УДК 004.421.5

ПОГРАММНАЯ РЕАЛИЗАЦИЯ КОДИРОВАНИЯ И ДЕКОДИРОВАНИЯ СООБЩЕНИЙ С ИСПОЛЬЗОВАНИЕМ КОДА РИДА-СОЛОМОНА

М.В. Ковтун
Студент группы Б21-503 НИЯУ МИФИ, frostohelp@ya.ru

Аннотация. Рассматривается помехоустойчивое кодирование как средство повышения надёжности передачи данных по каналам связи. Проводится сравнение характеристик блочных и непрерывных корректирующих кодов и описана целесообразность их комбинации в разных задачах. Реализовано кодирование и декодирование информации кодом Рида-Соломона с исправлением ошибок на языке Python. Применение схожих свойств БЧХ-кодов позволяет достичь лучшей производительности и простоты реализации (за счет синдромного декодирования). Экспериментально доказано, что при увеличении числа проверочных символов возрастают требования к вычислительной мощности, а время декодирования значительно превышает время кодирования.

Ключевые слова: помехоустойчивое кодирование, код Рида-Соломона, сверточный код, Python, исправление ошибок.

Введение

В системах передачи информации качественной характеристикой выступает надежность – способность системы гарантировать передачу данных с допустимыми ошибками. При этом кабельные и беспроводные линии передачи информации подвержены внешним помехам различных физических свойств (в виде шорохов, тресков, неразборчивости речи абонентов и появления звуков с других каналов), что приводит к неоднозначному распознаванию информации на стороне приемника. Магнитные и оптические носители информации часто в процессе использования приобретают физические повреждения, что делает невозможным считывание информации с отдельных участков поверхности носителя. В такой ситуации актуальным становится применение специальных технологий, позволяющих обнаруживать и исправлять ошибки, появляющиеся при воздействии помех.

Классифицируя методы защиты от помех, можно выделить технические методы (экранирование, заземление, фильтрация, изоляция и т.д.) и методы защиты на основе использования помехоустойчивого кодирования.

Кодирование позволяет согласовать формат сообщения с конкретным каналом связи или другим устройством, предназначенным для преобразования или хранения информации. Рассмотрим типовую структурную схему системы передачи дискретной информации (СПДИ) (рис. 1): сигнал формируется в источнике информации, поступает на вход кодера канала, где происходит кодирование с целью улучшения качества передачи сигнала путем повышения помехоустойчивости системы. Далее кодированный сигнал модулируется, проходит через зашумленный канал, где происходят некоторые

искажения сигнала, после чего демодулируется и поступает в декодер, где сообщение восстанавливается. Кодер с декодером образуют кодек источника – устройство или программное обеспечение, которое кодирует и декодирует информацию. Для его построения необходимы регистры сдвига, логические элементы и ключи [1].

Повышение скорости передачи информации в реальных СПДИ приводит к снижению помехоустойчивости и достоверности передачи.

Помехоустойчивое кодирование осуществляется путем введения в передаваемое кодовое слово достаточного количества избыточной информации (например, в виде проверочных символов). В реальных ситуациях длина кода ограничена допустимой сложностью устройств кодирования и декодирования, поэтому эффективность использования корректирующих кодов зависит от параметров кода и ограничений реализации канального кодека.

