МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Политехнический институт Факультет информационных технологий и электроники	Кафедра «Информационная безопасность систем и технологий»
Специальность 10.05.03–	Информационная безопасность автоматизированных систем
Специализация -	Защищенные автоматизированные системы управления

ДИПЛОМНАЯ РАБОТА

на тему

ИМИТАЦИОННАЯ МОДЕЛЬ СИСТЕМЫ ПЕРЕДАЧИ ДАННЫХ НА ОСНОВЕ СВЕРТОЧНОГО КОДИРОВАНИЯ

Студент _____ Ольшевская Ольга Игоревна

Руководитель д.т.н., доцент _____ Егорова Н.А.

Нормоконтролёр 7.06.23 Егорова Н.А.

Рецензент начальник лаборатории АО «ПНИЭИ» 16.06.23 Старкин А.В.

Работа допущена к защите (протокол заседания кафедры от 16.06.23 № 10)

Заведующий кафедрой _____ С.Л. Зефиров

Работа защищена с отметкой отлично (протокол заседания ГЭК от 27.06.23 № 6)

Секретарь ГЭК _____ А.П. Иванов

Подпись Старкина А.А. удостоверяю

Ведущий инженер _____ И.Л. Орлова

АКЦИОНЕРНОЕ ОБЩЕСТВО «ПНИЭИ» ОТДЕЛ КАДРОВ

Пенза 2023

Содержание

Введение 4
1 Анализ угроз безопасности информации при ее передаче по каналам связи ...6
2 Анализ алгоритма сверточного кодирования и алгоритмов сверточного декодирования 8
2.1 Принципы работы и параметры сверточного кодера 8
2.2 Обзор методов декодирования сверточных кодов 11
2.2.1 Метод последовательного декодирования 11
2.2.2 Метод порогового декодирования 13
2.2.3 Метод максимального правдоподобия 15
3 Разработка концептуальной модели системы передачи данных 18
4 Проектирование программы, реализующей имитационную модель системы передачи данных на основе сверточного кодирования 20
4.1 Описание блок-схем алгоритмов программы 20
4.2 Разработка диаграммы классов 32
5 Программная реализация имитационной модели 35
5.1 Описание программы 35
5.2 Программная реализация алгоритма сверточного кодирования 35
5.3 Программная реализация алгоритмов декодирования сверточного кода 36
5.4 Программная реализация источника информации и дискретного канала с искажениями 40
5.5 Программная реализация пользовательского интерфейса 41
6 Тестирование имитационной модели 43
6.1 Тестирование имитационных моделей 43
6.2 Подготовка тестовых сценариев 43
6.3 Написание модульных тестов 51
7 Проведение экспериментов с использованием разработанной имитационной модели 53

8 Вредные психофизиологические факторы, влияющие на опорно-двигательный аппарат специалиста по защите информации. Методы и средства физической культуры, снижающие их воздействие. 56
Заключение....58
Список использованных источников....59
Приложение А. Обязательное. Текст заголовочного файла encoder.h....61
Приложение Б. Обязательное. Текст основного файла программного модуля encoder.cpp....63
Приложение В. Обязательное. Текст заголовочного файла decoder.h....66
Приложение Г. Обязательное. Текст основного файла программного модуля decoder.cpp....71
Приложение Д. Обязательное.Руководство разработчика....80
Приложение Е. Обязательное. Текст модуля sequence.h....85
Приложение Ж. Обязательное. Текст главного модуля main.cpp....87
Приложение И. Обязательное. Руководство пользователя....90
Приложение К. Обязательное. Модульные тесты....93

Введение

На сегодняшний день уровень развития телекоммуникационной отрасли позволяет передавать большие объемы данных на высоких скоростях, однако при этом приходится сталкиваться с повышенной зашумленностью каналов связи. Наличие помех вызывает искажение данных при передаче, что может привести к тяжелым последствиям, особенно в современных комплексах автоматизации и управления, где информация используется для управления различными процессами и обеспечение ее целостности, доступности и достоверности является приоритетной задачей. Таким образом, высокие показатели надежности и качества передачи информации не теряют своей значимости.

Для повышения достоверности информации, передаваемой в системах связи, используется помехоустойчивое кодирование. На сегодняшний день создано большое число различных помехоустойчивых кодов, отличающихся по своим характеристикам и областям применения. Один из наиболее популярных – сверточный код, который получил широкое распространение за счет относительной простоты реализации кодера и декодера, и возможности достижения высокой степени кодирования.

Сверточные коды используются во многих системах передачи информации. Они применяются в системах спутниковой и сотовой связи, системах цифрового телевидения и радиовещания, а также в системах радиосвязи с подвижными объектами. Можно отметить, что сверточные коды, несмотря на свою популярность, исследованы в меньшей мере, в отличие от тех же блоковых кодов, алгебраическая теория которых хорошо разработана. Однако проблема декодирования сверточных кодов является одним из немаловажных направлений современных научных исследований. На практике же используются предопределенные структуры сверточных кодов, полученные в ходе исследований, т.к. при произвольном задании характеристик кода существует риск выбора катастрофических сверточных кодов.

Таким образом, изучение сверточных кодов необходимо для понимания работы многих современных систем связи. С этой целью, а также для проведения исследования возможностей сверточных кодов по обеспечению защиты информации от искажений в канале связи необходима имитационная модель системы передачи данных, моделирующая работу кодирующего и декодирующего устройств и позволяющая на основе достоверности передаваемых последовательностей принимать решение об эффективности сверточного кодирования.

Цель дипломной работы – разработка имитационной модели системы передачи данных для изучения помехоустойчивости при использовании сверточных кодов в интересах обеспечения целостности информации в каналах связи. Данная имитационная модель предназначена для проведения статистических исследований помехоустойчивости сверточных кодов при естественных и преднамеретных помехах в канале связи.

Дипломная работа состоит из восьми глав. Первая глава посвящена анализу угроз информации при ее передаче по каналам связи. Во второй главе рассмотрены алгоритмы сверточного кодирования и декодирования. Третья глава содержит описание разработанной концептуальной модели системы передачи данных. В четвертой и пятой главах приведены результаты проектирования и разработки программы, реализующей имитационную модель системы передачи данных на основе сверточного кодирования. В шестой главе описан ход проведения тестирования ее работоспособности. Далее представлены результаты проведения экспериментов с использованием разработанной имитационной модели. В заключительной главе рассмотрены вредные психофизиологические факторы, влияющие на опорно-двигательный аппарат специалиста по защите информации.

1 Анализ угроз безопасности информации при ее передаче по каналам связи

Канал связи – это система технических средств и среда распространения сигналов для односторонней передачи данных от отправителя к получателю. Выделяют несколько видов каналов связи, при использовании каждой из которых необходимо учитывать большое количество факторов помех и других искажений [1]. Кроме того, информационная система, в которой организована передача данных по каналам связи, является объектом нападения со стороны нарушителя, располагающего тем или иным потенциалом. В связи с этим необходимо обеспечение безопасности передаваемых данных от вредоносных воздействий естественного и искусственного характера. При этом можно выделить следующие наиболее характерные угрозы безопасности информации при ее передаче по каналам связи:

- перехват передаваемых данных, то есть их неправомерное получение с помощью технических средств, которые принимают и обрабатывают сигналы, по параметрам которых можно восстановить защищаемую информацию;
- анализ трафика, с помощью которого нарушитель отслеживает передачу данных по каналу связи с целью определения шаблонов связи;
- изменение потока сообщений, при котором нарушитель вносит искажения в передаваемые по каналу данные или нарушает общий порядок следования сообщений или пакетов;
- нарушение связи – угроза блокирования, которая может привести к потере доступности.

Кроме того, можно отметить угрозу имитации, при которой нарушитель «вставляет» в канал связи сфабрикованное им сообщение. При этом нарушитель рассчитывает на то, что получатель воспримет его как подлинное.

Таким образом, в результате реализации вышеописанных угроз безопасности информации при ее передаче по каналам связи могут быть нарушены ее следующие свойства [2]:

– целостность данных (в результате воздействия помех в канале связи, вредоносных атак злоумышленников, направленных на передаваемые данные);

– конфиденциальность данных (в результате перехвата и раскрытия передаваемых данных);

– доступность данных (в результате нарушения структуры системы или канала).

Можно сделать вывод, что в любом канале передачи данных в целях защиты информации необходимо решать задачи двух направлений:

– защита от несанкционированного доступа;

– защита от помех.

2 Анализ алгоритма сверточного кодирования и алгоритмов сверточного декодирования

2.1 Принципы работы и параметры сверточного кодера

Сверточные коды относятся к классу непрерывных кодов с возможностью исправления ошибок. В основе их построения лежит рекуррентный принцип кодирования и декодирования, когда на вход кодера поступает непрерывная последовательность информационных символов источника, а с выхода кодера снимается также непрерывная последовательность символов, являющихся функцией входных символов и структуры кодера.

На рисунке 1 представлена упрощенная структурная схема системы с применением сверточного кодирования [3].

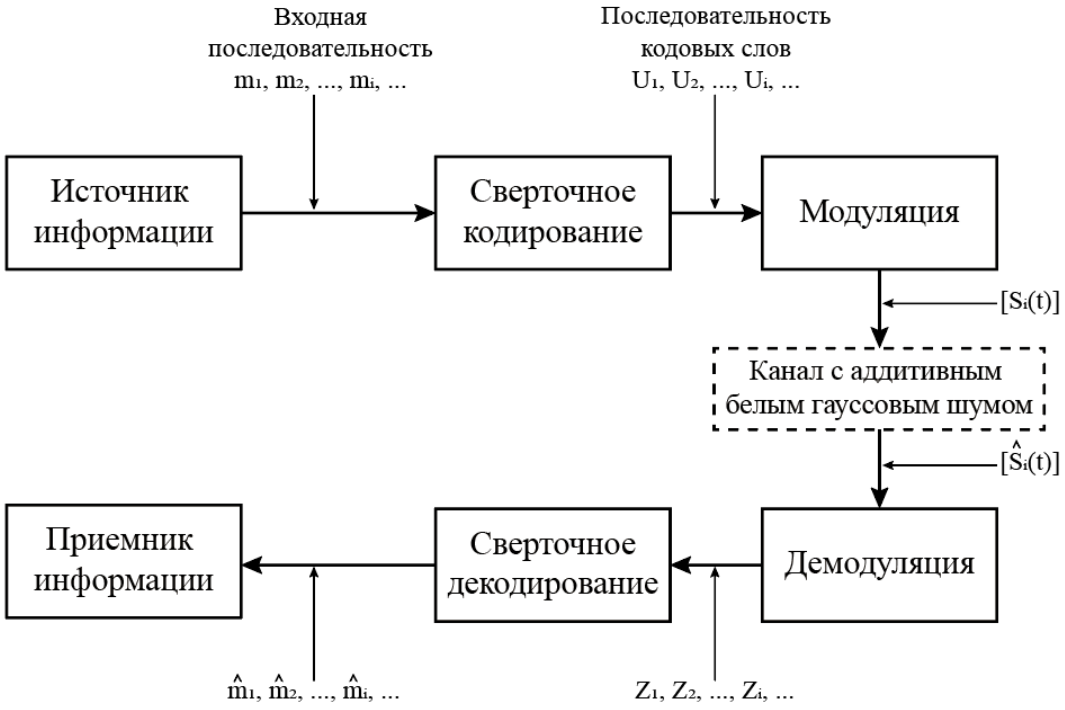

Рисунок 1 – Упрощенная схема системы связи с применением сверточного кодирования

Источником информации является функциональный модуль, генерирующий входную последовательность (1):

$$m = m_1, m_2, \dots, m_i, \dots, \quad (1)$$

где m_i – двоичный бит. Задачей кодера является преобразование последовательности m в последовательность кодовых слов (2).

$$U = G(m) = U_1, U_2, \dots, U_i, \dots \quad (2)$$

Последовательность кодовых слов модулируется сигналом S(t), во время прохождения по каналу связи он поражается шумом, в результате чего с выхода канала снимается сигнал $\hat{S}(t)$. После демодуляции на входе декодера появляется последовательность Z. Задачей декодера в этом случае является определение исходной последовательности, исходя из априорных знаний о структуре кодера и ошибочной последовательности Z.

В отечественной и зарубежной литературе сверточное кодирование описывается с привязкой к работе кодирующего устройства, поэтому будет целесообразно рассмотреть принцип построения сверточного кодера. Общая структура кодера показана на рисунке 2.

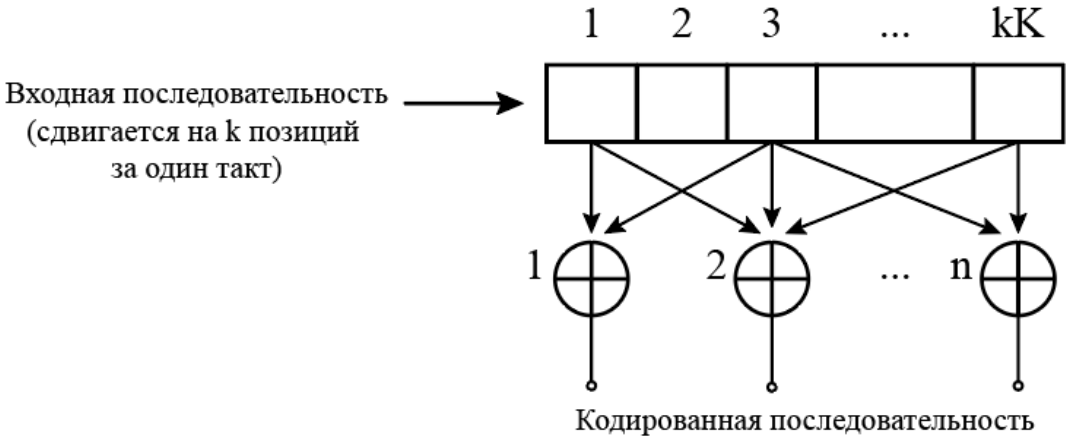

Рисунок 2 – Сверточный кодер с длиной кодового ограничения K и степенью кодирования k/n

Сверточный кодер – устройство, воспринимающее за каждый такт работы в общем случае k входных информационных символов, и выдающее на выход за тот же такт n выходных символов, подлежащих передаче по каналу связи [4]. Отношение (3) называют относительной скоростью кода.

$$R = k/n \quad (3)$$

Ряд исследований подтверждает, что сверточные коды дают значительный выигрыш от кодирования при скорости кода $R \geq 1/3$ [5].

Выходные символы, создаваемые кодером на одном такте, зависят от m информационных символов, поступивших на этом и предыдущем тактах. Таким образом, выходные символы сверточного кодера однозначно определяются его входным сигналом и состоянием, зависящим от m–k предыдущих информационных символов. Значение m определяет память кода, а параметр m + 1 или K называется длиной кодового ограничения, которая показывает число информационных входных символов, влияющих на выходные символы кодера в каждый момент времени [6].

На рисунке 3 в качестве модели кодера представлен сдвиговый регистр, подключенный к двум сумматорам по модулю 2. Выбор связи между сумматорами и разрядами регистра влияет на характеристики кода. Задача выбора связей, дающая оптимальные характеристики, сложна и в общем случае не решается.

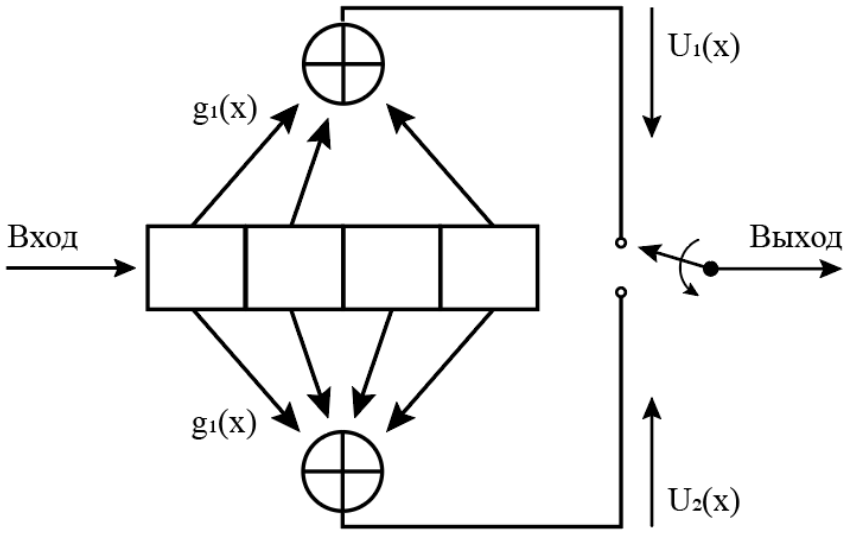

Рисунок 3 – Сверточный кодер (15, 17) с R = ½, K = 4

Выводы, подключаемые к одному сумматору, определяются порождающим полиномом g(x). Обычно такие многочлены указываются в виде восьмеричного числа, единицы двоичной формы которого соответствуют выводам. Если информационную последовательность представить в виде многочлена (4):

$$m(x) = m_0 + m_1 x^1 + m_2 x^2 + \ldots, \quad (4)$$

где m_i – информационные символы (для двоичных кодов 0 и 1), выходные последовательности будут получены путем умножения m(x) на порождающие полиномы g1(x) и g2(x), как в выражении (5):

$$U_1(x) = m(x)g_1(x);\ U_2(x) = m(x)g_2(x) \quad (5)$$

При каждом поступлении бит помещается в крайний левый разряд, а биты регистра смещаются на одну позицию вправо. Затем коммутатор на выходе дискретизирует выходы всех сумматоров по модулю 2, в результате чего формируются кадры кодовых символов, связанные с битами, находящимися в регистре кодера [7]. Эта операция выполняется для каждого входного бита.

Представленный на рисунке 3 сверточный код является несистематическим, так как в выходной последовательности в явном виде не присутствуют информационные символы последовательности m(x). Для систематического сверточного кода один из образующих полиномов должен быть равен x^i, где i = 0, 1 или 2.

Основным недостатком сверточных кодов является сложность декодирования, т.к. не существует четкой методики. Как правило, процесс декодирования является «угадыванием».

Развитие теории сверточных кодов происходило в трех направлениях, в соответствии с тремя важнейшими методами декодирования:

- метод последовательного декодирования;
- метод порогового декодирования;
- декодирование по максимуму правдоподобия.

2.2 Обзор методов декодирования сверточных кодов

2.2.1 Метод последовательного декодирования

В ходе работы последовательного декодера генерируется гипотеза о переданной последовательности кодовых слов и рассчитывается метрика между этой гипотезой и принятым сигналом. Если метрика показывает, что выбор гипотезы не правдоподобен, гипотеза последовательно заменяется, пока не будет найдена наиболее правдоподобная.

Число операций при последовательном декодировании является функцией скорости передачи и шумов в канале. Последовательный декодер строится по логической схеме, позволяющей проводить вычисления со средней скоростью, в несколько раз большей скорости передачи символов, и включает в свой состав буферное запоминающее устройство, предназначенное для хранения поступающих данных при повышении уровня шумов в канале. В случае если число возникших в канале ошибок превысит корректирующую способность кода, могут возникать ошибки. Ошибки так же могут возникать из-за переполнения запоминающего устройства.

При рассмотрении алгоритмов последовательного декодирования удобно представлять сверточный код в виде кодового дерева, как показано на рисунке 4.

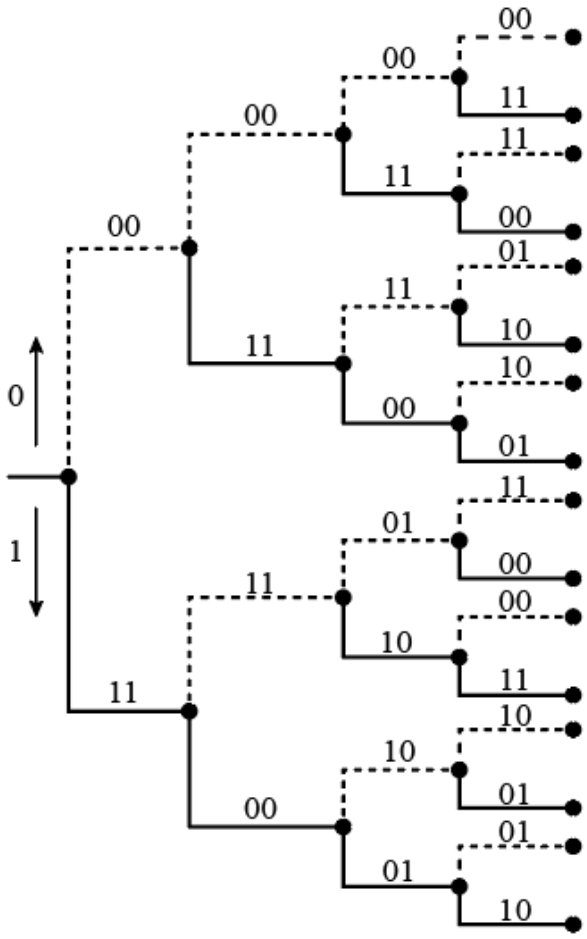

Рисунок 4 – Кодовое дерево кодера с R = ½, K = 4

Исходному нулевому состоянию регистра кодера соответствует начальный узел дерева. Если входной бит, поступающий в регистр, равен 1, то ему приписывается ребро дерева, идущее вниз, а если 0 – вверх. Для двух

полученных в результате узлов дерево строится аналогичным образом и т.д. Над каждым ребром дерева записываются n кодовых символов, получаемых при этом на выходе кодера. Для двоичных кодов в общем случае из каждого узла дерева будет исходить 2^k ребер, каждому из которых сопоставлено n двоичных символов.

Допустим, что принятая последовательность Z является искаженной последовательностью U. Декодер начинает с узла дерева в момент t_1 и генерирует оба пути, выходящие из него. Если принятые n кодовых символов совпадают с одним из сгенерированных путей, декодер следует этому пути. Если согласования нет, то декодер следует наиболее вероятному пути, но при этом ведет подсчет несовпадений между принятыми символами и ответвляющимися словами на пути следования. Если две ветви оказываются равновероятными, то приемник делает произвольный выбор. На каждом уровне дерева декодер генерирует новые ветви и сравнивает их со следующим набором n принятых кодовых символов. Поиск продолжается до тех пор, пока все дерево не будет пройдено по наиболее вероятному пути.

Если счет несовпадений превышает некоторое число, декодер решает, что он находится на неправильном пути, отказывается от него и повторяет все снова.

2.2.2 Метод порогового декодирования

При пороговом декодировании сверточных кодов вычисляются синдромы (признаки места ошибочных символов), затем они подаются на входы порогового элемента, где выносится решение о значении декодируемого символа. При пороговом декодировании используют, в основном, систематические коды. Общая схема декодера для сверточного кода ($R = ½$) представлена на рисунке 5 [8].

Моделью двоичного канала являются сумматоры по модулю 2, на входы которых кроме кодовых последовательностей U_1 и U_2 поступают ошибки e_1 и e_2. Декодер содержит аналог кодера, в котором по принимаемой последовательности Z_1 формируется копия проверочной. В формирователе

синдрома образуется последовательность синдромов S, которая поступает на вход синдромного регистра.

