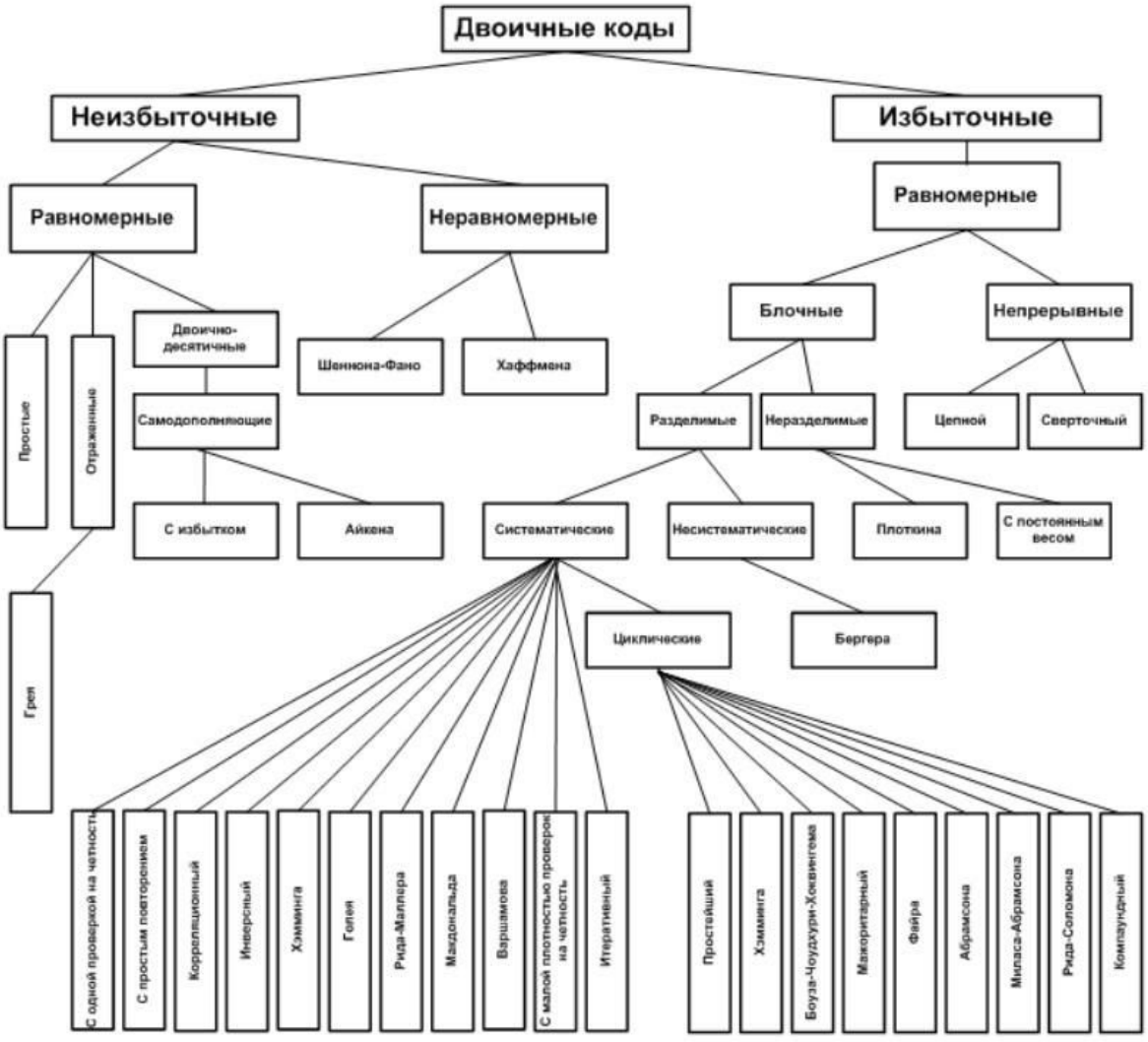
Коды Рида-Соломона для исправления ошибок.

Введение

Код Рида — Соломона был изобретён в 1960 году сотрудниками лаборатории Линкольна Массачусетского технологического института Ирвингом Ридоми Густавом Соломоном. Идея использования этого кода была представлена в статье «Polynomial Codes over Certain Finite Fields». Эффективные алгоритмы декодирования были предложены в 1969 году Элвином Берлекэмпом и Джэймсом Месси (алгоритм Берлекэмпа — Мэсси) и в 1977 году Давидом Мандельбаумом (метод, использующий Алгоритм Евклида). Первое широкое применение код Рида — Соломона получил в 1982 году в серийном выпуске компакт-дисков. [1]

Стоит отметить, что код Рида-Соломона относится к избыточным равномерным блочным разделимым систематическим циклическим кодам.



Теоретическая часть

Код Рида-Соломона (РС) будет иметь расстояние Хэмминга $D = n - k + 1$. (Расстояние Хэмминга - это минимальное число различных позиций между двумя разными кодовыми словами). В соответствии с теорией кодирования, код, имеющий расстояние Хэмминга $D = 2t + 1$, позволяет исправлять t ошибок.

Код Рида — Соломона над $GF(q^m)$, исправляющий t ошибок, требует 2t проверочных символов, и с его помощью исправляются произвольные комбинации ошибок длиной t и меньше. Данный код является оптимальным, потому что для него достигается граница Синглтона ($D \leq n - k + 1$). Поэтому данный код относится к кодам с максимально достижимым расстоянием (МДР).