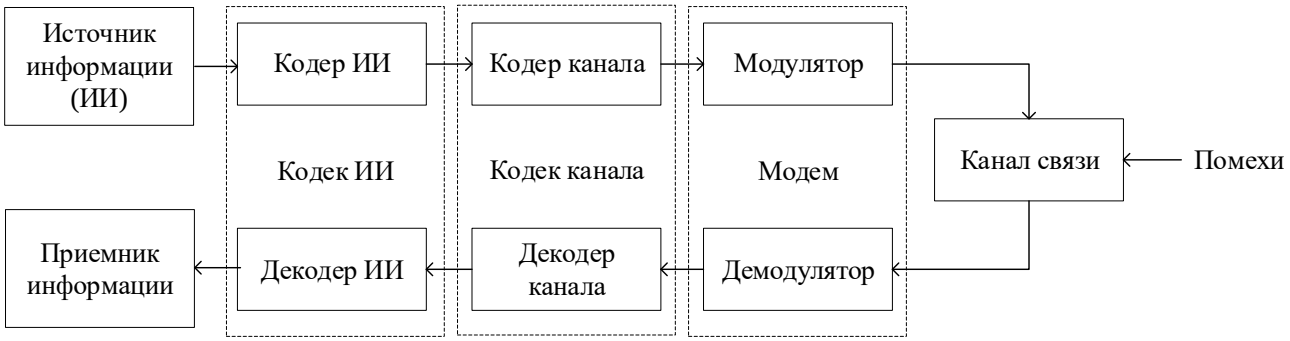

Рисунок 1 – Структурная схема СПДИ.

Кодирование включает в себя процесс преобразования сообщения (информационного слова) в кодовое слово. Каждому сообщению присваивается определенный набор кодовых символов, образующих кодовое слово. Набор кодовых слов, представляющих отдельные закодированные сообщения, образует код. К основным параметрам помехоустойчивых кодов относятся:

n – разрядность (количество символов) кодового слова (1),

k – разрядность (количество символов) информационного слова,

t – количество проверочных символов,

d – кодовое расстояние (минимальное число элементов, которыми любое кодовое слово отличается от другого по всем парам кодовых слов),

w – вес кода (количество ненулевых символов в кодовой комбинации).

$$n = k + t \qquad (1)$$

Существует два класса помехоустойчивых кодов: блочные коды и непрерывные коды.

Блочный код делит передаваемую информационную последовательность на отдельные блоки и добавляет к каждому блоку фиксированное количество

проверочных символов. Блоки кодируются и декодируются независимо друг от друга. Оптимальным линейным блочным кодом является недвоичный код Рида-Соломона, который имеет больше степеней свободы и может достигать максимального кодового расстояния (2):

$$d_{max} = t + 1 \qquad (2)$$

В непрерывных кодах, также известных как цепные, рекуррентные или сверточные, передаваемая информационная последовательность не разбивается на блоки, а проверочные символы размещаются между информационными в определенном порядке. Процессы кодирования и декодирования также выполняются в непрерывном режиме.

Поскольку коды Рида-Соломона и сверточные коды часто применяются при коррекции ошибок, сравним их основные характеристики (табл. 1) [2].

Каждый из этих кодов имеет свои достоинства и недостатки. Например, использование сверточных кодов приводит к меньшему расширению полосы пропускания, а коды Рида-Соломона могут указывать на наличие неисправимых ошибок. В блочных кодах в отличие от сверточных кодов нет памяти, кодирующее устройство по определенным правилам вносит избыточность. Поэтому на практике часто используется их комбинация: для повышения производительности в качестве внутреннего кода используется сверточный код, а в качестве внешнего – код Рида-Соломона.

Для исследования и демонстрации процессов кодирования и декодирования кодом Рида-Соломона была выполнена программная реализация на языке Python.

Коды Рида – Соломона – недвоичные блочные циклические коды (элементами кодового слова являются не биты (двоичные символы), а группы битов (недвоичные символы), например, байты), позволяющие исправлять ошибки в блоках данных.

В общем случае код Рида-Соломона определяется как RS (n, k) s-битных символов (кодер воспринимает k информационных символов по s битов каждый и добавляет проверочные символы для формирования n символов кодового слова).

Коды Рида-Соломона являются частным случаем кодов Боуза-Чоудхури-Хоквингема (БЧХ-кодов), корни порождающего полинома которого лежат в том же поле, над которым строится код, поэтому при реализации кода Рида-Соломона будем использовать логику БЧХ-кода, что позволит достичь лучшей производительности и простоты реализации [3].

Рассмотрим преимущества использования данной логики.

Оригинальная реализация кода Рида-Соломона включает полиномиальную интерполяцию и может требовать выбора наиболее популярного декодирования из

всех возможных. Это делает алгоритм сложным и неэффективным для реализации, особенно для больших значений n и k.

