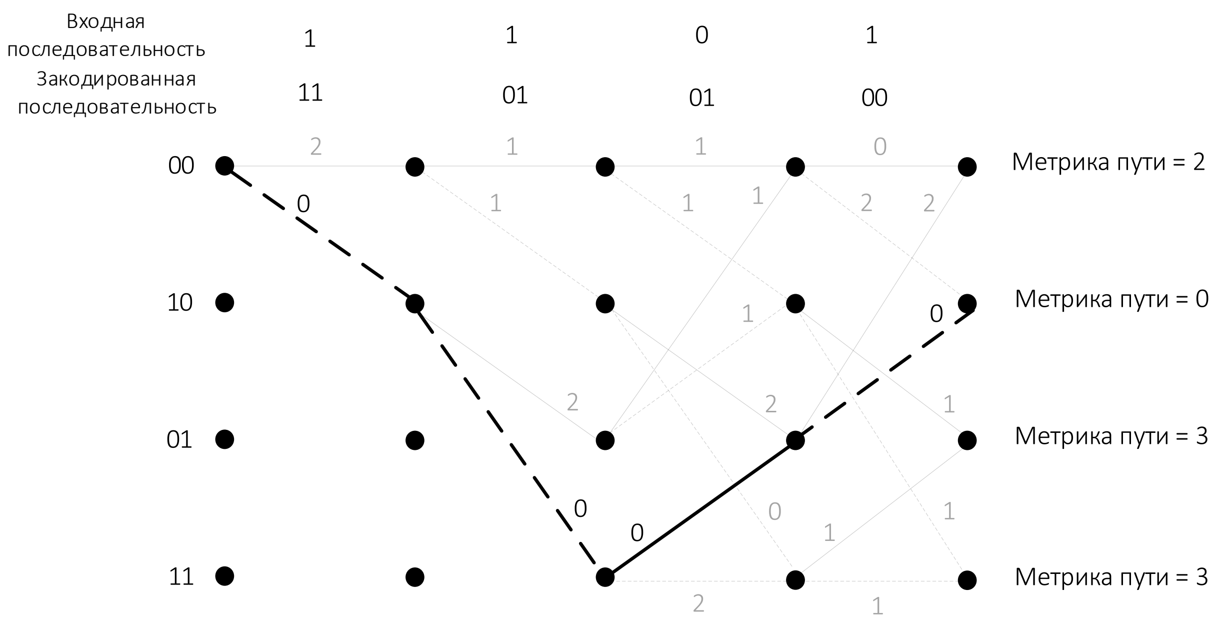


Рисунок 5 – Расчет метрики пути

При построении алгоритма напрямую, как и в случае с декодером, построенным по критерию максимального правдоподобия, в памяти необходимо держать 2^N путей и метрик. Однако, основываясь на решетчатой диаграмме можно заметить, что если в каждую точку приходит два пути, то один из них является более вероятным, а второй можно отбросить как менее вероятный, рисунок 6. Следующие состояния не зависят от предыдущих путей, а только от состояния сдвигового регистра, и поэтому, в каждом следующем такте достаточно знать минимальную метрику для каждого состояния и соответствующий ей путь.

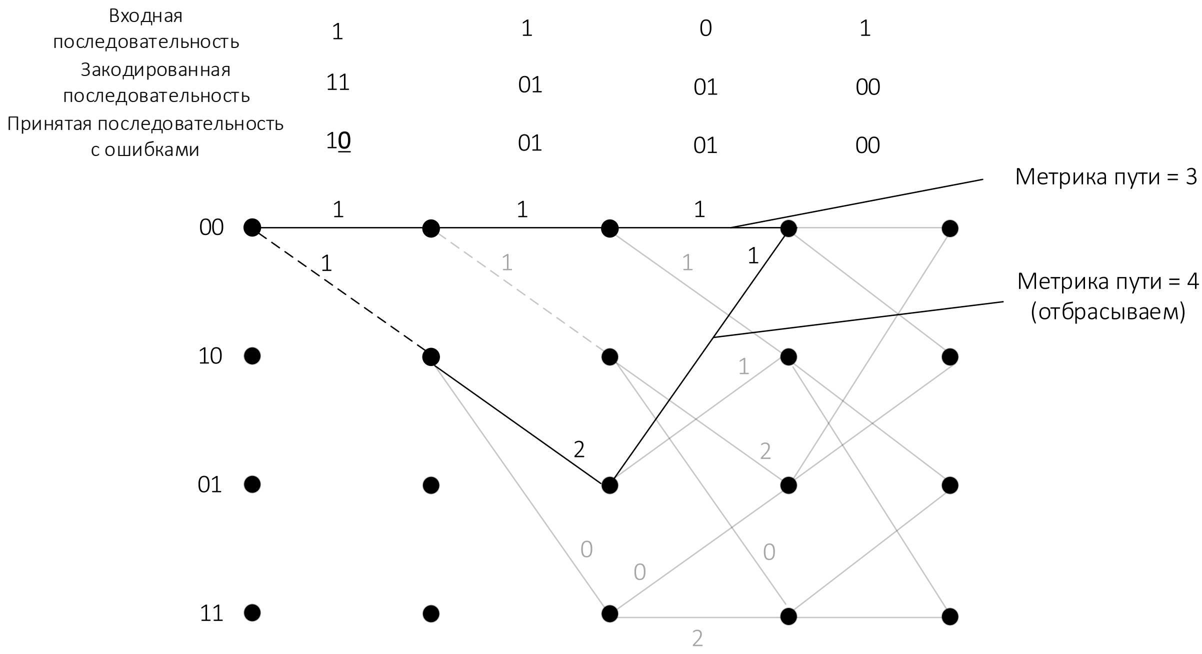


Рисунок 6 – Отбрасывание менее вероятного пути

На рисунке 7 приведен пример построения путей и расчета метрик в случае появления битовой ошибки в принятой последовательности.

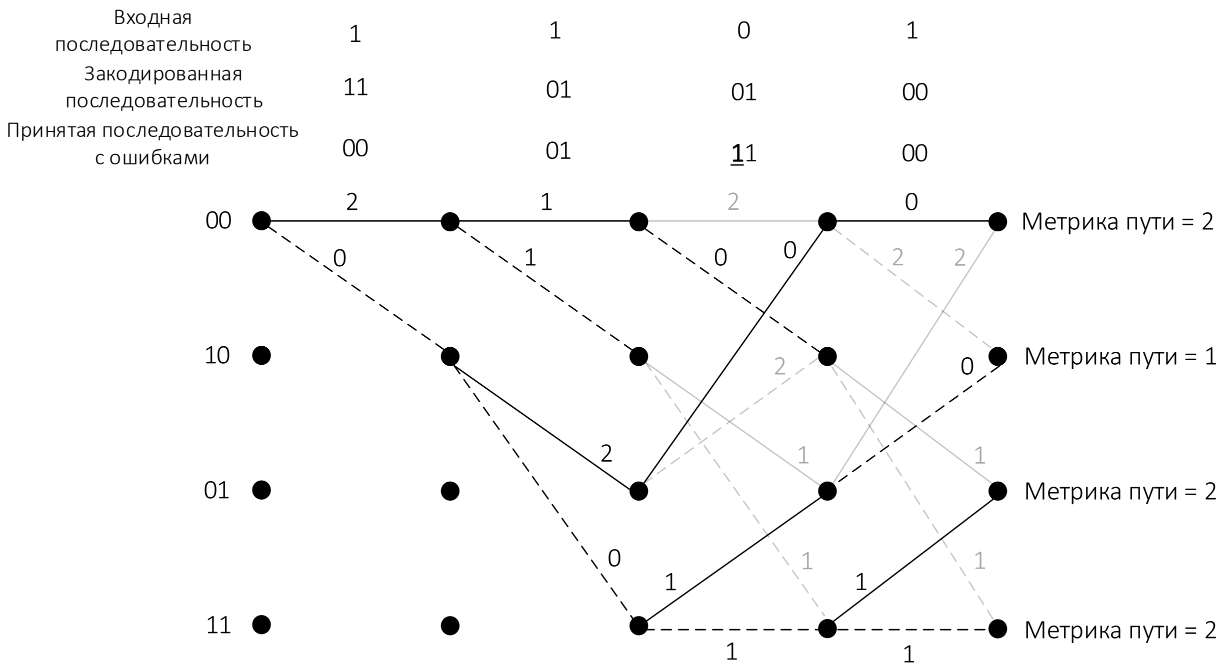


Рисунок 7 – Построение метрик пути при наличии ошибок

Ход работы

Основной частью работы будет реализация декодера Витерби.

Создайте функцию декодера Витерби и назовите ее *conv_decoder*, выходное значение *decoded_bits*, входное *bits*.

Прежде всего задайте переменную *N*, которая соответствует длине декодированного сообщения (половине длины кодированного):

N = length(bits)/2;

Далее, создайте массив в строках которого будут записаны узлы решетчатой диаграммы:

```
nodes  = [ 0 0;
           1 0;
           0 1;
           1 1];
```

Далее, создайте массив со связями, в строки (соответствующие узлам) которого запишите номера узлов в предыдущем такте, с которыми соединен текущий узел:

```
connects   = [ 1 3
               1 3
               2 4
               2 4];
```

Создайте следующий массив, который соответствует выходным значениям, формируемым при прохождении через ветвь, указанную в предыдущем массиве:

```
in_values  = [ [0 0], [1 1]
               [1 1], [0 0]
               [1 0], [0 1]
               [0 1],[1 0] ];
```

Обратите внимание, что здесь создается массив размером 4,4.

Далее, объявляем пустой вектор, в котором будем хранить значения метрик:

```
metrics        = zeros(4,1);
```

Поскольку исходное состояние регистра 00, для иных состояний искусственно завышаем метрику и тем самым делаем их менее вероятными:

```
metrics(2:4)    = 2;
```

В данном случае задается значение 2, но может быть указано любое другое.

Также задаем массив путей:

```
way=ones(4,1);
```

Каждый такт этот вектор будет увеличиваться в 2 раза (превратится в матрицу размером 4 на *n*, где *n* – номер такта).

После этого в цикле от 1 до *N* начинаем анализировать входной вектор закодированных битов:

```
for i = 1:N
```

Выделяем текущий дибит:

```
dibit = bits(2*(i-1)+1:2*i);
```

И анализируем каждый из узлов в цикле от 1 до 4:

```
for k = 1:4
```

Считаем веса для каждой входящей в узел ветви:

```
ves(k,1)   = sum(xor(in_values(k,1:2), dibit));
ves(k,2)   = sum(xor(in_values(k,3:4), dibit));
```

Новая метрика для этого узла – наименьшая из двух входящих. А каждая из входящих равна сумме метрики связанного предыдущего узла и веса ветви. Выбираем наименьшую, записываем ее в вектор новых метрик (*metrics_new*) и сохраняем путь в обновленный вектор:

```
if metrics(connects(k,1)) + ves(k,1) < metrics(connects(k,2)) + ves(k,2)
way_new(k,1:i) = [way(connects(k,1),1:i-1),connects(k,1)];
metrics_new(k,1)  = metrics(connects(k,1)) + ves(k,1);
else
way_new(k,1:i) = [way(connects(k,2),1:i-1),connects(k,2)];
metrics_new(k,1)  = metrics(connects(k,2)) + ves(k,2);
end
```

Это основной шаг алгоритма, связанный с обновлениями метрик и отбрасыванием наименее вероятных путей. Тщательно проанализируйте его.

На этом итерация для всех четырех заканчивается. Далее, записываем обновленные вектора метрик и путей в текущие:

metrics = metrics_new;

way = way_new;

И далее рассматриваем следующий дибит.

На этом основной цикл заканчивается. Находим номер минимальной метрики:

[~,num_min] = min(metrics);

И считаем, что он соответствует верному пути, который также обновляем. Первое значение вектора *way* отбрасывается (т.к. оно соответствует моменту до первого такта), а в последнюю позицию записываем номер текущего узла нашего пути.

way_min = [way(num_min,2:end) num_min];

Правильная последовательность исходных битов определяется как значение в левом разряде узла в каждой итерации (там находятся входные биты в n+1 такте):

for i=1:N

decoded_bits(1,i) = nodes(way_min(i),1);

end

Реализуйте модель, построенную по схеме, приведенной на рисунке 1. Моделирование проведите для *SNR* в диапазоне от 0 до 15 дБ. Постройте графики зависимостей *BER* (*SNR*). Подробное описание реализации этой части работы приведено в методичке, посвященной кодированию повторением. Сравните помехоустойчивость системы связи со сверточных кодированием с некодированной передачей и с кодированием повторением.

Для отчета по проделанной работе сделайте скриншоты зависимостей BER от SNR для сверточного кода и незакодированного сообщения.

Дополнительное задание для самостоятельной работы

1. При реализации декодера Витерби, описанной выше в памяти хранится массив путей 4 на *N*. Однако, за счет того, что наименее вероятные пути отбрасываются, к последней итерации большая часть первых путей для всех четырех узлов будет совпадать, и таким образом, длину этого массива можно сократить. Реализуйте такое сокращение для сохранения в памяти не более путей глубиной не более 10.

2. Сделайте универсальный декодер Витерби со входными аргументами *bits*, *L* (длина кодового ограничения) и *G1*, *G2* (векторы связи).

Контрольные вопросы:

1. Как рассчитываются веса путей на решетчатой диаграмме?
2. Каким образом отбрасываются наименее вероятные пути на решетчатой диаграмме?
3. Как зная путь восстановить исходную последовательность битов?
4. Как связано количество узлов древовидной диаграммы и длина сдвигового регистра?

Практическая работа № 6
Турбокоды.

Цель работы: Исследование системы связи с турбокодами.

Задачи практической работы:

1) Изучение теории турбокодов;
2) Реализация модели системы связи с турбокодами на основе готовых функций.

Оборудование и программное обеспечение: Octave, version 6.4.0.

Теоретический материал

Большинство разработанных помехоустойчивых кодов являются наиболее эффективными только в случае статистически независимых ошибок (разнесенных, независимых ошибок). Но реальная среда распространения может вызвать не единичные, а пакетные ошибки. Под пакетной ошибкой понимается N неправильно демодулированных символов, стоящих на соседних позициях.

Сверточные коды являются эффективным классом помехоустойчивых кодов, которые способны исправлять большое число ошибок. В то же время для них критически важно положение возникших ошибок – сверточные коды не способны бороться с групповыми ошибками. Это объясняется самой структурой сверточного кодера – устройства с памятью, в котором выходные биты зависят не только от битов на входе, но и от L (длина сдвигового регистра) предыдущих бит. Таким образом, при скорости кода R = 1/2 текущий бит закодированной последовательности оказывает влияние на 2L битов после него. Появление ошибок, расположенных на расстоянии менее 2L друг от друга критически снижает исправляющую способность кодеров.

С другой стороны, увеличение длины сдвигового регистра L при сохранении скорости R (а соответственно «удельной» исправляющей способности) позволяет справиться с большим числом ошибок в группе. Например, при L=3 и R=1/2 появление двух ошибок в соседних битах не позволит их правильно декодировать. С другой стороны, при L=8 и R=1/2 декодер их гарантированно исправит.

Это свойство связано с фундаментальными основами теории помехоустойчивого кодирования, сформулированных Шенноном: скорость передачи в канала с бесконечно малой вероятностью битовых ошибок может достичь пропускной способности $C=B\log_2(1+SNR)$ при использовании помехоустойчивых кодов с длинными кодовыми словами. Для таких кодов не важно положение ошибок. Это можно проиллюстрировать с помощью рисунка 1 (пример приведен для блочных кодов с четкими границами кодовых слов).