Код Рида — Соломона над полем $GF(q^m)$ с кодовым расстоянием $D = 2t + 1$ можно рассматривать как $((q^m - 1)m, (q^m - 1)m)$-код над полем $GF(q)$, который может исправлять любую комбинацию ошибок, сосредоточенную в t или меньшем числе блоков из m символов. Наибольшее число блоков длины m, которые может затронуть пакет длины l_i, где $l_i \leq mt_i - (m - 1)$, не превосходит t_i, поэтому код, который может исправить t блоков ошибок, всегда может исправить и любую комбинацию из p пакетов общей длины l, если $l + (m - 1) \leq mt$. [1]

Кодирование

Мы рассмотрим простые коды Рида-Соломона. Все операции будут производиться над полями простого порядка - аналогичные действия можно проводить и над полями с примарным числом элементов.

Последовательность, которую мы хотим закодировать, является вектором. Любой вектор размера n можно представить в виде полинома степени n-1. Так как элементы вектора являются символами некоторого конечного алфавита, то мы задали многочлен над конечным полем (это накладывает ограничения на размеры алфавита – он может быть только степенью простого числа).

Данный полином можно представить как функцию. Из курса математического анализа мы знаем, что любую функцию можно разложить на набор других функций с помощью преобразования Фурье. В случае конечного поля нам нужно использовать дискретное преобразование Фурье над конечным полем. Это один из видов дискретного преобразования Фурье для вектора $\vec{v} = (v_0, v_1, \ldots, v_{n-1})$, $v_i \in GF(q^m)$ над конечным полем $GF(q^m)$, определяемое как вектор $\vec{V} = (V_0, V_1, \ldots, V_{n-1})$, $V_i \in GF(q^m)$, где n делит $q^m - 1$ (это нужно для существования n корней из единицы над данным полем $GF(q^m)$) при некотором целом положительном m, с компонентами, вычисляемыми как

$$V_j = \sum_{i=0}^{n-1} \alpha^{ij} v_i, \quad j = 0, 1, \ldots, n-1.$$

где α — элемент порядка nв поле $GF(q^m)$ (то есть такой, что $\alpha^n = 1$, $\alpha^k \neq 1$, $k < n$).

Индекс i можно назвать временем, а $\vec{v}$ — временной функцией или сигналом. Аналогично индекс j — частотой, а $\vec{V}$ — частотной функцией или спектром. [2]

Приближенно представляя функцию набором значений в n равноотстоящих точках, мы приближаем функцию конечным рядом Фурье. Поэтому элемент α должен быть примитивным (возводя такой элемент в определенную степень мы можем получить любой элемент, кроме нулевого; примитивный элемент имеет порядок $n = q^m - 1$), или должен быть степенью примитивного элемента (то есть n будет делить $q^m - 1$).

Также использование примитивного элемента позволит нам сохранить порядок, в котором элементы подставляются в функцию (номер ячейки будет интерпретироваться как степень примитивного элемента).

Данный вектор будет иметь размерность n, у которого 2L правых позиций будут проверочными (для исправления до L ошибок, которые могут возникнуть в этом векторе). Остальные позиции будут содержать передаваемую информацию.

Для того чтобы найти спектр разложения нашего многочлена, мы должны по очереди подставить в него все степени элемент α от α^0 до α^{n-1}, получив образ Фурье для нашего полинома.

Иными словами, если $\vec{v} = (v_0, v_1, \ldots, v_{n-1})$, $v_i \in GF(q^m)$ — вектор информационных символов, $v(x) = v_0 + v_1 x + v_2 x^2 + \ldots + v_{n-1} x^{n-1}$ соответствующий полином, то кодовым вектором кода Рида-Соломона будет

$$\vec{V} = (v(1), v(\alpha), v(\alpha^2), \ldots, v(\alpha^{n-1}))$$

, то есть значения полинома в точках $1, \alpha, \alpha^2, \ldots, \alpha^{n-1}$.

Данный вектор $\vec{V}$ мы и передаем по каналу связи.

Декодирование

При прохождении сигнала через канал связи на него накладываются помехи. Для того, чтобы определить, наложились ли помехи на передаваемое сообщение, нужно попробовать его раскодировать, применив обратное преобразование Фурье.

Если после декодирования мы получаем ненулевые проверочные элементы (как это было в исходном сообщении), мы делаем вывод о том, что в канале произошло наложение помех и мы получили спектр сигнала, сложенный со спектром помех.

Обратное преобразование Фурье в данном случае определяется таким образом

$$v_j = (n)^{-1} \sum_{j=0}^{n-1} \alpha^{-ij} V_j, \quad i = 0, 1, \ldots, n-1.$$

Где n интерпретируется как порядок примитивного элемента поля $GF(q^m)$. [2]

То есть мы попытались собрать функцию из того спектра, который мы получили на выходе. В результате мы получили функцию, которая является суммой нашего изначального многочлена-сообщения и некой функции, которая описывает спектр помехи.

Теперь наша задача заключается в том, чтобы отделить функцию исходного многочлена от функции помехи. Данная операция возможна и основываются на свойстве линейности преобразования Фурье $F(\alpha x + \beta y) = \alpha f(x) + \beta f(y)$ (если мы берем какую-то линейную комбинацию функций, то преобразование Фурье этой комбинации будет такой же линейной комбинацией образов Фурье этих функций).

Определение ошибочных позиций

Предположим, что существует некоторый полином ошибок, который генерирует ошибку, наложившуюся на сигнал в канале связи. Общем случае данный полином имеет вид

$$e(x) = k_1 x^{j_1} + k_2 x^{j_2} + \ldots + k_r x^{j_r}, \text{ где } r \le t.$$

Декодер вычисляет последовательность синдромов вида

$$s_i = v'(a^i) = v(a^i) + e(a^i) = 0 + e(a^i) = e(a^i), \quad 1 \le i \le \delta - 1 = 2t.$$

Получим 2t синдромов

$$s_1 = e(a^1) = k_1 a^{j_1} + k_2 a^{j_2} + \ldots + k_r a^{j_r}$$

$$s_2 = e(a^2) = k_1 a^{2j_1} + k_2 a^{2j_2} + \ldots + k_r a^{2j_r}$$

$$\ldots$$

$$s_{2t} = e(a^{2t}) = k_1 a^{2tj_1} + k_2 a^{2tj_2} + \ldots + k_r a^{2tj_r}$$

Наша задача – найти позиции ошибок $j_1, j_2, \ldots, j_r$, $r \le t$, пользуясь знанием синдромов $s_1, s_2, \ldots, s_{2t}$.