Таблица 1 – Сравнение кодов Рида-Соломона и сверточных кодов.

Основные характеристики	Коды Рида-Соломона (РС)	Сверточные коды (СК)
Тип	Блоковые коды	Последовательные коды
Быстродействие	Зависит от алгоритма декодирования, требует больше вычислительных ресурсов по сравнению с простыми алгоритмами СК	Обычно быстрее при декодировании по сравнению с кодом РС благодаря простым и быстрым алгоритмам (например, алгоритма Витерби)
Коррекция ошибок	Способны корректировать множественные ошибки и потерю данных, эффективны в сценариях с пакетами ошибок	Эффективны для исправления случайных ошибок, особенно в системах реального времени
Структура	Работают с фиксированным размером блоков данных. Представляют данные в виде полиномов над конечным полем	Работают с непрерывным потоком данных. Используют регистры сдвига и логические операции
Применение	Используются в запоминающих устройствах, беспроводной или мобильной связи, спутниковой связи, цифровом телевидении (DVB), высокоскоростных модемах (ADSL, xDSL, и т.д.)	Используются в мобильной связи, спутниковых коммуникациях и в других системах, где важна низкая задержка
Преимущества	Высокая способность к коррекции множества ошибок. Эффективны при работе с пакетами ошибок	Более высокая скорость декодирования. Меньшая задержка при обработке данных. Подходят для систем реального времени
Недостатки	Высокая вычислительная сложность при декодировании. Меньшая скорость декодирования по сравнению с некоторыми другими типами кодов	Менее эффективны при коррекции пакетов ошибок по сравнению с кодами РС. Обычно требуют больше памяти для реализации декодера

Логика БЧХ-кода предоставляет более простой и эффективный метод декодирования, использует синдромное декодирование, которое позволяет обнаруживать и исправлять ошибки, основываясь на вычисленных синдромах, что значительно упрощает алгоритм. Таким образом, можно быстро определять ошибки и их позиции, что делает кодирование и декодирование более практичными для реальных приложений.

Коды Рида-Соломона и BCH-коды являются подклассами циклических кодов и имеют схожие математические основы. Это позволяет использовать методы и алгоритмы, разработанные для одного подкласса, для другого. При этом можно сохранить преимущества обеих систем.

В реальных системах, таких как цифровое телевидение, DVD и компакт-диски, часто используется комбинация этих методов для обеспечения надежной коррекции ошибок при разумных вычислительных затратах.

Рассмотрим этапы кодирования (работаем в поле Галуа $\mathrm{GF}(2^8)$):

Шаг 1. Задаём примитивный полином (чтобы определить операции в $\mathrm{GF}(2^8)$), а также количество проверочных символов $t = 6$.

Примитивный полином (3) в шестнадцатеричной системе:

$$z^8 + z^6 + z^5 + z^4 + 1 = 0\text{x}171 \qquad (3)$$

Шаг 2. Генерируем порождающий полином $g(x)$ (4) (порождающий элемент $\alpha = 02$)

$$g(x) = \prod_{j=1}^{t}\left(x - \alpha^j\right) =$$

$$= 01x^6 + 7\text{E}x^5 + 36x^4 + \text{A}3x^3 + \text{BE}x^2 + 49x + 1\text{A} \qquad (4)$$

Функции для поиска корней и создания порождающего полинома представлены в классе class ReedSolomon (рис. 2), логика функционирования поля описана в классе class G2_8 (рис. 3).

```
def generator_poly(self):
    g = [1]
    for root in self._generator_roots():
        g = GF2_8.poly_mul(g, [1, root])
    return g
```

```
def _generator_roots(self):
    for i in range(1, self._t + 1):
        yield GF2_8.EXP[i]
```

Рисунок 2 – class ReedSolomon.

```
@staticmethod
def _init_class():
    GF2_8.EXP = [0] * (GF2_8.ORDER * 2)
    GF2_8.LOG = [0] * GF2_8.SIZE

    x = 1
    for i in range(GF2_8.ORDER):
        GF2_8.EXP[i] = x
        GF2_8.LOG[x] = i
        x <<= 1
        if x & GF2_8.SIZE:
            x ^= GF2_8.PRIM

    for i in range(GF2_8.ORDER, GF2_8.ORDER * 2):
        GF2_8.EXP[i] = GF2_8.EXP[i - GF2_8.ORDER]
```

Рисунок 3 – class G2_8.