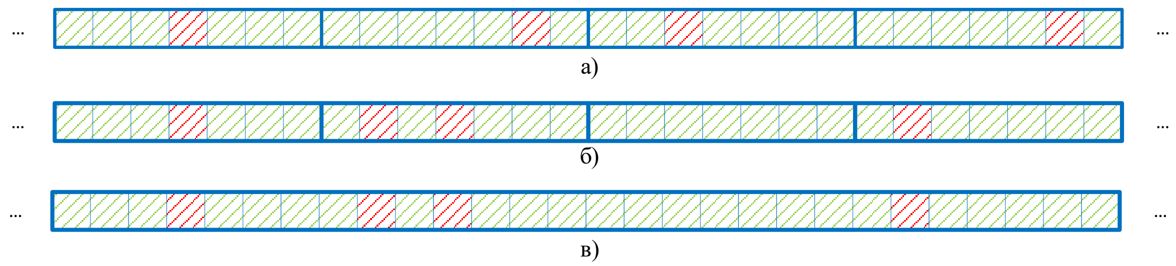


Рисунок 1 – Пример исправления ошибок в кодах различной длины

Пусть дан код с длиной кодового слова N=7, способный исправить 1 ошибку. При передаче пакета длиной 28 битов в него входит четыре кодовых слова. В случае, если возникло 4 ошибки, каждая из которых приходится на разные кодовые слова (рисунок 1а), помехоустойчивый кодек их исправит. Однако, весьма вероятна ситуация, в которой при сохранении общего числа ошибок (4) на одно кодовое слово придется две – в этом случае кодек не сможет их исправить. Использование кода с длиной кодового слова N=28 (рисунок 1в) и сохранении «удельной» исправляющей способности (4 ошибки) способен исправить ошибки с той же локализацией.

Увеличить длину кодового слова можно увеличив длину сдвигового регистра. Однако это приводит к росту вычислительной сложности алгоритма

Витерби (количество анализируемых узлов 2^{L-1}), и делает невозможным использование длинных сверточных кодов на практике.

Для решения этой проблемы совместно со сверточными кодами применяют перемежители (перемешиватели), рисунок 3. Перемежитель переставляет биты сообщения по определенному закону (как правило, заданному таблицей, или формулой), и обеспечивает разнесение групповых ошибок, образовавшихся в канале. На рисунке 2 изображен пример перемежителя.

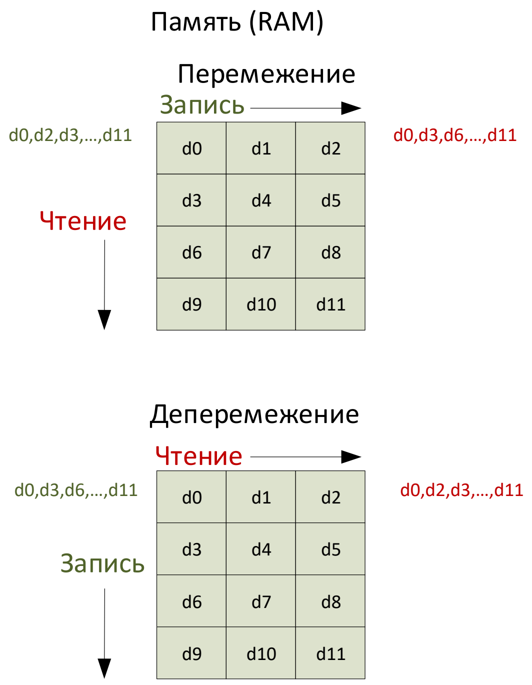


Рисунок 2 – Перемежитель

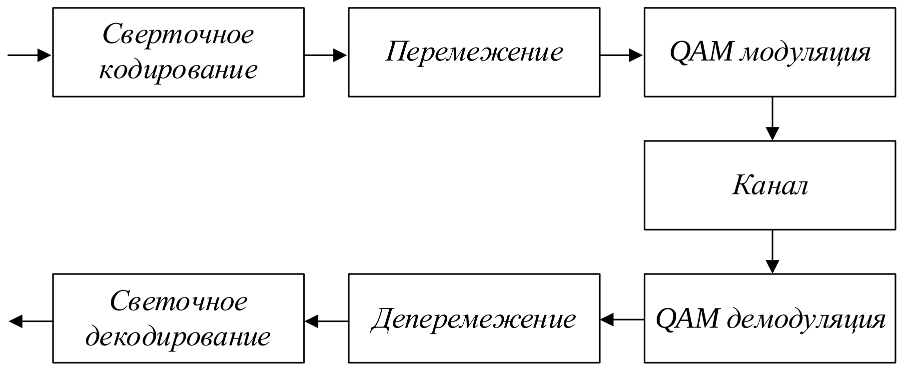


Рисунок 3 – Использование перемежителя совместно со сверточным кодеком

Другой подход заключается в использовании каскадного кодирования – например, сверточного кодера совместно с кодером Рида-Соломона, рисунок 4. В такой схеме сверточный кодер справляется с большим числом ошибок, а оставшиеся исправляет дополнительный кодер с большой длиной кодовых слов и высокой скоростью (и соответственно, низкой исправляющей способностью).

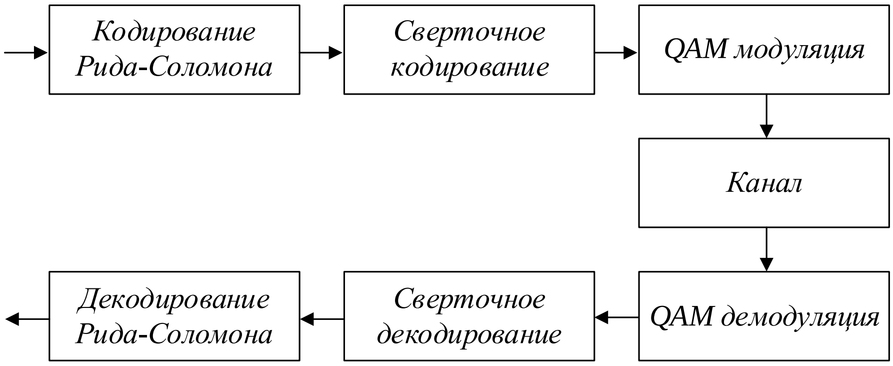


Рисунок 4 – Каскадное кодирование

Наконец подход, показавший наибольшую эффективность – турбокоды. Эти коды используются в ряде современных системах связи (CDMA2000, UMTS, 4G LTE, спутниковое телевидение) и могут обеспечить скорость, максимально приближенную к пропускной способности канала. Турбокоды, наряду с LDPC и полярными кодами считаются наиболее эффективными видами помехоустойчивого кодирования.

Идея турбокодов заключается в использовании двух или более простых кодеров, на часть из которых биты поступают после перемежения, рисунок 5.

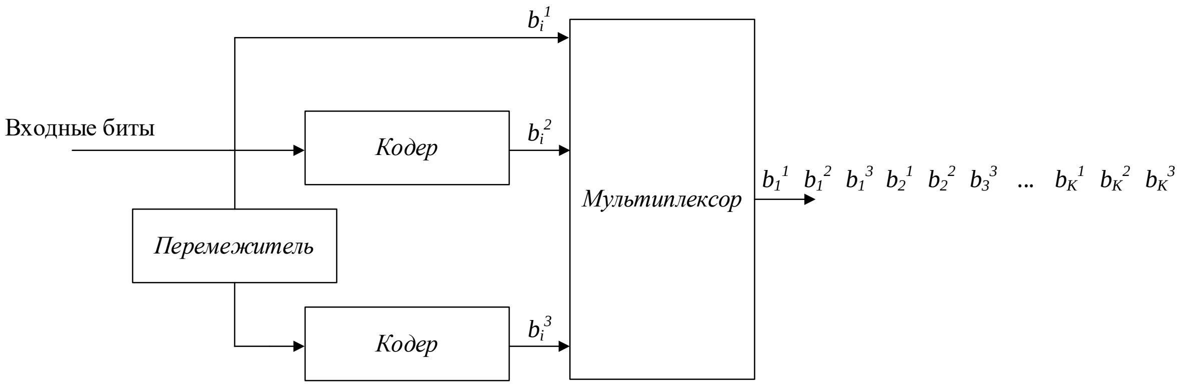


Рисунок 5 – Обобщенный турбокодер со скоростью 1/3

В кодере, который изображен на рисунке 4, формируется три потока выходных битов: систематический, содержащий неизменные исходные биты b_i^1, проверочные биты после первого кодера b_i^2 (скорость R=1, содержит только проверочные биты) и проверочные биты после второго кодера b_i^3, перед которым стоит перемежитель. Биты объединяются в последовательный поток с помощью мультиплексора: $b_1^1\ b_1^2\ b_1^3 \ldots b_i^1\ b_i^2\ b_i^3 \ldots b_K^1\ b_K^2\ b_K^3$.

В качестве элементарных кодеров могут выступать различные реализации. В случае использования сверточных кодов турбокод можно построить по схеме, приведенной на рисунке 6 (такая реализация используется в системе LTE).

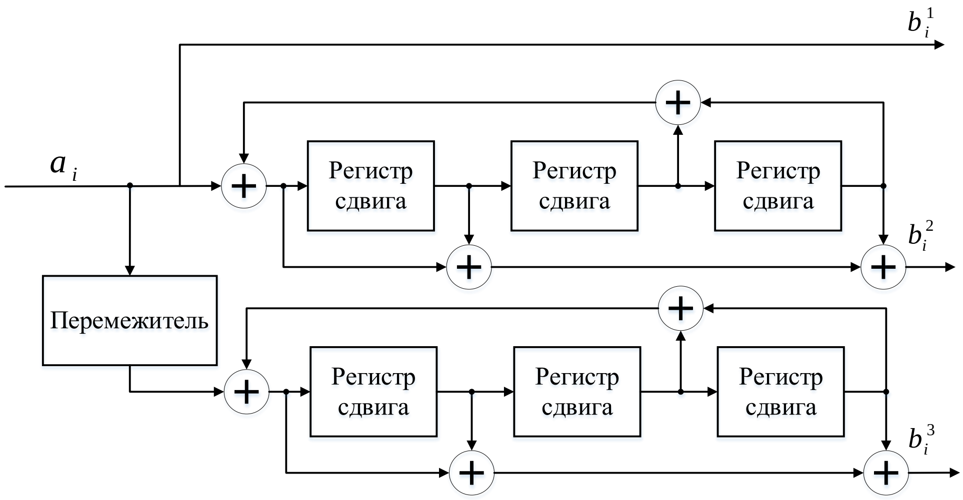


Рисунок 6 – Реализация турбокода на сверточных кодерах

Для декодирования турбокодов используют итерационные декодеры с «мягким» входом. Для формирования мягких решений демодулятор не принимает решения о том, что принятый символ соответствует «0», или «1», а рассчитывает значения логарифмических отношений правдоподобия LLR (Log Likelihood Ratio):

$$\mathrm{LLR}(\mathrm{u_k}) = \ln\left(\frac{\mathrm{P}(\mathrm{u_k} = 1)}{\mathrm{P}(\mathrm{u_k} = 0)}\right)$$

Это натуральный логарифм отношения вероятности того, что переданный символ соответствует «1» к вероятности того, что принятый символ соответствует «0». В случае, когда вероятность «1» выше, то отношение больше 1, а логарифм положителен. В противном случае отношение меньше 1, а логарифм отрицателен. Соотношение вероятностей напрямую влияет на модуль LLR – чем он больше, тем больше можно быть уверенным в правдивости гипотезы.

Для BPSK модуляции LLR может быть рассчитан как:

$$\mathrm{LLR}(\mathrm{u_k}) = \ln\left(\frac{e^{-\frac{1}{\sigma^2}(y_k-(-1))^2}}{e^{-\frac{1}{\sigma^2}(y_k-1)^2}}\right),$$

где σ^2 – мощность (дисперсия) шума.

В упрощенном случае, когда мощность шума не оценивается и условно считается разной единице:

$$\mathrm{LLR}(\mathrm{u_k}) = (y_k + 1)^2 - (y_k - 1)^2 = 4y_k.$$

Декодер турбокода на рисунке 6 приведен на рисунке 7.

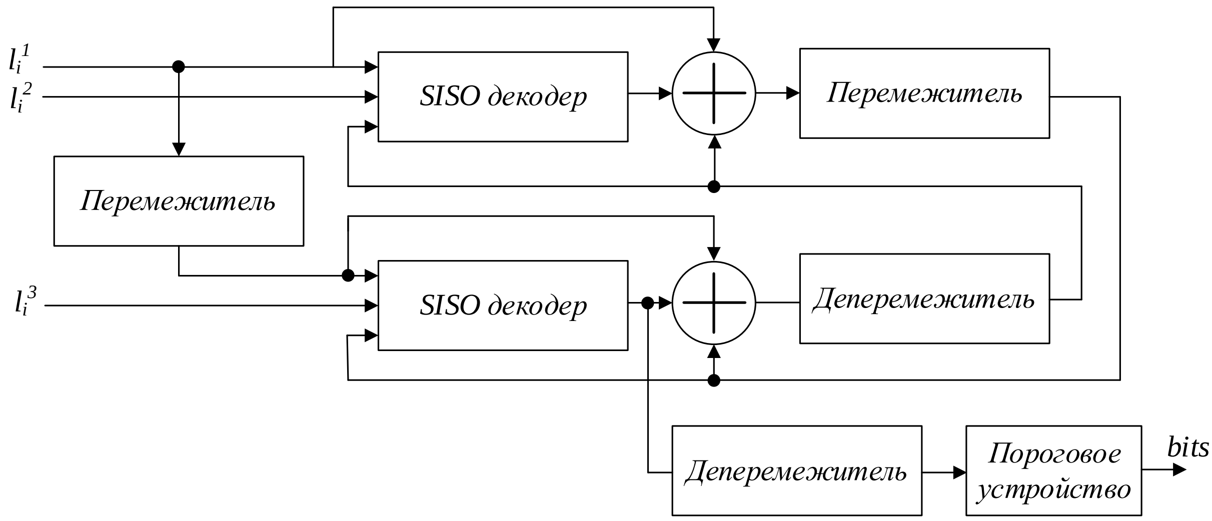


Рисунок 7 – Схема турбодекодера

Основной частью этого декодера является элементарный SISO (Soft Input Soft Output) декодер с мягким входом и мягким выходом. SISO декодирование является обратной процедурой элементарного сверточного кодирования и основано на алгоритме Витерби. Его основное назначение – пересчет оценки LLR на основании входящих данных. Такой декодер может иметь разное число входов, в зависимости от количества анализируемых значений LLR. В конечном итоге их анализ сводится к расчету оценки вероятности того, что тройка битов соответствовали «0» и «1»: $P(l_1,l_2,l_3)$ =

$P(l_1)P(l_1)P(l_1)$. Т.к. исходные значения логарифмы, то эта операция сводится к их сумме.

Декодер можно построит и по другой схеме, с SISO декодером с двумя входами, рисунок 8.

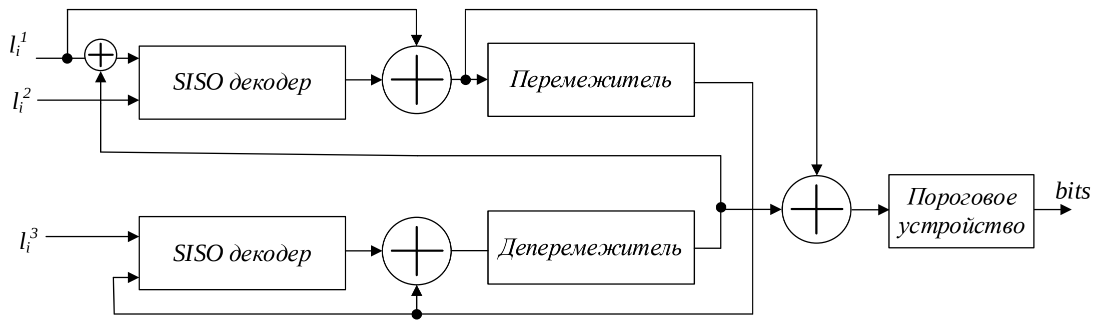


Рисунок 8 – Схема турбодекодера с SISO декодером с двумя входами

В этой схеме операции сложения с третьим значением LLR осуществляются непосредственно после сумматоров (после SISO декодеров).

Пересчет вероятностей осуществляется в течении ряда итераций. Каждая итерация включает в себя следующие шаги:

1. Пересчет оценок верхним SISO декодером. Пересчитанные оценки поступают на перемежитель и далее, на вход второго SISO декодера.
2. Пересчет оценок нижним SISO декодером. Пересчитанные оценки поступают на деперемежитель и далее, на вход первого SISO декодера.