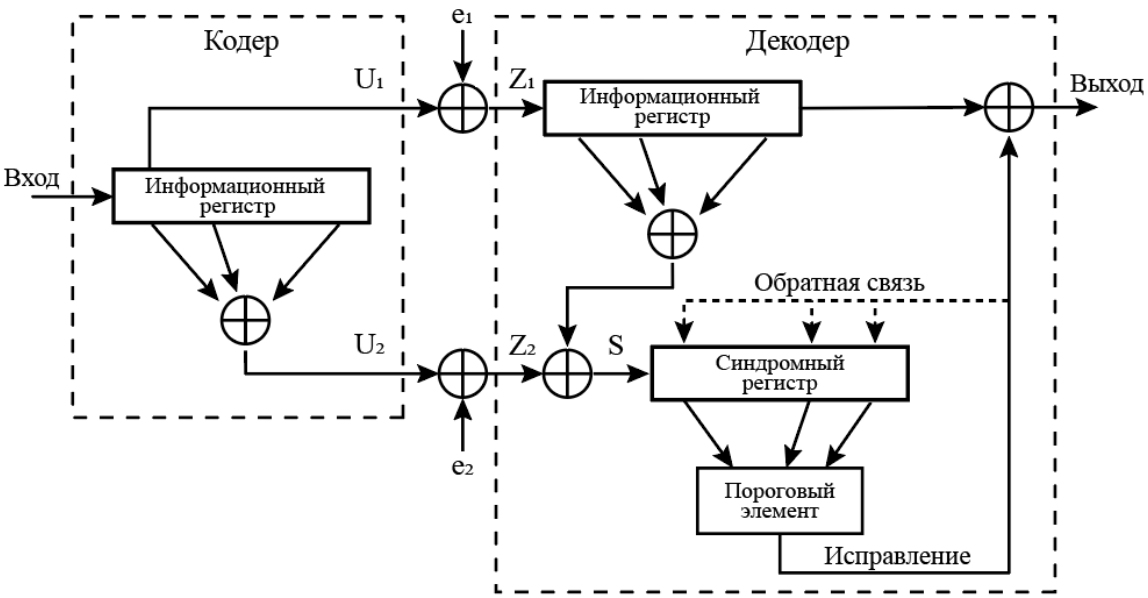

Рисунок 5 – Общая схема порогового декодирования

При отсутствии в канале ошибок последовательности на входах формирователя синдрома всегда совпадают, и S состоит из одних нулей. В противном случае в синдромных последовательностях на определенных позициях появляются единичные символы в соответствии с набором ошибок. По структуре синдромной последовательности должно быть возможно определить искаженные символы. Если имеется комбинация ошибок, которая может быть исправлена, то на выходной сумматор подаются единичные символы в моменты выхода из информационного регистра искаженных символов.

Ошибки, исправляемые в очередном информационном блоке, могут влиять на символы синдромов, соответствующих последующим блокам. Исключение их влияния достигается за счет обратной связи, которая на рисунке 5 представлена пунктирной линией.

Следует учитывать, что обратная связь может привести к явлению «размножения ошибок».

2.2.3 Метод максимального правдоподобия

Декодирование по методу максимального правдоподобия, по существу, представляет собой алгоритм поиска максимально правдоподобного пути на решеточной диаграмме кода (решетке).

Решетчатая диаграмма показывает все возможные переходы кодера из предыдущего состояния в последующее. Каждый узел характеризует состояние кодера, то есть состояние регистра сдвига. При этом одна ветвь соответствует входному нулевому биту, а другая – входной единице. Свойства решетчатого представления сверточного кода:

- решетка сверточного кода имеет регулярную структуру;
- два символа, стоящие у каждого ребра (при R = ½), обозначают переход между состояниями регистра и представляют собой фрагменты сверточного кода, порожденные на каждом такте символом кода источника;
- при решетчатом представлении сверточного кода с ростом числа входных символов количество вершин в решетке не растет экспоненциально, как в дереве, а остается постоянным, равным 2^{K-1}.

Пример решетчатой структуры сверточного кода показан на рисунке 6.

С ростом длины последовательностей количество путей растет экспоненциально, поэтому декодирование по максимуму правдоподобия при большой длине информационных последовательностей слишком сложно и непрактично для реализации. Однако эта задача решается довольно просто с применением метрики Хемминга, вычисляемой для каждого пути на решетке, что и реализует алгоритм Витерби.

Алгоритм Витерби состоит в том, что на каждом шаге строятся все возможные пути для этого шага, вычисляются метрики ветвей и метрики путей с учётом принятых на шаге кодированных бит и далее отсекаются пути с худшими значениями метрик. На каждом шаге имеется ровно 2^k ветвей. Метрики оставшихся путей сохраняются для вычислений на следующем шаге.

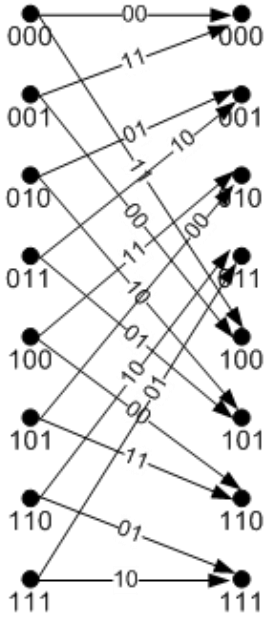

Рисунок 6 – Решетчатая структура для сверточного кода с R = ½, K = 4

Под метрикой ветви понимается хэмингово расстояние между принятой на шаге кодовой комбинацией, и комбинацией, которую должен был бы сгенерировать кодер при переходе из одного узла в другой. Хэмингово расстояние рассчитывается следующим образом: принятая комбинация суммируется по модулю два со сгенерированной комбинацией и далее рассчитывается количество единиц в получившемся значении. Сумма метрик ветвей, входящих в путь, называется метрикой пути.

Несмотря на то, что изначально путей на решетке может быть очень много, в дальнейшем удается исключить некоторые из них, оставив сравнительно небольшой список, содержащий наиболее правдоподобный путь.

Важным достоинством декодера Витерби является то, что если при декодировании сделана ошибка в выборе пути на решетчатой диаграмме кода, то за несколько тактов, декодер вновь выходит на правильный путь, как показано на рисунке 7.

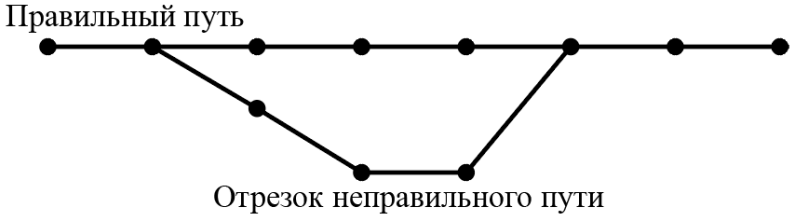

Рисунок 7 – Случай выбора декодером Витерби неправильного пути

Это позволяет начать процесс декодирования с любого момента времени, не заботясь о взаимной синхронизации кодера и декодера по началу работы. Декодер, начав декодирование в произвольный момент, через несколько тактов сам выйдет на правильный путь, автоматически установив указанную синхронизацию.

3 Разработка концептуальной модели системы передачи данных

В общем случае концептуальная модель предназначена для определения структуры моделируемой системы, свойств ее элементов и причинно-следственных связей, присущих системе и существенных для достижения цели моделирования.

Разработка концептуальной модели системы передачи данных обусловлена необходимостью описания основных составляющих моделируемой системы и их взаимосвязей перед началом проектирования разрабатываемой имитационной модели.

Концептуальная модель системы передачи данных на основе сверточного кодирования представлена на рисунке 8.

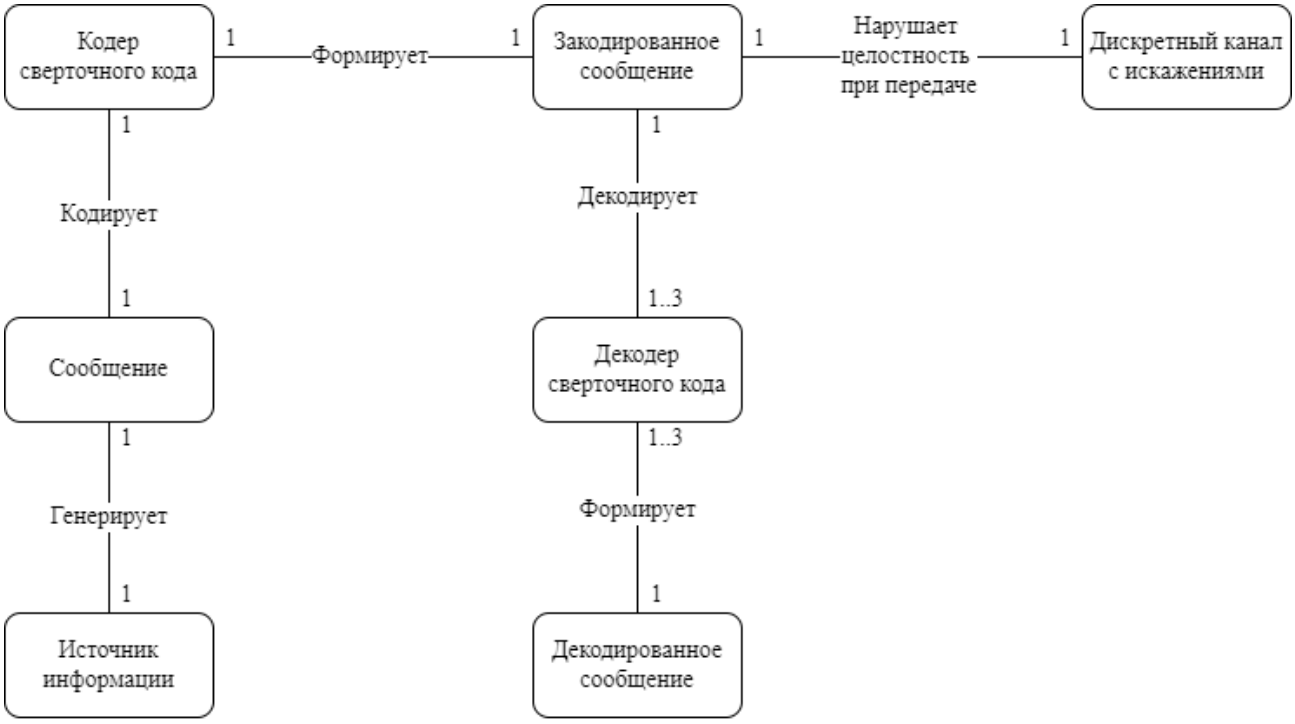

Рисунок 8 – Концептуальная модель системы передачи данных

Данная концептуальная модель включает источник сообщений, кодер сверточного кода, дискретный канал с искажениями, декодер и устанавливает связи между ними через преобразование исходного сообщения в декодированное. Так, источник информации генерирует сообщение, под которым понимается двоичная последовательность бит фиксированной длины. Кодер сверточного кода кодирует его и тем самым формирует закодированное

сообщение, целостность которого нарушается при его передаче по дискретному каналу связи с искажениями. При этом биты закодированного сообщения искажаются независимо друг от друга в соответствии с некоторой вероятностью. Декодер сверточного кода может декодировать сообщение по трем алгоритмам, формируя таким образом соответствующее количество декодированных сообщений.

4 Проектирование программы, реализующей имитационную модель системы передачи данных на основе сверточного кодирования

4.1 Описание блок-схем алгоритмов программы

На рисунке 9 представлен алгоритм работы программы, реализующей имитационную модель системы передачи данных на основе сверточного кодирования.

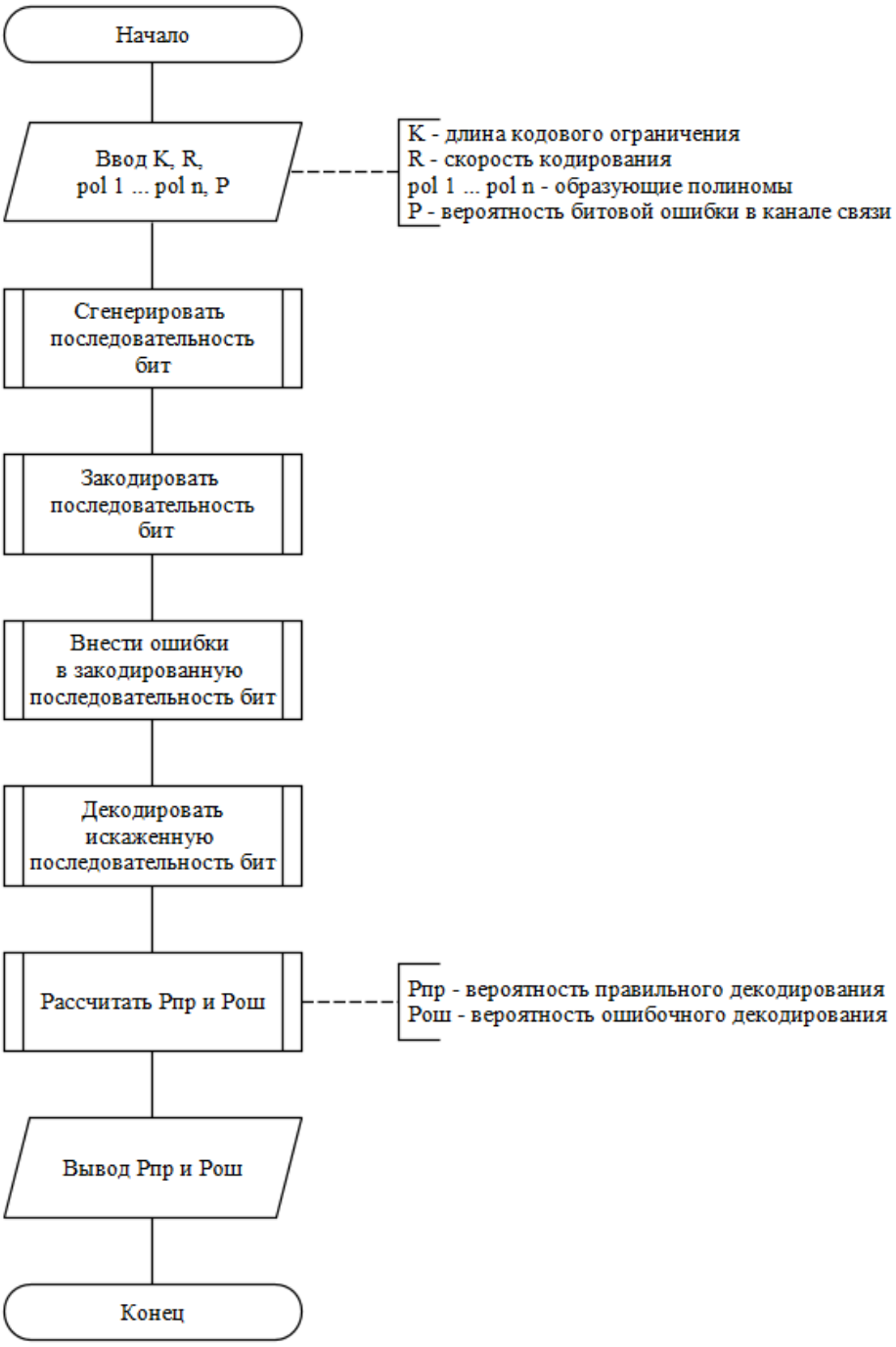

Рисунок 9 – Общий алгоритм работы разрабатываемой программы

Из рисунка 9 видно, что программа ожидает на вход параметры сверточного кода и вероятность битовой ошибки в дискретном канале связи. Далее должна быть сгенерирована двоичная последовательность бит, которую необходимо передать на вход программной реализации кодера сверточного кода. В закодированную последовательность бит требуется внести битовые ошибки в соответствии с заданной вероятностью, тем самым смоделировав передачу последовательности по дискретному каналу с искажениями. Полученную последовательность необходимо декодировать, и на основе сравнения результата декодирования с исходной последовательностью следует рассчитать вероятность правильного и ошибочного декодирования сверточного кода, заданного поступившими на вход программы параметрами.

Алгоритм генерации двоичной последовательности бит представлен на рисунке 10. В данном случае под двоичной последовательностью бит понимается псевдослучайная последовательность нулей и единиц заданной длины. Очередной бит такой последовательности представляет собой результат деления псевдослучайного числа по модулю два.

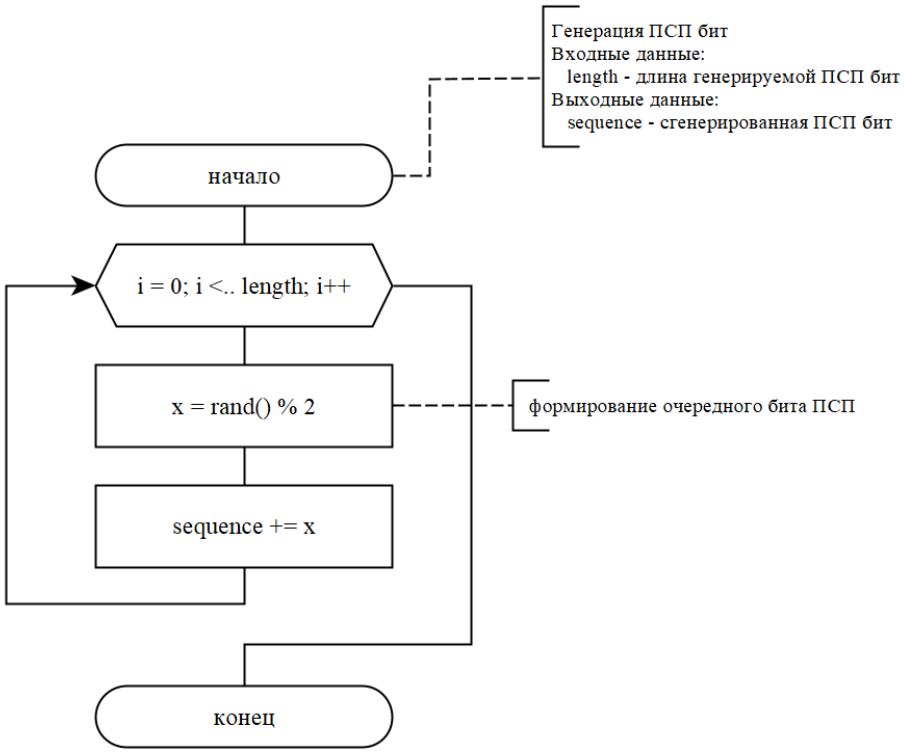

Рисунок 10 – Алгоритм генерации двоичной последовательности бит

Блок-схема алгоритма сверточного кодирования показана на рисунке 11.

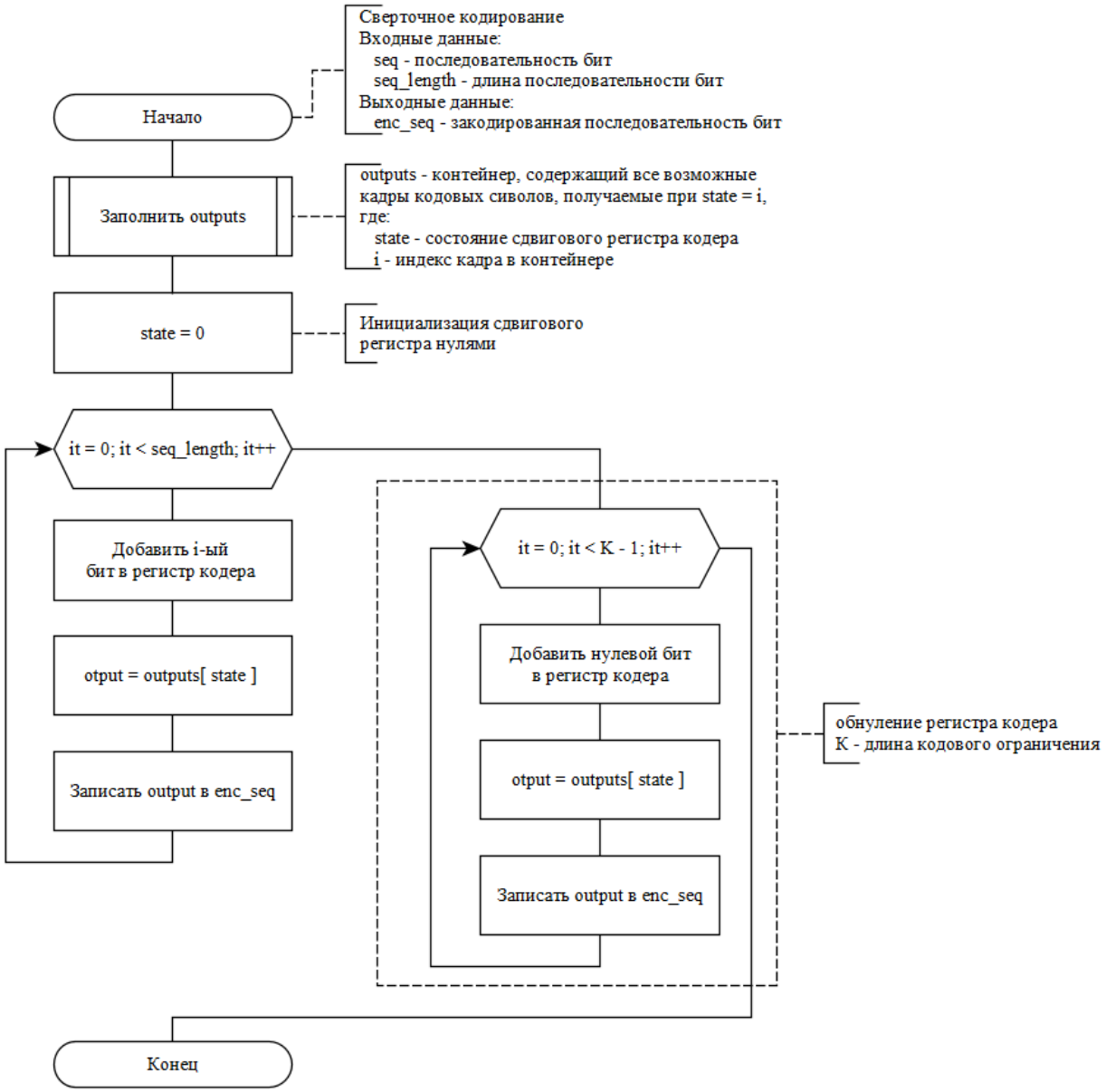

Рисунок 11 – Блок-схема алгоритма сверточного кодирования

Перед началом процедуры сверточного кодирования сгенерированной псевдослучайной последовательности бит составляется таблица, содержащая кадры кодовых символов, получаемые из всех возможных состояний регистра кодера. Такая таблица содержит 2 столбца, первый из которых представляет собой состояния регистра от 00…00 до 11…11 (количество разрядов варьируется в зависимости от длины кодового ограничения), а второй – комбинацию бит, получаемую из соответствующего состояния. При программной реализации удобнее всего хранить такую таблицу в виде контейнера outputs, каждый элемент

которого – кадр кодовых символов, получаемый из состояния, равного индексу элемента. Один символ такого кадра получается при побитовом умножении индекса и одного из образующих полиномов.

Блок-схема процедуры заполнения контейнера outputs показана на рисунке 12.