Шаг 3. Задаём сообщение или в текстовом виде (рис. 4, *a*), или как массив целых положительных чисел (рис. 4, *b*). Размер сообщений не должен превышать $n = k + t$.

```
message = 'Reed-Solomon'
bmessage = message.encode('utf-8')
encoded_message = rs.encode(bmessage)
```

a

```
length = random_int(0, 256 - rs._t)
input_data = np.random.randint(0, 255, length).astype(np.uint8)
```

b

Рисунок 4 – Формирование сообщений:
a – в текстовом виде, *b* – генерацией массива.

Шаг 4. Производим кодирование (рис. 5), для чего:

– определим полином (5), коэффициенты которого являются элементами сообщения:

$$p(x) = \sum_{j=1}^{k} a_j x^{j-1} = a_k x^{k-1} + \cdots + a_2 x + a_1 \qquad (5)$$

– сдвинем $p(x)$ на t символов влево, чтобы освободить место под проверочные символы (6):

$$p(x) \cdot x^t \qquad (6)$$

– найдём остаток от деления на $g(x)$ (7):

$$s_r(x) = p(x) \cdot x^t \bmod g(x) \qquad (7)$$

– вычислим закодированное сообщение (8):

$$s(x) = p(x) \cdot x^t - s_r(x) \qquad (8)$$

```
class ReedSolomon:
    def __init__(self, t):
        self._t = t
        self._generator_polynomial = self.generator_poly()

    def encode(self, msg):
        p_shifted = list(msg) + [0] * self._t
        _, sR = GF2_8.poly_div(p_shifted, self._generator_polynomial)
        s = GF2_8.poly_sub(p_shifted, sR)
        return s
```

Рисунок 5 – Кодирование.

Шаг 5. Вносим ошибки явно (рис. 6, *a*) или случайным образом (рис. 6, *b*).

```
# Introduce errors
erroneous_message = encoded_message
erroneous_message[0] = ord('A')
erroneous_message[1] = ord('b')
```

a

```
for _ in range(num_errors):
    encoded[random_int(0, len(encoded) - 1)] = \
    random_int(0, 255)
```

b

Рисунок 6 – Внесение ошибок: *a* – явно, *b* – случайным образом.

Шаг 6. Восстановление полученного сообщения.

Получатель имеет сообщение вида (9), состоящее из отправленного сообщения и ошибок:

$$r(x) = sm(x) + e(x) \qquad (9)$$

Вычисляем синдром *s* и определяем, есть ли ненулевые разряды (рис. 7):

```
def syndromes(self, r):
    return [GF2_8.poly_eval(r, root) for root in self._generator_roots()]
```

```
is_valid_codeword = all(s == 0 for s in syndromes)
if is_valid_codeword:
    return r
```

Рисунок 7 – Вычисление синдрома.

Вычисляем локатор ошибок и количество ошибок (10) (переменная *l* в коде) (рис. 8):

$$\Lambda(x) = \prod_{k=1}^{\nu}(1 - xX_k)\text{, где } X_k = \alpha^{i_k} \qquad (10)$$

Вычисляем (11) позиции ошибок (рис. 9):

$$i_k = \log_\alpha(\alpha^{i_k}) = \log_\alpha(X_k) \qquad (11)$$

```
def error_locator(self, syndromes):
    err_loc = [1]
    old_loc = [1]
    l = 0

    for i in range(self._t):
        delta = GF2_8.poly_mul_at(err_loc, syndromes, i)
        old_loc.append(0)

        if delta != 0:
            if 2 * l <= i:
                new_loc = GF2_8.poly_sub(err_loc, GF2_8.poly_scale(old_loc, delta))
                old_loc = GF2_8.poly_scale(err_loc, GF2_8.div(1, delta))
                err_loc = new_loc
                l = i + 1 - l
            else:
                err_loc = GF2_8.poly_sub(err_loc, GF2_8.poly_scale(old_loc, delta))
    return err_loc[-(l + 1):]
```

Рисунок 8 – Вычисление локатора ошибок.

```
def error_positions(self, err_loc):
    err_pos = []
    for i in range(GF2_8.SIZE):
        if GF2_8.poly_eval(err_loc, i) == 0:
            err_pos.append(GF2_8.LOG[GF2_8.div(1, i)])
    return err_pos
```

Рисунок 9 – Вычисление позиций ошибок.

Определяем корректность позиций ошибок (рис. 10): если мы не нашли ν разных корней или если эти позиции находятся вне границ нашего сообщения, то сообщение передано больше, чем с $t/2$ ошибками, из-за чего восстановить его невозможно.

```
if not self.error_positions_valid(err_pos, err_loc, r):
    raise ReedSolomonException('Could not decode message.')
```

```
def error_positions_valid(self, err_pos, err_loc, r):
    if len(err_loc) - 1 != len(err_pos):
        return False
    return max(err_pos) < len(r)
```

Рисунок 10 – Вычисление позиций ошибок.

Используя алгоритм Форни, получим значения ошибок:

– найдем полиномиальный вычислитель (12) ошибок (рис. 11)

$$\Omega(x) = S(x) * \Lambda(\mathrm{x}) \qquad (12)$$

```
def error_evaluator(self, syndromes, err_loc):
    syndromes = syndromes[::-1]
    mul = GF2_8.poly_mul(syndromes, err_loc)
    syndromes = syndromes[::-1]
    return mul[-self._t:]
```

Рисунок 11 – Нахождение вычислителя ошибок.

– вычислим значения ошибок e_j (13), где $\Lambda(x)$ – локатор ошибок, рассчитанный ранее (рис. 12):

$$e_j = -\frac{\Omega(X_j^{-1})}{\Lambda'(X_j^{-1})} \qquad (13)$$

```
def error_polynomial(self, syndromes, err_loc, err_pos):
    err_eval = self.error_evaluator(syndromes, err_loc)
    err_loc_deriv = GF2_8.poly_deriv(err_loc)
    error_polynomial = [0] * (max(err_pos) + 1)

    for pos in err_pos:
        x_inv = GF2_8.div(1, GF2_8.EXP[pos])
        n = GF2_8.poly_eval(err_eval, x_inv)
        d = GF2_8.poly_eval(err_loc_deriv, x_inv)
        magnitude = GF2_8.div(n, d)
        error_polynomial[len(error_polynomial) - pos - 1] = magnitude
    return error_polynomial
```

Рисунок 12 – Нахождение значений ошибок.

Восстанавливаем исходное сообщение и проверяем, что восстановили корректно, т.е. количество ошибок было не более t/2 и все разряды синдрома равны 0 (рис. 13).

$$s'(x) = r(x) - e(x) \qquad (14)$$

```
e = self.error_polynomial(syndromes, err_loc, err_pos)
repaired = GF2_8.poly_sub(r, e)

if not self.is_valid_codeword(repaired):
    raise ReedSolomonException('Could not decode message.')
```

Рисунок 13 – Восстановление сообщения.

Шаг 7. Декодируем сообщение (рис. 14).

```
def decode(self, r):
    repaired = self.repair(r)
    decoded = self.remove_check_symbols(repaired)
    return decoded
```

```
def remove_check_symbols(self, s):
    return s[:len(s) - self._t]
```

Рисунок 14 – Декодирование сообщения.