Рассмотрим полином локаторов ошибок $\sigma(y)$

$$\sigma(y) = (y - a^{j_1})(y - a^{j_2}) * \ldots * (y - a^{j_r})$$

Нам неизвестен точный вид этого полинома, т.к. неизвестны значения $a^{j_1}, a^{j_2}, \ldots, a^{j_r}{}_r$. Раскроем скобки, получим полином r-ой степени

$$\sigma(y) = 1 + \sigma_1 y + \sigma_2 y^2 + \ldots + \sigma_r y^r$$

Коэффициенты $\sigma_i \in GF(q)$, $1 \le i \le r$ нам также неизвестны.

Предположим, что каким-то образом мы узнали корни $\sigma(y)$ (например, перебрав все элементы рассматриваемого поля):

$$a^{j_1}, a^{j_2}, \ldots, a^{j_r}$$

Тогда легко сможем найти позиции ошибок, вычислив дискретный логарифм

$$j_1 = \log_a a^{j_1}$$

$$j_2 = \log_a a^{j_2}$$

$$\ldots$$

$$j_r = log_a a^{j_r}$$

Итого, знаем полином локаторов ошибок, а значит знаем его корни, и с помощью них найдем позиции ошибок.

Существуют различные способы установить вид характеристического полинома по набору значений $s_1, s_2, \ldots, s_{2t}$.

Можно показать, что синдромы $s_1, s_2, \ldots, s_{2t}$ и коэффициенты полинома локаторов ошибок $\sigma_1, \sigma_2, \ldots, \sigma_r$ соотносятся следующим образом

$$s_i = \sigma_1 s_{i-1} + \sigma_2 s_{i-2} + \ldots + \sigma_t s_{i-t}, \text{ при } i = t+1, \ldots, 2t.$$

Такие же соотношения выполнены для регистра сдвига с линейной обратной связью с начальным заполнением $s_1, s_2, \ldots, s_{2t}$ и характеристическим полиномом $\sigma(y)$. Для нахождения минимального характеристического полинома можно применить алгоритм Берлекэмпа-Месси. [3]

Алгоритм Берлекэмпа-Месси

Исходными данными алгоритма является последовательность s длины N. Алгоритм состоит из трех шагов.

Шаг 1. Инициализация:

- Введем несколько переменных: L – длина регистра сдвига, r – номер итерации, Δ_r – вспомогательная переменная.
- Введем несколько многочленов: C(x) – текущий регистр сдвига с линейной обратной связью, T(x) и B(x) – вспомогательные многочлены.
- $r = 0,\ L = 0,\ C(x) = 1,\ B(x) = 1$

Шаг 2. Работа алгоритма состоит из пяти пунктов:

1. $r = r + 1;\ \Delta_r = \sum_{j=0}^{L} c_j s_{r-j}$

2. Если $\Delta_r = 0$, то переходим к пункту 4;

в противном случае $T(x) = C(x) - \Delta_r x B(x)$

3. Если $2L \le r - 1$, то $B(x) = \Delta_r^{-1} C(x);\ C(x) = T(x);\ L = r - L$ и переходим к пункту 5;

в противном случае $C(x) = T(x)$;

4. $B(x) = xB(x)$;

5. Если $r = N$, то завершаем шаг 2;

в противном случае переходим к пункту 1.

Шаг 3. Выход алгоритма:

- L – длина регистра сдвига;
- C(x) – регистр сдвига с линейной обратной связью. [4]

Исправление найденных ошибок

После того как мы нашли искомый характеристический полином, мы с его помощью сможем не только найти ошибочные позиции, но и исправить их.

Так как нам известно количество символов в векторе ошибки (оно равно длине вектора образа Фурье от исходного сообщения), то мы можем восстановить информацию о значениях недостающих позиций в векторе ошибки (потому что через них выражены остальные ячейки вектора, которые идут после них). То есть мы сами генерируем прошлое этой последовательности, зная её многочлен. Таким образом мы достраиваем количество позиций, равное количеству значащих позиций в коде (которые являются смысловыми, а не проверочными) и получаем образ помех. Это называется методом Форни.

Допустим, у нас есть k смысловых позиций. Первая «ближайшая» позиция из «прошлого» является символом s_{-1}, который выдал генератор. Так как генерируемые символы периодичны, то данным элементом будет последний элемент в периоде; по аналогии предпоследний элемент будет равен элементу последовательности s_{-2}. Поэтому для вычисления k первых позиций вектора ошибки нам нужно взять k позиций с конца периода последовательности.

Численное значение символа s_{-1} можно получить из выражения: $s_{t-1} = \sigma_1 s_{t-2} + \sigma_2 s_{i-3} + \ldots + \sigma_t s_{-1}$, s_{-2} найдем из выражения: $s_{t-2} = \sigma_1 s_{t-3} + \sigma_2 s_{i-4} + \ldots + \sigma_t s_{-2}$, ведь мы уже вычислили s_{-1}. Таким же образом последовательно вычисляем элементы до s_{-k}.