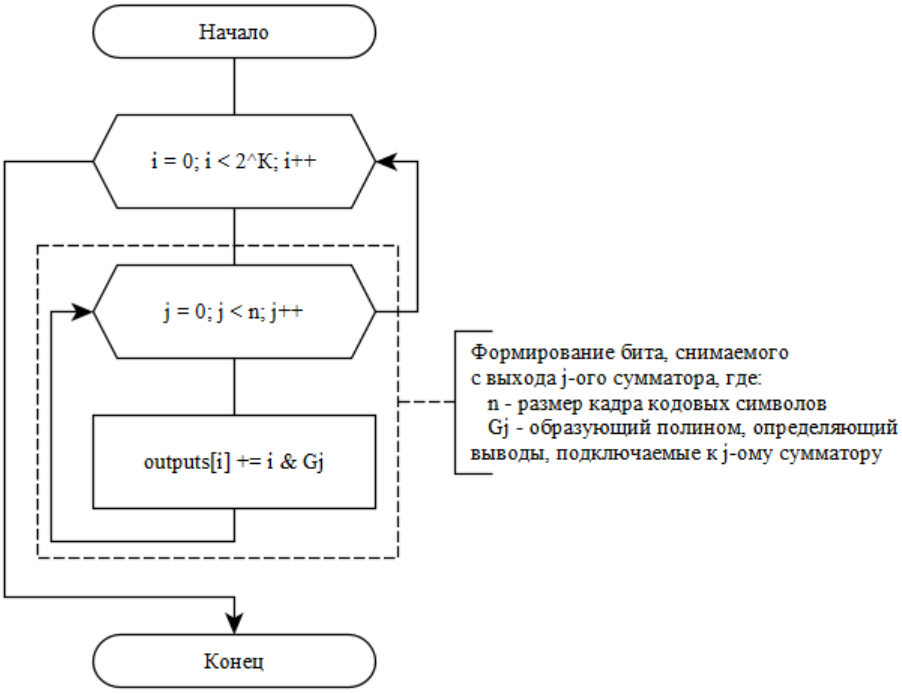

Рисунок 12 – Блок-схема процедуры заполнения контейнера outputs

После заполнения контейнера outputs задается начальное нулевое состояние регистра кодера. С приходом очередного бита кодируемой последовательности состояние регистра должно обновляться, а в конец закодированной последовательности должен быть добавлен элемент контейнера outputs по индексу, равному обновленному состоянию регистра. После того как в кодер поступил последний бит кодируемой последовательности, регистр кодера должен быть очищен. С этой целью на вход кодера K-1 раз подаются нулевые биты. Получаемые при этом кадры кодовых символов так же записываются в конец закодированной последовательности.

Алгоритм внесения ошибок в закодированную последовательность бит представлен блок-схемой на рисунке 13. Для каждого бита принятой последовательности генерируется псевдослучайное число в диапазоне от 0 до

100. Если это число оказывается меньше значения вероятности битовой ошибки, то текущий бит инвертируется, в противном случае – остается без изменений.

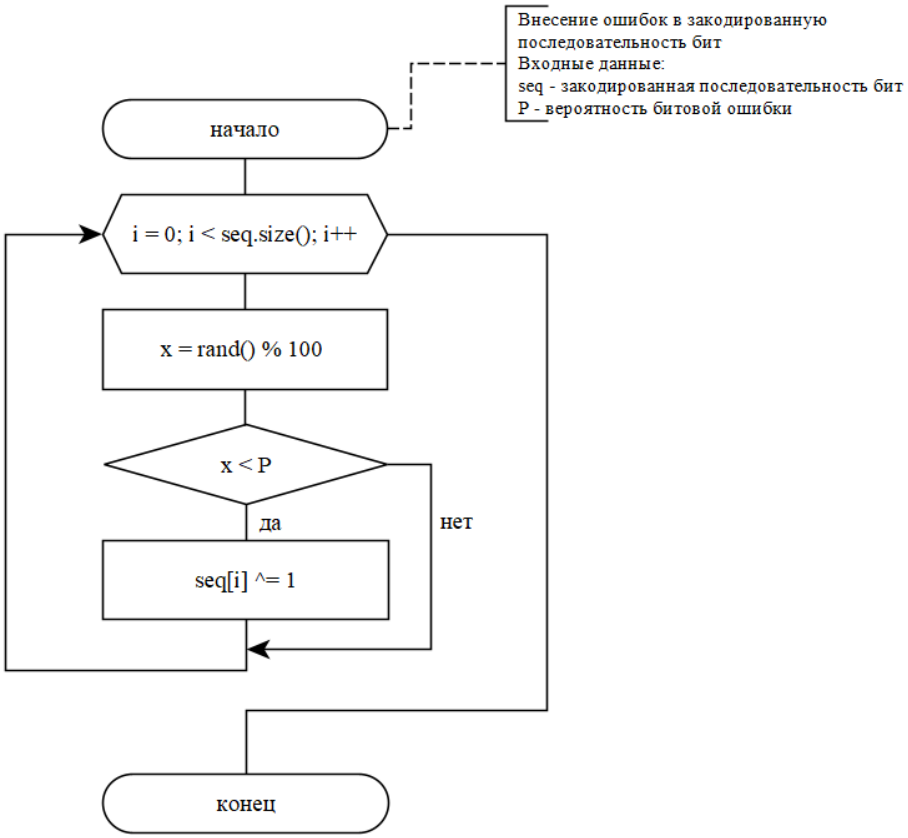

Рисунок 13 – Блок-схема алгоритма внесения ошибок в закодированную последовательность бит

Декодирование сверточного кода должно выполняться по следующим алгоритмам:

– по алгоритму декодирования по максимуму правдоподобия (алгоритм Витерби);

– по алгоритму последовательного декодирования;

– по алгоритму порогового декодирования систематического кода.

В каждом случае перед запуском процедуры декодирования необходимо заполнять контейнер outputs.

Блок-схема алгоритма декодирования Витерби представлена на рисунке 14. В основе алгоритма лежит подсчет метрик всех путей, которые могут быть построены на кодовой решетке. С этой целью для каждого кадра кодовых

символов вызывается процедура обновления метрик путей, блок-схема которой представлена рисунке 15.

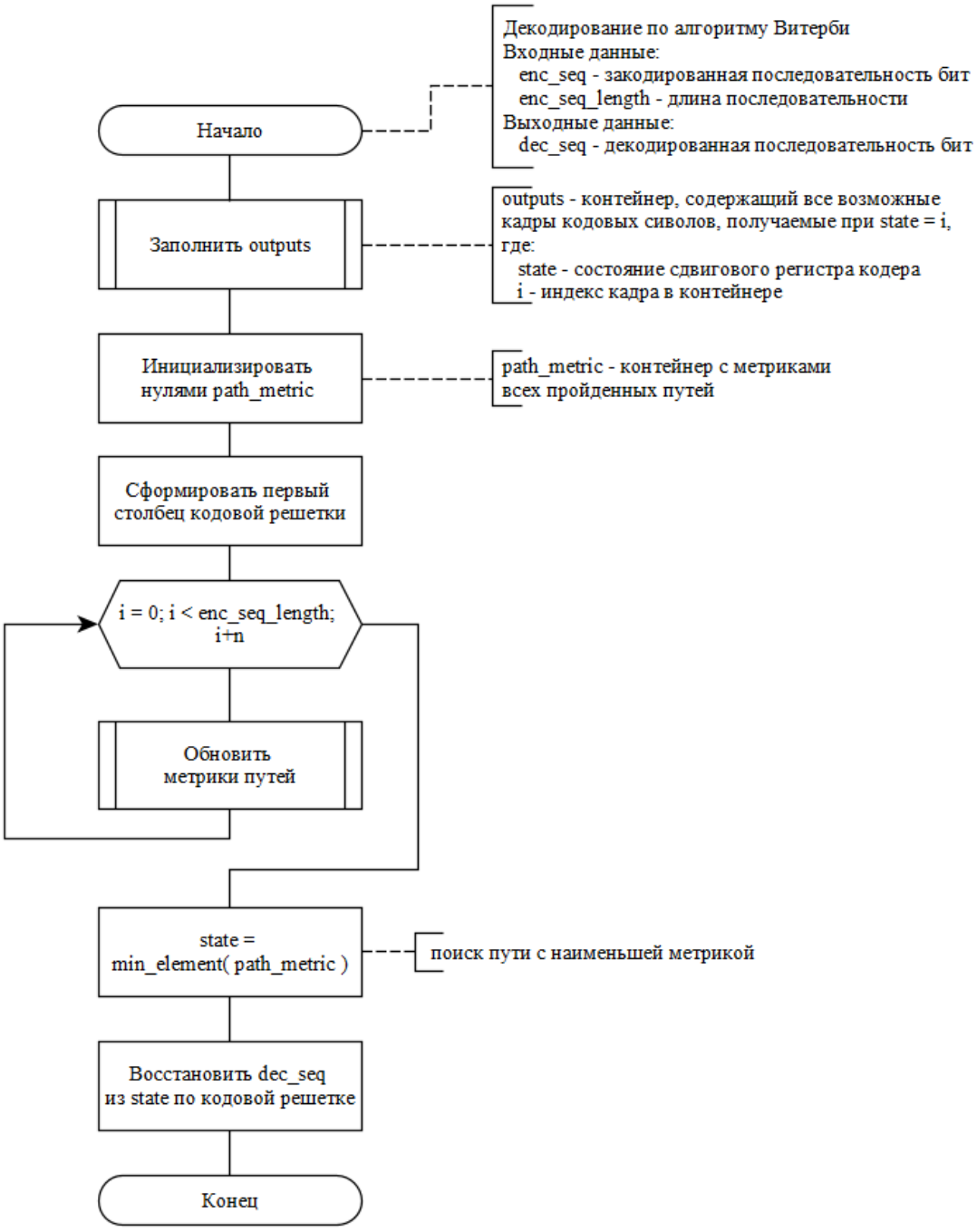

Рисунок 14 – Блок-схема алгоритма декодирования сверточного кода по алгоритму Витерби

По итогам такой обработки закодированной последовательности должны быть сформированы все наиболее вероятные пути на кодовой решетке и рассчитаны метрики для каждого из них. Путь с наименьшей метрикой считается

выжившим, на его основе воспроизводится предполагаемая исходная двоичная последовательность бит.

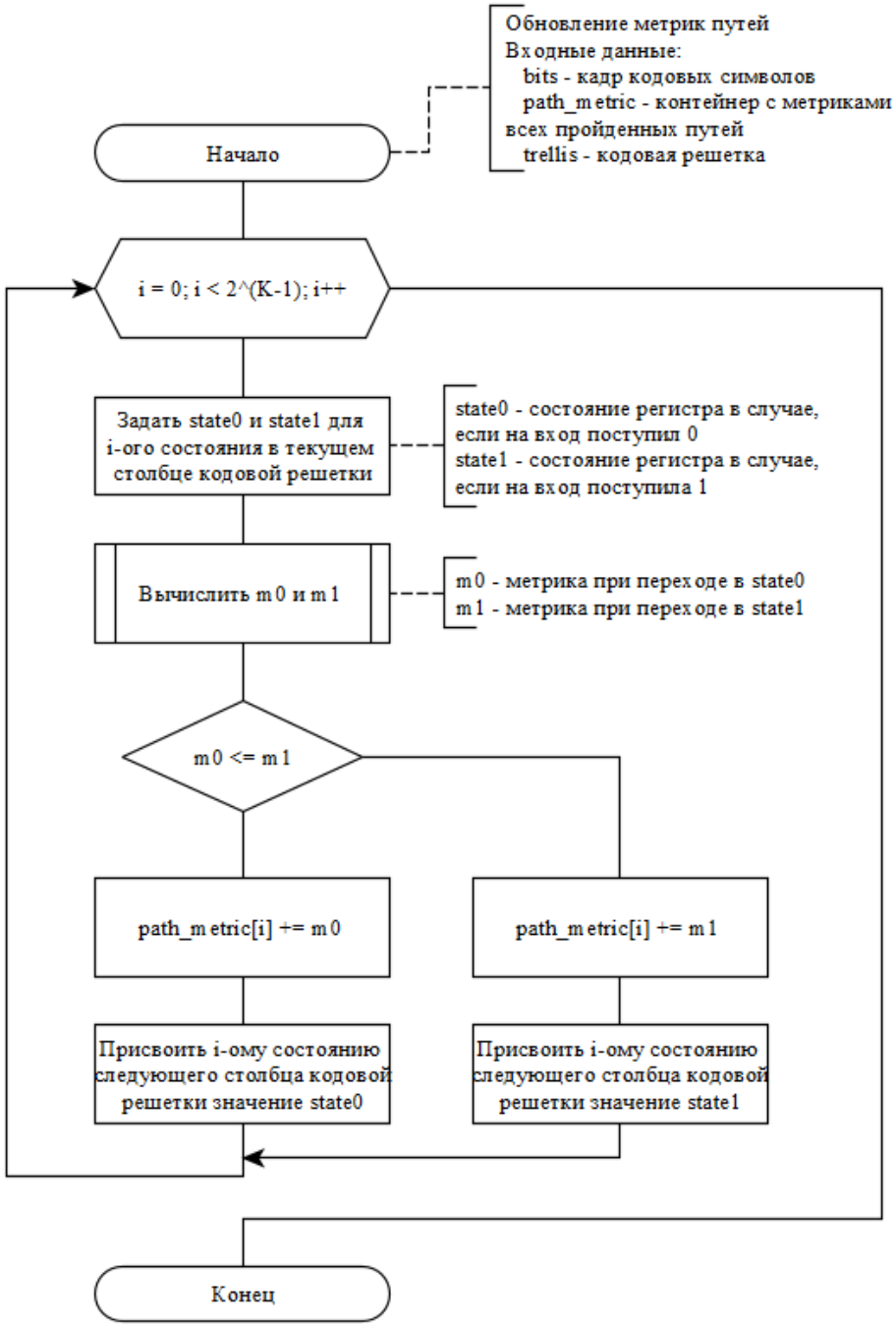

Рисунок 15 – Блок-схема процедуры обновления метрик путей

В ходе выполнения процедуры обновления метрик путей формируется новый столбец кодовой решетки, с учетом которого пересчитываются все метрики. Для продолжения пути из одно узла кодовой решетки должны быть заданы состояния регистра для обоих случаев, когда поступает 0 или 1. Для кадров кодовых символов, получаемых для каждого из таких состояний,

необходимо вычислить метрики (расстояние Хэмминга), так как показано на рисунке 16. Путь в состояние с наибольшей метрикой отбрасывается.

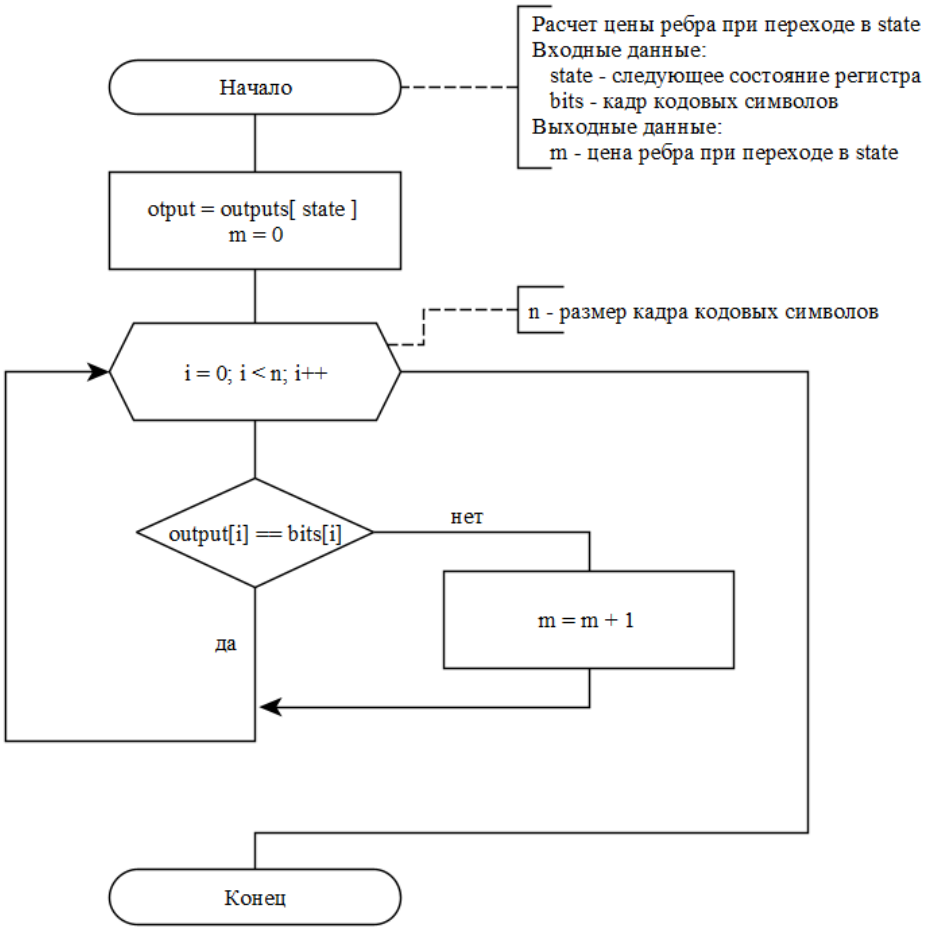

Рисунок 16 – Блок-схема процедуры расчета метрики при переходе в следующий узел на кодовой решетке

В основе алгоритма последовательного декодирования лежит движение декодера по кодовому дереву, которое сводится к тому, что декодер может двигаться по пути только в том случае, если его цена лежит выше текущего порога [9, 10]. Таким образом, перед тем как обработать очередной кадр кодовых символов и построить следующее ребро декодер сравнивает цену пути и порог. Если цена пути превышает порог, то путь считается правильным. Если же цена пути оказывается меньше значения порога, декодер возвращается на один узел назад и рассматривает ребро, которое не было пройдено ранее при текущем значении порога. Если оно не удовлетворяет порогу, декодер возвращается на

еще один узел назад и повторяет поиск лучшего пути. Блок-схема алгоритма последовательного декодирования представлена на рисунке 17.

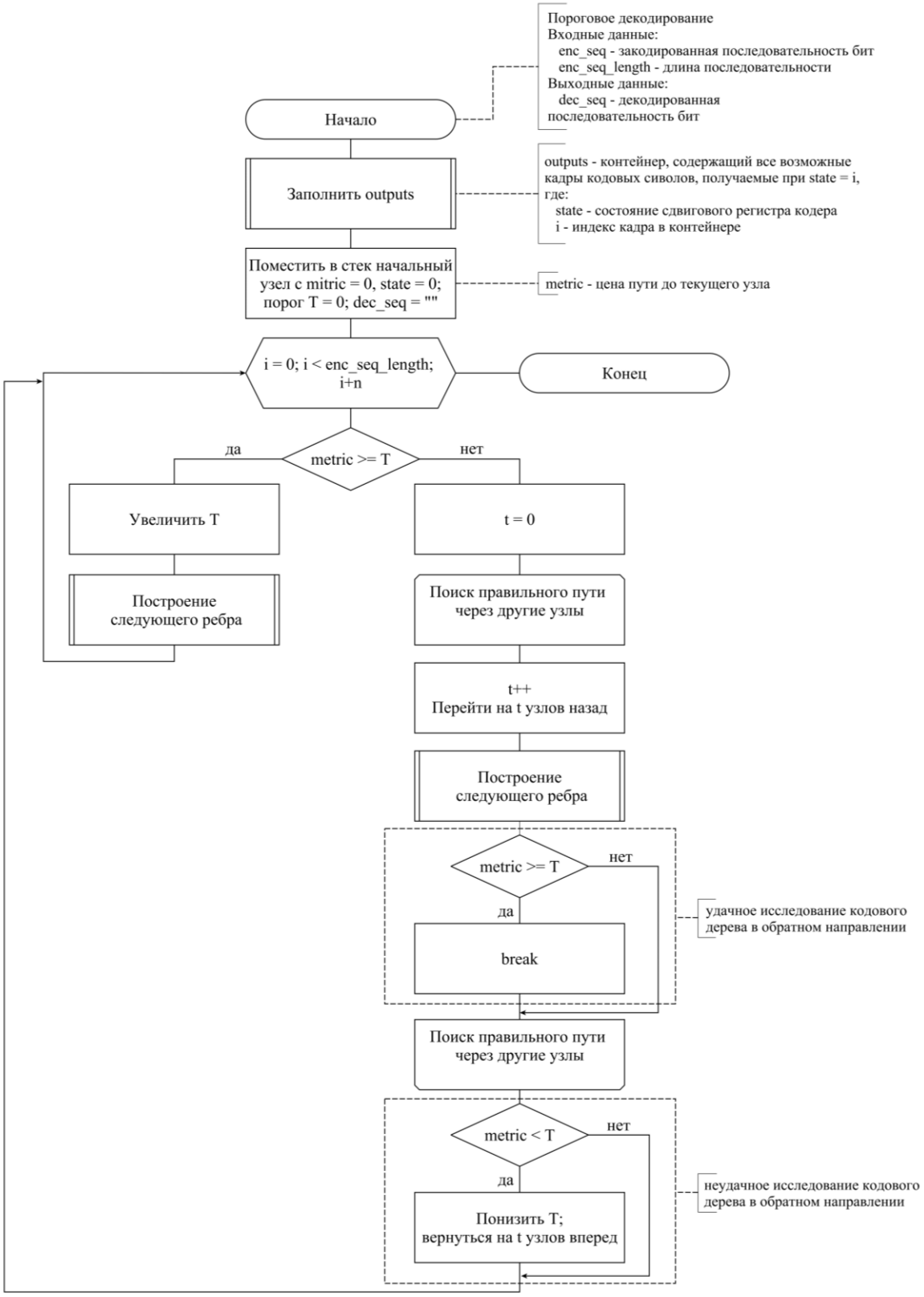

Рисунок 17 – Блок-схема алгоритма последовательного декодирования сверточного кода

При поиске лучшего пути в ходе движения в обратном направлении декодер считает правильным путь до узла, цена пути которого еще лежит выше текущего значения порога. В случае, если исследование в обратном направлении оказалось неудачным, порог уменьшается, и декодер возвращается к узлу, с которого начал движение назад. Так декодер должен найти правильный путь, вдоль которого цена пути возрастает, и на его основе декодировать принятую закодированную последовательность. При этом каждый узел, вошедший в итоговый путь, должен быть занесен в стек для упрощения движения в обратном направлении и обеспечения того, что декодер не пройдет по отброшенному пути второй раз.

Блок-схема процедуры построения нового ребра показана на рисунке 18.