После всех итераций оценки верхнего (в которую также включена LLR систематической части кода) и нижнего SISO декодеров суммируются и поступают на пороговое устройство, в котором принимаются жесткие решения.

В рамках этой работы мы не будем подробно рассматривать и реализовывать SISO декодер и турбокод в целом, а сосредоточимся на его практическом применении с использованием готовых функций (часть взята из https://github.com/robmaunder/turbo-3gpp-matlab).

Ход работы

В ходе работы вам необходимо реализовать имитационную модель системы связи, схема которой приведена на рисунке 9.

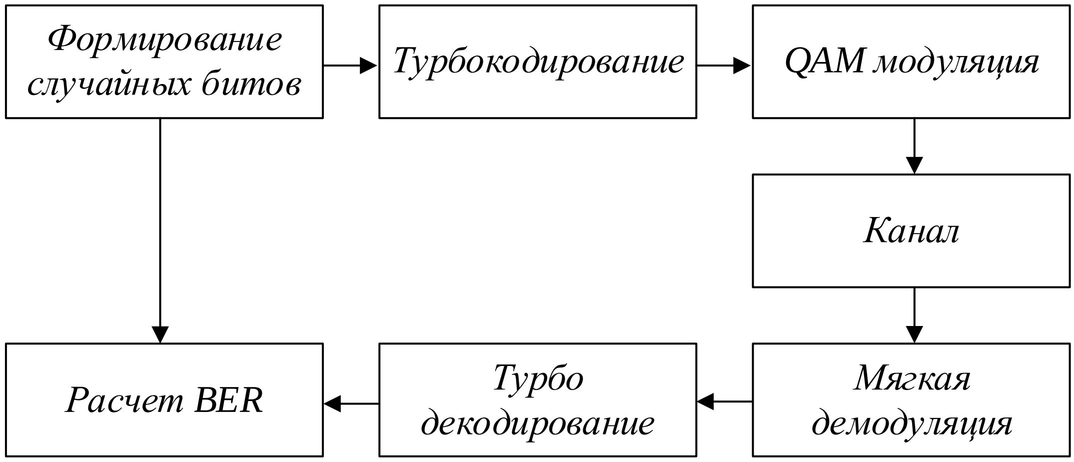


Рисунок 9 – Схема имитируемой системы связи

Скопируйте в папку с проектом файлы

turbo_encoder.m

turbo_decoder.m

parameters.mat

maxstar.m

internal_interleaver.m

conv_encoder_turbo.m

constituent_encoder.m

constituent_decoder.m

Из этих файлов вы будете обращаться к функции турбокодера turbo_encoder, которая имеет аргумент c (вектор длиной K) и возвращает значение вектора закодированных битов d. Размерность c определена стандартом (она привязана к параметрам перемежителя) и составляет K = 40, 48, 56, 64, 72, … 6144. Всего определено 188 различных значений K, которые записаны в таблице 5.1.3-3 спецификации ETSI TS 136 212 LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding.

Второй функцией, которую вы будете вызывать – турбодекодер *turbo_decoder.m*. Эта функция имеет аргументы: вектор *d_a* – входные значения LLR и *max_iterations* – максимальное число итераций (подайте значение 5), а возвращает вектор декодированных битов c.

Вам необходимо самостоятельно реализовать функцию демодулятора с мягким выходом *LLR_demod*, на вход которой поступает аргумент – вектор символов после канала связи *chan_sym*, а на выходе формируется вектор *llr*. Основная логика работы функции

```
sc = [-1 1];
for i = 1:length(chan_sym)
llr(i) =  ((chan_sym(i)-sc(1)).^2 - (chan_sym(i)-sc(2)).^2);
end
```

Реализуйте модель, приведенную на рисунке 8. Глобальными параметрами модели будут

```
M   = 2;
N   = 10;
K   = 40;
SNR = [-10 5];
```

Где *K* – длина блока входных битов турбокода, *N* – количество пакетов.

Моделирование проведите для *SNR* в диапазоне от -10 до 5 дБ. Постройте графики зависимостей *BER (SNR)* для кодов с длиной *K*=40, 6144. Оцените разницу в помехоустойчивости и сравните ее с другими кодами.

Для отчета по проделанной работе сделайте:

1. Скриншоты зависимостей *BER* от *SNR* для разной длины кода (*K*=40, *K*=6144);
2. Скриншот зависимости *BER* от *SNR* для сверточного и турбо кода.

Дополнительное задание для самостоятельной работы

1. Получите зависимости вероятности битовых ошибок от *SNR* и от числа итераций: *BER (SNR,N_iter)* для *N_iter* от 1 до 8. Оцените прирост помехоустойчивости с ростом числа итераций.

Контрольные вопросы:

1. Какие недостатки сверточных кодов устраняют турбокоды?
2. Что такое перемежитель и как он реализуется?
3. Что такое LLR и как его рассчитать для модуляции BPSK?
4. Какова функция SISO декодера?

Практическая работа № 7
Код Хэмминга

Цель работы: Реализация кодека Хэмминга.

Задачи практической работы:

1) Изучение теории кодирования Хэмминга;
2) Реализация алгоритма кодера Хэмминга;
3) Реализация алгоритма декодера Хэмминга;
4) Построение имитационной модели системы связи с кодеком Хэмминга;
5) Изучение помехоустойчивости системы связи с кодеком Хэмминга.

Оборудование и программное обеспечение: Octave, version 6.4.0.

Теоретический материал

В рамках текущей работы необходимо реализовать алгоритмы кодирования и декодирования Хэмминга и встроить их в разработанную ранее имитационную модель, приведенную на рисунке 1.

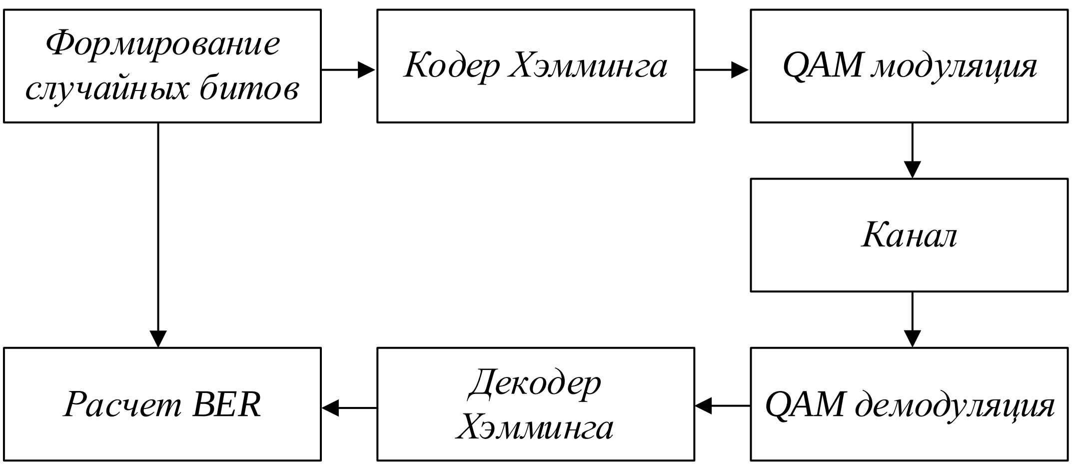


Рисунок 1 – Схема реализуемой в текущей работе модели

Логика построения модели совпадает с моделью системы с моделью расчета контрольной суммы. На входе формируется N векторов случайных битов (длиной L = 4), который кодируется и отображается в QAM символы. После канала связи и демодуляции биты поступают на декодер кода Хэмминга. Декодированные биты сравниваются с переданными, рассчитывается BER и строится его зависимость от SNR.

Коды Хэмминга являются простейшим видом блочного кодирования с параметрами n = 7 (длина кодового слова), k = 4 (длина сообщения), d_{min} = 3 (код способен исправить одну ошибку в пакете). Код Хэмминга систематический, т.е. кодовые слова содержат исходное сообщение.

Код Хемминга, как и другие блочные коды описываются с помощью порождающей **G** и проверочной **H** матриц.

Матрицу **G** для систематического кода можно представить в следующем виде:

$$\mathbf{G} = \begin{bmatrix} \mathbf{I} & \mathbf{P} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 & p_{11} & p_{12} & \cdots & p_{1n-k} \\ 0 & 1 & \cdots & 0 & p_{21} & p_{22} & \cdots & p_{2n-k} \\ \vdots & \vdots & \ddots & 0 & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & p_{k1} & p_{k2} & \cdots & pkn-k \end{bmatrix},$$

где **I –** единичная матрица, размером k×k, отвечающая за систематическую часть кодового слова, а **P –** матрица размером k×(n-k), отвечающая за проверочные биты кодового слова. Матрица H, в свою очередь имеет следующий вид $\mathbf{H} = \begin{bmatrix} -\mathbf{P}^T & \mathbf{I}_{n-k} \end{bmatrix}$, отрицательный знак можно отбросить, в случае двоичных кодов.

Рассмотрим подробнее работу кодера Хэмминга.На вход кодера поступает k информационных бит, на выходе формируется n бит кодового слова. Процедура кодирования (получения кодового слова **y**) сводится к умножению битового сообщения **x** на генераторную матрицу **G:**

$$\mathbf{y} = \mathbf{xG} = [\mathrm{x}_1, \mathrm{x}_2, \mathrm{x}_3, \mathrm{x}_4] \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}.$$

Можно заметить, что левая часть (первые 4 столбца) представляет из собой диагональную единичную матрицу **I**. Умножения сообщения на **I** дает систематическую часть кодового слова.

$$y_1 = x_1$$

$$y_2 = x_2$$

$$y_3 = x_3$$

$$y_4 = x_4$$

Три правых столбца матрицы при умножении на **x** формируют проверочную часть кодового слова, т.е.

$$y_5 = x_1 + x_2 + x_3$$

$$y_6 = x_2 + x_3 + x_4$$

$$y_7 = x_1 + x_2 + x_4$$

Здесь и везде далее сложения выполняются по модулю 2.

Процедуру кодирования также можно наглядно представить в виде структурной схемы (рисунок 2).

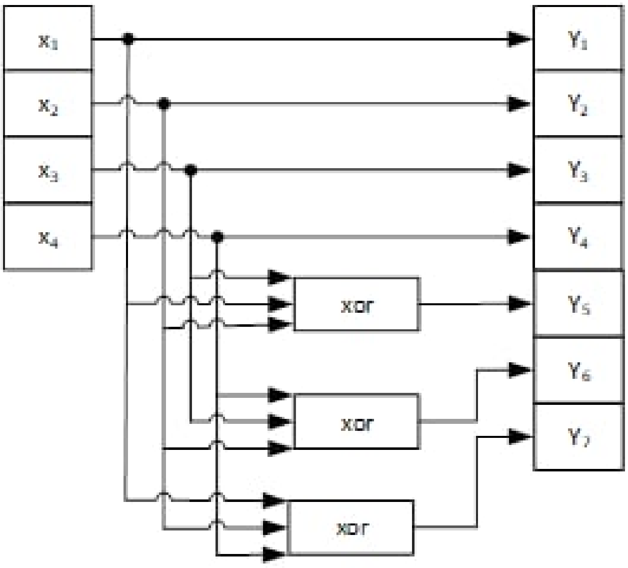


Рисунок 2 – Кодер Хемминга

При процедуре декодирования принятый вектор бит умножается на проверочную матрицу **H**.

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Как было сказано ранее **H** тоже имеет определенную структуру: первые четыре строки повторяют правую часть матрицы **G**, а три нижние строки образуют единичную матрицу.

В случае, если при передаче не возникло ошибок, принятый вектор равен **y**. Для декодирования рассчитывается вектор синдромов **S**:

$$\mathbf{S} = \mathbf{yH} = [\mathrm{x}_1\mathrm{x}_2\mathrm{x}_3\mathrm{x}_4\mathrm{y}_5\mathrm{y}_6\mathrm{y}_7] \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = [\mathrm{s}_1\mathrm{s}_2\mathrm{s}_3]$$

Притом

$$s_1 = x_1+x_2+x_3+y_5 = x_1+x_2+x_3+ x_1 +x_2+x_3 = 0$$

$$s_2 = x_2+x_3+x_4+y_6 = x_1+x_2+x_3+ x_1+x_2+x_3 = 0$$

$$s_3 = x_1+x_2+x_4+y_7 = x_1+x_2+x_3+ x_1+x_2+x_3 = 0$$

Таким образом, вне зависимости от значения битов вектора **x** при отсутствии ошибок вектора **S** всегда будет содержать только нули, **yH** = **0**. Появление битовых ошибок можно задать с помощью вектора ошибок e (длина вектора n). Единичные элементы этого вектора означают появившиеся ошибки. Тогда принятый вектор бит с ошибками можно записать как **y‘** = **y** + **e**.

В этом случае вектор синдромов равен:

$$\mathbf{S} = \mathbf{y}'\mathbf{H} = (\mathbf{y} + \mathbf{e})\mathbf{H} = \mathbf{YH} + \mathbf{eH} = \mathbf{eH}.$$

Из этого выражения следует, что вне зависимости от значения битов вектора **x** значение вектора **S** зависит только от конфигурации вектора ошибки **e**. Подставляя различные значения битов вектора **e** можно получить таблицу синдромов:

Таблица 1 – Положение ошибки и соответствующие ей синдромы

Вектор ошибки e							Синдром S		
e_1	e_2	e_3	e_4	e_5	e_6	e_7	s_1	s_2	s_3
1	0	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	1	1	1
0	0	1	0	0	0	0	1	1	0
0	0	0	1	0	0	0	0	1	1
0	0	0	0	1	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0
0	0	0	0	0	0	1	0	0	1

Таким образом, однократной ошибке соответствует уникальный синдром. Используя это свойство, рассчитав в декодере $\mathbf{S} = \mathbf{y}'\mathbf{H}$ можно по получившемуся вектору **S** определить положение ошибки (получить вектор **e**) и исправить ее

$$\mathbf{y} = \mathbf{y}' + \mathbf{e}$$

Такой способ декодирования называется табличным и требует хранения в памяти всех конфигураций синдромов и соответствующих им векторов ошибок, что для длинных кодов является серьезным недостатком.

Ход работы

Реализуйте модель, построенную по схеме, приведенной на рисунке 1. Принципы построения модели схожи с работой, посвященной расчету контрольной суммы.

Создайте функцию *hamm_coder* с аргументом *bits* и выходным значением *coded_bits*. Запишите генераторную матрицу

G = [1 0 0 0 1 0 1

0 1 0 0 1 1 1

0 0 1 0 1 1 0

0 0 0 1 0 1 1];

Выполните процедуру кодирования (умножение входного вектора на генераторную матрицу)

*coded_bits = mod(bits*G,2);*

Все вычисления проводятся по модулю 2, поэтому необходима операция взятия остатка от деления на 2.

Создайте функцию *hamm_decoder* с аргументом *bits* и выходным значением *decoded_bits*. Запишите проверочную матрицу

H = [1 0 1

1 1 1

1 1 0

0 1 1

1 0 0

0 1 0

0 0 1];

Рассчитайте вектор синдромов

*S = mod(bits*H,2);*

По значению вектора синдромов определите вектор ошибки (на основании таблицы 1)

```
if      S ==  [0 0 0]
e = [0 0 0 0 0 0 0];
elseif  S ==  [0 0 1]
e = [0 0 0 0 0 0 1];
elseif  S ==  [0 1 0]
e = [0 0 0 0 0 1 0];
elseif  S ==  [0 1 1]
e = [0 0 0 1 0 0 0];
elseif  S ==  [1 0 0]
e = [0 0 0 0 1 0 0];
```

elseif S == [1 0 1]

e = [1 0 0 0 0 0 0];

elseif S == [1 1 0]

e = [0 0 1 0 0 0 0];

elseif S == [1 1 1]

e = [0 1 0 0 0 0 0];

end

Исправленный вектор битов будет равен сумме по модулю два входного вектора и вектора ошибок

corrected_bits = xor(bits,e);

Из исправленного вектора битов выделите систематическую часть (четыре первых бита)

decoded_bits = corrected_bits(1:4);

Соберите модель, приведенную на рисунке 1 в файле *main*. Основные параметры модели соответствуют предыдущим работам за исключением *K* – длины сообщения:

M = 2;

N = 1000;

K = 4;

SNR = [0 15];

k = 0;

Моделирование проводится в цикле

for snr_i = SNR(1):SNR(2)

Счетчик номера итерации

k = k+1

И обнуление количества ошибок в этой итерации

err = 0;

Далее обработка выполняется для *N* посылок. Для этого создается цикл

for n = 1:N

В каждой итерации выполняется последовательность процедур формирования и обработки, приведенная на рисунке 1.

bits = randi([0 M-1],1,K);

coded_bits = hamm_coder (bits);

qam_sym = modulator(coded_bits, M);

chan_sym = awgn(qam_sym,snr_i,'measured');

r_bits = demodulator(chan_sym, M);

decoded_bits = hamm_decoder (r_bits);

И к счетчику ошибок добавляется количество ошибок в этой посылке

err = err+length(find(xor(bits,decoded_bits)));

На этом обработка одного пакета заканчивается.