После проведенных вычислений мы получили вектор $(s_{-k}, \ldots, s_{-1}, s_0, \ldots, s_{2t-1})$. Для данного вектора выполняем преобразование Фурье, описанное в разделе кодирования, и получаем ошибку. Затем мы из полученного после канала связи вектора вычитаем только что полученный вектор ошибки, после чего применяем обратное преобразование Фурье для очищенного сигнала и получаем исходное сообщение. [5]

Так как в данном описании процессов кодирования/декодирования и восстановления исходного сообщения природа элементов не играла роли, то мы можем провести аналогичные действия над конечными полями многочленов - тогда мы сможем кодировать вектора с элементами из поля мощностью q^m (обычно используют поле 2^8, так как на практике кодирование используется для хранения/передачи байт информации (один байт может принимать 256 значений, включая нулевое)).

Данная реализация процессов кодирования и декодирования кодов Рида-Соломона с использованием преобразования Фурье немного отличается от стандартной реализации с использованием порождающего полинома, основанной на алгоритме получения разрешенных кодовых комбинаций циклического кода из комбинаций простого кода. В своей работе я выбрал реализацию с преобразованием Фурье, потому что в этом случае алгоритмы кодирования, декодирования и исправления ошибок можно описать, основываясь на менее абстрактных вещах и пользуясь более понятными практическими аналогиями.

Практическая часть

Числовой пример

Приведем числовой пример использования кодов Рида-Соломона. Все операции будем проводить над конечным полем $GF(7)$. Для удобства выпишем степени всех элементов этого поля:

z	z^0	z^1	z^2	z^3	z^4	z^5	z^6
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
2	1	2	4	1	2	4	1
3	1	3	2	6	4	5	1
4	1	4	2	1	4	2	1

5	1	5	4	6	2	3	1
6	1	6	1	6	1	6	1

Красным цветом в таблице выделен максимальный примитивный элемент этого поля. Данная таблица нам пригодится для выполнения прямого и обратного преобразования Фурье, а также для быстрого деления и возведения в степень.

Для демонстрации способа возведения элемента поля в степень возведем $5^{4*4} = (5^4)^4 = 2^4 = 2$. Как видно из примера, для возведения элемента в степень мы просто берем значение из нужной ячейки таблицы выше.

Теперь приведем пример деления над конечным полем: $\frac{6}{5} = \frac{5^3}{5} = 5^2 = 4$. Приведем ещё один пример: $\frac{1}{5} = \frac{5^6}{5} = 5^5 = 3$.

Над данным полем $GF(7)$ рассмотрим вектор, который будет состоять из 6-ти элементов (так как порядок элемента «5» равен 6-ти). Пусть код сможет исправлять 1 ошибку, а значит будет иметь 4 значащие позиции и 2 служебные (такой код можно назвать кодом (6, 4)). Возьмем вектор (1,2,3,4,0,0), который мы передадим по каналу связи с помехами.

Кодирование

Данное сообщение можно представить как многочлен $a(x) = 1 + 2x + 3x^2 + 4x^3$.

Применим к нему дискретное преобразование Фурье над конечным полем (возьмем примитивный элемент 5). Для этого в многочлен подставим по очереди все степени элемента 5 начиная от 0-ой до 5-ой включительно:

$$a\left(5^0\right) = a(1) = 1 + 2 * 1 + 3 * 1^2 + 4 * 1^3 = 3$$

$$a\left(5^1\right) = a(5) = 1 + 2 * 5 + 3 * 5^2 + 4 * 5^3 = 5$$

$$a\left(5^2\right) = a(4) = 1 + 2 * 4 + 3 * 4^2 + 4 * 4^3 = 5$$

$$a\left(5^3\right) = a(6) = 1 + 2 * 6 + 3 * 6^2 + 4 * 6^3 = 5$$

$$a\left(5^4\right) = a(2) = 1 + 2 * 2 + 3 * 2^2 + 4 * 2^3 = 0$$

$$a\left(5^5\right) = a(3) = 1 + 2 * 3 + 3 * 3^2 + 4 * 3^3 = 2$$

После преобразования Фурье мы получили спектр (3,5,5,5,0,2), который мы и передадим по каналу связи.

Декодирование

Во время прохождения сообщения по каналу связи на него наложилась помеха $(0, 0, 3, 0, 0, 0)$, и в результате на выходе мы получили вектор $(3, 5, 5 + 3, 5, 0, 2) = (3, 5, 1, 5, 0, 2)$. Попробуем раскодировать данный зашумленный вектор, используя обратное дискретное преобразование Фурье над конечным полем:

$$a`\left(5^0\right) = a`(1) = \frac{3*1^0+5*1^{-1}+1*1^{-2}+5*1^{-3}+0*1^{-4}+2*1^{-5}}{6} = \frac{2}{6} = 5$$

$$a`\left(5^1\right) = a`(5) = \frac{3*5^0+5*5^{-1}+1*5^{-2}+5*5^{-3}+0*5^{-4}+2*5^{-5}}{6} = \frac{4}{6} = 3$$

$$a`\left(5^2\right) = a`(4) = \frac{3*4^0+5*4^{-1}+1*4^{-2}+5*4^{-3}+0*4^{-4}+2*4^{-5}}{6} = \frac{2}{6} = 5$$

$$a`\left(5^3\right) = a`(6) = \frac{3*6^0+5*6^{-1}+1*6^{-2}+5*6^{-3}+0*6^{-4}+2*6^{-5}}{6} = \frac{6}{6} = 1$$

$$a`\left(5^4\right) = a`(2) = \frac{3*2^0+5*2^{-1}+1*2^{-2}+5*2^{-3}+0*2^{-4}+2*2^{-5}}{6} = \frac{6}{6} = 1$$

$$a`\left(5^5\right) = a`(3) = \frac{3*3^0+5*3^{-1}+1*3^{-2}+5*3^{-3}+0*3^{-4}+2*3^{-5}}{6} = \frac{5}{6} = 2$$

В результате получили раскодированный вектор (5,3,5,1,1,2). Служебные позиции (1,2) не равны нулю, значит в канале связи произошло наложение помех.