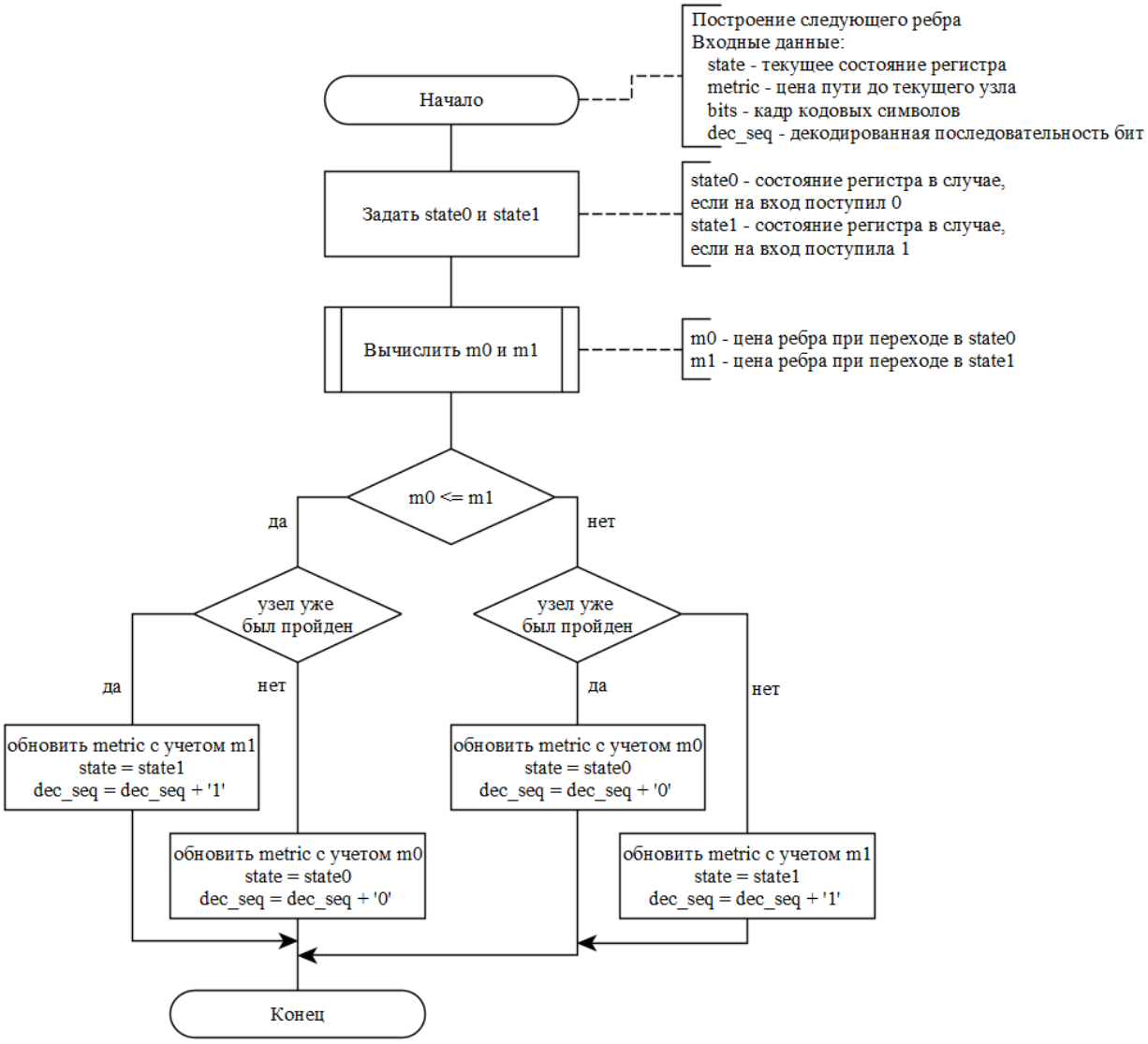

Рисунок 18 – Блок-схема процедуры построения нового ребра

Построение следующего ребра заключается в том, чтобы выбрать ребро, ведущее в узел, при переходе в который цена пути выше. Для этого необходимо рассчитать метрики переходов в состояния, когда на вход кодера могли поступить 0 и 1. Если цена пути и это ребро не рассматривалось раньше, то оно включается путь. Иначе в путь включается другое ребро.

Блок-схема алгоритма порогового декодирования систематического сверточного кода приведена на рисунке 19.

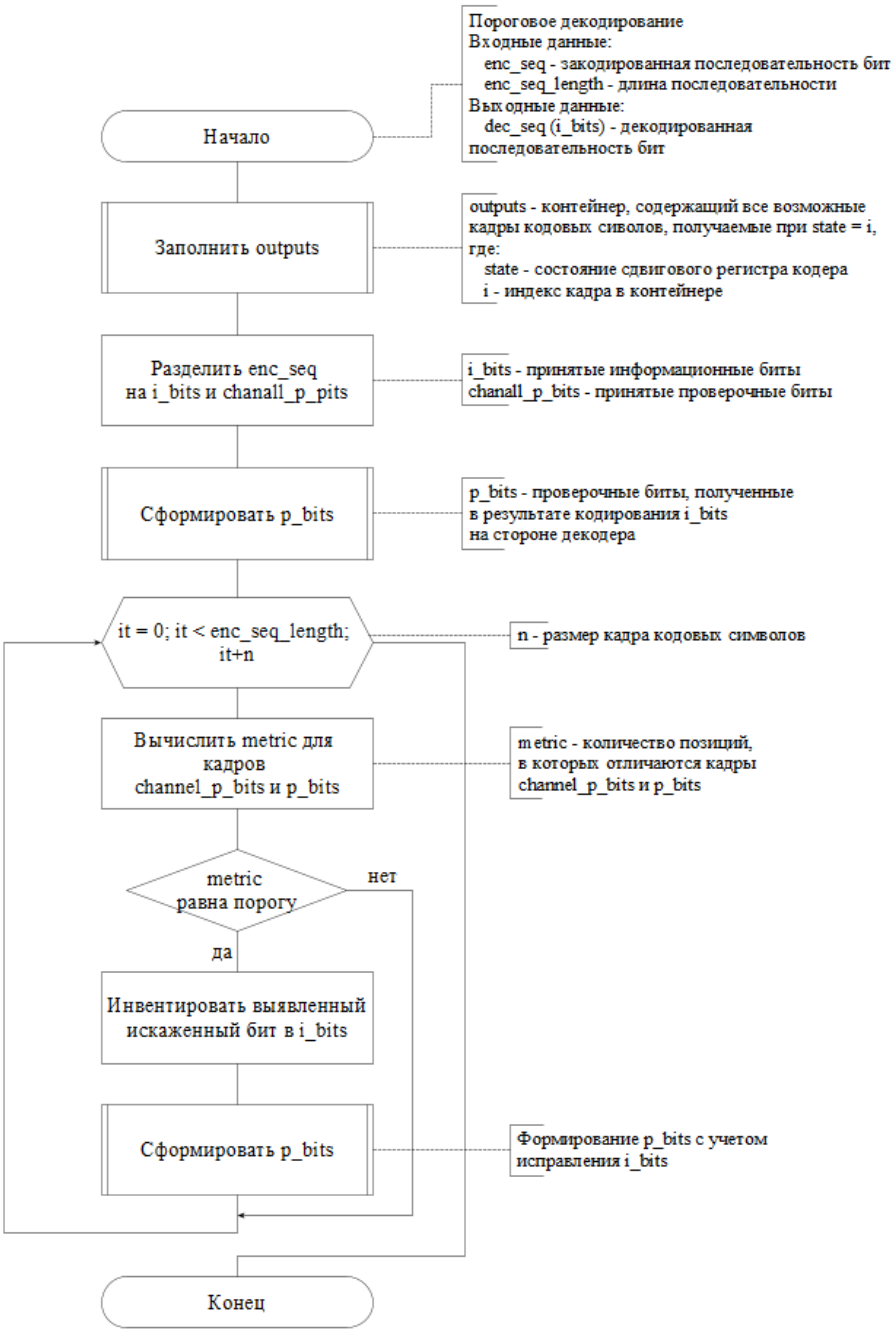

Рисунок 19 – Блок-схема алгоритма порогового декодирования сверточного кода

С учетом того, что программная реализация порогового декодера применяется для систематических сверточных кодов, принятую закодированную последовательность бит можно разделить на информационные и проверочные биты. После этого информационные биты кодируются, и из закодированной на стороне декодера последовательности вновь выделяются проверочные биты, как показано на рисунке 20. Процедура кодирования информационных бит аналогична алгоритму кодирования, который был рассмотрен выше. Принятые и полученные проверочные биты сравниваются между собой в рамках K кадров с учетом только тех бит, на которые мог влиять рассматриваемый информационный бит. Если несовпадение выявлено во всех проверочных битах (при учете меньшего числа несовпадений возможно размножение ошибок в ходе декодирования) одного информационного бита, то этот бит должен быть инвертирован, после чего проверочные биты декодера необходимо пересчитать с учетом изменений в информационной последовательности.

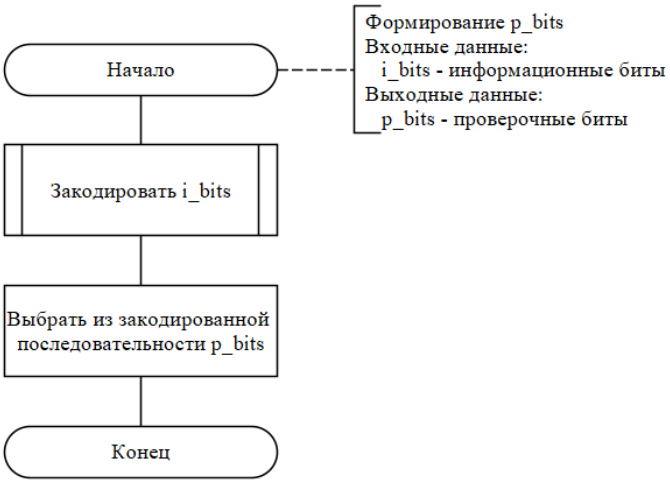

Рисунок 20 – Блок-схема процедуры формирования проверочных бит декодером

После завершения декодирования последовательности бит необходимо сравнить результат декодирования с исходной последовательностью и вычислить вероятности правильного и ошибочного декодирования сверточного кода. С этой целью сравнивается каждый бит указанных последовательностей, а также заводятся два счетчика: один необходим для подсчета совпадающих бит в последовательности, другой – для подсчета несовпадающих. Вероятность

правильного декодирования может быть получена путем вычисления отношения первого счетчика к длине последовательностей. Вероятность ошибочного декодирования может быть получена аналогично, но с использованием второго счетчика. Описанный алгоритм приведен на рисунке 21.

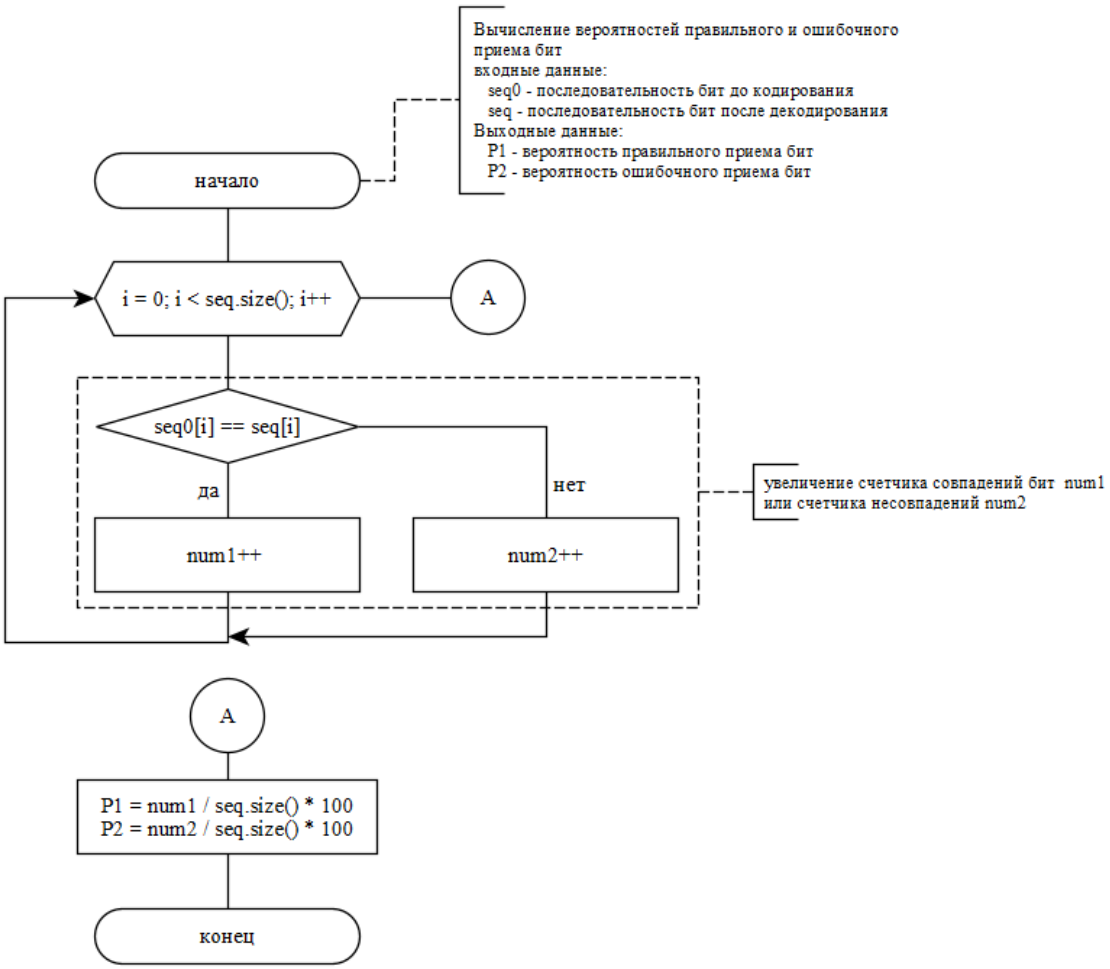

Рисунок 21 – Алгоритм вычисления вероятностей правильного и ошибочного декодирования сверточного кода

4.2 Разработка диаграммы классов

На основе разработанных алгоритмов сверточного кодирования и алгоритмов сверточного декодирования в целях проектирования внутренней структуры программы была разработана диаграмма классов, представленная на рисунке 22.

Для реализации кодера сверточного кода был спроектирован класс Coder. Класс содержит три атрибута, два из которых содержат параметры кодера (длину кодового ограничения и полиномы). В связи с тем, что образующих полиномов

должно быть не меньше двух, их хранение может быть осуществлено с помощью вектора, каждый элемент которого представляет собой запись одного полинома в виде целого числа. Аргумент outputs – вектор, заполняемый с помощью метода класса gen_outputs. Метод encode должен вызываться из интерфейса пользователя для запуска процедуры сверточного кодирования.

Рисунок 22 – Диаграмма классов программной реализации имитационной модели

При анализе алгоритмов декодирования сверточного кода можно выделить у декодеров общее: параметры кода, формирование outputs и вычисление метрик как расстояния Хэмминга. Это общее было перенесено в базовый класс Decoder, из которого наследуются все декодеры.

Класс ViterbiDecoder реализует декодер Витерби. Помимо основного метода, запускающего декодирование из пользовательского интерфейса, класс содержит вспомогательные методы для обновления и вычисления метрик путей, алгоритмы которых были представлены в предыдущем разделе.

Класс SequentialDecoder реализует последовательный декодер. Класс содержит аргумент stack_ для хранения пройденных узлов кодового дерева. Т.к. при переходе в узел декодер должен знать его состояние, цену пути до узла, количество уже построенных из него ребер, было решено использовать для представления каждого узла структуру Node, экземпляр которой создается

методом gen_node. Класс также содержит основной и вспомогательный методы, алгоритмы которых были представлены в предыдущем разделе.

Класс TresholdDecoder содержит аргументы для хранения информационных бит и проверочных бит, полученных на стороне декодера, а также 4 метода, один из которых вызывает пороговое декодирование из пользовательского интерфейса, остальные необходимы для выделения информационных и проверочных бит из принятой последовательности.

5 Программная реализация имитационной модели

5.1 Описание программы

Программная реализация имитационной модели предоставляет возможность расчета вероятности правильного и ошибочного декодирования сверточного кода.

На основе диаграммы классов были разработаны модули:

- encoder.h, реализующий функциональность сверточного кодера;
- decoder.h, реализующий функциональность декодеров сверточного кода.

Текст заголовочного файла модуля encoder.h приведен в приложении А, текст основного файла encoder.cpp – в приложении Б.

Текст заголовочного файла модуля decoder.h приведен в приложении В, текст основного файла decoder.cpp – в приложении Г.

Руководство разработчика к данным модулям приведено в приложении Д.

Также был разработан модуль sequence.h, включающий описание функции, генерирующей последовательности бит, функции, вносящей ошибки в битовую последовательность, и функции расчета правильного и ошибочного декодирования сверточного кода. Текст модуля приведен в приложении Е.

Программа была написана на высокоуровневом языке C++ стандарта C++11 с использованием компилятора g++.

5.2 Программная реализация алгоритма сверточного кодирования

Для представления кодера был реализован класс Coder. В закрытой части класса объявлено три атрибута: длина кодового ограничения K_, контейнеры для хранения образующих полиномов pol_ и всех возможных комбинаций на выходе кодера в зависимости от состояния сдвигового регистра outputs_. Закрытая часть класса также включает метод gen_outputs, задача которого – заполнение вектора outputs_. В открытой части класса определены конструктор класса и метод encode, при вызове которого из интерфейса пользователя запускается процедура сверточного кодирования.

В конструкторе класса Coder реализовано задание длины кодового ограничения и образующих полиномов с помощью списка инициализации. Внутри конструктора выполнен вызов метода gen_outputs.

В методе gen_outputs осуществляется выделение памяти под outputs_, размер вектора определен как 1 << K_, т.е. 2^K. Далее реализовано вычисление кодовых комбинаций при переборе элементов outputs_, за состояние регистра сдвига на каждом шаге принимается индекс соответствующего элемента.

Метод encode принимает по ссылке строку бит bits, которую необходимо закодировать, и возвращает строку, содержащую закодированную последовательность. Для хранения текущего состояния регистра сдвига в методе объявлена переменная state, инициализируемая 0. Перебор символов (бит) принимаемой на вход последовательности реализован в цикле, на каждом шаге которого происходит обновление state и добавление к закодированной последовательности комбинации, получаемой при обращении к outputs_ по индексу state. Далее в цикле на основе диапазона от 0 до K_ - 1 выполняется добавление бит, формируемых за счет памяти кодера для его обнуления.

5.3 Программная реализация алгоритмов декодирования сверточного кода

Программная реализация любого декодера является более сложной, чем кодера, в связи с тем, что процесс декодирования предполагает не только обратное преобразование принятой последовательности бит, но и нахождение и исправление искаженных символов.

Для представления рассматриваемых декодеров были реализованы базовый класс Decoder и производные от него классы ViterbiDecoder, SequentialDecoder, TresholdDecoder.

Закрытая часть и конструктор класса Decoder дублируют класс Coder. В открытой части определен метод hamming_distance, возвращающий расстояние Хэмминга между принятой кодовой комбинацией x и комбинацией y, которую должен был бы сгенерировать кодер при переходе из одного узла в другой. В методе реализован перебор символов принятых комбинаций, в ходе которого

выполняется их сравнение и добавление его результата к расстоянию в виде distance += x[i] != y[i].

Класс ViterbiDecoder включает конструктор, инициализирующий аргументы класса Decoder, метод do_viterbi_decode, необходимый для запуска процедуры декодирования по алгоритму Витерби, а также псевдоним typedef для двумерного целочисленного вектора Trellis, имитирующего кодовую решетку, и вспомогательные методы для построения путей на кодовой решетке.

Метод do_viterbi_decode принимает по ссылке строку с закодированной последовательностью bits. В методе объявляется объект типа Trellis и контейнер path_metrics для хранения метрик всех построенных путей. Перебор комбинаций закодированной последовательности реализован в цикле. Выделенные кадры кодовых символов current_bits передаются в метод update_path_metrics для построения возможных путей на данном шаге. После выхода из цикла выполняется поиск в path_metrics индекса минимального элемента, т.е. последнего состояния кодера на пути с наименьшей метрикой. Далее реализовано восстановление исходной последовательности с помощью движения по сформированному к этому моменту пути в решетке trellis в обратном направлении.

В методе update_path_metrics реализовано вычисление новых метрик путей с учётом переданных на шаге кодированных бит. Метод принимает рассматриваемый кадр кодовых символов current_bits, указатели на path_metrics и trellis. Для хранения обновленных метрик путей объявляется вектор new_path_metrics и новый столбец кодовой решетки – вектор new_trellis_column. Для каждого пути с целью вычисления метрики следующего ребра вызывается метод path_metric.

Метод path_metric принимает рассматриваемый кадр кодовых символов current_bits, state – текущее состояние регистра и указатель на path_metrics. В методе реализовано обновление state до target_state в случаях поступления нулевого бита или единицы. Для каждого случая вычисляется метрика ребра с помощью метода branch_metric. Метод возвращает пару для ребра с наименьшей

метрикой, состоящую из метрики и состояния регистра, в которое ведет ребро.

Метод branch_metric необходим для формирования кодовой комбинации output, получаемой при обращении к outputs_ по индексу target_state и вызова метода hamming_distance (bits, output).

В классе SequentialDecoder определены конструктор, деструктор, метод do_sequential_decode, а также структура Node, стек для хранения пройденных узлов кодового дерева steck_, вспомогательные методы gen_branch и gen_node.

В конструкторе реализована инициализация аргументов класса Decoder, а также создание указателя на экземпляр структуры Node, представляющего собой начальный узел дерева (исходное нулевое состояние регистра кодера). В деструкторе же осуществлено удаление всех указателей на экземпляры структуры Node в контейнере steck_.

В метод do_sequential_decode передается по ссылке строка с закодированной последовательностью bits. Перебор комбинаций закодированной последовательности реализован в цикле. На каждом шаге цикла цена рассматриваемого пути tau, вычисляемая как 0.5*pol_.size()*i - steck_[steck_.size()-1]→metric сравнивается с порогом T [11]. Если цена пути не меньше T, то для выделенного кадра кодовых символов current_bits вызывается метод gen_branch для генерации следующего узла кодового дерева и пути к нему. При этом T вычисляется как 2*[tau/2], тем самым допуская небольшое снижение цены пути. Если первое условие не было выполнено, то gen_branch вызывается для элемента steck_ по индексу steck_[pos] → previos_node_pos, т.е. для предыдущего узла на пути. Процедура повторяется до тех пор, пока не будет выполнено одно из условий: tau’ >= T (найден путь через другой узел) или previosT[previosT.size() - 1].second > pos (путь через другой узел не найден, необходимо уменьшить T), где previosT – контейнер пар, хранящий предыдущие значения порога и индексы узлов, при переходе в которые порог был увеличен.

Метод gen_branch принимает рассматриваемую кодовую комбинацию current_bits, и его индекс pos в steck_. В методе реализовано обновление state в

случаях поступления нулевого бита или единицы. Для каждого случая вычисляется метрика ветви с помощью метода hamming_distance. Для того, чтобы избежать повторного построения пути через формируемый узел, счетчик steck_[pos]→nodes_num увеличивается. Далее с помощью метода gen_node генерируется и добавляется в steck_ узел либо для ветви с наименьшей метрикой, либо для еще не рассмотренной (при nodes_num % 2 == 0).

В методе gen_node реализовано создание указателя на экземпляр структуры Node и инициализация его полей переданными в метод параметрами.