После передачи *N* пакетов рассчитывается значение *BER*

*BER(k) = err/(N*K);*

На этом текущая итерация глобального цикла заканчивается

Моделирование проведите для *SNR* в диапазоне от 0 до 15 дБ.

Для отчета по проделанной работе сделайте скриншоты зависимостей *BER* от *SNR* для сверточного кода и кода Хемминга.

Дополнительное задание для самостоятельной работы

1. Постройте зависимость *BER (Eb/N0)* для кода Хэмминга и сверточного кода. Сравните результат.

Контрольные вопросы:

1. Сколько ошибок способен исправить кодек Хэмминга?
2. Каково минимальное межкодовое расстояние для кода Хэмминга?
3. В чем недостаток декодирования табличным методом?
4. Как параметры n и k связаны с длиной вектора синдромов?

Практическая работа № 8

LDPC кодирование по стандарту 5G NR

Цель работы: Реализация LDPDC кодера в соответствии со стандартом 5G NR.

Задачи практической работы:

1) Формирование базовой матрицы LDPC.
2) Реализация алгоритма кодирования.
3) Проверка правильности кодирования.

Оборудование и программное обеспечение: Octave, version 6.4.0.

Теоретический материал

Классические блочные и LDPC коды

Линейные блочные LDPC коды были предложены Р. Галлагером и иногда называются по его имени. Классические блочные коды, например, коды Хэмминга, подразумевают хранение в памяти генераторной и проверочных матриц. Кодирование сводится к умножению входного вектора бит на генераторную матрицу, а декодирование – на проверочную. Результатом умножения вектора битов на выходе демодулятора на проверочную матрицу становится вектор синдромов, зная значение которых можно определить наличие и положение ошибок. Однако, удлинение кодового слова ведет к увеличению размеров как проверочной и генераторной матриц, так и хранящихся в памяти таблиц синдромов.

Решением этой проблемы является применение кодов с малой плотностью проверок на четность. Количество единиц в генераторной и проверочной матрицах таких кодов мало (относительно размеров), и в памяти нужно держать только их положение, а не всю матрицу. Кроме того, для декодирования можно применить различные версии итеративного алгоритма передачи сообщений MPA (Message Passing Algorithm), имеющего значительно меньшую вычислительную сложность по сравнению с классическими решениями.

Применение LDPC кодов в 5G NR регламентирует спецификация 3GPP TS 38.212 3rd Generation Partnership Project; Technical Specification Group Radio Access Network; NR; Multiplexing and channel coding (Release 16). Задача LDPC, как и любого кодека – исправление ошибок на выходе демодулятора, которые зависят от отношения сигнал-шум и вида применяемой модуляции.

Стандартом определены следующие виды модуляции. Для нисходящего канала QPSK, 16QAM, 64QAM, и 256QAM (8 битов на символ). Для восходящего канала QPSK, 16QAM, 64QAM и 256QAM для CP-OFDM; π/2-BPSK, QPSK, 16QAM, 64QAM и 256QAM для DFT-s-OFDM.

Повышение индекса модуляции ведет к увеличению скорости передачи, что является одним из основных показателей качества телекоммуникационных систем. С другой стороны, повышение индекса модуляции ведет к увеличению вероятности битовых ошибок. С этим можно бороться используя LDPC код с низкой скоростью, который позволит исправить возникшие ошибки, но, в свою очередь, приведет к снижению скорости передачи. Таким образом, выбор параметров модуляции и кодирования является сложной оптимизационной задачей. Решение этой задачи приведено в спецификациях систем связи. Для 5G NR это 3GPP TS 38.214 3rd Generation Partnership Project; Technical Specification Group Radio Access Network; NR; Physical layer procedures for data. Этот документ содержит таблицы с параметрами модуляции, кодирования (сигнально-кодовые конструкции, Modulation and Coding Scheme, MCS), зависящих от характеристики состояния канала передачи CSI (Channel State Information). Скорость LDPC кода в 5G NR может быть от 30/1024 до 948/1024, что в комбинации с соответствующими модуляциями обеспечивает спектральную эффективность от 0.0586 бит/с/Гц до 7.4063 бит/с/Гц.

Для обеспечения соответствующей скорости кодирования необходимо формировать разреженную проверочную матрицу с определенными параметрами.

Алгоритм построения LDPC кодов относится к квазициклическим QC-LDPC (Quasi-cyclic).

Проверочная матрица **H** LDPC кодов в системах 5G строится на основе базовых матриц, которые называются базовыми графами (BG). Стандартом задано два базовых графа: **BG1** и **BG2**. Размерность **BG1** составляет 46x68, **BG2** 42x52 элемента. На рисунке 1 изображена таблица, задающая **BG2** (для **BG2** вид и структура таблицы аналогичны).

Номер строки BG с ненулевым элементом

Номер столбца BG с ненулевым элементом

Коэффициент сдвига элементов подматрицы

$\mathbf{H}_{BG}$		$V_{i,j}$							
Row index i	Column index j	Set index i_{LS}							
		0	1	2	3	4	5	6	7
0	0	9	174	0	72	3	156	143	145
	1	117	97	0	110	26	143	19	131
	2	204	166	0	23	53	14	176	71
	3	26	66	0	181	35	3	165	21
	6	189	71	0	95	115	40	196	23
	9	205	172	0	8	127	123	13	112
	10	0	0	0	1	0	0	0	1
	11	0	0	0	0	0	0	0	0
1	0	167	27	137	53	19	17	18	142
	3	166	36	124	156	94	65	27	174
	4	253	48	0	115	104	63	3	183
	5	125	92	0	156	66	1	102	27
	6	226	31	88	115	84	55	185	96
	7	156	187	0	200	98	37	17	23
	8	224	185	0	29	69	171	14	9
	9	252	3	55	31	50	133	180	167
	11	0	0	0	0	0	0	0	0
	12	0	0	0	0	0	0	0	0
2	0	81	25	20	152	95	98	126	74
	1	114	114	94	131	106	168	163	31
	3	44	117	99	46	92	107	47	3
	4	52	110	9	191	110	82	183	53
	8	240	114	108	91	111	142	132	155
	10	1	1	1	0	1	1	1	0
	12	0	0	0	0	0	0	0	0
	13	0	0	0	0	0	0	0	0
3	1	8	136	38	185	120	53	36	239
	2	58	175	15	6	121	174	48	171
	4	158	113	102	36	22	174	18	95
	5	104	72	146	124	4	127	111	110
	6	209	123	12	124	73	17	203	159
	7	54	118	57	110	49	89	3	199
	8	18	28	53	156	128	17	191	43

$\mathbf{H}_{BG}$		$V_{i,j}$							
Row index i	Column index j	Set index i_{LS}							
		0	1	2	3	4	5	6	7
16	26	0	0	0	0	0	0	0	0
17	1	254	158	0	48	120	134	57	196
	5	124	23	24	132	43	23	201	173
	11	114	9	109	206	65	62	142	195
	12	64	6	18	2	42	163	35	218
	27	0	0	0	0	0	0	0	0
18	0	220	186	0	68	17	173	129	128
	6	194	6	18	16	106	31	203	211
	7	50	46	86	156	142	22	140	210
	28	0	0	0	0	0	0	0	0
19	0	87	58	0	35	79	13	110	39
	1	20	42	158	138	28	135	124	84
	10	185	156	154	86	41	145	52	88
	29	0	0	0	0	0	0	0	0
20	1	26	76	0	6	2	128	196	117
	4	105	61	148	20	103	52	35	227
	11	29	153	104	141	78	173	114	6
	30	0	0	0	0	0	0	0	0
21	0	76	157	0	80	91	156	10	238
	8	42	175	17	43	75	166	122	13
	13	210	67	33	81	81	40	23	11
	31	0	0	0	0	0	0	0	0
22	1	222	20	0	49	54	18	202	195
	2	63	52	4	1	132	163	126	44
	32	0	0	0	0	0	0	0	0
23	0	23	106	0	156	68	110	52	5
	3	235	86	75	54	115	132	170	94
	5	238	95	158	134	56	150	13	111
	33	0	0	0	0	0	0	0	0
24	1	46	182	0	153	30	113	113	81
	2	139	153	69	88	42	108	161	19
	9	8	64	87	63	101	61	88	130
	34	0	0	0	0	0	0	0	0

Рисунок 1 – Элемент таблицы стандарта, задающей BG2

Первый (Row index *i*) и второй (Column index *j*) столбцы задают номера строк и столбцов, на пересечении которых расположены ненулевые элементы. Так, например, в первой строке ненулевые элементы располагаются в столбцах с номерами 0, 1, 2, 3, 6, 9, 10, 11. В зависимости от принятого на этапе сегментации значения i_{LS} выбирается один из восьми столбцов. Значения элементов в этом столбце $V_{i,j}$ – коэффициенты логического сдвига вправо для элементов BG. Каждый элемент BG представляет из себя единичную матрицу **I** (единицы на главной диагонали) размерностью ZcxZc, со столбцами, сдвинутыми вправо на коэффициент логического сдвига. Таким образом, матрица H для BG2 имеет размерность 42Zcx52Zc.

Запишем коэффициент логического сдвига вправо в нижний индекс матрицы **I**, тогда при $Z_c = 3$ для различных коэффициентов матрицы **I** будут выглядеть следующим образом

$$\mathbf{I} = \mathbf{I}_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \ \mathbf{I}_1 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \ \mathbf{I}_2 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \ \mathbf{I}_{-1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Матрицы **BG1** и **BG2** имеют определенную структуру, рисунок 2.

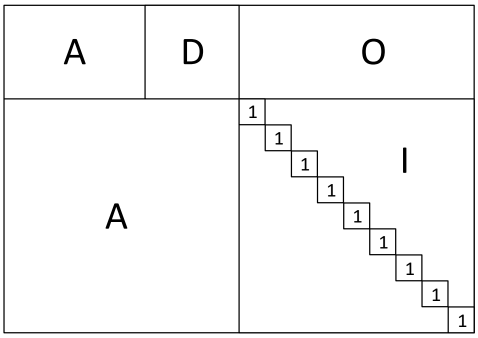
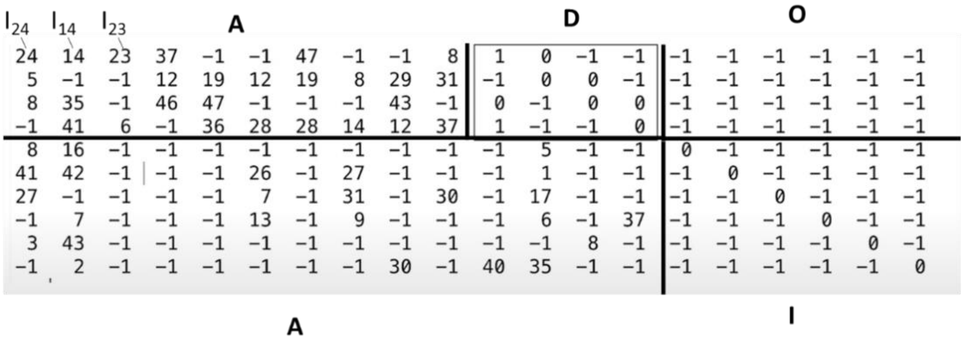


Рисунок 2 – Структура матриц **BG1** и **BG2**

На рисунке 3 приведена реализация матрицы **BG2** для первых 10 строк и 20 столбцов, $i_{LS} = 3$, $Z_c = 48$. Числа в этой матрицы – значения коэффициентов сдвига. Значение «0» соответствует отсутствию сдвига, значение «-1» - нулевая матрица.



24	14	23	37	−1	−1	47	−1	−1	8	1	0	−1	−1	−1	−1	−1	−1	−1	−1
5	−1	−1	12	19	12	19	8	29	31	−1	0	0	−1	−1	−1	−1	−1	−1	−1
8	35	−1	46	47	−1	−1	−1	43	−1	0	−1	0	0	−1	−1	−1	−1	−1	−1
−1	41	6	−1	36	28	28	14	12	37	1	−1	−1	0	−1	−1	−1	−1	−1	−1
8	16	−1	−1	−1	−1	−1	−1	−1	−1	−1	5	−1	−1	0	−1	−1	−1	−1	−1
41	42	−1	−1	−1	26	−1	27	−1	−1	−1	1	−1	−1	−1	0	−1	−1	−1	−1
27	−1	−1	−1	−1	7	−1	31	−1	30	−1	17	−1	−1	−1	−1	0	−1	−1	−1
−1	7	−1	−1	−1	13	−1	9	−1	−1	−1	6	−1	37	−1	−1	−1	0	−1	−1
3	43	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	8	−1	−1	−1	−1	−1	0	−1
−1	2	−1	−1	−1	−1	−1	−1	30	−1	40	35	−1	−1	−1	−1	−1	−1	−1	0

Рисунок 3 – Реализация матрицы **BG2** для первых 10 строк и 20 столбцов, $i_{LS} = 3$, $Z_c = 48$

Элементы **BG1** и **BG2** – подматрица **A**, **D**, **O**, **I**. Матрица **I** – диагональная матрица, значения всех коэффициентов сдвига в ней равны нулю (в подматрицах размерностью ZcxZc отсутствует логический сдвиг). Матрица **O** – нулевая, т.е. все элементы – нули. Матрица **D** является продолжением матрицы **I** и помимо элементов, расположенных на главной диагонали содержит дополнительные проверочные биты. Матрица **A** (фактически две матрицы **A**) содержит проверочные биты и является разреженной.

LDPC кодирование

Проверочная матрица **H** полностью определяет алгоритм кодирования. Однако, зная только матрицу **H** невозможно напрямую получить кодовые слова. Рассмотрим основные принципы и алгоритм кодирования. Кодовые слова **d** = [d_0, d_1, … d_{N-1}] LDPC кода в 5G систематические, т.е. содержат исходное сообщение $\mathbf{c} = \left[c_0, c_1, c_2, ..., c_{K-1}\right]$ и проверочные биты $\mathbf{w} = \left[w_0, w_1, w_2, ..., w_{N+2Z_c-K-1}\right]$. Обратите внимание, что длина кодового слова N = 66Zc для **BG1** и 50Zc для **BG2**. Таким образом, **d**=[**c** **w**]. Согласно свойствам блочных кодов:

$$\mathbf{H} \times \begin{bmatrix} \mathbf{c} \\ \mathbf{w} \end{bmatrix} = \mathbf{S} = \mathbf{0}, \qquad (1)$$

где **S** – вектор синдромов, если ошибок нет, то он равен **0** – вектору нулей.