Определение ошибочных позиций

Воспользуемся алгоритмом Берлекэмпа-Месси:

	s_0	s_1
Синдром ошибки равен:	1	2

Начальное состояние алгоритма: $c(x) = 1,\ b(x) = 1,\ L = 0,\ r = 0.$

1) $r = 1,\ \Delta_1 = 1 * 1 = 1 \neq 0,\ \Rightarrow T(x) = 1 - x = 1 + 6x$

$2 * 0 = 0,\ \Rightarrow b(x) = 1,\ c(x) = 1 + 6x,\ L = 1 - 0 = 1$

2) $r = 2,\ \Delta_2 = 1 * 2 + 6 * 1 = 1 \neq 0,\ \Rightarrow T(x) = 1 + 6x - x = 1 + 5x$

$2 * 1 > 1,\ \Rightarrow b(x) = x,\ c(x) = 1 + 5x$

В результате $c(x) = 5x + 1$. Нормализуем коэффициенты:

$$c(x) = \frac{5x+1}{5} = x + 3.$$

Корнем данного уравнения является $x = 4 = 5^2$, значит ошибка произошла в позиции 2 (позиция определена верно).

Восстановление искаженных позиций

Восстановим испорченные символы, используя алгоритм Форни:

Найдем $(s_{-4}, s_{-3}, s_{-2}, s_{-1}, s_0, s_1)$

$$s_{-1}: \; s_{-1} * 1 + s_0 * 3 = s_{-1} * 1 + 1 * 3 = 0, \; => s_{-1} = 4$$

$$s_{-2}: \; s_{-2} * 1 + s_{-1} * 3 = s_{-2} * 1 + 4 * 3 = 0, \; => s_{-2} = 2$$

$$s_{-3}: \; s_{-3} * 1 + s_{-2} * 3 = s_{-3} * 1 + 2 * 3 = 0, \; => s_{-3} = 1$$

$$s_{-4}: \; s_{-4} * 1 + s_{-3} * 3 = s_{-4} * 1 + 1 * 3 = 0, \; => s_{-4} = 4$$

В итоге получили вектор (4,1,2,4,1,2), который можно представить как $d(x)$ - полином-прообраз ошибки. Именно он сложился с изначальным вектором и после преобразования Фурье превратился в искаженный вектор, который мы получили на выходе. Но свойство линейности преобразования Фурье позволила нам найти образ многочлена ошибки и вычесть его из сигнала:

$$d\left(5^0\right) = d(1) = 4 + 1 * 1 + 2 * 1^2 + 4 * 1^3 + 1 * 1^4 + 2 * 1^5 = 0$$

$$d\left(5^1\right) = d(5) = 4 + 1 * 5 + 2 * 5^2 + 4 * 5^3 + 1 * 5^4 + 2 * 5^5 = 0$$

$$d\left(5^2\right) = d(4) = 4 + 1 * 4 + 2 * 4^2 + 4 * 4^3 + 1 * 4^4 + 2 * 4^5 = 3$$

$$d\left(5^3\right) = d(6) = 4 + 1 * 6 + 2 * 6^2 + 4 * 6^3 + 1 * 6^4 + 2 * 6^5 = 0$$

$$d\left(5^4\right) = d(2) = 4 + 1 * 2 + 2 * 2^2 + 4 * 2^3 + 1 * 2^4 + 2 * 2^5 = 0$$

$$d\left(5^5\right) = d(3) = 4 + 1 * 3 + 2 * 3^2 + 4 * 3^3 + 1 * 3^4 + 2 * 3^5 = 0$$

В результате преобразования Фурье мы получили помеху (0,0,3,0,0,0), которую можно вычесть из сигнала: (3,5,1-3,5,0,2)=(3,5,5,5,0,2) – мы получили образ изначального сообщения без помех. Убедимся в этом и раскодируем полученное сообщение.

$$a`\left(5^0\right) = a`(1) = \frac{3*1^0+5*1^{-1}+5*1^{-2}+5*1^{-3}+0*1^{-4}+2*1^{-5}}{6} = \frac{6}{6} = 1$$

$$a`\left(5^1\right) = a`(5) = \frac{3*5^0+5*5^{-1}+5*5^{-2}+5*5^{-3}+0*5^{-4}+2*5^{-5}}{6} = \frac{5}{6} = 2$$

$$a`\left(5^2\right) = a`(4) = \frac{3*4^0+5*4^{-1}+5*4^{-2}+5*4^{-3}+0*4^{-4}+2*4^{-5}}{6} = \frac{1}{6} = 3$$

$$a`\left(5^3\right) = a`(6) = \frac{3*6^0+5*6^{-1}+5*6^{-2}+5*6^{-3}+0*6^{-4}+2*6^{-5}}{6} = \frac{3}{6} = 4$$

$$a`\left(5^4\right) = a`(2) = \frac{3*2^0+5*2^{-1}+5*2^{-2}+5*2^{-3}+0*2^{-4}+2*2^{-5}}{6} = \frac{0}{6} = 0$$

$$a`\left(5^5\right) = a`(3) = \frac{3*3^0+5*3^{-1}+5*3^{-2}+5*3^{-3}+0*3^{-4}+2*3^{-5}}{6} = \frac{0}{6} = 0$$

В результате преобразования мы получили исходное сообщение (1,2,3,4,0,0), значит проделанные преобразования можно считать верными.