В классе TresholdDecoder определены аргументы i_bits_ - принятые информационные биты, p_bits_ - сформированные на стороне декодера проверочные биты. Реализованы методы класса: do_treshold_decode для вызова процедуры порогового декодирования, get_i_bits для выделения из принятой последовательности информационных бит, get_p_bits для выделения проверочных бит, encode_i_bits для кодирования принятых информационных бит.

В методе do_treshold_decode для принятой строки закодированных систематическим кодом бит bits последовательно вызываются методы get_i_bits и get_p_bits, возвращаемая строка которого заносится в переменную chanal_p_bits, а также метод encode_i_bits. Далее в цикле реализовано выделение комбинаций проверочных бит из chanal_p_bits и из p_bits_. Комбинации сравниваются при помощи метода hamming_distance. Если в результате его выполнения возвращаемое расстояние Хэмминга говорит о том, что искажены все проверочные биты для одного информационного, то соответствующий информационный бит исправляется и снова вызывается encode_i_bits для перерасчета проверочных бит на стороне декодера.

Методы get_i_bits и get_p_bits реализованы таким образом, что кодовая комбинация разбивается на первый информационный бит, который добавляется к строке i_bits_, и оставшиеся проверочные, которые добавляются к строке p_bits_.

Метод encode_i_bits дублирует метод encode класса Coder за исключением того, что он возвращает результат вызова метода get_p_bits для сформированной закодированной последовательности.

5.4 Программная реализация источника информации и дискретного канала с искажениями

В целях программной реализации источника информации была написана функция gen_sequence. На вход функция принимает требуемый размер генерируемой битовой последовательности length, по умолчанию установлен 10000. Сначала объявляется пустая строка sequence. Далее в цикле на основе интервала от 0 до length генерируются случайные числа, остаток деления на 2 которых приводится к типу std::string и добавляется к sequence.

Для реализации дискретного канала с искажениями была написана функция imit_chanell, принимающая на вход вероятность битовой ошибки P и закодированную последовательность бит enc_seq. Далее в цикле на основе интервала от 0 до enc_seq.size() генерируются случайные числа в диапазоне rand() % 100 (от 0 до 100), каждое число сравнивается со значением P * 100. В случае если сгенерированное число меньше, то текущий бит инвертируется путем его сложения с 1 по модулю 2.

Для расчета вероятности правильного и ошибочного декодирования была написана функция count_P. Функция принимает на вход ссылки на строку s1 с исходной, сгенерированной с помощью gen_sequence последовательностью бит и строку s2 с декодированной последовательностью, полученной в результате вызова методов do_viterbi_decode, do_sequential_decode и do_treshold_decode. В теле функции объявляются счетчики num_1 для совпадающих битов и num_2 для несовпадающих. В цикле реализован перебор символов принятых строк и их сравнение. Функция возвращает пару num_1 / double(s1.size()) и num_2 / double(s1.size()), где первый параметр – вероятность правильного декодирования, а второй – неправильного.

5.5 Программная реализация пользовательского интерфейса

Тип интерфейса — командная строка. В этом случае кодовое ограничение, скорость кодирования, образующие полиномы и вероятность битовой ошибки должны быть переданы параметрами argc и argv[] функции main.

Было решено, что первый аргумент командной строки — режим работы имитационной модели, второй — длина кодового ограничения, третий — скорость кодирования, далее – образующие полиномы и последний аргумент – вероятность битовой ошибки.

В начале программы было проверено, сколько параметров было указано при запуске. Если их недопустимое количество, программа должна быть завершена с ошибкой.

В целях обеспечения получения разных последовательностей при вызове функции gen_sequence была использована комбинация стандартных функций srand(time(NULL)).

Разбор аргументов командной строки реализован следующим образом. Объявляются переменные K (кодовое ограничение), n (количество выходных символов за один такт работы кодера), P (вероятность битовой ошибки) и вектор polinoms. Переменная K инициализируется результатом вызова atoi(argv[2]), n – stoi(argv[2].erase(0,2)). В цикле на основе интервала от 0 до n-1 в вектор polinoms заносятся образующие полиномы с помощью выражения polinoms.push_back(atoi(argv[3 + i])), где i – инкремент цикла. Переменная P инициализируется вызовом atof(argv[3 + n + 1]).

Далее объявляется переменная типа std::string, для которой вызывается функция gen_sequence(10000).

Создается объект класса Coder coder(K, polinoms) и вызывается метод encode для сгенерированной ранее последовательности. Результат вызова метода encode и переменная P передаются в функцию imit_chanell для внесения ошибок в закодированную последовательность.

Далее в зависимости от значения аргумента argv[1] производится декодирование требуемыми методами. В общем случае, когда argv[1] == “VST”,

создаются объекты классов ViterbiDecoder, SequentialDecoder, TresholdDecoder. В конструкторе передаются K и polinoms. Вызываются методы do_viterbi_decode, do_sequential_decode и do_treshold_decode объектов соответствующих классов. Перед созданием объекта класса TresholdDecoder необходимо проверить, что polinoms[0] == 1, т.к. пороговый декодер реализован для систематического кода. Если проверка не пройдена, то декодирование пороговым методом не выполняется.

Результат декодирования передается в функцию count_P вместе с исходной последовательностью. Результат работы функции выводится в консоль.

Текст главного модуля main.cpp приведен в приложении Ж. Руководство пользователя к имитационной модели представлено в приложении И.

6 Тестирование имитационной модели

6.1 Тестирование имитационных моделей

Для оценки достоверности имитационной модели и укрепления доверия к ней необходимо убедиться, что полученные с ее помощью результаты являются верными, а выводы, сделанные на их основе, будут правильными и применимыми для реальной системы. С этой целью на этапе испытания разработанной модели проводится тестирование, направленное на нахождение ошибок и неточностей. При создании моделей не в средах имитационного моделирования, а с помощью инструментов программирования общего назначения (в интегрированных средах разработки) могут быть использованы подходы к тестированию, являющиеся стандартными в разработке программного обеспечения [12].

Модульное тестирование – широко применяемая в программной разработке разновидность тестирования. Идея модульного тестирования – написание тестов для функциональных модулей или процессов. Это обеспечивает быструю проверку кода, а также облегчает обнаружение и устранение ошибок.

Перед тем, как приступить к тестированию, необходимо подготовить наборы тестов – тестовые сценарии. На их основе выполняется проверка совпадения результатов, полученных при обработке имитационной моделью заранее определенных входных данных, с ожидаемыми результатами.

6.2 Подготовка тестовых сценариев

Тестовый сценарий — это описание начальных условий, входных данных и ожидаемого результата. Для его подготовки необходимо понимать задачи, решаемые рассматриваемой имитационной моделью. Таким образом, можно выделить:

- сверточное кодирование;
- декодирование сверточного кода по алгоритму Витерби;
- последовательное декодирование сверточного кода;

– пороговое декодирование сверточного кода;

– декодирование в случае наличия ошибок в принятой на входе декодера последовательности бит.

Соответственно, было написано 5 тестовых сценариев.

Тестовый сценарий для проверки правильности программной реализации сверточного кодирования представлен таблицей 1. В рамках его создания был подготовлен набор ожидаемых битовых последовательностей, полученный в результате кодирования 15 бит с учетом всех допустимых скоростей кодирования и трех значений длины кодового ограничения, включая пограничные. Каждый тест из данного тестового сценария требует сравнения ожидаемой и полученной в результате кодирования последовательностей бит.

Таблица 1 – Тестовый сценарий для проверки правильности программной реализации сверточного кодирования

№	Тест	Входные данные	Ожидаемый результат
1.1	Сверточное кодирование (7, 5) при К=3, R=$^1/_2$	Последовательность бит 101100101110001	Закодированная последовательность бит 111000010111111000011001110011011
1.2	Сверточное кодирование (7, 7, 5) при К=3, R=$^1/_3$	Последовательность бит 101100101110001	Закодированная последовательность бит 111110000001001111111110000001110001111000111110111
1.3	Сверточное кодирование (109, 79) при К=7, R=$^1/_2$	Последовательность бит 101100101110001	Закодированная последовательность бит 110100011010111101100111110110000011001011
1.4	Сверточное кодирование (109, 95, 83) при К=7, R=$^1/_3$	Последовательность бит 101100101110001	Закодированная последовательность бит 111011001010110100101100010101001111110000111100000111001110011100111
1.5	Сверточное кодирование (15015, 20111) при К=15, R=$^1/_2$	Последовательность бит 101100101110001	Закодированная последовательность бит 111100010000101010010111100001011011000101000100001110100 1
1.6	Сверточное кодирование (15015, 20111, 32514) при К=15, R=$^1/_3$	Последовательность бит 101100101110001	Закодированная последовательность бит 110111000011001000100101101010011110101001010010100110000001010001010001000111101101011

В тестовые сценарии, написанные для проверки правильности работы декодера Витерби и последовательного декодера, входящих в состав имитационной модели, были включены тесты, требующие сравнения исходных 15 бит с результатом декодирования последовательностей, подготовленных для тестирования кодера в рамках предыдущего тестового сценария. Для проверки порогового декодера были подготовлены последовательности, закодированные систематическим кодом. Тестовые сценарии сведены в таблицу 2.

Таблица 2 – Тестовый сценарий для проверки правильности программной реализации декодирования

№	Тест	Входные данные	Ожидаемый результат
Декодирование по алгоритму Витерби			
2.1	Декодирование по алгоритму Витерби (7, 5) при К=3, R=$^1/_2$	Закодированная последовательность бит 111000010111111000011001110011101 1	Декодированная последовательность бит 101100101110001
2.2	Декодирование по алгоритму Витерби (7, 7, 5) при К=3, R=$^1/_3$	Закодированная последовательность бит 111110000001001111111110000001110001 111000111110111	Декодированная последовательность бит 101100101110001
2.3	Декодирование по алгоритму Витерби (109, 79) при К=7, R=$^1/_2$	Закодированная последовательность бит 111010001101011110110011111011000001 1001011	Декодированная последовательность бит 101100101110001
2.4	Декодирование по алгоритму Витерби (109, 95, 83) при К=7, R=$^1/_3$	Закодированная последовательность бит 111011001010110100101100010101001111 100001111000001110011100111	Декодированная последовательность бит 101100101110001
2.5	Декодирование по алгоритму Витерби (15015, 20111) при К=15, R=$^1/_2$	Закодированная последовательность бит 111100010000101010010111100001011011 000101000100001110100 1	Декодированная последовательность бит 101100101110001
2.6	Декодирование по алгоритму Витерби (15015, 20111, 32514) при К=15, R=$^1/_3$	Закодированная последовательность бит 110111000011001000100101101010011 101001010010100110000010010001010001000111101101011	Декодированная последовательность бит 101100101110001

Продолжение таблицы 2

№	Тест	Входные данные	Ожидаемый результат
Последовательное декодирование			
3.1	Последовательное декодирование (7, 5) при К=3, R=$^1/_2$	Закодированная последовательность бит 11100001011111100001100111001111011	Декодированная последовательность бит 101100101110001
3.2	Последовательное декодирование (7, 7, 5) при К=3, R=$^1/_3$	Закодированная последовательность бит 11111000000100111111111000000111000111111000111110111	Декодированная последовательность бит 101100101110001
3.3	Последовательное декодирование (109, 79) при К=7, R=$^1/_2$	Закодированная последовательность бит 11101000110101111011001111101100000110010111	Декодированная последовательность бит 101100101110001
3.4	Последовательное декодирование (109, 95, 83) при К=7, R=$^1/_3$	Закодированная последовательность бит 11101100101011010010110001010100111110000111100000111001110011100111	Декодированная последовательность бит 101100101110001
3.5	Последовательное декодирование (15015, 20111) при К=15, R=$^1/_2$	Закодированная последовательность бит 11110001000010101001011110000101101100010100010000111010011	Декодированная последовательность бит 101100101110001
3.6	Последовательное декодирование (15015, 20111, 32514) при К=15, R=$^1/_3$	Закодированная последовательность бит 11011100001100100010010110101001111010100101001010011000000100100010100001000111101101011	Декодированная последовательность бит 101100101110001
Пороговое декодирование			
4.1	Пороговое декодирование (1, 7) при К=3, R=$^1/_2$	Закодированная последовательность бит 11011010000111011010110001001101011	Декодированная последовательность бит 101100101110001
4.2	Пороговое декодирование (1, 5, 7) при К=3, R=$^1/_3$	Закодированная последовательность бит 11100110011001001111100110011010101001100000111001011	Декодированная последовательность бит 101100101110001
4.3	Пороговое декодирование (1, 109) при К=7, R=$^1/_2$	Закодированная последовательность бит 11001010010111011011100101001100000100010101	Декодированная последовательность бит 101100101110001
4.4	Пороговое декодирование (1, 95, 83) при К=7, R=$^1/_3$	Закодированная последовательность бит 11101110111001000010100011010110101100000011110000001010011000011	Декодированная последовательность бит 101100101110001

Продолжение таблицы 2

№	Тест	Входные данные	Ожидаемый результат
4.5	Пороговое декодирование (1, 20111) при К=15, R=$^1/_2$	Закодированная последовательность бит 1101101100001000101111010000110100010001010001000001000001	Декодированная последовательность бит 101100101110001
4.6	Пороговое декодирование (1, 15015, 20111) при К=15, R=$^1/_3$	Закодированная последовательность бит 1110111001010000001100101101011010110000101001010011000001001000001000000011010010001	Декодированная последовательность бит 101100101110001

Для проверки правильности декодирования в случае наличия ошибок было выполнено сверточное кодирование 10 бит при длине кодового ограничения равной 4 и с учетом всех допустимых скоростей кодирования. В полученные закодированные последовательности были внесены ошибки при помощи соответствующей функции, входящей в состав программной реализации имитационной модели, с вероятностью битовой ошибки от 0 до 1 с шагом 0,1. Далее для полученного набора битовых последовательностей были вручную рассчитаны ожидаемые результаты декодирования для каждого из декодеров. Таким образом, каждый тест данного тестового сценария требует проверки на соответствие ожидаемого результата декодирования и полученной в его результате последовательности бит. Тесты, входящие в состав данного сценария сведены в таблицу 3.

Таблица 3 – Тестовый сценарий для проверки правильности программной реализации декодирования в случае наличия ошибок

№	Тест	Входные данные	Ожидаемый результат
5.1	Декодирование (1, 10) при К=4, R=$^1/_2$, P=0,1	Закодированная последовательность бит с ошибками 101<u>0</u>0101110100011<u>0</u>0<u>0</u>0010<u>1</u>	Декодированная последовательность бит для: декодер Витерби - 1100100010 последовательный декодер - 1100100010 пороговый декодер - 1100100010
5.2	Декодирование (1, 7, 10) при К=4, R=$^1/_3$, P=0,1	Закодированная последовательность бит с ошибками 1101010<u>0</u>0111110110100011100110<u>1</u>1000	Декодированная последовательность бит для: декодер Витерби - 1100100010 последовательный декодер - 1100100010 пороговый декодер - 1100100010

Продолжение таблицы 3

№	Тест	Входные данные	Ожидаемый результат
5.3	Декодирование (1, 10) при К=4, R=$^1/_2$, P=0,2	Закодированная последовательность бит с ошибками 1<u>1</u>11010111<u>0</u>0000110<u>1</u>1000<u>1</u><u>0</u>	Декодированная последовательность бит для: декодер Витерби - 1100100010 последовательный декодер - 1100100011 пороговый декодер - 1100100011
5.4	Декодирование (1, 7, 10) при К=4, R=$^1/_3$, P=0,2	Закодированная последовательность бит с ошибками 1101010<u>0</u><u>0</u>1111<u>0</u><u>0</u>0100<u>1</u>111 00<u>1</u><u>0</u>11<u>0</u>1<u>0</u>1000	Декодированная последовательность бит для: декодер Витерби - 1100100010 последовательный декодер - 1100100010 пороговый декодер - 1100100010
5.5	Декодирование (1, 10) при К=4, R=$^1/_2$, P=0,3	Закодированная последовательность бит с ошибками 1<u>1</u>10<u>1</u>111101<u>0</u><u>0</u><u>0</u>0<u>0</u>100100	Декодированная последовательность бит для: декодер Витерби - 1<u>0</u>00<u>1</u>10<u>0</u><u>0</u> последовательный декодер - 11<u>0</u>1100<u>0</u><u>0</u> пороговый декодер - 11<u>0</u>1100<u>0</u><u>0</u>
5.6	Декодирование (1, 7, 10) при К=4, R=$^1/_3$, P=0,3	Закодированная последовательность бит с ошибками 1101010<u>0</u>0111<u>0</u>1110<u>1</u>0<u>1</u>111 0011011<u>1</u>01001	Декодированная последовательность бит для: декодер Витерби - 1100100010 последовательный декодер - 1100100010 пороговый декодер - 1100<u>1</u>10010
5.7	Декодирование (1, 10) при К=4, R=$^1/_2$, P=0,4	Закодированная последовательность бит с ошибками 1<u>1</u>11<u>1</u>011<u>0</u><u>0</u>1001100<u>0</u>1<u>0</u>1100	Декодированная последовательность бит для: декодер Витерби - 1<u>0</u><u>0</u><u>0</u><u>0</u>1<u>0</u>10 последовательный декодер - 11<u>1</u>010<u>1</u>010 пороговый декодер - 11<u>1</u>010<u>1</u>010
5.8	Декодирование (1, 7, 10) при К=4, R=$^1/_3$, P=0,4	Закодированная последовательность бит с ошибками 110<u>0</u>1100101<u>1</u>0<u>1</u>101<u>0</u>010<u>1</u>10<u>1</u><u>0</u> <u>1</u>10<u>1</u>10<u>1</u>01<u>1</u>01	Декодированная последовательность бит для: декодер Витерби - 1100<u>0</u><u>0</u>0<u>1</u><u>1</u> последовательный декодер - 1<u>0</u><u>0</u>1<u>0</u><u>0</u><u>0</u>111 пороговый декодер - 1<u>0</u>10<u>0</u><u>0</u><u>1</u><u>1</u>1
5.9	Декодирование (1, 10) при К=4, R=$^1/_2$, P=0,5	Закодированная последовательность бит с ошибками 10<u>0</u><u>0</u>1<u>1</u>1111<u>1</u><u>0</u><u>0</u>1<u>0</u>1111<u>1</u>1<u>1</u><u>0</u><u>0</u>1<u>0</u>	Декодированная последовательность бит для: декодер Витерби - <u>0</u><u>0</u>11<u>1</u><u>1</u>0<u>0</u>1<u>1</u> последовательный декодер - 1<u>0</u>11<u>1</u>10<u>0</u>1<u>1</u> пороговый декодер - <u>0</u><u>0</u>11<u>1</u><u>1</u>0<u>0</u>1<u>1</u>

Продолжение таблицы 3

№	Тест	Входные данные	Ожидаемый результат
5.10	Декодирование (1, 7, 10) при К=4, R=$^1/_3$, P=0,5	Закодированная последовательность бит с ошибками 1001100011100111110110111 00001110111101	Декодированная последовательность бит для: декодер Витерби - 0100011110 последовательный декодер - 0100011000 пороговый декодер - 1100011110
5.11	Декодирование (1, 10) при К=4, R=$^1/_2$, P=0,6	Закодированная последовательность бит с ошибками 011100001000001101111110011	Декодированная последовательность бит для: декодер Витерби - 0000100111 последовательный декодер - 0100100101 пороговый декодер - 1000100101
5.12	Декодирование (1, 7, 10) при К=4, R=$^1/_3$, P=0,6	Закодированная последовательность бит с ошибками 001111011100010000101100000100100100100010	Декодированная последовательность бит для: декодер Витерби - 0101000000 последовательный декодер - 0101000000 пороговый декодер - 0101010100
5.13	Декодирование (1, 10) при К=4, R=$^1/_2$, P=0,7	Закодированная последовательность бит с ошибками 00011010101001111010111011	Декодированная последовательность бит для: декодер Витерби - 0010110110 последовательный декодер - 0011110111 пороговый декодер - 0111110101
5.14	Декодирование (1, 7, 10) при К=4, R=$^1/_3$, P=0,7	Закодированная последовательность бит с ошибками 00101011100100110110010000 11100111110111	Декодированная последовательность бит для: декодер Витерби - 0011010001 последовательный декодер - 0010000001 пороговый декодер - 0010110101
5.15	Декодирование (1, 10) при К=4, R=$^1/_2$, P=0,8	Закодированная последовательность бит с ошибками 00101000011011000110111011	Декодированная последовательность бит для: декодер Витерби - 0100010001 последовательный декодер - 0110011001 пороговый декодер - 0110011001
5.16	Декодирование (1, 7, 10) при К=4, R=$^1/_3$, P=0,8	Закодированная последовательность бит с ошибками 00101001000010011011001000 1100100100001	Декодированная последовательность бит для: декодер Витерби - 0000010000 последовательный декодер - 0000010000 пороговый декодер - 0000111101

Продолжение таблицы 3

№	Тест	Входные данные	Ожидаемый результат
5.17	Декодирование (1, 10) при К=4, R=$^1/_2$, P=0,9	Закодированная последовательность бит с ошибками 010010101010111001101111111	Декодированная последовательность бит для: декодер Витерби - 0010111001 последовательный декодер - 0011111101 пороговый декодер - 0010111001
5.18	Декодирование (1, 7, 10) при К=4, R=$^1/_3$, P=0,9	Закодированная последовательность бит с ошибками 001110110100000101111110001100110110111	Декодированная последовательность бит для: декодер Витерби - 0101100101 последовательный декодер - 0101010000 пороговый декодер - 0111011101
5.19	Декодирование (1, 10) при К=4, R=$^1/_2$, P=1	Закодированная последовательность бит с ошибками 010010100010111001101110 11	Декодированная последовательность бит для: декодер Витерби - 0000010001 последовательный декодер - 0001001101 пороговый декодер - 0001101101
5.20	Декодирование (1, 7, 10) при К=4, R=$^1/_3$, P=1	Закодированная последовательность бит с ошибками 001010110100000100101110001100101110111	Декодированная последовательность бит для: декодер Витерби - 0000000110 последовательный декодер - 0011011101 пороговый декодер - 0011011101

Дополнительно был подготовлен тестовый сценарий для проверки корректности генерируемой последовательности бит, а также корректности последовательности бит после внесения в нее ошибок. Сценарий приведен в таблице 4.

Таблица 4 – Тестовый сценарий для проверки корректности генерируемых последовательностей бит

№	Тест	Входные данные	Ожидаемый результат
6.1	Генерация битовой последовательности заданной длины	-	Длина сгенерированной битовой последовательности равна 10000
6.2	Проверка на содержание в сгенерированной последовательности только 0 и 1	-	Отсутствие в сгенерированной битовой последовательности других символов, отличных от 0 и 1
6.3	Проверка длины последовательности бит после внесения битовых ошибок	Вероятность битовой ошибки, закодированная последовательность бит	Длина последовательности после внесения битовых ошибок не изменилась

Продолжение таблицы 4

№	Тест	Входные данные	Ожидаемый результат
6.4	Проверка на содержание в последовательности с ошибками только 0 и 1	Вероятность битовой ошибки, закодированная последовательность бит	Отсутствие в последовательности с ошибками других символов, отличных от 0 и 1

6.3 Написание модульных тестов

На основе подготовленных тестовых сценариев были написаны модульные тесты. Для этого были применены макросы, входящие в состав библиотеки UnitTest++.