Результат тестирования работы программы показывает, что сообщения восстанавливаются верно при наличии не более двух ошибок при количестве проверочных символов $t = 6$ (рис. 15).

```
Encoded:   [113, 42, 35, 55, 215, 180, 42, 213, 7, 245, 67, 71, 110, 91, 206]
Erroneous: [113, 42, 35, 55, 21, 180, 42, 213, 7, 245, 67, 71, 110, 91, 206]
Decoded:   [113, 42, 35, 55, 215, 180, 42, 213, 7]
******************************
Encoded:   [159, 95, 150, 144, 28, 123, 25, 14, 165, 9]
Erroneous: [159, 95, 150, 144, 28, 123, 25, 14, 165, 9]
Decoded:   [159, 95, 150, 144]
******************************
Encoded:   [134, 183, 213, 86, 224, 232, 166, 180, 234]
Erroneous: [134, 183, 213, 85, 224, 232, 166, 180, 3]
Decoded:   [134, 183, 213]
```

Рисунок 15 – Пример работы программы.

На следующем этапе было проведено тестирование с целью анализа зависимости времени работы программы от числа проверочных символов при фиксированном количестве информационных символов (табл. 2). На рис. 16 представлена зависимость среднего времени декодирования одного сообщения (AVG DEC TIME 1) от t.

Таблица 2 – Результаты экспериментов при заданных параметрах для кодирования и декодирования сообщений.

$k = 100$						
tests | errors	t	n	ENC TIME, c	DEC TIME, c	ENC TIME 1, c	DEC TIME 1, c
5000 | 3	6	106	3,818088	16,71241	0,0007636	0,0033425
5000 | 3	10	110	5,204664	25,04663	0,0010409	0,0050093
5000 | 3	20	120	8,39976	45,91281	0,00168	0,0091826
5000 | 3	50	150	18,24146	119,9818	0,0036483	0,0239964
5000 | 3	100	200	34,59369	278,108	0,0069187	0,0556216

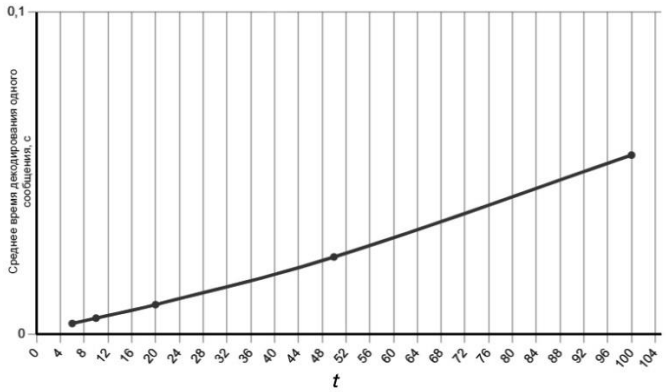

Рисунок 16 – График зависимости времени декодирования от t.

Заключение

Объем вычислительной мощности, необходимой для кодирования и декодирования для кодов Рида-Соломона при фиксированной длине сообщений, зависит от числа t. Большое значение t означает, что большее число ошибок может

быть исправлено, но это потребует большей вычислительной мощности по сравнению с вариантом при меньшем *t*.

Полученное время декодирования сообщений превышает время кодирования от четырех (при малых *t*) до восьми раз (при больших *t*).

Таким образом применение кодов Рида-Соломона для исправления ошибок делает задачу построения эффективных алгоритмов декодирования этих кодов весьма актуальной. В настоящее время реализуется множество различных алгоритмов декодирования кода Рида-Соломона, однако время декодирования все еще остается большим.

Список литературы

1. Пуговкин А. В. Основы построения инфокоммуникационных систем и сетей: учебное пособие. Томский Государственный университет систем управления и радиоэлектроники (ТУСУР). – Томск : Эль Контент, 2014. – 156 с.
2. TM Synchronization and Channel Coding, Recommendation for Space Data System Standards CCSDS 131.0-B-1, Issue 1, Blue Book, Consultative Committee for Space Data Systems, September, 2023. [Электронный ресурс]. https://public.ccsds.org/Pubs/131x0b5.pdf (Дата обращения: 08.08.2024).
3. Reed-Solomon error correction. [Электронный ресурс]. https://sigh.github.io/reed-solomon/ (Дата обращения: 08.08.2024).