Программная реализация

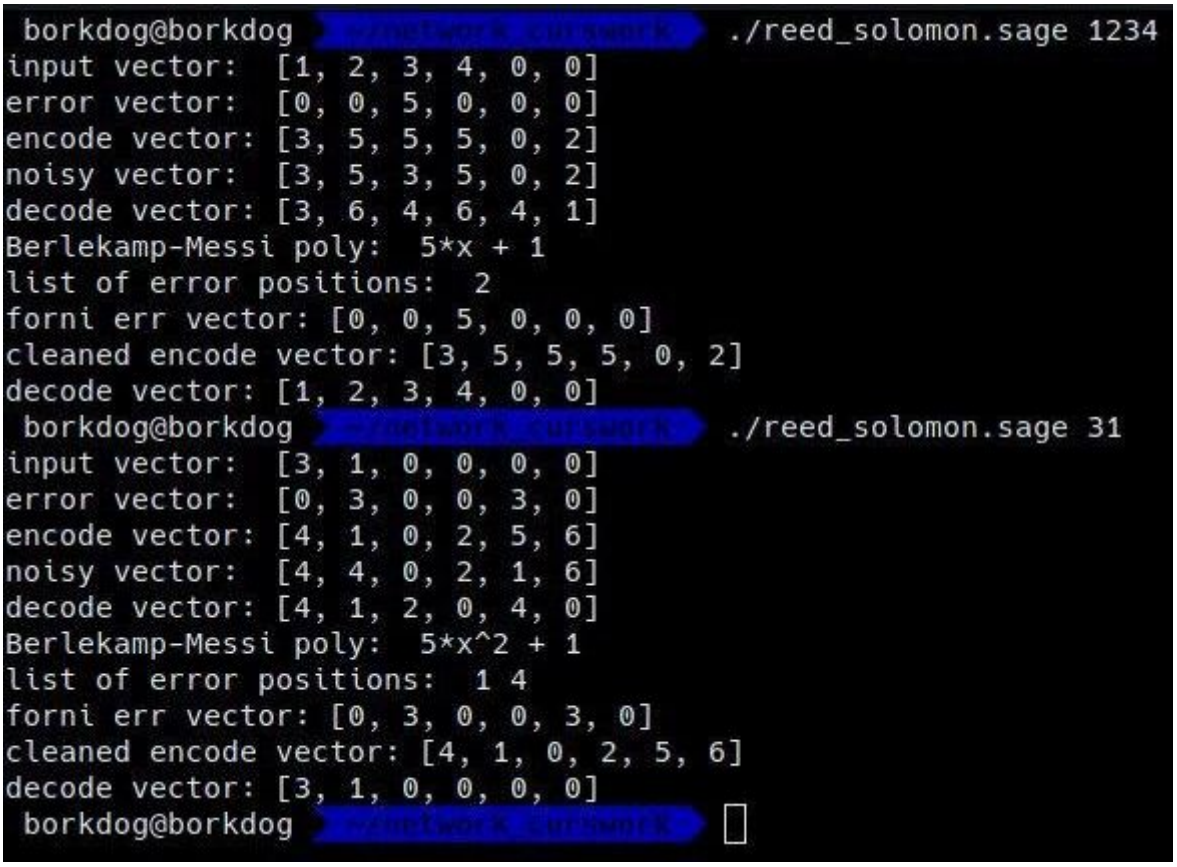
Так как все вычисления нам придется проводить над конечным полем, то лучше всего использовать язык программирования компьютерной алгебры. Мой выбор остановился на языке SageMath версии 9.0, так как в его основе лежит Python3, а значит на нем можно писать компактный и читаемый код, что облегчит его восприятие.

SageMath (англ. Sage - Мудрец) — система компьютерной алгебры покрывающая много областей математики, включая алгебру, комбинаторику, вычислительную математику и математический анализ. Первая версия SageMath была выпущена 24 февраля 2005 года в виде свободного программного обеспечения с лицензией GNU GPL. Первоначальной целью проекта было "создание открытого программного обеспечения альтернативного системам Magma, Maple, Mathematica, и MATLAB". Разработчиком SageMath является Уильям Стейн — математик Университета Вашингтона. [6]

На момент написания документа актуальной версией является SageMath 9.0. Данная версия включает в себя хорошо отлаженные модули вышеупомянутых систем компьютерной алгебры, управляющая программа для которых пишется с использованием синтаксиса ЯПВУ Python3. Также у SageMath есть консольная версия и поддержка *.sage скриптов, что делает

его очень удобным языком для использования в unix-образных системах как составную часть других программных комплексов.

В приложении 1 приведен листинг программы, написанной мной на языке SageMath. Теперь приведем вывод программы во время её работы:



Работу данной программы можно описать блок-схемой в приложении 2.

Опишем в словесной форме, за что отвечает каждая из функций кода.

Разработанная мной программа принимает один аргумент через командную строку – последовательность символов, которую нужно закодировать (из-за особенностей программы данное количество символов не может быть больше 4-х). Символами данной последовательности могут быть только числа от 0 до 6-ти, так как мы работаем над полем $GF(7)$. В данной программе не производится проверка входных данных – данный подход призван уменьшить количество кода в листинге, который не относится непосредственно к кодам Рида-Соломона.

Функция main() – это основная функция, из неё вызываются все остальные функции. Сначала входной вектор дополняется нулями до 6-ти позиций. После этого генерируется вектор ошибки случайным образом, оставляя такое количество позиций ненулевыми, чтобы кодовый вектор

после передачи можно было восстановить. За генерацию вектора ошибки отвечает функция gen_error. Далее к входному вектору применяется дискретное преобразование Фурье над полем $GF(7)$ (функция encode()), после чего на полученный образ накладываются помехи. Далее производится попытка декодирования полученного сообщения функцией decode(), которая представляет из себя обратное преобразование Фурье над полем $GF(7)$, и если синдром будет нулевым, то программа завершает работу. Если же синдром оказался ненулевым, то с помощью функции berlekamp_messi() мы из синдрома получаем ассоциированный с ним полином и по нему узнаем номера искаженных позиций, находя степени его корней. После чего мы восстанавливаем вектор ошибки с помощью функции forni(): она по ассоциированному многочлену методом Форни достраивает прообраз помехи и находит его Фурье-образ функцией encode() (то есть находим изначальную помеху канала связи). После чего мы вычитаем из выходного сигнала найденную помеху и применяем к очищенному сигналу функцию decode(), получая исходное сообщение. После этого программа завершается.