Для каждого тестового сценария был создан набор тестов с помощью макроса SUITE, для описания каждого теста был использован макрос TEST.

По тестовому сценарию проверки правильности программной реализации сверточного кодирования было написано 6 тестов. В каждом тесте был создан объект класса Coder. Требуемые длина кодового ограничения и полиномы заданы заранее и переданы в конструкторе класса. Далее с помощью макроса CHECK_EQUEL осуществлена проверка соответствия ожидаемой закодированной последовательности и результата вызова метода encode от заданной в тестовом сценарии входной последовательности бит.

Для проверки правильности программной реализации декодера Витерби было написано 6 тестов. В каждом тесте был создан объект класса ViterbiDecoder. Требуемые длина кодового ограничения и полиномы заданы заранее и переданы в конструкторе класса. Для успешного выполнения каждого теста должна быть пройдена проверка соответствия ожидаемой декодированной последовательности и результата вызова метода do_viterbi_decode от заданной входной закодированной последовательности бит. Данная проверка осуществлена с помощью макроса CHECK_EQUEL.

Для тестовых сценариев проверки правильности программной реализации последовательного и порогового декодеров было написано 12 аналогичных тестов, отличие заключается в создании объектов и вызова методов классов SequentialDecoder и TresholdDecoder соответственно.

Для проверки работоспособности декодеров в случаях наличия битовых ошибок в закодированной последовательности было написано 20 тестов. В тестах созданы и инициализированы объекты классов ViterbiDecoder, SequentialDecoder и TresholdDecoder. Далее для каждого декодера с помощью макроса CHECK_EQUEL задана проверка соответствия ожидаемой декодированной последовательности и результата вызова методов декодирования для последовательностей с заранее внесенными битовыми ошибками.

В SUITE для проверки корректности генерируемой последовательности бит было включено 4 теста. В первых двух тестах произведена проверка результата вызова функции gen_sequence(10000). В первом тесте в макрос CHECK_EQUEL передаются результат вызова size() от сгенерированной последовательности и требуемый размер. Во втором тесте после вызова gen_sequence(10000) написана процедура проверки каждого отдельного бита на отличие от 0 и 1. Аналогичные тесты прописаны для тестирования функции imit_chanal.

Реализация модульных тестов приведена в приложении К.

Таким образом, для оценки достоверности имитационной модели и правильности программной реализации сверточного кодирования и декодирования было запущено выполнение разработанных тестов в эмуляторе терминала Linux, результат:

```
Success: 48 tests passed.
Test time: 1.39 seconds.
```

7 Проведение экспериментов с использованием разработанной имитационной модели

Разработанная имитационная модель может быть использована для проведения статистических исследований помехоустойчивости сверточных кодов при естественных и преднамеренных помехах в канале связи. Результаты подобных исследований могут быть использованы на начальном этапе разработки систем помехоустойчивого кодирования, чтобы с заданной достоверностью гарантировать целостность данных и их защищенность от разрушающих воздействий в каналах связи. Кроме того, имитационная модель может быть включена в образовательный процесс в целях изучения сверточных кодов.

Для демонстрации экспериментов, проведение которых возможно с использованием разработанной модели, был выбран систематический сверточный код с длиной кодового ограничения 4 и скоростью кода 1/2. Результаты моделирования этого кода при возрастании битовой ошибки от 0 до 1 и при выполнении 100 прогонов модели приведены на рисунке 23.

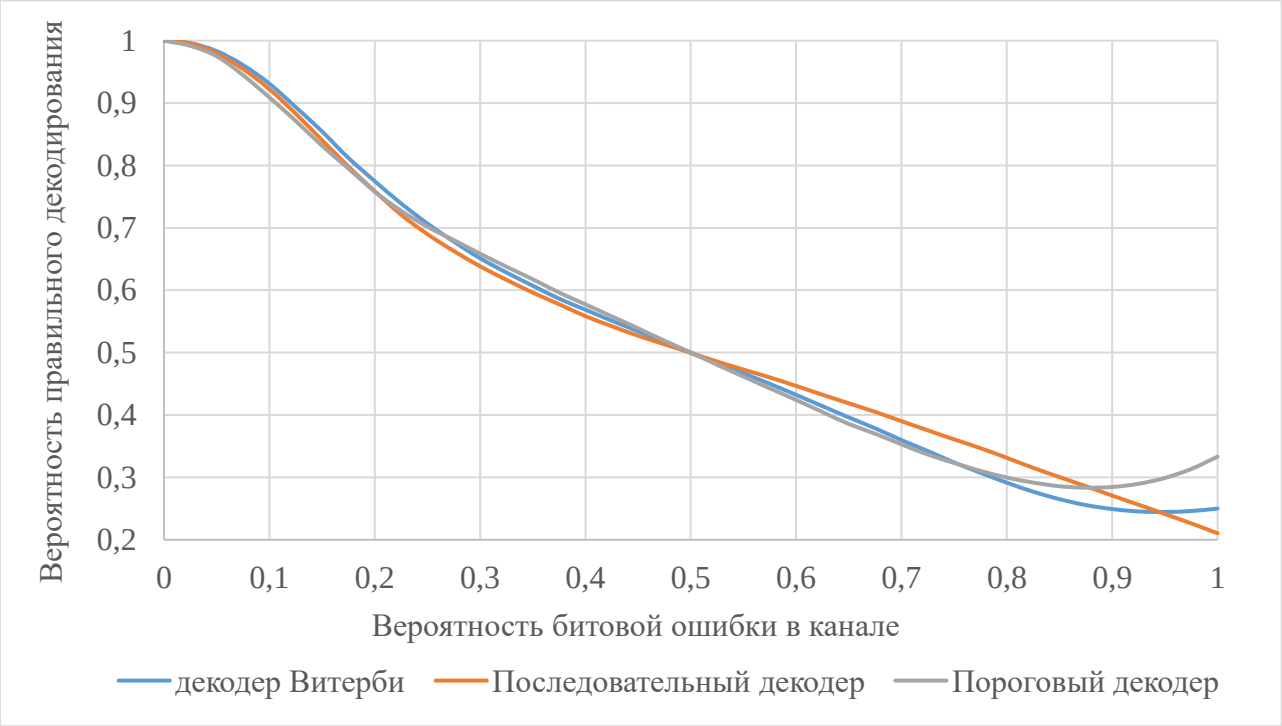

Рисунок 23 – Сравнение помехоустойчивости сверточных кодов при K=4 R=1/2 и декодировании разными методами

По графику на рисунке 23 видно, что рассматриваемые декодеры при данных параметрах кода обеспечивают примерно одинаковую вероятность правильного декодирования с ростом вероятности битовой ошибки. Декодеры исправляют ошибки в принимаемой последовательности при повышении вероятности искажения бита до 0,05. Далее вероятность правильного декодирования убывает линейно.

Далее был рассмотрен систематический сверточный код с такой же длиной кодового ограничения 4, но с большим количеством выходных символов за один такт работы кодера – скоростью кода 1/3. Результаты моделирования этого кода при возрастании битовой ошибки от 0 до 1 и выполнении 100 прогонов модели приведены на рисунке 24.

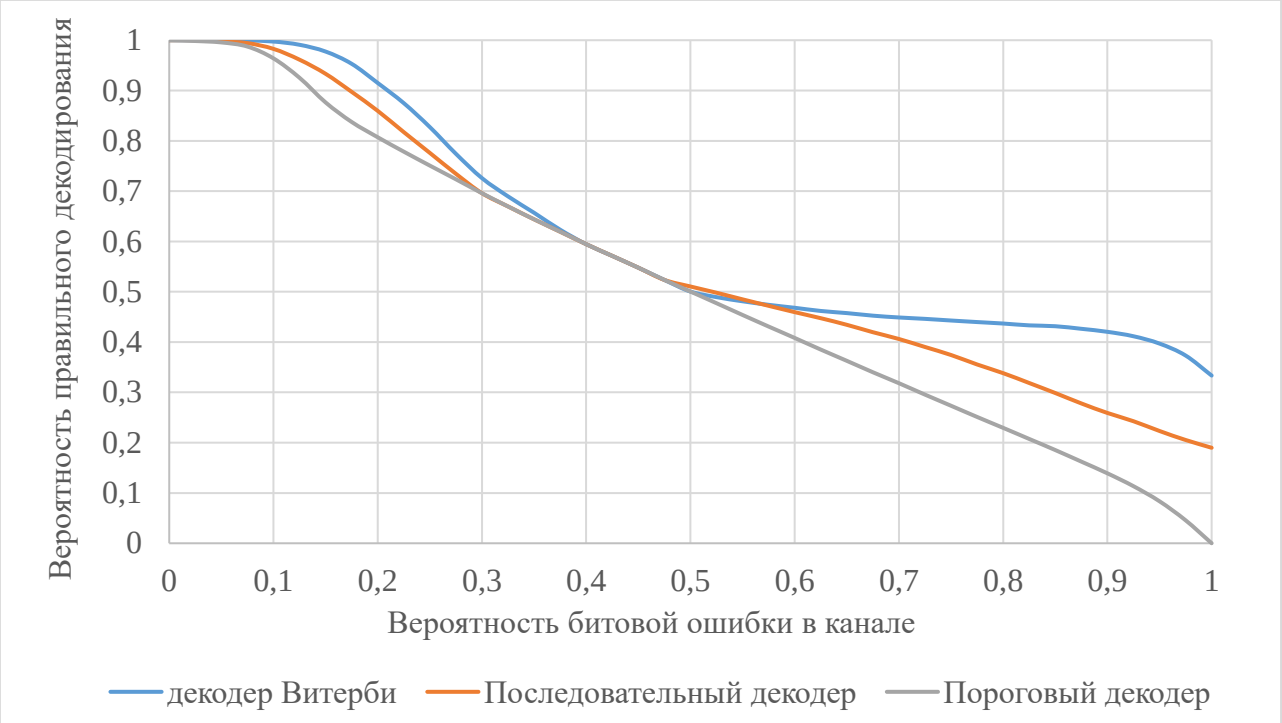

Рисунок 24 – Сравнение помехоустойчивости сверточных кодов при K=4 R=1/3 и декодировании разными методами

По графику на рисунке 24 видно, что рассматриваемый код в сравнении с предыдущим обеспечивает более высокую помехоустойчивость за счет введения в передаваемую последовательность большей избыточности. По графику также видно, что декодер Витерби при данных параметрах кода более эффективен в сравнении с последовательным и пороговым методами.

Таким образом, может быть сделан вывод, что при скорости кодирования 1/3 исправляющая способность кода выше, чем при скорости 1/2. Однако при увеличении количества выходных символов за один такт работы кодера до 5 было замечено, что вероятность правильного декодирования снижается.

Следующим был рассмотрен систематический код с большей длиной кодового ограничения K=9, скоростью кода 1/2. Результаты моделирования этого кода при возрастании битовой ошибки от 0 до 1 и при выполнении100 прогонов модели приведены на рисунке 25. По приведенному графику можно сделать вывод, что увеличение длины кодового ограничения также влияет на способность кода исправлять ошибки.

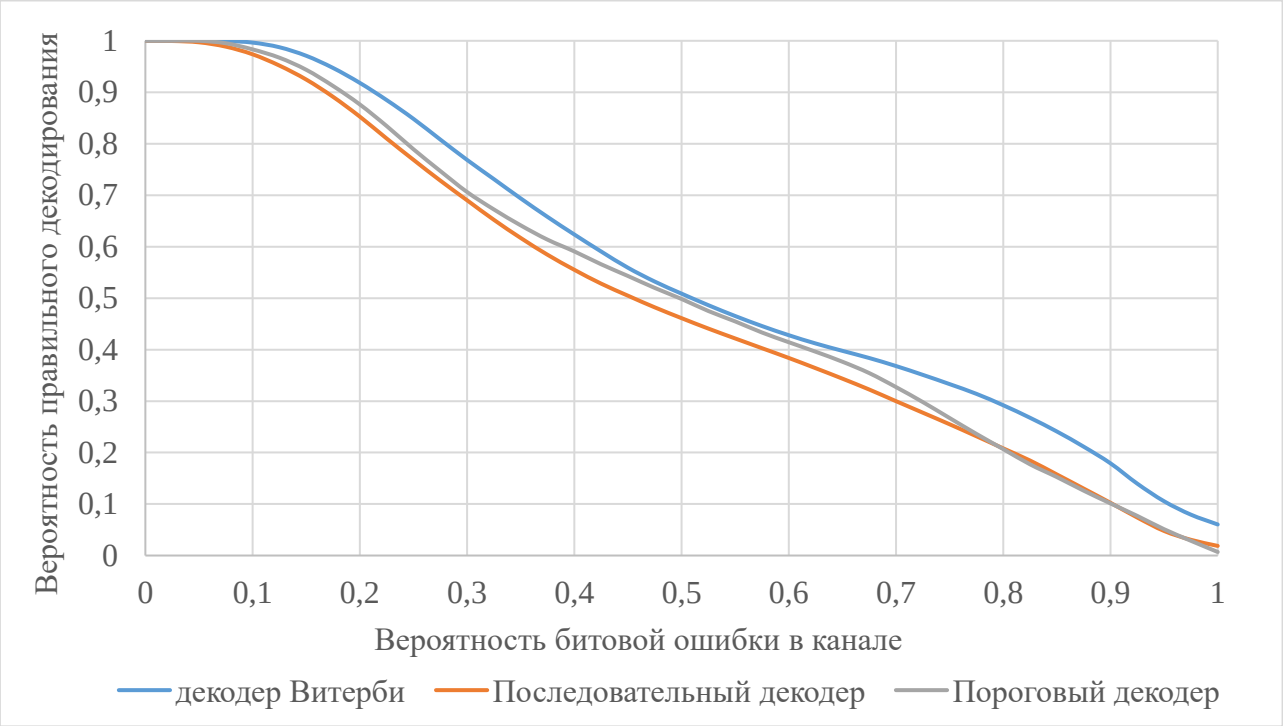

Рисунок 25 – Сравнение помехоустойчивости сверточных кодов при K=9 R=1/2 и декодировании разными методами

8 Вредные психофизиологические факторы, влияющие на опорно-двигательный аппарат специалиста по защите информации. Методы и средства физической культуры, снижающие их воздействие.

Трудовой деятельности специалиста по защите информации сопутствуют интенсивные психофизиологические процессы, связанные с использованием интеллектуальных, мышечных, психических и других ресурсов. По этой причине возможно возникновение дискомфортного психофизиологического состояния, что может влиять на опорно-двигательный аппарат, сосудисто-сердечную систему, органы зрения, нервную систему, а также психоэмоциональное состояние.

Специалисты по защите информации проводят сидя на рабочем месте за компьютером по несколько часов в день. С точки зрения физиологии сидячая поза – наиболее вредная, особенно при этом страдает опорно-двигательный аппарат, что может привести к преждевременному развитию таких заболеваний как остеохондроз, радикулит, артрит.

Опорно-двигательный аппарат – это единая функциональная система костей, их соединений и мышц. На него могут влиять следующие факторы:

- длительное нахождение в статичном положении, в результате которого мышечный каркас спины ослабевает, что негативно влияет на прочность позвонков и эластичность межпозвонковых дисков;
- длительная интенсивная работа на клавиатуре и однотипные монотонные движения при работе с компьютерной мышкой, следствием которых могут стать воспалительные процессы в кистях и предплечьях, нарушение координации движений пальцев, а также судороги, развитие туннельного или кистевого синдрома;
- неправильная осанка во время работы, которая увеличивает нагрузку на позвоночник, что приводит к невротическим состояниям и патологическим нарушениям со стороны опорно-двигательного аппарата. Напряжение мышц основания головы вследствие неудобной позы за компьютером ограничивает

кровоток в сосудах шеи, кровоснабжение головы и отток крови от нее, что влечет за собой головные боли, боли в шее, руках и кистях.

Во избежание проблем с опорно-двигательным аппаратом специалисту по защите информации необходимо придерживаться следующих рекомендаций:

- грамотно подходить к организации рабочего места;
- делать регулярные перерывы в работе с мышью и клавиатурой;
- поддерживать правильную осанку;
- придерживаться активного образа жизни.

Кроме того, для снижения воздействия вредных психофизиологических факторов на опорно-двигательный аппарат рекомендуется выполнять физические упражнения, которые относятся к основным средствам физической культуры. При этом важно уделять внимание таким упражнениям для туловища, как наклоны в стороны, круговые движения, разгибания в вертикальном положении. Каждое упражнение должно выполняться без рывков и в медленном темпе. Немаловажно сосредоточиться на формировании и закреплении навыка правильной осанки. Для этого необходимо регулярно выполнять упражнения, направленные на укрепление мышц спины, мышц брюшного пресса, мышц боковой поверхности туловища, симметричные и асимметричные упражнения [13].

Заключение

В ходе выполнения дипломной работы был рассмотрен и проанализирован необходимый теоретический материал в части угроз безопасности информации при ее передаче по каналам связи и построения сверточных кодеров и декодеров. Произведен обзор трех методов декодирования сверточного кода.

Была разработана концептуальная модель системы передачи данных, а также выполнено проектирование программы, реализующей имитационную модель системы передачи данных на основе сверточного кодирования.

На заключительном этапе была разработана программная реализация имитационной модели на языке C++, к которой также было составлено руководство пользователя. В составе разработанной программы были реализованы программный модуль кодера encoder.h и программный модуль декодеров decoder.h, к которым было составлено руководство разработчика. Для оценки достоверности имитационной модели были написаны и выполнены модульные тесты.

Кроме того, были проведены эксперименты с использованием разработанной имитационной модели, направленные на исследование влияния длины кодового ограничения и скорости кода на способность декодеров исправлять ошибки в принятых последовательностях бит.

Таким образом, техническое задание на дипломную работу выполнено в полном объеме.

Список использованных источников

1 Шиганова, М.В. Методы обхода искажений в беспроводных каналах связи [Электронный ресурс] / М.В. Шиганова, А.В. Поначугин // Приволжский научный вестник. – 2016.

2 Кузнецов, Н. А. Разработка алгоритмов и моделирование обеспечения целостности данных в информационных системах обмена дискретной информацией: дис. канд. тех. наук: 05.13.18 / Николай Алексеевич Кузнецов. – Ульяновск. – 2022. – 131 с.

3 Синицин, Д. В. Повышение помехоустойчивости радиотехнических систем передачи информации с использованием сверточных алгоритмов обработки сигналов: дис. канд. тех. наук: 05.12.04: защищена 22.01.14: утв. 15.07.14 / Дмитрий Вячеславович Синицин. – Владимир, 2014. С. 127.

4 Никитин, Г. И. Сверточные коды: учеб. Пособие. – СПб. : Изд-во СПбГУАП, 2001. – 80 с.

5 Морелос-Сарагоса, Р. Искусство помехоустойчивого кодирования. Методы, алгоритмы, применение. - М.: Техносфера, 2006. – 320 с.

6 Когновицкий, О. С. Теория помехоустойчивого кодирования. Часть 2. Сверточные коды. Турбокоды / О. С. Когновицкий, В. М. Охорзин, И. А. Небаев. – СПб. : Изд-во СПбГУТ, 2015. – 64 с.

7 Сизоненко, А. Б. Моделирование работы кодера сверточного кода посредством арифметических полиномов // Вестник ВИ МВД России. - 2010. - №3.

8 Банкет, В. Л. Цифровые методы в спутниковой связи / В. Л. Банкет, В. М. Дорофеев. - М.: Радио и связь, 1988. - 240 с.

9 Кларк, Д. К. Кодирование с исправлением ошибок в системах цифровой связи / Д. К. Кларк; Пер. с англ. С. И. Гельфанда под ред. Б. С. Цыбакова. - М. : Радио и связь. - 1987. - 391 с.

10 Касами Т. Теория кодирования / Т. Касами, Н. Токура, Е. Ивадари, Я. Инагаки. Пер. с японского А.В. Кузнецова под редакцией Б.С. Цыбакова и С.И.

Гельфанда. – М.: Издательство «Мир»: Редакция литературы по новой технике. - 1978.

11 Блейхут, Р. Теория и практика кодов, контролирующих ошибки / Р. Блейхут. Пер. с англ. – М.: Мир, 1986. – 576 с.

12 Гацко, А.А. Тестирование имитационных моделей [Электронный ресурс] / А.А. Гацко, О.С. Медведев, Л.Р. Коркин // 58-я научная конференция аспирантов, магистрантов и студентов: сб. статей. БГУИР. – Минск. - 2022.

13 Цеева, Н.А. Технология применения средств лечебной физкультуры при нарушениях опорно-двигательного аппарата на занятиях физическим воспитанием [Электронный ресурс] / Н.А. Цеева, А.Г. Заболотний, Е.В. Золотцева, О.О. Ельникова, Ю.Д. Ушхо // Вестник Адыгейского государственного университета: сб. статей. АГУ. – Майкоп. – 2017. - №4. – С. 136-139.