Таким образом, задача кодирования сводится к поиску значений **w**, удовлетворяющих (1). При этом, если матрица **H** имеет вид

$$\mathbf{H} = [\mathbf{A}\ \mathbf{I}], \qquad (2)$$

где A – проверочная часть, а I – единичная матрица (это справедливо например, для кода Хэмминга, который разбирался в предыдущей работе), для первого синдрома справедливо выражение:

$$m_{i1} + m_{i_2} + ... + m_{i_k} + w_k = 0, \qquad (3)$$

где $m_{i1} + m_{i_2} + ... + m_{i_k}$ - биты, участвующие в формировании проверочного бита w_k. Используя это свойство, можно получить все проверочные биты w_k и, таким образом закодировать сообщение. С учетом того, что вся арифметика выполняется по модулю 2, выражение (3) можно записать как:

$$w_k = m_{i1} + m_{i_2} + ... + m_{i_k}, \qquad (3)$$

Однако, проверочная матрица LDPC в 5G имеет более сложную структуру, поэтому процесс кодирования усложняется, но основан на тех же принципах.

Рассмотрим матрицу **H**, приведенную на рисунке 3. Правая часть матрицы H является диагональной, начиная с 5-ой строки, поэтому для элементов $\mathbf{w_k}$ при k=5…N-1 справедливо выражение (4). Для первых 4-х строк справедливы выражения:

$$\begin{aligned}
&\mathbf{I}_{24}\mathbf{m}_1+\mathbf{I}_{14}\mathbf{m}_2+\mathbf{I}_{23}\mathbf{m}_3+\mathbf{I}_{37}\mathbf{m}_4+\mathbf{I}_{47}\mathbf{m}_7+\mathbf{I}_{8}\mathbf{m}_{10}+\quad+\mathbf{I}_{1}\mathbf{p}_1+\mathbf{I}_{0}\mathbf{p}_2=0\\
&\mathbf{I}_{5}\mathbf{m}_1+\mathbf{I}_{12}\mathbf{m}_4+\mathbf{I}_{19}\mathbf{m}_5+\mathbf{I}_{12}\mathbf{m}_6+\mathbf{I}_{19}\mathbf{m}_7+\mathbf{I}_{8}\mathbf{m}_8+\mathbf{I}_{29}\mathbf{m}_9+\mathbf{I}_{31}\mathbf{m}_{10}+\quad+\mathbf{I}_{0}\mathbf{p}_2+\mathbf{I}_{0}\mathbf{p}_3=0\\
&\mathbf{I}_{8}\mathbf{m}_1+\mathbf{I}_{35}\mathbf{m}_2+\mathbf{I}_{46}\mathbf{m}_4+\mathbf{I}_{47}\mathbf{m}_5+\mathbf{I}_{43}\mathbf{m}_9+\quad+\mathbf{I}_{0}\mathbf{p}_1+\quad+\mathbf{I}_{0}\mathbf{p}_3+\mathbf{I}_{0}\mathbf{p}_4=0\\
&\mathbf{I}_{41}\mathbf{m}_2+\mathbf{I}_{6}\mathbf{m}_3+\mathbf{I}_{36}\mathbf{m}_5+\mathbf{I}_{28}\mathbf{m}_6+\mathbf{I}_{28}\mathbf{m}_7+\mathbf{I}_{14}\mathbf{m}_5+\mathbf{I}_{12}\mathbf{m}_9+\mathbf{I}_{37}\mathbf{m}_{10}+\mathbf{I}_{1}\mathbf{p}_1\quad+\mathbf{I}_{0}\mathbf{p}_4=0
\end{aligned}\tag{4}$$

где $\mathbf{m_i}$ – вектор входных, а p_i – вектор проверочных бит длиной Z_c. Следует отметить, что произведение $\mathbf{m_i I_t}$ эквивалентно следующей перестановке бит внутри вектора $\mathbf{m_i}$:

$$\mathbf{I}_1\mathbf{m}_i=\begin{bmatrix}0&1&0\\0&0&1\\1&0&0\end{bmatrix}\begin{bmatrix}m_1\\m_2\\m_3\end{bmatrix}=\begin{bmatrix}m_2\\m_3\\m_1\end{bmatrix}.$$

Если индекс единичной матрицы t больше значения Z_c, то сдвиг осуществляется на остаток от деления t на Z_c, например, при Z_c = 3, $\mathbf{I}_2=\mathbf{I}_5=\mathbf{I}_{23}$. Для того, чтобы получить из (4) вектор p_1 нужно сложить все четыре выражения. Т.к. сложение выполняется по модулю 2, $\mathbf{I}_0\mathbf{p}_1+\mathbf{I}_0\mathbf{p}_1=0$. Далее, зная $\mathbf{I}_0\mathbf{p}_1$ можно найти $\mathbf{I}_1\mathbf{p}_1$, а значит, из первого уравнения можно выделить $\mathbf{I}_0\mathbf{p}_2$ и $\mathbf{p}_2$. После этого по цепочке находятся и $\mathbf{p}_3$, $\mathbf{p}_4$. Остальные элементы p находятся из проверочной матрицы в явном виде.

Ход работы

Задача кодирования в соответствии со стандартом 5G весьма трудоемкая, поэтому в рамках этой работы вам предлагается использовать ряд основных готовых функций и собрать из них систему кодирования. В качестве дополнительного факультативного задания вы можете разработать часть функций самостоятельно.

1. Создайте скрипт, в котором будет реализована модель 5G LDPC кодека (например, *LDPC5G.m*).

1.1 Объявите переменные:

i_ls = 0;

Zc = 8;

1.2 Создайте вектор **msg** и присвойте ему значение, равное 10*Zc* случайных битов.

2. Формирование проверочной матрицы **H**.

2.1 На этом этапе вы будете использовать готовую функцию *tableBG2.m*, которая содержит таблицу всевозможных (определенных стандартом) значений логических сдвигов для матрицы **BG2**. У функции отсутствуют входные аргументы, а возвращает она таблицу **H_BG**. Также вы будете использовать функцию *generate_H.m*. Входные аргументы - параметры Z_c и i_{LS}. Функция *generate_H.m* формирует **H_shift** - матрицу сдвигов и **H** – проверочную матрицу, содержащую биты. Следует отметить, что для реального кодера достаточно знания матрицы **H_shift**, а матрица **H** в нашем случае формируется для ознакомления и для возможности подключения разработанного декодера. Функция *tableBG2.m*, подключена внутрь функции *generate_H.m*.

2.2 Подключите функцию формирования проверочной матрицы:

[H_shift,H] = generate_H(H_BG_type,i_ls, Zc)

3. Кодирование.

3.1 На первом этапе необходимо разработать функцию логического сдвига *mul_sh.m*. Аргументы функции: **x** – входной вектор бит, *k* – величина сдвига. Функция возвращает значение **y** – выходной вектор бит. Логика работы следующая. Если значение сдвига равно «-1», на выходе формируется вектор с одной строкой и *length(x)* столбцов, заполненный нулями, иначе формируется вектор:

y=[x(k+1:end) x(1:k)];

3.2 Создайте функцию кодирования *nrldpc_encoder.m.* Входные аргументы **H_shift** –матрица сдвигов, *Zc* – значение коэффициента сдвига, *msg* – сформированный на первом этапе вектор сообщения. Функция возвращает значение **CW** – кодовое слово.

Логика работы функции может быть следующая:

3.2.1 Определите размеры матрицы **H_shift**, *m* – количество строк, *n* – количество столбцов. Создайте вектор **CW** длиной $n \cdot Zc$. Т.к. код систематический, присвойте первым $(n\text{-}m)Z_c$ битам **CW** значения входного сообщения **msg**. Создайте вектор *temp* размерностью (1, *Zc*).

3.2.2 Для того, чтобы определить первый вектор проверочных бит $\mathbf{w_1}$ ($\mathbf{p_1}$) необходимо выполнить сложение информационных бит, соответствующих первым четырем строкам проверочной матрицы со сдвигом. Создайте цикл с *i* от 1 до 4. Внутри него следующий цикл с j = 1:n-m. Внутри выполните следующее присвоение:

*temp=mod(temp+mul_sh(msg((j-1)*Zc+1:j*Zc), H_shift(i,j)), 2);*

В зависимости от конфигурации проверочной матрицы вектор первых проверочных бит w_1 (p_1) может быть либо во второй, либо в третьей строке. Это нужно проверить с помощью условия: если элемент *H_shift(2,n-m+1)* равен -1 (пустой), тогда *pl_sh = H_shift(3,n-m+1)*;, иначе *pl_sh = H_shift(2,n-m+1);* Присвойте первым проверочным битам кодового слова смещенные на соответствующее значение биты вектора **temp**:

*CW((n-m)*Zc+1:(n-m+1)*Zc) = mul_sh(temp,Zc-pl_sh);*

3.2.3 Для поиска векторов проверочных бит w_2, w_3 и w_4 создайте цикл с *i*=1:3, каждый раз обнуляйте значение вектора **temp**. Далее, внутри этого цикла создайте еще один с *j=1:n-m+i*. Внутри цикла выполните присвоение

*temp = mod(temp + mul_sh(CW((j-1)*Zc+1:j*Zc),H_shift(i,j)),2);*

Закройте этот цикл и внутри цикла с *i* присвойте соответствующей группе проверочных бит внутри кодового слова значение **temp**:

*CW((n-m+i)*Zc+1:(n-m+i+1)*Zc) = temp;*

3.2.4 Найдите оставшиеся значения проверочных бит. Для этого создайте цикл с *i=5:m*. , каждый раз обнуляйте значение вектора **temp**. Далее, внутри этого цикла создайте еще один *с j=1:n-m+4*. Внутри цикла выполните присвоение

*temp = mod(temp + mul_sh(CW((j-1)*Zc+1:j*Zc),H_shift(i,j)),2);*

Закройте этот цикл и внутри цикла с *i* присвойте соответствующей группе проверочных бит внутри кодового слова значение **temp**:

*CW((n-m+i-1)*Zc+1:(n-m+i)*Zc) = temp;*

На этом найдены все проверочные биты и функция кодирования заканчивается.

3.2.5 Подключите функцию *nrldpc_encoder* в скрипт *LDPC5G.m*, подайте на ее входы **H_shift**, *Zc*, **msg**.

4. Для проверки правильности кодирования можно воспользоваться основным свойством проверочной матрицы: ее произведение на кодовое слово при отсутствии ошибок дает нулевой вектор синдромов. Вам необходимо создать функцию проверки *check_cword.m*. Аргументы функции ***H_shift***, *Zc*, ***CW***. Функция возвращает значение *out*, которое в случае прохождения проверки равно «1», в случае не прохождения равно «0».

4.1 Определите размеры матрицы ***H_shift***, *m* – количество строк, *n* – количество столбцов. Создайте вектор **syn** длиной $m \cdot Zc$, 1.

4.2 Создайте цикл с *i=1:m*, внутри него еще один цикл с *j = 1:n*. Внутри этого цикла последовательно рассчитайте $m \cdot Zc$ синдромов:

*syn((i-1)*Zc+1:i*Zc) = mod(syn((i-1)*Zc+1:i*Zc) + mul_sh(CW((j-1)*Zc+1:j*Zc),H_shift(i,j)).',2);*

На этом оба цикла заканчиваются.

4.3 Если хоть один из элементов вектора синдромов равен нулю, значит входной аргумент **CW** не является кодовым словом и функция возвращает значение «0», иначе «1»:

if any(syn)

```
out = 0;
else
out = 1;
end
```

4.4 Проверьте правильность работы алгоритма кодирования: подайте на вход функции *check_cword.m* кодовое слово.

4.5 Создайте вектор **CW_false** содержащий *68·Zc* случайных бит.

4.6 Подайте вектор на вход функции check_cword.m, вектор **CW_false**, убедитесь в том, что функция возвращает значение 0.

В качестве отчета приведите скриншот выхода функции *check_cword* при подаче на вход **CW_false**.

На этом основная часть работы заканчивается.

Дополнительные задания для самостоятельной работы

5.1 Дополнительное задание 1. Внесите в кодовое слово битовые ошибки по аналогии с предыдущей работой. Подключите разработанную функцию BP декодирования *LDPC_decoder.m*. Определите количество ошибок, которое способен исправлять декодер. Обратите внимание, что декодер работает не с матрицей сдвига, а непосредственно с проверочной матрицей **H**. Кроме того, матрица **H**, подаваемая на вход функции должна быть транспонирована (*H.'*).

5.2 Дополнительное задание 2. Постройте график зависимости *BER* при передаче сигналов в канале с АБГШ, по аналогии с факультативным заданием в предыдущей работе. Сравните исправляющую способность LDPC кода в 5G и кода Хэмминга.

Контрольные вопросы:

1. Что такое параметр *Zc* и на что он влияет?

2. Какие размеры матрицы **BG2**?

3. Как осуществляется проверка правильности формирования кодовых слов?

4. В чем заключается функция логического сдвига, выполняемого функцией *mul_sh.m*.

Листинг

Файл tableBG2.m

```
function [H_BG] = tableBG2()
H_BG (1) = {[0  9    174  0    72   3    156  143  145
1    117  97   0    110  26   143  19   131
2    204  166  0    23   53   14   176  71
3    26   66   0    181  35   3    165  21
6    189  71   0    95   115  40   196  23
9    205  172  0    8    127  123  13   112
10   0    0    0    1    0    0    0    1
11   0    0    0    0    0    0    0    0]};
H_BG (2) = {[0  167  27   137  53   19   17   18   142
3    166  36   124  156  94   65   27   174
4    253  48   0    115  104  63   3    183
5    125  92   0    156  66   1    102  27
6    226  31   88   115  84   55   185  96
7    156  187  0    200  98   37   17   23
8    224  185  0    29   69   171  14   9
9    252  3    55   31   50   133  180  167
11   0    0    0    0    0    0    0    0
12   0    0    0    0    0    0    0    0]};
H_BG (3) = {[0  81   25   20   152  95   98   126  74
1    114  114  94   131  106  168  163  31
3    44   117  99   46   92   107  47   3
4    52   110  9    191  110  82   183  53
8    240  114  108  91   111  142  132  155
10   1    1    1    0    1    1    1    0
12   0    0    0    0    0    0    0    0
13   0    0    0    0    0    0    0    0]};
H_BG (4) = {[1  8    136  38   185  120  53   36   239
2    58   175  15   6    121  174  48   171
4    158  113  102  36   22   174  18   95
5    104  72   146  124  4    127  111  110
6    209  123  12   124  73   17   203  159
7    54   118  57   110  49   89   3    199
8    18   28   53   156  128  17   191  43
9    128  186  46   133  79   105  160  75
10   0    0    0    1    0    0    0    1
13   0    0    0    0    0    0    0    0]};
H_BG (5) = {[0  179  72   0    200  42   86   43   29
1    214  74   136  16   24   67   27   140
11   71   29   157  101  51   83   117  180
14   0    0    0    0    0    0    0    0]};
H_BG (6) = {[0  231  10   0    185  40   79   136  121
1    41   44   131  138  140  84   49   41
5    194  121  142  170  84   35   36   169
7    159  80   141  219  137  103  132  88
11   103  48   64   193  71   60   62   207
15   0    0    0    0    0    0    0    0]};
H_BG (7) = {[0  155  129  0    123  109  47   7    137
5    228  92   124  55   87   154  34   72
```

```
7     45    100   99    31    107   10    198   172
9     28    49    45    222   133   155   168   124
11    158   184   148   209   139   29    12    56
16    0     0     0     0     0     0     0     0]};
H_BG (8) = {[1  129   80    0     103   97    48    163   86
5     147   186   45    13    135   125   78    186
7     140   16    148   105   35    24    143   87
11    3     102   96    150   108   47    107   172
13    116   143   78    181   65    55    58    154
17    0     0     0     0     0     0     0     0]};
H_BG (9) = {[0  142   118   0     147   70    53    101   176
1     94    70    65    43    69    31    177   169
12    230   152   87    152   88    161   22    225
18    0     0     0     0     0     0     0     0]};
H_BG (10) = {[1 203   28    0     2     97    104   186   167
8     205   132   97    30    40    142   27    238
10    61    185   51    184   24    99    205   48
11    247   178   85    83    49    64    81    68
19    0     0     0     0     0     0     0     0]};
H_BG (11) = {[0 11    59    0     174   46    111   125   38
1     185   104   17    150   41    25    60    217
6     0     22    156   8     101   174   177   208
7     117   52    20    56    96    23    51    232
20    0     0     0     0     0     0     0     0]};
H_BG (12) = {[0 11    32    0     99    28    91    39    178
7     236   92    7     138   30    175   29    214
9     210   174   4     110   116   24    35    168
13    56    154   2     99    64    141   8     51
21    0     0     0     0     0     0     0     0]};
H_BG (13) = {[1 63    39    0     46    33    122   18    124
3     111   93    113   217   122   11    155   122
11    14    11    48    109   131   4     49    72
22    0     0     0     0     0     0     0     0]};
H_BG (14) = {[0 83    49    0     37    76    29    32    48
1     2     125   112   113   37    91    53    57
8     38    35    102   143   62    27    95    167
13    222   166   26    140   47    127   186   219
23    0     0     0     0     0     0     0     0]};
H_BG (15) = {[1 115   19    0     36    143   11    91    82
6     145   118   138   95    51    145   20    232
11    3     21    57    40    130   8     52    204
13    232   163   27    116   97    166   109   162
24    0     0     0     0     0     0     0     0]};
H_BG (16) = {[0 51    68    0     116   139   137   174   38
10    175   63    73    200   96    103   108   217
11    213   81    99    110   128   40    102   157
25    0     0     0     0     0     0     0     0]};
H_BG (17) = {[1 203   87    0     75    48    78    125   170
9     142   177   79    158   9     158   31    23
11    8     135   111   134   28    17    54    175
12    242   64    143   97    8     165   176   202
26    0     0     0     0     0     0     0     0]};
H_BG (18) = {[1 254   158   0     48    120   134   57    196
5     124   23    24    132   43    23    201   173
11    114   9     109   206   65    62    142   195
12    64    6     18    2     42    163   35    218
27    0     0     0     0     0     0     0     0]};
```

```
H_BG (19) = {[0 220  186  0    68   17   173  129  128
6    194  6    18   16   106  31   203  211
7    50   46   86   156  142  22   140  210
28   0    0    0    0    0    0    0    0]};
H_BG (20) = {[0 87   58   0    35   79   13   110  39
1    20   42   158  138  28   135  124  84
10   185  156  154  86   41   145  52   88
29   0    0    0    0    0    0    0    0]};
H_BG (21) = {[1 26   76   0    6    2    128  196  117
4    105  61   148  20   103  52   35   227
11   29   153  104  141  78   173  114  6
30   0    0    0    0    0    0    0    0]};
H_BG (22) = {[0 76   157  0    80   91   156  10   238
8    42   175  17   43   75   166  122  13
13   210  67   33   81   81   40   23   11
31   0    0    0    0    0    0    0    0]};
H_BG (23) = {[1 222  20   0    49   54   18   202  195
2    63   52   4    1    132  163  126  44
32   0    0    0    0    0    0    0    0]};
H_BG (24) = {[0 23   106  0    156  68   110  52   5
3    235  86   75   54   115  132  170  94
5    238  95   158  134  56   150  13   111
33   0    0    0    0    0    0    0    0]};
H_BG (25) = {[1 46   182  0    153  30   113  113  81
2    139  153  69   88   42   108  161  19
9    8    64   87   63   101  61   88   130
34   0    0    0    0    0    0    0    0]};
H_BG (26) = {[0 228  45   0    211  128  72   197  66
5    156  21   65   94   63   136  194  95
35   0    0    0    0    0    0    0    0]};
H_BG (27) = {[2 29   67   0    90   142  36   164  146
7    143  137  100  6    28   38   172  66
12   160  55   13   221  100  53   49   190
13   122  85   7    6    133  145  161  86
36   0    0    0    0    0    0    0    0]};
H_BG (28) = {[0 8    103  0    27   13   42   168  64
6    151  50   32   118  10   104  193  181
37   0    0    0    0    0    0    0    0]};
H_BG (29) = {[1 98   70   0    216  106  64   14   7
2    101  111  126  212  77   24   186  144
5    135  168  110  193  43   149  46   16
38   0    0    0    0    0    0    0    0]};
H_BG (30) = {[0 18   110  0    108  133  139  50   25
4    28   17   154  61   25   161  27   57
39   0    0    0    0    0    0    0    0]};
H_BG (31) = {[2 71   120  0    106  87   84   70   37
5    240  154  35   44   56   173  17   139
7    9    52   51   185  104  93   50   221
9    84   56   134  176  70   29   6    17
40   0    0    0    0    0    0    0    0]};
H_BG (32) = {[1 106  3    0    147  80   117  115  201
13   1    170  20   182  139  148  189  46
41   0    0    0    0    0    0    0    0]};
H_BG (33) = {[0 242  84   0    108  32   116  110  179
5    44   8    20   21   89   73   0    14
12   166  17   122  110  71   142  163  116
42   0    0    0    0    0    0    0    0]};
```

```
H_BG (34) = {[2 132  165  0    71   135  105  163  46
7    164  179  88   12   6    137  173  2
10   235  124  13   109  2    29   179  106
43   0    0    0    0    0    0    0    0]};
H_BG (35) = {[0 147  173  0    29   37   11   197  184
12   85   177  19   201  25   41   191  135
13   36   12   78   69   114  162  193  141
44   0    0    0    0    0    0    0    0]};
H_BG (36) = {[1 57   77   0    91   60   126  157  85
5    40   184  157  165  137  152  167  225
11   63   18   6    55   93   172  181  175
45   0    0    0    0    0    0    0    0]};
H_BG (37) = {[0 140  25   0    1    121  73   197  178
2    38   151  63   175  129  154  167  112
7    154  170  82   83   26   129  179  106
46   0    0    0    0    0    0    0    0]};
H_BG (38) = {[10219  37   0    40   97   167  181  154
13   151  31   144  12   56   38   193  114
47   0    0    0    0    0    0    0    0]};
H_BG (39) = {[1 31   84   0    37   1    112  157  42
5    66   151  93   97   70   7    173  41
11   38   190  19   46   1    19   191  105
48   0    0    0    0    0    0    0    0]};
H_BG (40) = {[0 239  93   0    106  119  109  181  167
7    172  132  24   181  32   6    157  45
12   34   57   138  154  142  105  173  189
49   0    0    0    0    0    0    0    0]};
H_BG (41) = {[2 0    103  0    98   6    160  193  78
10   75   107  36   35   73   156  163  67
13   120  163  143  36   102  82   179  180
50   0    0    0    0    0    0    0    0]};
H_BG (42) = {[1 129  147  0    120  48   132  191  53
5    229  7    2    101  47   6    197  215
11   118  60   55   81   19   8    167  230
51   0    0    0    0    0    0    0    0]};
```

Файл generate_H.m

```
function [H_shift,H] = generate_H(H_BG_type,i_ls, Zc)
if isequal(H_BG_type,'H_BG1')
    H_BG_table = tableBG1();
    H_BG_I      = 46;
    H_BG_J      = 68;
elseif isequal(H_BG_type,'H_BG2')
    H_BG_table = tableBG2();
    H_BG_I      = 42;
    H_BG_J      = 52;
end
    H_BG        =  zeros(H_BG_I,H_BG_J);
    H_shift     =  -1*ones(H_BG_I,H_BG_J);
    Vij_mass = -1*ones(H_BG_I,H_BG_J);
    for i = 1:H_BG_I
        table_row =cell2mat(H_BG_table(i));
        for k=1:size(table_row,1)
            H_BG(i,table_row(k,1)+1) = 1;
            Vij_mass(i,table_row(k,1)+1) = table_row(k,i_ls+2);
        end
    end
```

```
I_matrix = eye(Zc,Zc);

for i = 1:H_BG_I
    for j = 1:H_BG_J
        if H_BG(i,j)==0
            H((i-1)*Zc+1:i*Zc, (j-1)*Zc+1:j*Zc) = zeros(Zc,Zc);
        else
            table_row   =   cell2mat(H_BG_table(i));
            Vij         =   Vij_mass(i,j);
            Pij         =   mod(Vij,Zc);
            H_shift(i,j)=   Pij;
            I_shift     =   circshift(I_matrix, -Pij);
H((i-1)*Zc+1:i*Zc, (j-1)*Zc+1:j*Zc) = I_shift;
        end
    end
end
```

Практическая работа № 9

Декодирование LDPC кодов алгоритмом Bit Flipping

Цель работы: Реализация и исследование алгоритма декодирования Bit Flipping.

Задачи практической работы:

1) Формирование битового сообщения.
2) Реализация блочного кодера и формирование кодовых слов.
3) Реализация декодера на основе алгоритма Bit Flipping.
4) Исследование процесса исправления ошибок.

Оборудование и программное обеспечение: Octave, version 6.4.0.

Теоретический материал

Блочные коды

Рассмотрим принципы работы систематических блочных кодов, рисунок 1. Информационная последовательность разбиваются на блоки из K символов. Кодирование заключается в отображении этого блока в блок длиной N > K, который называется кодовым словом. Величина N-K является избыточностью кода, а R = K/N – скоростью кода. Такое отображение можно представить в матричном виде:

$$\mathbf{xG} = \mathbf{cw}, \tag{1}$$

где **x** – блок кодируемых символов длиной K, **G** – порождающая матрица размерностью K × N, **cw** – кодовое слово длиной N.

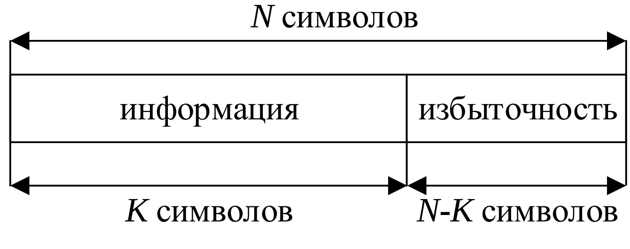


Рисунок 1 – Добавление избыточности в блочный код (N, K)

Часто кодовые слова представляют в систематической форме **C** = (**U**, **V**), где **V** – вектор с добавленной избыточной информацией. В этом случае генераторная матрица должна иметь вид:

$$\mathbf{G}=[\mathbf{IP}]=\begin{bmatrix} 1 & 0 & \dots & 0 & g_{1,1} & \dots & g_{1,\mathrm{n-k}} \\ 0 & 1 & \dots & 0 & g_{2,1} & \dots & g_{2,\mathrm{n-k}} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 & g_{k,1} & \dots & g_{\mathrm{k,n-k}} \end{bmatrix}, \quad (2)$$

где **I** – единичная матрица размером K × K, матрица **P** формирует проверочные символы.

Пусть y – битовая последовательность, поступающая на вход декодера,

$$\mathbf{y}=\mathbf{cw}+\mathbf{e},$$

где **e** – вектор ошибок, длина которого совпадает с длиной кодового слова (N), если ошибок нет, **e** содержит только нулевые элементы. Если в канале связи произошла ошибка, например, в третьем бите, а N = 7, то **e**=[0 0 1 0 0 0 0].

Обнаружение ошибок может быть осуществлено путем вычисления синдромов декодером:

$$\mathbf{S}=\mathbf{yH}, \quad (3)$$

где **S** – вектор синдромов длиной N - K, **H** – проверочная матрица размерностью N - K × N такая, что выполняется свойство ортогональности:

$$\mathbf{GH}=0. \quad (4)$$

Такую матрицу можно получить из порождающей вида (2) как,

$$\mathbf{H}=\begin{bmatrix} \mathbf{P} \\ \mathbf{I}_{\mathrm{n\text{-}k}} \end{bmatrix}.$$

Если при прохождении сигнала через канал и его демодуляции, было принято сообщение без ошибок, то каждый блок этого сообщения по-прежнему является кодовым словом и значит, в соответствии с (1), (3) и (4) выполняется условие

$$\mathbf{S}=\mathbf{cwH}=\mathbf{UGH}=0. \quad (5)$$

Это важное свойство позволяет обнаружить и исправить ошибки. Так, если вектор синдромов **S** отличен от нуля (поэтому также вычисление синдромов называют проверками на четность), то принятый блок не является кодовым словом, и по значению **S** мы можем определить какой бит был определен неверно. На этом основано исправление ошибок по таблице синдромов.

LDPC коды

Классические алгоритмы декодирования малоприменимы для LDPC кодов, т.к. количество всевозможных кодовых слов растет экспоненциально с увеличением длины кодового слова.

На практике для декодирования LDPC кодов используются различные варианты итерационных декодеров. Галлагером было предложено два варианта декодирования: инверсия бит (BF, Bit Flip) и алгоритм распространения доверия (BP, Belief Propagation).

BF – алгоритм, работающий с демодуляторами, имеющими «жесткий» выход. Суть данного метода заключается в следующем:

1) Вычисляются проверки на четность, в соответствии с матрицей **H**.
2) Инверсия бита, который был задействован в наибольшем количестве непрошедших проверок.

Т.к. алгоритм итеративный, то после каждой итерации проверяется синдром $\mathbf{S} = \mathbf{H} \cdot \mathbf{c}^{T}$, где **c** – кодовое слово. Декодирование будет продолжаться, пока синдром не станет нулевым или количество итераций не достигнет максимума.

Алгоритм BP имеет схожие черты с алгоритмом BF, но сложнее, т.к. он работает не с жесткими решениями демодулятора, а с LLR.

В первозданном виде алгоритм Галлагера практически не применяется по причине ограниченной эффективности. Вместо него применяются различные вариации данного алгоритма, такие, как: MPA (Message Passing Algorithm, алгоритм передачи сообщения) и SPA (Sum-Product Algorithm, алгоритм суммы-произведения), они имеют сравнительно меньшую вычислительную.

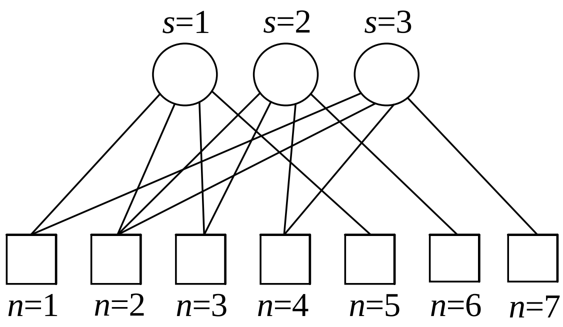
Важным понятием в теории LDPC кодов является представление разреженных матриц в виде двудольных графов, которые также называются графами Таннера. На рисунке 2 изображена генераторная (**G**), проверочная (**H**) матрицы кода Хэмминга (K=4, N=7) и соответствующий матрице **H** граф Таннера.

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

(а)

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(б)



(в)

Рисунок 2 – Генераторная (а) и проверочная (б) матрицы кода Хэмминга (4,7), граф Таннера (в), соответствующий проверочной матрице

Вершины графа Таннера для LDPC кодов разделены на две группы. Первая группа (круги) соответствует столбцам, а вторая (квадраты) строкам проверочной матрицы **H**. Ребро графа соединяет вершину из первой группы n и вершину из второй группы s, если на пересечении n-ой строки и s-го столбца матрицы **H** расположен единичный элемент. Так, например, первая вершина из группы столбцов соединена с первой, второй, третьей и пятой вершинами из группы строк, поскольку в этих строках первого столбца расположены единицы. Нетрудно догадаться, что группа столбцов соответствует синдромам, рассчитанным с помощью (3). Так, первый синдром можно получить, сложив первый, второй, третий и пятый биты принятой последовательности.

Алгоритм Bit Flipping

В данной работе будет рассмотрен только алгоритм Bit Flipping, поэтому рассмотрим его подробней.

Процесс декодирования можно представить с помощью графа Таннера. В качестве примера, для уменьшения размера графа приведен не LDPC, а код Хэмминга, генераторная и проверочная матрицы которого приведены на рисунке 2.

Предположим, что на вход кодера поступает битовое сообщение **x** = [1 1 0 1], на выходе кодера ему соответствует кодовое слово **cw** = [1 1 0 1 0 0 1], которое поступает в канал связи. В канале произошла ошибка во втором бите (вектор ошибок **e** = [0 1 0 0 0 0 0]). Таким образом, на выходе канала имеется вектор [1 0 0 1 0 0 1]. Процесс декодирования приведен на рисунке 3.

Входные биты декодера записываются в вершины графа, соответствующие строкам проверочной матрицы. После этого рассчитываются проверки (синдромы) и записываются в вершины, соответствующие столбцам. Для каждой вершины из группы строк рассчитывается количество непрошедших проверок (с единичным значением синдромов). Далее инвертируется бит, участвующий в максимальном числе непрошедших поверок. В случае, если после этого все синдромы равны нулю, принимается решение о верном декодировании, в противном случае процесс декодирования продолжается.

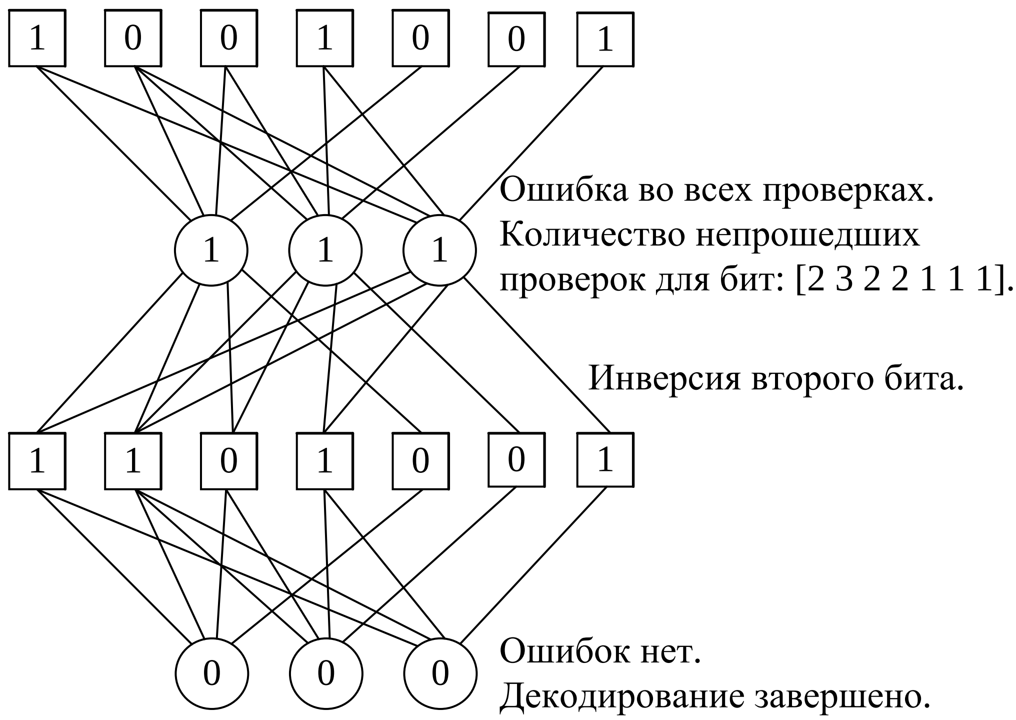


Рисунок 3 – Иллюстрация метода BF на графе Таннера

В вышеописанном примере количество непрошедших проверок максимально для второго бита. Однако, может возникнуть ситуация, когда это количество будет одинаковым для нескольких бит. В этом случае нужно посмотреть процент непрошедших проверок от общего количества проверок для каждого из этих бит. Бит с максимальным процентом непрошедших проверок инвертируется.

Ход работы

Вам предстоит реализовать формирование битового сообщения, его кодирование, добавление ошибки, разработать декодер на основе алгоритма BF и провести исследование кодека. В данной работе исследование LDPC кодека мы будем проводить на примере работы с кодом Хэмминга (4,7).

1. Создайте скрипт, в котором будет реализована модель LDPC кодека (например, *LDPC.m*).

1.1 Объявите переменные:

$K = 4$ – длина информационного сообщения,

N *= 7* – длина кодового слова,

N_iter_BF = 7 – максимальное число итераций алгоритма BF.

G и ***H*** , заполните значениями матриц кода Хэмминга, рисунок 2.

1.2 Объявите вектор ошибок ***e*** *= [0 0 0 0 0 0 0]*, в дальнейшем вы будете менять его значения для исследования исправляющей способности кода.

1.3 Создайте вектор **bits** и присвойте ему значение, равное **K** случайных битов. Сделать это можно с помощью функции *rand*, которая формирует случайные значения чисел от 0 до 1 и последующим округлением функцией *round*.:

bits = round(rand(1,K));

2. Создайте функцию блочного LDPC кодера, например, *ldpc_encoder.m*. Аргументы функции – вектор входных бит **bits**, и генераторная матрица **G**. Выходное значение **codeword**.

2.1 Выполните кодирование – присвойте выходному значению **codeword** результат умножения вектора **bits** на генераторную матрицу **G**. Т.к все процессы кодирования и декодирования выполняются с помощью операций по модулю два, проверьте результат умножения на четность, сделать это можно с помощью функции поиска остатка от деления mod.

codeword = mod(codeword,2);

3. Создайте функцию LDPC декодера, например, *LDPC_decoder.m*. Аргументы функции – вектор входных бит **bits**, проверочная матрица **H**, длина информационного сообщения **K**, длина кодового слова ***N***, количество итераций алгоритма **N_iter_BF**. Выходное значение **r_bits**.

Логика работы декодера может выглядеть следующим образом. Возможна и альтернативная реализация кода.

3.1 Рассчитайте значения проверок (вектора синдромов): умножьте входное сообщение на проверочную матрицу и проверьте результат на четность:

$$S = y*H;$$
$$S = mod(S,2);$$

3.2 Объявите счетчик итераций алгоритма BF **n** и присвойте ему нулевое значение. Создайте цикл с условием (*while*): сумма синдромов не равна нулю (если хотя бы один синдром будет ненулевым, сумма тоже будет ненулевой) и счетчик синдромов не превысил максимальное значение.

Внутри цикла должна быть реализована следующая логика:

3.2.1 К счетчику каждую итерацию добавляется единица.

3.2.2 Для каждого бита рассчитывается количество непрошедших проверок. Создается счетчик непрошедших проверок *failed_check_bits* размерностью 1x***N*** и заполняется нулями. Создается цикл со счетчиком *i* от 1 до *N*-*K* (количество синдромов). Внутри цикла *i*-ый синдром **s**(*i*) умножается на столбец проверочной матрицы **H**(:,*i*) и добавляется к значению *failed_check_bits*.

3.2.3 С помощью функции max ищется максимальное число непрошедших проверок *max_failed_bit* и индекс соответствующего бита *max_failed_bit_num* в векторе счетчика непрошедших проверок *failed_check_bits*:

[max_failed_bit,max_failed_bit_num] = max(failed_check_bits);

3.2.4 Далее необходимо найти количество бит с максимальным числом непрошедших проверок *num_max_failed*:

num_max_failed=length(find(failed_check_bits==max_failed_bit));

3.2.5 Если такой бит один, он инвертируется и заново рассчитывается вектор синдромов **S**. На этом текущая итерация цикла *while* заканчивается.

3.2.6 Если бит не один, для каждого из них необходимо рассчитать процент непрошедших проверок от общего числа. Для каждого бита в цикле с *j* от 1 до *num_max_failed* выполняем следующие действия:

записываем номер бита (*max_failed_bit_num*) в переменную *number_of_max_failed_bits(j)*, обнуляем для него значение счетчика *failed_check_bits(max_failed_bit_num)* и рассчитываем для *j*-го бита отношение числа непрошедших проверок к общему числу проверок (количеству единиц в соответствующей биту строке проверочной матрицы). Ищем следующий бит с максимальным числом непрошедших проверок и возвращаемся к началу цикла.

После нахождения для всех бит с максимальным числом непрошедших проверок отношения этого значения к общему числу проверок, находим бит с максимальным процентом. Инвертируем этот бит.

Рассчитываем вектор синдромов **S**. На этом текущая итерация цикла *while* заканчивается.

3.3 В конце программы, после основного цикла выходному значению функции *r_bits* присваивается *K* первых бит исправленного вектора **y**.

4. Подключите функцию кодирования разработанный на первом этапе скрипт (*LDPC.m*). К вектору кодового слова **codeword** добавьте вектор ошибок **e**. Результат y подайте на вход функции декодирования. Убедитесь, что при отсутствии ошибок в канале связи (вектор **e** содержит только нули), декодированное сообщение *r_bits* соадает с исходным *bits*.

num_errors = length(find(bits-r_bits))

Добавьте одиночную ошибку, убедитесь, что декодер исправляет ее вне зависимости от позиции.

4.1 Проверьте исправляющую способность кода при другом числе ошибок.

В качестве отчета по работе приведите скриншот зависимости количества ошибок на выходе декодера от количества ошибок на входе кодера.

Дополнительные задания для самостоятельной работы

1. Проведите тестирование декодера для разработанного на прошлом занятии кодера.

2. Встройте LDPC кодек в разработанную ранее имитационную модель системы связи. Постройте зависимости *BER(SNR)* системы связи с LDPC кодеком.

Контрольные вопросы:

1. Что такое генераторная и проверочная матрицы блочного кода? Как закодировать сообщение зная генераторную матрицу?

2. Что такое граф Таннера, как его можно получить из проверочной матрицы?

2. В каком случае принимается решение о выходе из итераций алгоритма BF?

Практическая работа №10
Полярные коды.

Цель работы: Изучение принципов полярного кодирования.

Задачи практической работы:

1) Изучение теории полярных кодов;

2) Формирование генераторной матрицы;

3) Реализация модели полярного кодера.

Оборудование и программное обеспечение: Octave.

Теоретический материал

Полярные коды – несистематические блочные коды, длина кодовых слов составляет $N = 2^n$ (2, 4, 8, 16 и т.д.). Длина сообщения K составляет от 1 до N. Кодовое расстояние d_{min} этого кода, а соответственно и его исправляющая способность зависит от скорости кода, то есть отношения K к N. Чем меньше K по отношению к N, тем большее число ошибок получится исправить.

Процесс кодирования полярных (как и любых блочных) кодов может быть задан как произведение входных битов на генераторную матрицу:

$$\mathbf{d} = \mathbf{uG}, \tag{1}$$

где **u** – последовательность битов, поступающая на вход умножителя (в общем виде не совпадает с сообщением **m**).

Базовой генераторной матрицей размером (2,2) является так называемая матрица Арикана (образующая матрица полярного кода):

$$\mathbf{G} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}. \tag{2}$$

Эту матрицу также называют образующей матрицей полярного кода, потому что все остальные матрицы получаются из нее. Отметим, что все генераторные матрицы для полярных кодов вне зависимости от N квадратны, то есть количество строк и столбцов равны.

Матрицы **G** размерностью N больше 2 получаются путем произведения Кронекера матрицы Арикана на матрицу размером N/2. Так, чтобы получить матрицу размером N=4 нужно заменить единичные элементы образующей матрицы на ее саму, а вместо нулевого элемента поставить матрицу заполненную нулями.

Чтобы прояснить процедуру кодирования, вернемся к формирующей матрице размером N=2. В этом случае процедура кодирования сводится к:

$$\mathbf{d} = \begin{bmatrix} d_1 & d_2 \end{bmatrix} = \begin{bmatrix} u_1 & u_2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} u_1 + u_2 & u_2 \end{bmatrix}. \tag{3}$$

В этой процедуре мы можем рассматривать u_1, u_2 как входные битовые каналы, а d_1 и d_2 как выходные. Таким образом информация о первом входном канале u_1 находится только в первом выходном канале d_1. А информация о u_2 находится сразу в двух выходных каналах d_1 и d_2. Такая операция называется полярным преобразованием, потому что мы как бы поляризуем входные битовые каналы – делаем одни более помехоустойчивыми (надежными), а другие менее.

Здесь канал битов u_2 будет более надежным, потому что помимо того, что его биты передаются в неизменном виде в d_2, так еще и передаются параллельно с u_1 в d_1. Канал u_1 напротив, менее помехоустойчивый, поскольку информация о нем есть только в канале d_1. Также эта процедура элементарного полярного преобразования иллюстрируется рисунком 1

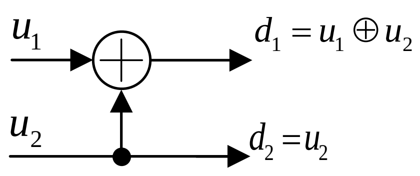


Рисунок 1– Элементарное полярное преобразование

Аналогичные схемы можно привести и для больших N. На рисунке 2 приведена генераторная матрица $\mathbf{G}_4$ и соответствующая ей схема кодера.

$$\mathbf{G}_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

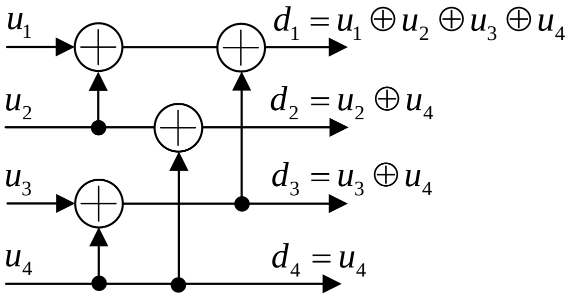


Рисунок 2 – Генераторная матрица и схема кодера для $N = 4$

Обобщенный канал передачи системы связи с полярным кодированием приведен на рисунке 3 Поток данных **u** поступает на блок генераторной матрицы G_N, после чего формируется кодовое слово **d**. После модуляции (например, BPSK) и прохождения канала распространения радиоволн (РРВ) на приемную сторону поступает вектор $\mathbf{r} = [r_1, r_2, \ldots, r_N]$.

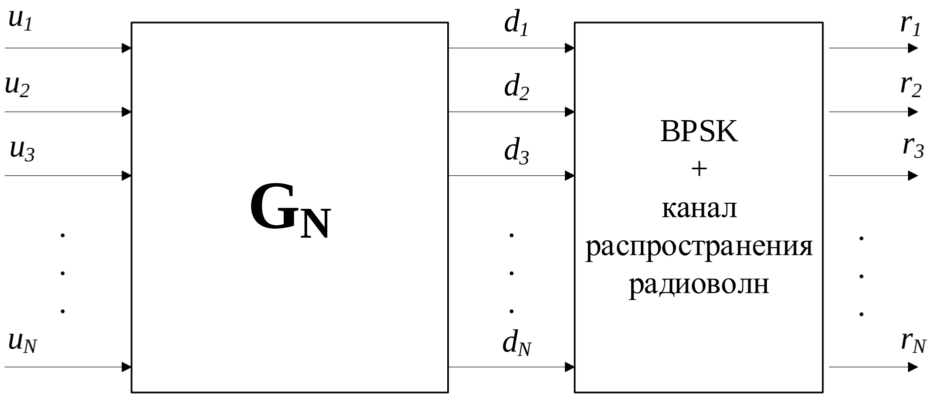


Рисунок 3 – Канал передачи полярной последовательности

Надежность входного канала полярного кода зависит от количества проверочных битов (каналов), в формировании которых он участвует. Чем больше каналов несет информацию о входном бите, тем более помехоустойчива его передача. Информацию о помехоустойчивости каналов можно получить из так называемой последовательности надежности **Q**.

Рассмотрим алгоритм ее получения. На рисунке 4 приведена генераторная матрица **G** для *N*=8, а справа от ее каждой строки приведены значения V_n, равные весу (количеству единиц) ее строк.

$$\mathbf{G}_8 = \begin{array}{c} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \end{array} \begin{array}{l} V_1 = 1 \\ V_2 = 2 \\ V_3 = 2 \\ V_4 = 4 \\ V_5 = 2 \\ V_6 = 4 \\ V_7 = 4 \\ V_8 = 8 \end{array}$$

$$\boxed{\mathbf{Q} = [1\ 2\ 3\ 5\ 4\ 6\ 7\ 8]}$$

Рисунок 4 – Генераторная матрица, веса строк и последовательность надежности полярных кодов

Сортируя каналы от наименее к наиболее надежным можно получить **Q** (также называемую полярной последовательностью). В полярной последовательности первый бит u_1 имеет наименьшую надежность, так как участвует в формировании только одного канала d_1.