Аппаратная реализация кодера

Аппаратная реализация кодера и декодера кодов Рида-Соломона достаточно трудна в силу используемых преобразований в алгоритмах. Поэтому в данной работе мы ограничимся только программной реализацией механизмов кодирования/декодирования и исправления ошибок. В ходе проводимого исследования выяснилось наличие аппаратных реализаций в сети интернет (например, патент [7]).

Применение в практических устройствах

Сразу после появления коды Рида — Соломона стали применяться в качестве внешних кодов в каскадных конструкциях, использующихся в спутниковой связи.

В настоящий момент коды Рида — Соломона имеют очень широкую область применения благодаря их способности находить и исправлять многократные пакеты ошибок.

Запись и хранение информации

Код Рида — Соломона используется при записи и чтении в контроллерах оперативной памяти, при архивировании данных, записи

информации на жесткие диски (ECC), записи на CD/DVD диски. Даже если поврежден значительный объем информации, испорчено несколько секторов дискового носителя, то коды Рида — Соломона позволяют восстановить большую часть потерянной информации. Также используется при записи на такие носители, как магнитные ленты и штрихкоды.

Запись на CD-ROM

Возможные ошибки при чтении с диска появляются уже на этапе производства диска, так как сделать идеальный диск при современных технологиях невозможно. Также ошибки могут быть вызваны царапинами на поверхности диска, пылью и т.д. Поэтому при изготовлении читаемого компакт-диска используется система коррекции CIRC (CrossInterleavedReedSolomonCode). Эта коррекция реализована во всех устройствах, позволяющих считывать данные с CD дисков, в виде чипа с прошивкой firmware. Нахождение и коррекция ошибок основана на избыточности и перемежении (redundancy&interleaving). Избыточность составляет примерно 25 % от исходной информации.

При записи на аудиокомпакт-диски используется стандарт RedBook. Коррекция ошибок происходит на двух уровнях, C1 и C2. При кодировании на первом этапе происходит добавление проверочных символов к исходным данным, на втором этапе информация снова кодируется. Кроме кодирования осуществляется также перемешивание (перемежение) байтов, чтобы при коррекции блоки ошибок распались на отдельные биты, которые легче исправляются.

Беспроводная и мобильная связь

Данный алгоритм кодирования используется при передаче данных по сетям WiMAX, в оптических линиях связи, в спутниковой и радиорелейной связи. Метод прямой коррекции ошибок в проходящем трафике (ForwardErrorCorrection, FEC) основывается на кодах Рида — Соломона. [1]

RAID 6

Код Рида — Соломона используется в некоторых прикладных программах в области хранения данных, например в RAID 6;

RAID (англ. RedundantArrayofIndependentDisks — избыточный массив независимых (самостоятельных) дисков) — технология виртуализации

данных для объединения нескольких физических дисковых устройств в логический модуль для повышения отказоустойчивости и производительности. Технология RAID 6 похож на широко используемую технологию RAID 5, но имеет более высокую степень надёжности, так как её реализация содержит два (или более) диска данных и два диска контроля чётности. Основана на кодах Рида — Соломона и обеспечивает работоспособность после одновременного выхода из строя любых двух дисков. [8]

Рассеивающие преобразования на кодах РС

Коды Рида-Соломона можно использовать для построения рассеивающих преобразований в блочных шифрах и хэш-функциях. Примером алгоритма с такими преобразованиями является SHARK, так как он для линейного преобразования использует МДР-коды, а именно коды Рида-Соломона. [9]

Асимметричные криптосистемы на кодах РС

Существует ряд ассиметричных криптосистем, основанных на кодах Рида-Соломона.

Криптосистема McEliece

Примером такой системы является криптоситема McEliece с открытыми ключами на основе теории алгебраического кодирования, разработанная в 1978 году Робертом Мак-Элисом. Это была первая схема, использующая рандомизацию в процессе шифрования. Алгоритм не получил широко признания в криптографии, но в то же время является кандидатом для постквантовой криптографии, так как устойчив к атаке с использованием Алгоритма Шора.

Алгоритм основан на сложности декодирования полных линейных кодов. В качестве линейного кода данный алгоритм использует коды Рида-Соломона. Общая задача декодирования является NP-сложной. [10]На данный момент уже найдены алгоритмы, взламывающие криптосистему за полиномиальное, либо субэкспоненциальное время работы.[11]

Криптосистема Сидельникова

Криптосистема Сидельникова (Мак-Элиса-Сидельникова) — криптографическая система с открытым ключом, основанная на криптосистеме McEliece. Была предложена математиком, академиком Академии криптографии РФ Владимиром Михайловичем Сидельниковым в 1994 году. Для генерации раундовых ключей в данном алгоритме используется алгоритм SHARK, описанный выше.[12] Данная криптосистема на сегодняшний момент тоже не является стойкой для ряда атак. [11]

Заключение

Коды Рида-Соломона находят широкое применение в различных областях человеческой деятельности – от хранения данных до криптографических приложений. Также, как уже было указано ранее, они обладают оптимальным соотношением количества проверочных символов к количеству исправляемых ошибок, что делает их наиболее эффективными кодами - это позволяет строить системы, которые очень устойчивы к искажениям искусственного и естественного характера.