Приложение А

(обязательное)

Текст заголовочного файла encoder.h

```
#include <string>
#include <vector>
#include <cassert>

using namespace std;

class Coder
{
public:
    /// конструктор класса
    /// in: кодовое ограничение
    ///     образующие полиномы
    Coder( int K, vector< int >& pol );

    /// сверточное кодирование
    /// in: битовая последовательность
    /// out: закодированная битовая последовательность
    string encode( const string& bits );

private:

    /// парсинг образующих полиномов
    /// in: кодовое ограничение
    ///     полином
    /// out: адаптированный полином
    int pars_polinom_bits( int num_bits, int polinom_bits );

    /// расчет всех возможных комбинаций на выходе кодера
    void gen_outputs();

    vector <int> pol_; /// образующие полиномы
    vector <string> outputs_; /// все возможные комбинации на выходе
кодера
    int K_; /// кодовое ограничение
};
```

Приложение Б

(обязательное)

Текст основного файла программного модуля encoder.cpp

```
#include <math.h>
#include "encoder.h"

/// парсинг образующих полиномов
int Coder::pars_polinom_bits( int num_bits, int polinom_bits )
{
    int output = 0;
    while( num_bits-- > 0 )
    {
        output = (output << 1) + (polinom_bits & 1);
        polinom_bits >>= 1;
    }

    return output;
}

/// расчет всех возможных комбинаций на выходе кодера
void Coder::gen_outputs()
{
    outputs_.resize(1 << K_);
    for( int i = 0; i < outputs_.size(); i++ )
    {
        for( int j = 0; j < pol_.size(); j++ )
        {
            int polinom = pars_polinom_bits(K_, pol_[j]);
            int input = i; int output = 0;
            for( int k = 0; k < K_; k++ )
            {
                output ^= (input & 1) & (polinom & 1);
                polinom >>= 1; input >>= 1;
            }
        outputs_[i] += output ? "1" : "0";
        }
    }
}

/// конструктор класса
Coder::Coder( int K, vector <int>& pol ) : K_( K ), pol_( pol )
{
    for( int i = 0; i < pol.size(); i++ )
        assert(pol_[i] > 0);

    gen_outputs();
}

/// сверточное кодирование
string Coder::encode( const string& bits )
{
    string encoded;
    int state = 0;

    for( int i = 0; i < bits.size(); i++)
    {
```

```
        char c = bits[i];
        int input = c - '0';
        encoded += outputs_.at( state | ( input << ( K_-1 ) ) );
        state = ( state >> 1 ) | ( input << K_ - 2 );
    }

    for( int i = 0; i < K_ - 1; i++ )
    {
        encoded += outputs_.at( state );
        state = state >> 1;
    }

    return encoded;
}
```

Приложение В

(обязательное)

Текст заголовочного файла decoder.h

```
#include <string>
#include <vector>
#include <algorithm>
#include <limits>
#include <utility>
#include <cassert>
#include <tuple>
#include <cstdint>
#include <math.h>

using namespace std;


class Decoder
{
    /// парсинг образующих полиномов
    /// in: кодовое ограничение
    ///     полином
    /// out: адаптированный полином
    int pars_polinom_bits( int num_bits, int polinom_bits );

    /// расчет всех возможных комбинаций на выходе кодера
    void gen_outputs();

public:
    /// конструктор класса
    /// in: кодовое ограничение
    ///     образующие полиномы
    Decoder( int K, vector <int>& pol );

    /// рассчет расстояния Хэмминга
    /// in: принятая комбинация
    ///     рассчитанная комбинация
    /// out: количество отличающихся позиций
    unsigned int hamming_distance( const string x, const string y
);

    int K_; /// кодовое ограничение
    vector <int> pol_; /// образующие полиномы
    vector <string> outputs_; /// все возможные комбинации на
выходе кодера
};

/// декодер Витерби
class ViterbiDecoder : public Decoder
{
public:
    /// конструктор класса
    /// in: кодовое ограничение
    ///     образующие полиномы
    ViterbiDecoder( int K, vector <int>& pol ) : Decoder( K, pol )
{};
```

```
    /// декодирование по алгоритму Витерби
    /// in: закодированная последовательность бит
    /// out: декодированная последовательность бит
    string do_viterbi_decode( const string& bits );

private:
    /// кодовая решетка
    typedef vector < vector<int> > Trellis;

    /// построение отрезка пути на кодовой решетке
    /// in: принятая кодовая комбинация
    ///     текущее состояние регистра
    ///     следующее состояние регистра
    /// out: расстояние Хэмминга для пути
    int branch_metric( const string& bits,
                       int state,
                       int target_state );

    /// расчет метрики пути
    /// in: принятая кодовая комбинация
    ///     метрики уже построенных путей
    ///     текущее состояние регистра
    /// out: пара <метрика пути, текущее состояние регистра>
    pair <int, int> path_metric( const string& bits,
                                 const std::vector< int >&
old_path_metrics,
                                 int state );

    /// построение пути на кодовой решетке
    /// in: принятая кодовая комбинация
    ///     метрики уже построенных путей
    ///     кодовая решетка
    void update_path_metrics( const string& bits,
                              vector<int>* path_metrics,
                              Trellis* trellis );
};


/// последовательный декодер
class SequentialDecoder : public Decoder
{
public:
    /// конструктор класса
    /// in: кодовое ограничение
    ///     образующие полиномы
    SequentialDecoder( int K, vector <int>& pol );

    /// последовательное декодирование
    /// in: закодированная последовательность бит
    /// out: декодированная последовательность бит
    string do_sequential_decode( const string& bits );

    /// деструктор класса
```

```
    ~SequentialDecoder();

private:
    /// узел кодового дерева
    struct Node
    {
        unsigned short state; /// текущее состояние регистра
        unsigned int previos_node_pos; /// позиция предыдущего
узла в дереве
        unsigned int nodes_num; /// число рассмотренных ветвей,
исходящих из узла
        int metric; /// цена пути
        string decoded_sequence; /// гипотеза
    };

    /// пройденные узлы
    vector <struct Node*> steck_;

    /// исследование дерева
    /// in: принятая кодовая комбинация
    ///     позиция текущего узла в дереве
    void gen_branch( const string& bits, unsigned int pos );

    /// выбор следующего узла в пути
    Node* gen_node( unsigned int metric,
                    unsigned short state,
                    unsigned int pos,
                    const string& c );

/// пороговое декодирование
class TresholdDecoder : public Decoder
{
public:
    /// конструктор класса
    /// in: кодовое ограничение
    ///     образующие полиномы
    TresholdDecoder( int K, vector <int>& pol ) : Decoder( K, pol
){ i_bits_ = ""; p_bits_ = ""; };

    /// пороговое декодирование
    /// in: закодированная последовательность бит
    /// out: декодированная последовательность бит
    string do_treshold_decode(const string& bits );

private:
    string i_bits_; /// принятые информационные биты
    string p_bits_; /// принятые проверочные биты

    /// кодирование информационных битов
    void encode_i_bits();

    /// выделение информационных бит из закодированной
последовательности
```

```
    void get_i_bits( const string& bits );

    /// выделение проверочных бит из закодированной
последовательности
    string get_p_bits( const string& bits );
};
```

Приложение Г

(обязательное)

Текст основного файла программного модуля decoder.cpp

```
#include "decoder.h"
#include <iostream>

/// парсинг образующих полиномов
int Decoder::pars_polinom_bits( int num_bits, int polinom_bits )
{
    int output = 0;
    while( num_bits-- > 0 )
    {
        output = (output << 1) + (polinom_bits & 1);
        polinom_bits >>= 1;
    }

    return output;
}

/// расчет всех возможных комбинаций на выходе кодера
void Decoder::gen_outputs()
{
    outputs_.resize(1 << K_);
    for( int i = 0; i < outputs_.size(); i++ )
    {
        for( int j = 0; j < pol_.size(); j++ )
        {
            int polinom = pars_polinom_bits(K_, pol_[j]);
            int input = i; int output = 0;
            for( int k = 0; k < K_; k++ )
            {
                output ^= (input & 1) & (polinom & 1);
                polinom >>= 1; input >>= 1;
            }
        outputs_[i] += output ? "1" : "0";
        }
    }
}

/// конструктор класса
Decoder::Decoder( int K, vector <int>& pol ) : K_( K ), pol_( pol )
{
    for( int i = 0; i < pol.size(); i++ )
    {
        assert(pol_[i] > 0);
    }

    gen_outputs();
}

/// рассчет расстояния Хэмминга
unsigned int Decoder::hamming_distance( const string x, const string
y )
{
    assert( x.size() == y.size() );
    unsigned short distance = 0;
```

```
    for( int i = 0; i < x.size(); i++ )
    {
        distance += x[i] != y[i];
    }
    return distance;
}

////////////////////////////////////////////
/// декодер Витерби
////////////////////////////////////////////

/// построение отрезка пути на кодовой решетке
int ViterbiDecoder::branch_metric( const string& bits,
                                   int state,
                                   int target_state )
{
    const string output = outputs_.at( state | ( ( target_state >>
( K_ - 2 ) ) << ( K_ - 1 ) ) );
    return hamming_distance( bits, output );
}

/// расчет метрики пути
pair <int, int> ViterbiDecoder::path_metric( const string& bits,
                                             const std::vector< int
>& old_path_metrics,
                                             int state )
{
    int state_0 = (state & ( ( 1 << ( K_ - 2 ) ) - 1 ) ) << 1;
    int state_1 = state_0 | 0;
    int state_2 = state_0 | 1;

    int pm1 = old_path_metrics[state_1];
    {
        pm1 += branch_metric(bits, state_1, state);
    }
    int pm2 = old_path_metrics[state_2];
    {
        pm2 += branch_metric(bits, state_2, state);
    }

    if( pm1 <= pm2 )
    {
        return make_pair( pm1, state_1 );
    }
    else
    {
        return make_pair( pm2, state_2 );
    }
}

/// построение пути на кодовой решетке
void ViterbiDecoder::update_path_metrics(const string& bits,
                                  vector <int>* path_metrics,
```

```
                                    Trellis* trellis)
{
    vector <int> new_path_metrics( path_metrics->size() );
    vector <int> new_trellis_column( 1 << ( K_ - 1 ) );
    for( int i = 0; i < path_metrics->size(); i++ )
    {
        pair<int, int> p = path_metric(bits, *path_metrics, i);
        new_path_metrics[i] = p.first;
        new_trellis_column[i] = p.second;
    }

    *path_metrics = new_path_metrics;
    trellis->push_back(new_trellis_column);
}

/// декодирование по алгоритму Витерби
string ViterbiDecoder::do_viterbi_decode( const string& bits )
{
    Trellis trellis;
    vector <int> path_metrics( 1 << ( K_ - 1 ), 0);
    ///path_metrics.front() = 0;
    for( int i = 0; i < bits.size(); i += pol_.size() )
    {
        string current_bits( bits, i, pol_.size() );
        update_path_metrics( current_bits, &path_metrics, &trellis
);
    }

    string decoded_sequence;
    int     state     =     min_element(     path_metrics.begin(),
path_metrics.end() ) - path_metrics.begin();
    for( int i = trellis.size() - 1; i >= 0; i-- )
    {
        decoded_sequence += state >> ( K_ - 2 ) ? "1" : "0";
        state = trellis[i][state];
    }
    reverse( decoded_sequence.begin(), decoded_sequence.end() );

    return decoded_sequence.substr( 0, decoded_sequence.size() - K_
+ 1 );
}

////////////////////////////////////////////
/// последовательный декодер
////////////////////////////////////////////

/// конструктор класса
SequentialDecoder::SequentialDecoder( int K, vector <int>& pol ) :
Decoder( K, pol )
{
    Node *node = new Node;

    node -> state = 0;
```

```
    node -> previos_node_pos = 0;
    node -> nodes_num = 0;
    node -> metric = 0;
    node -> decoded_sequence = "";

    steck_.push_back( node );
}

/// деструктор класса
SequentialDecoder::~SequentialDecoder()
{
    for( auto it : steck_ )
        delete it;
    steck_.clear();
}

/// рассчет цены пути
int SequentialDecoder::compare( const string s1, const string s2 )
{
    int num = 0;
    for (int i = 0; i < s1.size(); i++)
        if (s1[i] == s2[i]) num++;
        else num -= 2;
    return num;
}

/// выбор следующего узла в пути
SequentialDecoder::Node* SequentialDecoder::gen_node( unsigned int
m, unsigned short state, unsigned int pos, const string& c )
{
    Node *node = new Node;
    node -> previos_node_pos = pos;
    node -> nodes_num = 0;
    node -> state = state;
    node -> metric = ( steck_[pos] -> metric + m);
    node -> decoded_sequence = ( steck_[pos] -> decoded_sequence +
c );

    return(node);
}

/// исследование дерева
void SequentialDecoder::gen_branch( const string& bits, unsigned int
pos )
{
    int state_0 = ( steck_[pos] -> state >> 1 );
    int state_1 = ( ( steck_[pos] -> state >> 1 ) | (1 << ( K_ - 1
) ) );

    int m1 = hamming_distance( bits, outputs_.at( state_0 ) );
    int m2 = hamming_distance( bits, outputs_.at( state_1 ) );

    steck_[pos] -> nodes_num += 1;
```

```
    if( m1 <= m2 )
        if( steck_[pos] -> nodes_num % 2 )
            steck_.push_back( gen_node( m1, state_0, pos, "0" ) );
        else
        {
            steck_.push_back(gen_node(m2, state_1, pos, "1"));
        }
    else if( steck_[pos]->nodes_num % 2 )
        steck_.push_back( gen_node( m2, state_1, pos, "1" ) );
    else
    {
        steck_.push_back(gen_node(m1, state_0, pos, "0"));
    }
}

/// последовательное декодирование
string SequentialDecoder::do_sequential_decode( const string& bits)
{
    int T = 0;
    int max_node = 2;
    vector <pair<int, int>> previosT = { {0, 0} };
    for( int i = 0; i < bits.size(); i += pol_.size() )
    {
        double tau = 0.5 * (steck_[steck_.size() - 1]-
>decoded_sequence).size() * pol_.size() - steck_[steck_.size() - 1]-
>metric;
        if (steck_.size() == 1 || tau >= T )
        {
            if ( steck_.size() != 1 )
            {
                T = tau / 2 * 2;
                if (T > previosT[previosT.size() - 1].first)
                {
                    previosT.push_back({ T, steck_.size() - 1 });
                    max_node = 2;
                }
            }
            string current_bits(bits, i, pol_.size());
            gen_branch( current_bits, steck_.size() - 1 );
        }
        else
        {
            unsigned int pos0 = steck_.size() - 1; int i0 = i -
pol_.size();
            unsigned int pos = steck_[steck_.size() - 1] ->
previos_node_pos;
            i -= pol_.size();
            while( 1 )
            {
                if (steck_[pos]->nodes_num < max_node)
                {
                    string current_bits( bits, i, pol_.size() );
```

```
                    gen_branch( current_bits, pos );
                    tau  =  0.5  *  (steck_[steck_.size()  -  1]-
>decoded_sequence).size() * pol_.size() - steck_[steck_.size() - 1]-
>metric;
                    if (tau >= T)
                    {
                        break;
                    }
                }
                if( previosT[previosT.size() - 1].second > pos )
                {
                    {
                        T = previosT[previosT.size() - 2].first;
                        previosT.pop_back();
                    }
                    max_node += 2;
                    for( vector< Node* >::iterator it = steck_.end()
- 1; it != (steck_.begin() + pos0 + 1); it-- )
                    {
                        delete *it;
                        steck_.pop_back();
                    }
                    i = i0;
                    break;
                }
                pos   =   steck_[pos]->previos_node_pos;   i   -=
pol_.size();
            }
        }
    }
    return (steck_[steck_.size() - 1] -> decoded_sequence).substr(
0, steck_[steck_.size() - 1] -> decoded_sequence.size() - K_ + 1);
}

////////////////////////////////////////////
/// пороговый декодер
////////////////////////////////////////////

///    выделение    информационных    бит    из    закодированной
последовательности
void TresholdDecoder::get_i_bits( const string& bits )
{
    for( int i = 0; i < bits.size() - (K_ - 1)*pol_.size(); i +=
pol_.size() )
        i_bits_ += bits[i];
}

/// выделение проверочных бит из закодированной последовательности
string TresholdDecoder::get_p_bits( const string& bits )
{
    string p_bits = "";
    for( int i = 0; i < bits.size(); i += pol_.size() )
        p_bits += bits.substr( i + 1, pol_.size() - 1 );
```

```
    return p_bits;
}

/// кодирование информационных битов
void TresholdDecoder::encode_i_bits()
{
    string encoded;
    int state = 0;

    for( int i = 0; i < i_bits_.size(); i++)
    {
        char c = i_bits_[i];
        assert( c == '0' || c == '1' );
        int input = c - '0';
        encoded += outputs_.at( state | ( input << ( K_ - 1 ) ) );
        state = ( state >> 1 ) | ( input << K_ - 2 );
    }

    for( int i = 0; i < K_ - 1; i++ )
    {
        encoded += outputs_.at( state );
        state = state >> 1;
    }

    p_bits_ = get_p_bits( encoded );
}

/// пороговое декодирование
string TresholdDecoder::do_treshold_decode( const string& bits )
{
    int sindrom = 0; int num_s = 0;

    for (int i = 0; i < K_; i++)
    {
        for (int j = 1; j < pol_.size(); j++)
        {
            sindrom <<= 1;
            if ((pol_[j] >> i) & 1 == 1)
            {
                sindrom = sindrom | 1;
            }
        }
    }
    for (int i = 0; i < K_ * (pol_.size() - 1); i++)
        if ((sindrom >> i) & 1 == 1)
            num_s++;
    cout << sindrom << "  " << num_s << endl;

    get_i_bits( bits );
    encode_i_bits();
    string chanal_p_bits = get_p_bits( bits );
    int count = 0;
```

```
    for( int i = 0; i < p_bits_.size() - (K_ - 1) * (pol_.size() -
1); i += pol_.size() - 1 )
    {
        string current_bits = ""; string p_current_bits = "";
        string temp1( chanal_p_bits, i, K_ * (pol_.size() - 1) );
        string temp2(p_bits_, i, K_ * (pol_.size() - 1));
        for( int j = 0; j < K_ * (pol_.size() - 1); j++ )
            if ( (sindrom >> j) & 1 == 1 )
            {
                current_bits += temp1[temp1.size() - 1 - j];
                p_current_bits += temp2[temp1.size() - 1 - j];
            }
        if( hamming_distance( current_bits, p_current_bits ) ==
num_s )
        {
            if( i_bits_[count] == '0' ) i_bits_.at(count) = '1';
            else i_bits_[count] = '0';
            encode_i_bits();
        }
        count++;
    }

    return i_bits_;
}
```

Приложение Д

(обязательное)

Руководство разработчика

1 Введение

Данный документ предназначен для программистов, которым необходимо включить в состав разрабатываемых программ кодер и декодер сверточного кода с использованием модулей encoder и decoder.

В документе рассматривается подключение библиотек к проекту и дается описание интерфейсных функций, предоставляемых библиотеками.

Библиотеки написаны на языке C++, все объявления функций приводятся на языке C++.

2 Подключение библиотеки к проекту

Для работы ПО, разработанного с использованием модулей encoder и decoder необходим ПК с одной из поддерживаемых ОС:

- 64-разрядная ОС Windows 7/10;
- 64-разрядные ОС Debian GNU/Linux 9 и выше.

Системные требования определяются ПО верхнего уровня. В общем случае системные требования соответствуют требованиям используемой ОС.

Для начала работы необходимо подключить к проекту заголовочные файлы encoder.h и decoder.h с помощью директивы #include. При этом необходимо убедиться, что все нужные файлы загружены либо в директорию с программой, либо в директорию, заданную через переменную окружения.

3 Стандартная последовательность вызовов

3.1 Сверточное кодирование

Модуль encoder.h предоставляет возможность сверточного кодирования двоичных последовательностей бит со следующими параметрами:

– скорость кодирования – 1/2, 1/3 (рекомендуемые значения, однако ограничения на количество выходных символов не накладываются);

– кодовое ограничение – от 3 до 15.

Порядок действий при реализации сверточного кодирования последовательности бит с помощью модуля encoder.h выглядит следующим образом:

а) объявление и инициализация переменной типа `std::string` для хранения последовательности бит, которую необходимо закодировать (последовательность должна содержать только символы «0» и «1», ограничения на длину последовательности не накладываются);

б) объявление и инициализация переменной типа `int` для задания длины кодового ограничения;

в) объявление и инициализация вектора `std::vector <int>` для задания образующих полиномов (полиномы указываются в виде десятичных чисел, размер вектора должен соответствовать количеству выходных символов за один такт работы кодера);

г) создание объекта класса `Coder`, при этом в конструкторе должны быть переданы длина кодового ограничения и вектор образующих полиномов;

д) вызов метода `encode( const string& bits )`, где `bits` – последовательность бит, которую необходимо закодировать; метод возвращает строку `std:string` с закодированной последовательностью.

Пример реализации описанной последовательности действий:

```
int K = 4; /// задание длины кодового ограничения
std::vector<int> pol; /// вектор под полиномы
pol.push_back(13); /// задание полинома x^3+x^2+1
pol.push_back(15); /// задание полинома x^3+x^2+x+1
/// генерация двоичной последовательности бит
std::string seq;
for( int i = 0; i < 1000; i++ )
    seq += to_string( rand() % 2 );
/// создание объекта класса Coder
Coder coder(K, pol);
/// кодирование двоичной последовательности бит
seq = coder.encode(seq);
```

3.2 Сверточное декодирование

Модуль decoder.h предоставляет возможность декодирования двоичных последовательностей бит по следующим алгоритмам:

– алгоритм порогового декодирования систематического кода;

– алгоритм последовательного декодирования;

– алгоритм декодирования по максимуму правдоподобия (алгоритм Витерби);

Порядок действий при реализации декодирования последовательности бит с помощью модуля decoder.h выглядит следующим образом:

а) создание объекта класса одного или нескольких декодеров (классы `ViterbiDecoder`, `SequentialDecoder`, `TresholdDecoder`) при этом в конструкторе должны быть переданы длина кодового ограничения и вектор образующих полиномов (при подключении порогового декодера первый элемент вектора обязательно должен быть равен 1);

б) вызов метода `do_viterbi_decode(const string& bits)` для декодирования последовательности бит `bits` по алгоритму Витерби, метод возвращает строку с декодированной последовательностью;

в) вызов метода `do_sequential_decode(const string& bits)` для последовательного декодирования последовательности бит `bits`, метод возвращает строку с декодированной последовательностью;

г) вызов метода `do_treshold_decode(const string& bits)` для последовательного декодирования последовательности бит `bits`, метод возвращает строку с декодированной последовательностью;

Пример реализации описанной последовательности действий:

```
/// декодер Витерби
ViterbiDecoder v_decoder(K, pol);
/// последовательный декодер
SequentialDecoder s_decoder(K, pol);
/// пороговый декодер
TresholdDecoder t_decoder(K, pol);
/// декодирование по алгоритму Витерби
std::string seq_1 = v_decoder.do_viterbi_decode(seq);
/// последовательное декодирование
```

```
std::string seq_2 =s_decoder.do_sequential_decode(seq);
/// пороговое декодировние
std::string seq_3 = t_decoder.do_treshold_decode(seq);
```

Приложение Е

(обязательное)

Текст программного модуля sequence.h

```
#include <vector>
#include <string>

using namespace std;

/// генерация последовательности бит
/// in: длина последовательности
/// out: сгенерированная последовательность
string gen_sequence( int length )
{
    string sequence = "";
    for( int i = 0; i < length; i++ )
        sequence += to_string( rand() % 2 );
    return sequence;
}

/// внесение ошибок в последовательность бит
/// in: вероятность битовой ошибки
///     последовательность бит
void imit_chanell(double P, string& s)
{
    int count = 0;
        for( int i = 0; i < s.size(); i++ )
        {
            if( ( (double)rand() / (double)RAND_MAX ) < P )
            {
                if( s[i] == '1' ) s[i] = '0';
                else s[i] = '1';
                count++;
            }
        }
}

/// рассчет вероятности правильного и вероятности неправильного
декодирования
/// in: последовательность бит до кодирования
///     последовательность бит после декодирования
pair < double, double > count_P( string& s1, string& s2 )
{
    double num_1 = 0.0, num_2 = 0.0;
    for( int i = 0; i < s1.size(); i++ )
    {
        if( s1[i] == s2[i] ) num_1++;
        else num_2++;
    }
    pair < double, double > p;
    p = make_pair(num_1 / double(s1.size()), num_2 /
double(s1.size()));
    return p;
}
```

Приложение Ж

(обязательное)

Текст главного модуля main.cpp

```
#include <iostream>
#include <limits>
#include <cstring>
#include <time.h>

#include <fstream>

#include "encoder.h"
#include "decoder.h"
#include "sequences.h"

int main(int argc, char* argv[] )
{
   srand( time(0) );

    vector <int> polinoms;
    int K, n;
    double P;

   K = atoi( argv[2] );
   string s( argv[3] );
   s = s.erase(0,2);
   n = stoi( s );

    for( int i = 1; i <= n; i++ )
       polinoms.push_back( atoi(argv[3 + i]) );

    P = atof( argv[3 + n + 1] );

    string seq_0 = gen_sequence(10000);

    Coder coder(K, polinoms);
    string seq = coder.encode(seq_0);

    imit_chanell(P, seq);

    pair < double, double > p;

    string seq_1;
    string sp(argv[1]);


    if (sp.find("T") != string::npos)
       if (polinoms[0] == 1)
       {
          TresholdDecoder t_decoder(K, polinoms);
          seq_1 = t_decoder.do_treshold_decode(seq);
          p = count_P(seq_0, seq_1);
          cout << "Treshold decoder" << endl;
          cout << "P of correct decoding: " << p.first << endl <<
"P of incorrect decoding: " << p.second << endl << endl;
       }
```

```
    if (sp.find("V") != string::npos)
    {
       ViterbiDecoder v_decoder(K, polinoms);
       seq_1 = v_decoder.do_viterbi_decode(seq);
       p = count_P(seq_0, seq_1);
       cout << "Viterbi decoder" << endl;
       cout << "P of correct decoding: " << p.first << endl << "P
of incorrect decoding: " << p.second << endl << endl;
    }

    if (sp.find("S") != string::npos)
    {
         SequentialDecoder s_decoder(K, polinoms);
         seq_1 = s_decoder.do_sequential_decode(seq);
         p = count_P(seq_0, seq_1);
         cout << "Sequential decoder" << endl;
         cout << "P of correct decoding: " << p.first << endl <<
"P of incorrect decoding: " << p.second << endl << endl;
    }

    return 0;
}
```

Приложение И

(обязательное)

Руководство пользователя

1.1 Введение

Имитационная модель системы передачи данных (далее ИМ) предназначена для проведения статистических исследований помехоустойчивости сверточных кодов при различных помехах в канале связи.

Перед проведением экспериментов на ИМ рекомендуется ознакомиться с теоретическими основами кодирования с использованием сверточных кодов.

1.2 Назначение и условия применения

ИМ предоставляет возможность расчета вероятности правильного приема и вероятности подавления сверточного кода.

Для успешного применения и эффективной эксплуатации ИМ требуется персональный компьютер с 64-разрядной ОС Windows 7/10 или Debian GNU/Linux 9 и выше.

1.3 Начало работы

Для запуска ИМ в Linux необходим эмулятор терминала, в Windows – окно командной строки, а также требуется указать правильный путь к исполняемому файлу программы VKR(.exe), реализующей ИМ.

1.4 Синтаксис команды командной строки

Запуск эксперимента на ИМ:

`./VKR <режим работы> <длина кодового ограничения> <скорость кодирования> <полином 1> … <полином n> <вероятность битовой ошибки>`

Подробное описание аргументов командной строки приведено в таблице И.1.

Таблица И.1 – Описание аргументов командной строки

Аргумент	Описание
`<режим работы>`	Необходимо указать декодеры, участвующие в эксперименте: V – декодер Витерби, S – последовательный декодер, T – пороговый декодер. Возможно комбинирование декодеров. Так, для включения в состав эксперимента всех трех методов декодирования необходимо ввести `VST`.

Продолжение таблицы И.1

Аргумент	Описание
	Прим.: пороговый декодер, реализованный в рамках ИМ, предназначен для декодирования только систематического кода, т.е. аргумент <полином 1> должен быть указан как 1.
<длина кодового ограничения>	Необходимо указать длину кодового ограничения сверточного кода в виде целого числа в десятичной системе счисления. Не рекомендуется вводить значение меньше 3 и больше 15.
<скорость кодирования>	Необходимо указать скорость кодирования сверточного кода в виде дроби 1/n, где n – количество выходных символов за один такт работы кодера. Рекомендуется проводить эксперименты при скоростях 1/2 и 1/3, однако программная реализация ИМ не накладывает ограничений на значение n.
<полином 1> … <полином n>	Необходимо указать образующие полиномы сверточного кода в виде целых чисел в десятичной системе счисления. Прим. 1: количество аргументов командной строки, передающих образующие полиномы, должно соответствовать количеству выходных символов за один такт работы кодера. Прим. 2: количество разрядов в двоичной записи каждого полинома не должно превышать максимальное число разрядов двоичной записи длины кодового ограничения.
<вероятность битовой ошибки>	Необходимо ввести вероятность битовой ошибки в канале связи. Вероятность указывается в долях в диапазоне от 0 до 1.

Прим.: в рамках ИМ сверточное кодирование производится на ПСП длиной 10000 бит.

Пример запуска эксперимента для декодера Витерби, длины кодового ограничения 4, скорости кодирования 1/2, полиномах x^3+x^2+1 и x^3+x^2+x+1, вероятности битовой ошибки 0,1:

```
./VKR V 4 1/2 13 15 0.1
```

Приложение К

(обязательное)

Модульные тесты

```
#include <unittest++/UnitTest++.h>
using namespace std;

/// генерация битовой последовательности
SUITE( bits )
{
    TEST( gen_seq_size )
    {
        string s = gen_sequence( 10000 );
        CHECK_EQUAL( 10000, s.size() );
    }

    TEST( gen_seq_bits )
    {
        string s = gen_sequence(10000);
        int count = 0;
        for( int i = 0; i < s.size(); i++ )
            if( s[i] != '0' && s[i] != '1' )
                count++;

        CHECK_EQUAL( 0, count );
    }

    TEST( imit_chahell_size )
    {
        string s = gen_sequence(10000);
        imit_chanell( 0.5, s );
        CHECK_EQUAL( 10000, s.size() );
    }

    TEST( imit_chahell_bits )
    {
        string s = gen_sequence(10000);
        imit_chanell( 0.5, s );
        int count = 0;
        for( int i = 0; i < s.size(); i++ )
            if( s[i] != '0' && s[i] != '1' )
                count++;

        CHECK_EQUAL( 0, count );
    }
}

/// кодирование
SUITE( encoding )
{
    /// кодирование при K = 3, R = 1/2
    TEST( Enc_K3_R2 )
    {
        vector <int> polinoms;
        polinoms.push_back(7);
        polinoms.push_back(5);
```

```
        Coder coder(3, polinoms);

        string s = "101100101110001";
        string s_enc = "111000010111111000011001110011101l";

        CHECK_EQUAL( s_enc, coder.encode( s ));
    }

    /// кодирование при K = 3, R = 1/3
    TEST( Enc_K3_R3 )
    {
        vector <int> polinoms;
        polinoms.push_back(7);
        polinoms.push_back(7);
        polinoms.push_back(5);

        Coder coder(3, polinoms);

        string s = "101100101110001";
        string                          s_enc                          =
"111110000001001111111100000011100011110001111101l1";

        CHECK_EQUAL( s_enc, coder.encode( s ));
    }

    /// кодирование при K = 7, R = 1/2
    TEST( Enc_K7_R2 )
    {
        vector <int> polinoms;
        polinoms.push_back(109);
        polinoms.push_back(79);

        Coder coder(7, polinoms);

        string s = "101100101110001";
        string                          s_enc                          =
"11010001101011110110011111011000001l001011";

        CHECK_EQUAL( s_enc, coder.encode( s ));
    }

    /// кодирование при K = 7, R = 1/3
    TEST( Enc_K7_R3 )
    {
        vector <int> polinoms;
        polinoms.push_back(109);
        polinoms.push_back(95);
        polinoms.push_back(83);

        Coder coder(7, polinoms);
```

```
        string s = "101100101110001";
        string                          s_enc                           =
"11101100101011010010110001010100111110000111100000111001110011100111";

        CHECK_EQUAL( s_enc, coder.encode( s ));
    }

    /// кодирование при K = 15, R = 1/2
    TEST( Enc_K15_R2 )
    {
        vector <int> polinoms;
        polinoms.push_back(15015);
        polinoms.push_back(20111);

        Coder coder(15, polinoms);

        string s = "101100101110001";
        string                          s_enc                           =
"11110001000010101001011110000101101100010100010000111010011";

        CHECK_EQUAL( s_enc, coder.encode( s ));
    }

    /// кодирование при K = 15, R = 1/3
    TEST( Enc_K15_R3 )
    {
        vector <int> polinoms;
        polinoms.push_back(15015);
        polinoms.push_back(20111);
        polinoms.push_back(32514);

        Coder coder(15, polinoms);

        string s = "101100101110001";
        string                          s_enc                           =
"11011100001100100010010110101001111010100101001010011000001001000
10100010001111011010111";

        CHECK_EQUAL( s_enc, coder.encode( s ));
    }
}

/// декодер Витерби
SUITE( viterbi_decoding )
{
    /// декодирование по алгоритму Витерби при K = 3 R = 1/2
    TEST( viterbi_K3_R2 )
    {
        vector <int> polinoms;
        polinoms.push_back(7);
        polinoms.push_back(5);
```

```
        ViterbiDecoder vDecoder(3, polinoms);

        string s = "101100101110001";
        string s_enc = "11100001011111100001100111001110111";

        CHECK_EQUAL( s, vDecoder.do_viterbi_decode( s_enc ));
    }

    /// декодирование по алгоритму Витерби при K = 3 R = 1/3
    TEST( viterbi_K3_R3 )
    {
        vector <int> polinoms;
        polinoms.push_back(7);
        polinoms.push_back(7);
        polinoms.push_back(5);

        ViterbiDecoder vDecoder(3, polinoms);

        string s = "101100101110001";
        string                          s_enc                          =
"111110000001001111111110000001110001111000111110111";

        CHECK_EQUAL( s, vDecoder.do_viterbi_decode( s_enc ) );
    }

    /// декодирование по флгоритму Витерби при K = 7, R = 1/2
    TEST( viterbi_K7_R2 )
    {
        vector <int> polinoms;
        polinoms.push_back(109);
        polinoms.push_back(79);

        ViterbiDecoder vDecoder(7, polinoms);

        string s = "101100101110001";
        string                          s_enc                          =
"111010001101011110110011111011000011001011";

        CHECK_EQUAL( s, vDecoder.do_viterbi_decode( s_enc ) );
    }

    /// декодирование по методу Витерби при K = 7, R = 1/3
    TEST( viterbi_K7_R3 )
    {
        vector <int> polinoms;
        polinoms.push_back(109);
        polinoms.push_back(95);
        polinoms.push_back(83);

        ViterbiDecoder vDecoder(7, polinoms);

        string s = "101100101110001";
```

```
        string                          s_enc                           =
"11101100101011010010110001010100111110000111100000111001110011111";

        CHECK_EQUAL( s, vDecoder.do_viterbi_decode( s_enc ) );
    }

    /// декодирование по алгоритму Витерби при K = 15, R = 1/2
    TEST( viterbi_K15_R2 )
    {
        vector <int> polinoms;
        polinoms.push_back(15015);
        polinoms.push_back(20111);

        ViterbiDecoder vDecoder(15, polinoms);

        string s = "101100101110001";
        string                          s_enc                           =
"11110001000010101001011110000101101100010100010000111010011";

        CHECK_EQUAL( s, vDecoder.do_viterbi_decode( s_enc ) );
    }

    /// декодирование по алгоритму Витерби при K = 15, R = 1/3
    TEST( viterbi_K15_R3 )
    {
        vector <int> polinoms;
        polinoms.push_back(15015);
        polinoms.push_back(20111);
        polinoms.push_back(32514);

        ViterbiDecoder vDecoder(15, polinoms);

        string s = "101100101110001";
        string                          s_enc                           =
"11011100001100100010010110101001111010100101001010011000001001000
10100010000111101101011";

        CHECK_EQUAL( s, vDecoder.do_viterbi_decode( s_enc ) );
    }
}

/// последовательный декодер
SUITE( sequential_decoding )
{
    /// последовательное декодирование при K = 3 R = 1/2
    TEST( sequential_K3_R2 )
    {
        vector <int> polinoms;
        polinoms.push_back(7);
        polinoms.push_back(5);

        SequentialDecoder sDecoder(3, polinoms);
```

```
        string s = "101100101110001";
        string s_enc = "11100001011111100001100111001110111";

        CHECK_EQUAL( s, sDecoder.do_sequential_decode( s_enc ));
    }

    /// последовательное декодирование при K = 3 R = 1/3
    TEST( sequential_K3_R3 )
    {
        vector <int> polinoms;
        polinoms.push_back(7);
        polinoms.push_back(7);
        polinoms.push_back(5);

        SequentialDecoder sDecoder(3, polinoms);

        string s = "101100101110001";
        string                          s_enc                          =
"11111000000100111111111000000111000111100011111 0111";

        CHECK_EQUAL( s, sDecoder.do_sequential_decode( s_enc ) );
    }

    /// последовательное декодирование при K = 7, R = 1/2
    TEST( sequential_K7_R2 )
    {
        vector <int> polinoms;
        polinoms.push_back(109);
        polinoms.push_back(79);

        SequentialDecoder sDecoder(7, polinoms);

        string s = "101100101110001";
        string                          s_enc                          =
"11101000110101111011001111101100000110010 11";

        CHECK_EQUAL( s, sDecoder.do_sequential_decode( s_enc ) );
    }

    /// последовательное декодирование при K = 7, R = 1/3
    TEST( sequential_K7_R3 )
    {
        vector <int> polinoms;
        polinoms.push_back(109);
        polinoms.push_back(95);
        polinoms.push_back(85);

        SequentialDecoder sDecoder(7, polinoms);

        string s = "101100101110001";
```

```
        string                          s_enc                           =
"11101100101011010010110001010100111110000111100000111001110011111";

        CHECK_EQUAL( s, sDecoder.do_sequential_decode( s_enc ) );
    }

    /// последовательное декодирование при K = 15, R = 1/2
    TEST( sequential_K15_R2 )
    {
        vector <int> polinoms;
        polinoms.push_back(15015);
        polinoms.push_back(20111);

        SequentialDecoder sDecoder(15, polinoms);

        string s = "101100101110001";
        string                          s_enc                           =
"11110001000010101001011110000101101100010100010000111010011";

        CHECK_EQUAL( s, sDecoder.do_sequential_decode( s_enc ) );
    }

    /// последовательное декодирование при K = 15, R = 1/3
    TEST( sequential_K15_R3 )
    {
        vector <int> polinoms;
        polinoms.push_back(15015);
        polinoms.push_back(20111);
        polinoms.push_back(32514);

        SequentialDecoder sDecoder(15, polinoms);

        string s = "101100101110001";
        string                          s_enc                           =
"11011100001100100010010110101001111010100101001010011000001001000
10100010000111101101011";

        CHECK_EQUAL( s, sDecoder.do_sequential_decode( s_enc ) );
    }
}

/// пороговый декодер
SUITE( treshold_decoding )
{
    /// пороговое декодирование при K = 3 R = 1/2
    TEST( treshold_K3_R2 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(7);

        TresholdDecoder tDecoder(3, polinoms);
```

```
        string s = "101100101110001";
        string s_enc = "1101101000011101101011000100110101";

        CHECK_EQUAL( s, tDecoder.do_treshold_decode( s_enc ));
    }

    /// пороговое декодирование при K = 3 R = 1/3
    TEST( treshold_K3_R3 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(5);
        polinoms.push_back(7);

        TresholdDecoder tDecoder(3, polinoms);

        string s = "101100101110001";
        string                          s_enc                          =
"111001100110010011111001100110101010011000111001011";

        CHECK_EQUAL( s, tDecoder.do_treshold_decode( s_enc ) );
    }

    /// пороговое декодирование при K = 7, R = 1/2
    TEST( treshold_K7_R2 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(109);

        TresholdDecoder tDecoder(7, polinoms);

        string s = "101100101110001";
        string                          s_enc                          =
"110010100101110110111001010011000001000101";

        CHECK_EQUAL( s, tDecoder.do_treshold_decode( s_enc ) );
    }

    /// пороговое декодирование при K = 7, R = 1/3
    TEST( treshold_K7_R3 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(95);
        polinoms.push_back(83);

        TresholdDecoder tDecoder(7, polinoms);

        string s = "101100101110001";
```

```
        string                          s_enc                           =
"11101110111001000010100011010110101100000111100000101001100000 11";

        CHECK_EQUAL( s, tDecoder.do_treshold_decode( s_enc ) );
    }

    /// пороговое декодирование при K = 15, R = 1/2
    TEST( treshold_K15_R2 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(20111);

        TresholdDecoder tDecoder(15, polinoms);

        string s = "101100101110001";
        string                          s_enc                           =
"11011011000010001011110100001101000100010100010000010000001";

        CHECK_EQUAL( s, tDecoder.do_treshold_decode( s_enc ) );
    }

    /// пороговое декодирование при K = 15, R = 1/3
    TEST( treshold_K15_R3 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(20111);
        polinoms.push_back(15015);

        TresholdDecoder tDecoder(15, polinoms);

        string s = "101100101110001";
        string                          s_enc                           =
"11101110010100000011001011010110101101000010100101001100000100100
00010000000110100100001";

        CHECK_EQUAL( s, tDecoder.do_treshold_decode( s_enc ) );
    }
}

/// декодирование последовательностей с ошибками
SUITE( P_decoding )
{
    /// декодирование при K=4, R=1/2, P=0.1
    TEST( dec2_10 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(10);

        string s_enc = "10100101110100011000000101";
```

```
        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "1100100010",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "1100100010",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "1100100010",   tDecoder.do_treshold_decode(
s_enc ) );
    }

    /// декодирование при K=4, R=1/3, P=0.1
    TEST( dec3_10 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(7);
        polinoms.push_back(10);

        string s_enc = "110101000011111011010001110011010011000";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "1100100010",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "1100100010",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "1100100010",   tDecoder.do_treshold_decode(
s_enc ) );
    }

    /// декодирование при K=4, R=1/2, P=0.2
    TEST( dec2_20 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(10);

        string s_enc = "11110101110000011011000110";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "1100100010",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "1100100011",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "1100100011",   tDecoder.do_treshold_decode(
s_enc ) );
    }
```

```
    /// декодирование при K=4, R=1/3, P=0.2
    TEST( dec3_20 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(7);
        polinoms.push_back(10);

        string s_enc = "1101010000011110000100111100101101010000";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "1100100010",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "1100100010",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "1100100010",   tDecoder.do_treshold_decode(
s_enc ) );
    }

    /// декодирование при K=4, R=1/2, P=0.3
    TEST( dec2_30 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(10);

        string s_enc = "11100111110101000000100100";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "1001100000",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "1101100000",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "1101100000",   tDecoder.do_treshold_decode(
s_enc ) );
    }

    /// декодирование при K=4, R=1/3, P=0.3
    TEST( dec3_30 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(7);
        polinoms.push_back(10);

        string s_enc = "1101010000111101110110111011011101001";
```

```
        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "1100100010",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "1100100010",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "1100110010",   tDecoder.do_treshold_decode(
s_enc ) );
    }

    /// декодирование при K=4, R=1/2, P=0.4
    TEST( dec2_40 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(10);

        string s_enc = "1111110110001001100010110 0";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "1000001010",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "1110101010",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "1110101010",   tDecoder.do_treshold_decode(
s_enc ) );
    }

    /// декодирование при K=4, R=1/3, P=0.4
    TEST( dec3_40 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(7);
        polinoms.push_back(10);

        string s_enc = "110011001011011010010110100110110101101";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "1100000111",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "1001000111",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "1010000111",   tDecoder.do_treshold_decode(
s_enc ) );
    }
```

```
    /// декодирование при K=4, R=1/2, P=0.5
    TEST( dec2_50 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(10);

        string s_enc = "10001111111001011111110010";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "0011110011",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "1011110011",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "0011110011",   tDecoder.do_treshold_decode(
s_enc ) );
    }

    /// декодирование при K=4, R=1/3, P=0.5
    TEST( dec3_50 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(7);
        polinoms.push_back(10);

        string s_enc = "10011000111001111110110111000011011101";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "0100011110",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "0100011000",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "1100011110",   tDecoder.do_treshold_decode(
s_enc ) );
    }

    /// декодирование при K=4, R=1/2, P=0.6
    TEST( dec2_60 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(10);

        string s_enc = "01110000100000110111110011";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
```

```
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "0000100111",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "0100100101",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "1000100101",   tDecoder.do_treshold_decode(
s_enc ) );
    }

    /// декодирование при K=4, R=1/3, P=0.6
    TEST( dec3_60 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(7);
        polinoms.push_back(10);

        string s_enc = "001111011100010000101100000100100100010";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "0101000000",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "0101000000",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "0101001101",   tDecoder.do_treshold_decode(
s_enc ) );
    }

    /// декодирование при K=4, R=1/2, P=0.7
    TEST( dec2_70 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(10);

        string s_enc = "00011010101001111010111011";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "0010110110",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "0011110111",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "0111110101",   tDecoder.do_treshold_decode(
s_enc ) );
    }

    /// декодирование при K=4, R=1/3, P=0.7
    TEST( dec3_70 )
```

```
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(7);
        polinoms.push_back(10);

        string s_enc = "00101011100100110110010000111001111011111";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "0011010001",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "0010000001",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "0010110101",   tDecoder.do_treshold_decode(
s_enc ) );
    }

    /// декодирование при K=4, R=1/2, P=0.8
    TEST( dec2_80 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(10);

        string s_enc = "00101000011011000110111011";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "0100010001",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "0110011001",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "0100011001",   tDecoder.do_treshold_decode(
s_enc ) );
    }

    /// декодирование при K=4, R=1/3, P=0.8
    TEST( dec3_80 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(7);
        polinoms.push_back(10);

        string s_enc = "00101001000010011011001000110010010001";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
```

```
        CHECK_EQUAL(   "0000010000",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "0000010000",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "0000111101",   tDecoder.do_treshold_decode(
s_enc ) );
    }

    /// декодирование при K=4, R=1/2, P=0.9
    TEST( dec2_90 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(10);

        string s_enc = "01001010101011100110111111";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "0010111001",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "0011111101",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "0010111001",   tDecoder.do_treshold_decode(
s_enc ) );
    }

    /// декодирование при K=4, R=1/3, P=0.9
    TEST( dec3_90 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(7);
        polinoms.push_back(10);

        string s_enc = "001110110100000101111110001100110110111";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "0101100101",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "0101010000",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "0111011101",   tDecoder.do_treshold_decode(
s_enc ) );
    }

    /// декодирование при K=4, R=1/2, P=1
    TEST( dec2_100 )
    {
```

```
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(10);

        string s_enc = "01001010001011100110111011";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "0000010001",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "0001001101",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "0001101101",   tDecoder.do_treshold_decode(
s_enc ) );
    }

    /// декодирование при K=4, R=1/3, P=1
    TEST( dec3_100 )
    {
        vector <int> polinoms;
        polinoms.push_back(1);
        polinoms.push_back(7);
        polinoms.push_back(10);

        string s_enc = "00101011010000010010111000110010101110111";

        ViterbiDecoder vDecoder( 4, polinoms );
        SequentialDecoder sDecoder( 4, polinoms );
        TresholdDecoder tDecoder( 4, polinoms );
        CHECK_EQUAL(   "0000000110",   vDecoder.do_viterbi_decode(
s_enc ) );
        CHECK_EQUAL(  "0011011101",  sDecoder.do_sequential_decode(
s_enc ) );
        CHECK_EQUAL(   "0011011101",   tDecoder.do_treshold_decode(
s_enc ) );
    }
}

int main(int argc, char **argv)
{
    return UnitTest::RunAllTests();
}
```