Идея полярного кодирования заключается в заморозке наименее надежных каналов. Это значит, что в них мы будем передавать так называемые замороженные биты, или frozen bits. То есть биты с постоянным значением, например 0. Если длина кодового слова *N*, а длина сообщения *K*, то по этому принципу замороженными будут *N*-*K* битов. Тогда процедура кодирования описывается выражением:

$$\mathbf{d} = \begin{bmatrix} f_1 & \ldots & f_{N-K} & m_1 & \ldots & m_K \end{bmatrix} \mathbf{G}, \tag{4}$$

где $f_i = 0$, m_i – биты сообщения.

Ход работы

В ходе работы вам предстоит реализовать алгоритм полярного кодирования.

Скопируйте в папку с проектом файл:

PolarSeq.mat

В данной модели кодер будет реализован через произведение вектора сообщения на генераторную матрицу.

Для начала введем исходные данные.

```
clc; clear; close all;
```

Одним из важнейших параметров является полярная последовательность, её нужно загрузить используя функцию *load()*.

```
load('PolarSeq.mat')
```

Введем оставшиеся параметры. Задайте длину сообщения равную 4, размер кодового слова 8 и значения битов вектора сообщения *msg*.

```
K = ...;                    % Длина сообщения
N = ...;                    % Длина кодового слова
msg = [1 1 0 1]             % Последовательность битов
```

Для полярного кодирования требуется определить полярную последовательность и сформировать генераторную матрицу.

Первым этапом определим полярную последовательность. Для N=1024 последовательность Q сохранена в файле *PolarSeq.mat*. Получить последовательность *Q1* для меньших N можно исключая из Q все элементы больше N:

```
Q1 = Q(Q<N)+1;
```

Единица добавляется, поскольку по стандарту в Q индексация начинается с нулевого элемента, а в Octave с первого.

Создадим вектор последовательности битов на входе кодера *u* и заполним его нулями:

u = zeros(1,N);

Первые *N-K* битов в u будут замороженными (имеют значения 0). В оставшиеся позиции (всего их будет *K*) запишем вектор сообщения:

u(Q1(N-K+1:end)) = msg;

После разделения каналов от менее надежных до более надежных, приступим к формированию кодового слова с помощью генераторной матрицы. Для начала задайте матрицу Арикана *A1:*

A1 = […];

Теперь сформируем генераторную матрицу, она должны быть квадратной, то есть количество строк должно быть равно количеству столбцов. Т.к. матрица формируется с помощью произведения Кронекера, воспользуемся встроенной функцией *kron*:

G = 1;

for i = 1:log2(N)

G = kron(G,A1);

end

При правильно заданной матрице Арикана и написанном алгоритме для кодового слова длиной 8, результат должен быть следующим:

G [8x8 double]

	1	2	3	4	5	6	7	8
1	1	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0
3	1	0	1	0	0	0	0	0
4	1	1	1	1	0	0	0	0
5	1	0	0	0	1	0	0	0
6	1	1	0	0	1	1	0	0
7	1	0	1	0	1	0	1	0
8	1	1	1	1	1	1	1	1

Рисунок 5 – Генераторная матрица 8×8

Заключительным этапом кодирования поводится перемножение генераторной матрицы и сформированной последовательности **u**, для формирования кодового слова.

*cword = mod(u*G,2)';*

Для отчета по проделанной работе выведите в командную строку закодированную последовательность битов, при подаче на кодер сообщения [1 1 0 1] и сделайте скриншот.

Дополнительное задание для самостоятельной работы:

1. Закодируйте сообщение [0 0 1 0 1 1] (не забудьте поменять размер сообщения в исходных данных) при длине кодового слова 16.

Контрольные вопросы:

1. В чем заключается суть поляризации канала?
2. Что такое полярная последовательность? Для чего используется заморозка бит?
3. Каким образом происходит формирование кодового слова в полярных кодах? Что такое генераторная матрица Арикана и как она выглядит?

Приложение. Алгоритм SC для декодирования полярных кодов

Простейшим алгоритмом декодирования полярных кодов является SC, потому что в данном алгоритме в каждой итерации принимается единственное решение. Декодирование выполняется в течении ряда итераций. Их количество совпадает с количеством ветвей древовидной диаграммы, исключая младшее поколение потомков. Для каждой вершины выполняются следующие действия:

- Пересчет значений для левого потомка (левой входящей ветви вершины) на основании входных значений L (LLR).
- Принятие решений для правого потомка.
- Восстановление кодового слова.

Последовательность шагов поясним далее. Пусть $a_1 \ldots a_M$ – значения LLR, поступающие для левого потомка, $b_1 \ldots b_M$ – LLR – значения, поступающие от правого потомка. Значение M зависит от рассматриваемого поколения ветвей. Так, во втором поколении M=1 (на вход от каждого потомка поступает по одному значению), в третьем M=2 и т.д.

На рисунке 6 приведено графическое пояснение первого шага декодирования.

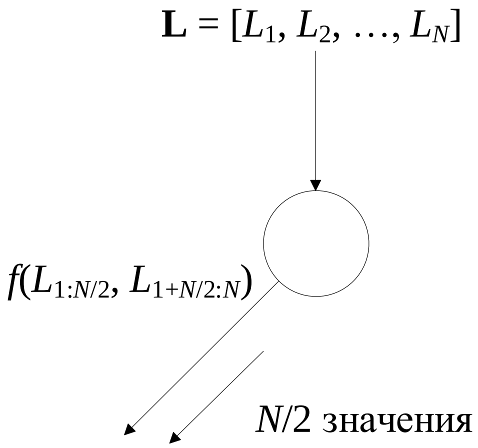


Рисунок 6 – Принятие решения для левого потомка в старшем поколении

Первый этап сводится к принятию решений для левого потомка M-ого порядка. Его можно представить следующим образом:

$$\mathrm{minsum}(a_{1:M},b_{1:M})=[\ \mathrm{minsum}(a_1,b_1),\ \mathrm{minsum}(a_2,b_2),\ \ldots,\ \mathrm{minsum}(a_M,b_M)],$$

где minsum – операция минимальной суммы аппроксимации [14]

$$f(a,b) = \mathrm{sign}(a) \cdot \mathrm{sign}(b) \cdot \min(|a|,|b|),$$

где sign – оператор получения знака числа, min – минимальное значение пары чисел.

Второй этап - пересчет значений для правого потомка по кодовому древу, представлен на рисунке 1.13:

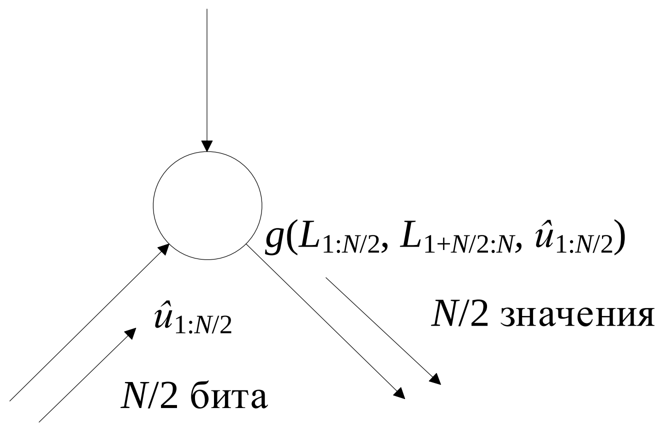


Рисунок 7 – Принятие решения для правого потомка в старшем поколении

Второй этап – принятие решения для правого потомка в старшем поколении можно описать следующим образом:

$$g(a_{1:M},b_{1:M},\hat{u}_{1:N/2}) = [g(a_1,b_1,\hat{u}_1), g(a_2,b_2,\hat{u}_2), \ldots, g(a_M,b_M,\hat{u}_{N/2})], \quad (1.6)$$

где $\hat{u}$ – жесткое решение, полученное от LLR, пересчитанных в левой ветви. Значение $\hat{u}$ может быть получено по знаку LLR: LLR>0 соответствует значению бита 0 и наоборот, LLR<0 соответствует значению бита 1.

Расчёт каждой условной суммы между двумя LLR и оцененного бита в предыдущей метрике выполняется с помощью функции

$$g(a,b,c) = b + (1-2c) \cdot a, \quad (1.7)$$

где a – значение LLR левой ветви, b – значение LLR правой ветви.

Таким образом, если значения бита $\hat{u}$ в левой ветви 0, то a и b складываются, а если значение $\hat{u} = 1$, то из b вычитается а.

Следующим этапом является процедура восстановления кодового слова, представленная на рисунке 8. Выполняется во всех итерациях, кроме последней. Этот этап эквивалентен полярной трансформации - кодированию.

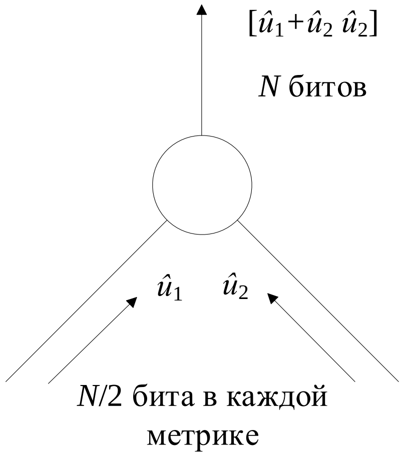


Рисунок 8 – Восстановление кодового слова

В следующей итерации эти же этапы повторяются для следующей ветви в порядке обхода графа (порядок будет указан далее).

Приведем пример декодирования кодового слова для K=2, N=2 без избыточности и добавления шумов (пример базовой полярной трансформации). Пусть **m** = [0 1]. Поскольку избыточность не добавляется, то **u**=**m**. Тогда кодовое слово **d** = [u_1+u_2 u_2] = [1 1]. Без добавления шумов LLR на входе приемника будет [-4 -4].

Тогда этап 1 алгоритма SC даст:

$$f(-4,-4) = sign(-4)\cdot sign(-4)\cdot \min(|-4|,|-4|) = -1\cdot(-1)\cdot 4 = 4 .$$

Жесткая оценка этого результата дает $\hat{u}_1$ = 0.

Этап 2 выполняется следующим образом:

$$g(a,b,c) = g(-4,-4,0) = b + (1-2c)\cdot a = -4 + (1-0)\cdot(-4) = -8 .$$

Жесткая оценка $\hat{u}_2$ = 1. На этом декодирование завершается.

В случае, если эта итерация не последняя, выполнится последующее восстановление кодового слова.

Рассмотрим пример алгоритма декодирования SC с добавлением избыточности и внесением искажений в канал связи. Пусть сообщение msg = [1 0 1], тогда u = [0 1 0 1] (первый бит заморожен), кодовое слово **d** = [0 0 1 1] модулируется (BPSK) и проходит через канал. В канале возникли искажения и после демодулятора были сформированы следующие значения LLR для каждого бита: LLR_1 = 2.75, LLR_2 = 3.5, LLR_3 = - 3.3, LLR_4 = -4.12, тогда **r =** [2.75

3.5 -3.3 -4.12]. Последовательность шагов алгоритма оценивания приведена на рисунке 1.15.

Номер над каждым ребром дерева означает номер шага. На рисунке приняты следующие обозначения: r – значения *LLR* поступающие на декодер, L_l – оценка LLR в левой ветви, L_r – оценка LLR в правой ветви, L_1, L_2, L_3, L_4 – оценки LLR, f – функция минимальной суммы, g – функция условной суммы, $\hat{u}_1$, $\hat{u}_2$, $\hat{u}_3$, $\hat{u}_4$ – жесткие решения декодированных бит.

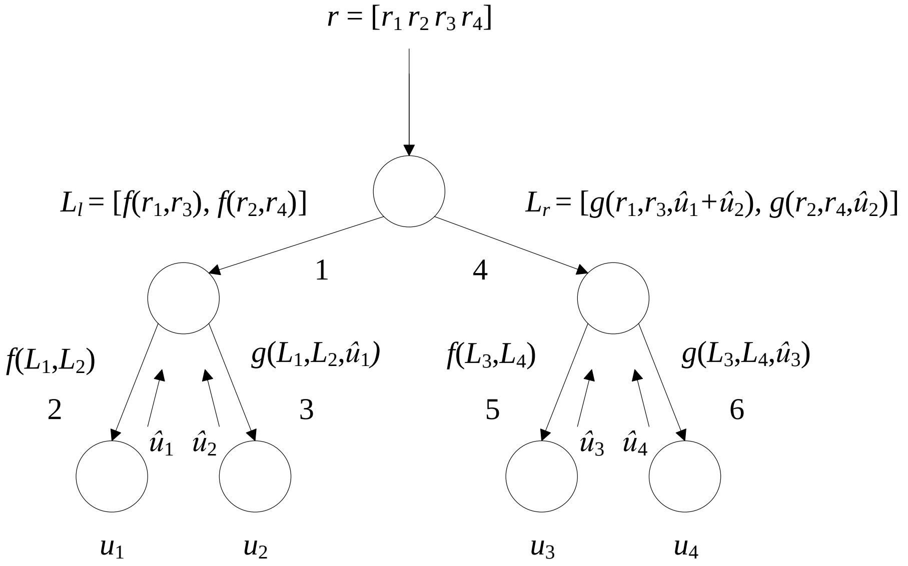


Рисунок 9 – Алгоритм декодирования методом SC

Проведем декодирование при заданных параметрах. Первым этапом проводим принятие решения по левому потомку в первой ветви (шаг 1):

$$L_l = [(\mathrm{sign}(r_1)\cdot\mathrm{sign}(r_3)\cdot\min(|r_1|,|r_3|)), (\mathrm{sign}(r_2)\cdot\mathrm{sign}(r_4)\cdot\min(|r_2|,|r_4|))] =$$

$$= [(\mathrm{sign}(2.75)\cdot\mathrm{sign}(-3.3)\cdot\min(|2.75|,|-3.3|)), (\mathrm{sign}(3.5)\cdot\mathrm{sign}(-4.12)\cdot\min(|3.5|,|-$$

$$-4.12|))] = [-2.75\ -3.5]$$

После оценки первого потомка, проводится принятие решения о следующих двух потомках, так же начинаем расчёт с левой метрики (шаг 2):

$$f = \mathrm{sign}(L_1)\cdot\mathrm{sign}(L_2)\cdot\min(|L_1|,|L_2|) = \mathrm{sign}(-2.75)\cdot\mathrm{sign}(-3.5)\cdot\min(|-2.75|,|-3.5|)=$$

$$= 2.75$$

На этом этапе можно сформировать жесткое решение для первого декодированного бита $\hat{u}_1 = 0$.

Далее по полученному результату в левой метрике, проводится оценка правой (шаг 3):

$$g = L_2 + (1 - 2\cdot\hat{u}_1)\cdot L_1 = -3.5 + (1 - 2\cdot 0)\cdot -2.75 = -6.25$$

Полученное значение имеет отрицательный знак, следовательно, оценка $\hat{u}_2 = 1$.

Пройдя левый путь и получив два декодированных бита приступаем к обходу правой ветви (шаг 4).

$$L_r = [(r_3 + (1\text{-}2\cdot(\hat{u}_1 + \hat{u}_2))\cdot r_1), (r_4 + (1\text{-}2\cdot \hat{u}_2)\cdot r_2)] =$$

$$= [(-3.3 + (1\text{-}2\cdot(0+1))\cdot(2.75)), (-4.12 + (1\text{-}2\cdot 1)\cdot 3.5)] = [-6.05\ -7.62]$$

Проводим оценку для левого потомка (шаг 5):

$$f = \mathrm{sign}(L_3)\cdot\mathrm{sign}(L_4)\cdot\min(|L_3|,|L_4|) = \mathrm{sign}(-6.05)\cdot\mathrm{sign}(-7.62)\cdot\min(|-6.05|,|-7.62|) = 6.05$$

Полученный результат имеет положительное значение, следовательно, $\hat{u}_3 = 0$.

Декодировав третий бит, можно провести оценку последнего потомка (шаг 6).

$$g = L_4 + (1\text{-}2\cdot \hat{u}_3)\cdot L_3 = -7.62 + (1\text{-}2\cdot 0)\cdot(-6.05) = -13.67$$

Результат оценки имеет отрицательное значение, из этого следует, что $\hat{u}_4 = 1$.

Результатом декодирования принятого сообщения является последовательность $\hat{\mathbf{u}} = [0\ 1\ 0\ 1]$. Последовательность совпадает с переданной, таким образом ошибок нет.