В данной работе была разобрана одна из возможных алгоритмических реализаций кодера, декодера, а также механизма выявления и исправления ошибок для кодов Рида-Соломона. Также рассмотрена работа алгоритмов на конкретном числовом примере и выполнена программная реализация данных алгоритмов на ЯПВУ SageMath.

Используемая литература

1. Статья «Код Рида-Соломона», материал из Википедии [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Код_Рида_—_Соломона, свободный (23.05.20)
2. Статья «Дискретное преобразование Фурье над конечным полем», материал из Википедии [Электронный ресурс]. -Режим доступа: https://ru.wikipedia.org/wiki/Дискретное_преобразование_Фурье_над_конечным_полем, свободный (23.05.20)
3. Ю.Л. Сагалович Введение в алгебраические коды, учебное пособие. — 2-е изд., перераб. и доп. — М.: ИППИ РАН, 2010.
4. Методические указания «Математические методы защиты информации», Министерство образования Российской Федерации Ярославский государственный университет им. П.Г. Демидова Кафедра компьютерных сетей, часть 3 [Электронный ресурс]. - Режим доступа: http://www.lib.uniyar.ac.ru/edocs/iuni/20130406.pdf, свободный (23.05.20)
5. Статья «Коды Рида-Соломона» [Электронный ресурс]. -Режим доступа: https://habr.com/ru/post/191418/, свободный (23.05.20)
6. Документация системы компьютерной алгебры SageMath [Электронный ресурс]. - Режим доступа: https://www.sagemath.org/ru/, свободный (23.05.20)
7. Патент 2 541 869 Рос. Федерация МПК H03M 13/45 Егоров С.И., Графов О.Б.; заявитель и патентообладатель —Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования "Юго-Западный государственный университет" (ЮЗ ГУ) № 2013145270/08; заявл. 10.10.13 ; опубл. 20.02.15, Бюл.№5 - Режим доступа: https://yandex.ru/patents/doc/RU2541869C1_20150220, свободный (23.05.20)
8. Статья «RAID», материал из Википедии [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/RAID,свободный (23.05.20)
9. Статья «SHARK», материал из Википедии [Электронный ресурс]. – Режим доступа:https://ru.wikipedia.org/wiki/SHARK, свободный (23.05.20)
10. Статья «McEliece», материал из Википедии [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/McEliece, свободный (23.05.20)
11. Minder L., Shokrollahi A.Cryptanalysis of the Sidelnikov Cryptosystem (англ.) // Advances in Cryptology — EUROCRYPT 2007: 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007. Proceedings / M. Naor — Springer Berlin

Heidelberg, 2007. — P. 347—360. — ISBN 978-3-540-72539-8 — doi:10.1007/978-3-540-72540-4_20
12. Статья «Криптосистема Сидельникова», материал из Википедии [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Криптосистема_Сидельникова, свободный (23.05.20)

Приложения

Приложение №1

Листинг программы «reed_solomon.sage»:

```
#!/usr/bin/env sage

import sys
import copy
import random
from sage.all import *


def gen_error(amnt_service_pos, L):
    seed()
    out = [L[0] for i in range(7-1-(amnt_service_pos//2))]
    for i in range(amnt_service_pos//2):
        out.insert(randint(0, 7-1-(amnt_service_pos//2)+i), L[randint(0,7-1)])
    return out


def encode(vec, L):
    R = PolynomialRing(GF(7), 'x')(vec)
    return [R(L[5]**i) for i in range(7-1)]


def decode(vec, L):
    R = PolynomialRing(GF(7), 'x')(vec)
    return [R(L[5]**(-i))/L[7-1] for i in range(7-1)]


def berlekamp_messi(S, L):
    p.<x> = PolynomialRing(GF(7), 'x')
    C = p([1])
    B = p([1])
    r = 0
    l = 0
    while (r < len(S)):
        delta = L[0]
        for i in range(len(C.list())):
            delta += S[r-i]*C.list()[i]
```

```
        if (delta != 0):
            T = C - delta * x * B
            if (2*l<=r):
                B = (delta**(-1))*C
                C = T
                l = r-l+1
            else:
                C = T
                B *= x
        else:
            B *= x
        r += 1
    return C


def forni(S, L, poly, amnt_service_pos):
    out = deepcopy(S)
    poly_list = poly.list()
    poly_len = len(poly_list)-1
    for i in range(7-1-amnt_service_pos):
        out.insert(0,L[0])
        for r in range(poly_len):
            out[0] -= out[poly_len-r]*poly_list[r]/poly_list[-1]
    return encode(out, L)


def main():
    L = GF(7).list()
    amnt_service_pos = 7-1-len(sys.argv[1])
    inp = [L[int(i)] for i in sys.argv[1]]
    for i in range(amnt_service_pos):
        inp.append(L[0])
    err = gen_error(amnt_service_pos, L)

    print("input vector: ",inp)
    print("error vector: ",err)
    enc = encode(inp, L)
    print("encode vector:", enc)
    noisy = [enc[i]+err[i] for i in range(7-1)]
    print("noisy vector: ", noisy)
    dec = decode(noisy, L)
    print("decode vector:", dec)
    if (dec[-amnt_service_pos:] != [L[0] for i in range(amnt_service_pos)]):
        poly = berlekamp_messi(dec[-amnt_service_pos:], L)
        print("Berlekamp-Messi poly: ",poly)
        print("list of error positions: "," ".join([(str)(r[0].log(5)) for r in p
oly.roots()]))
        err = forni(dec[-amnt_service_pos:], L, poly, amnt_service_pos)
        print("forni err vector:", err)
        enc = [noisy[i]-err[i] for i in range(7-1)]
        print("cleaned encode vector:", enc)
        dec = decode(enc, L)
        print("decode vector:", dec)
```

```
if __name__ == "__main__":
    main()
```

Приложение№2

Блок-схема алгоритма работы программы «reed_solomon.sage»: