Декодирование корректирующих кодов низкой плотности (LDPC) алгоритмом Sum-Product

Оглавление

08.08.2020, 32 с.

Пособие рассчитано на выпускников технических учебных заведений, специализирующихся на проблемах передачи информации. Перед изучением данного пособия рекомендуется изучить тему Линейные блочные коды, которая является базовой.

Введение

Коды низкой плотности – это двоичные линейные блочные коды, имеющие специфическую структуру проверочной матрицы H. Особенность этой матрицы заключается в том, что количество единиц K в каждой строке

фиксировано и намного меньше длины строки n. Как правило, длина строки (число столбцов матрицы) составляет тысячи и десятки тысяч, в то время как ненулевых элементов в строке – всего лишь несколько единиц. Каждая строка проверочной матрицы является случайной в том плане, что единицы разбросаны по случайным позициям. Англоязычная аббревиатура таких кодов – LDPC – Low-Density Parity Check codes.

Малость количества единиц вызвана необходимостью контроля вычислительных затрат на процесс декодирования, ведь скорость декодирования ограничивает возможную скорость передачи (приема) данных. Однако, от количества единиц K зависит корректирующая способность таких кодов: если $K << n$, то чем больше K, тем выше достоверность приема, но ниже скорость декодирования, поэтому всегда необходим некоторый компромисс между временем декодирования и требуемой вероятностью ошибки. Помимо количества единиц K на достоверность приема влияет длина строки n: чем больше, тем лучше. В случае выбора LDPC кодов приоритетом является моделирование (несмотря на его трудоемкость ввиду больших n), потому что теория как таковая отсутствует (скорее, существуют практические рекомендации построения кодов), и лучшим критерием является измеренная вероятность ошибки на выходе декодера для конкретного кода и алгоритма декодирования. Единственная теоретическая поддержка – теорема Шеннона, говорящая о том, что с ростом n потенциально (хотя, на самом деле, и теоретически, и практически) возрастает корректирующая способность кодов, и при этом вполне можно обойтись двоичными кодами, что упрощает аппаратную (hardware) реализацию кодека.

Из-за случайности проверочной матрицы кода необходим некий универсальный алгоритм декодирования. К счастью, такой алгоритм есть и имя ему – алгоритм распространения доверия (Belief Propagation Algorithm). Часто такой алгоритм называют как Sum-Product Algorithm, чем мы и будем пользоваться в дальнейшем. Этот алгоритм относится к классу «обменных» алгоритмов: алгоритмов обмена сообщениями (Message Passing Algorithm).

Существует также такое, довольно-таки обобщенное, понятие как фактор-граф (Factor Graph), с которым можно ознакомиться, например, здесь, но после изучения данного пособия, одна из целей которого – адаптировать ординарного читателя к чтению серьезных публикаций.

LDPC коды были предложены американским исследователем Робертом Галлагером (эти коды иногда так и называют – коды Галлагера), и представлены в его докторской диссертации в 63-м году 20-го столетия[1]. Однако, в те времена об их практической применимости не было и речи из-за нехватки вычислительных мощностей. Вспомнили о них лишь в середине лихих 90-х, но не в России, а в Англии/Канаде, где 90-е не были лихими.

Предваряя всякие рассуждения относительно разного рода существующих на сегодня декодеров, скажу, что каков бы ни был корректирующий код и каков бы ни был алгоритм декодирования, главенствующим является понятие в е р о я т н о с т и. Тот, кто даст миру лучшую конструктивную альтернативу, вероятно, будет иметь статус «Шеннон №2», если позволительны такие вольности…

Алгоритм Sum-Product

Подготовка

Название Sum-Product говорит о сумме произведений. Как мы знаем из теории вероятностей, есть теорема умножения и теорема сложения вероятностей – вот, в принципе, и весь теоретический базис данного алгоритма, как бы парадоксально это ни звучало.

Итак, все начинается с вероятности (с меры доверия), которая, как известно, является вещественным числом на интервале от 0 до 1. Сравнивая вероятность с некоторым пороговым уровнем (числом), будем получать логические нули и единицы – квантованные вероятности – результаты принятия некоторого решения. Степень квантования может быть разной: если на интервале от 0 до 1 выбрать несколько пороговых уровней, то получится

[1] https://web.stanford.edu/class/ee388/papers/ldpc.pdf

устройство (алгоритм), выходом которого будут номера – номера отрезков, в которые попадает заданная вероятность. Таким образом, выбирая степень квантования, от бесконечной (нет квантования) до начальной (бинарное квантование), будем получать решающие устройства с разной степенью мягкости – от полностью мягкого до жесткого. Такая классификация полезна из-за существования мягких (soft) и жестких (hard) алгоритмов декодирования корректирующих кодов, плюс при аппаратной (hardware) реализации любого декодера всегда имеется ограничение на разрядность чисел, что говорит об отсутствии полностью мягких схем декодирования – такие схемы существуют только в теории в виде математических формул и при всякой реализации возникает квантование той или иной степени.

Помимо квантования вероятностей (сужения динамического диапазона – отношения максимально возможной величины к минимально возможной), полезна и обратная операция – расширение динамического диапазона. Например, известно, что сумма по модулю два $a \oplus b = c$ (логическая операция XOR) равна нулю тогда и только тогда, когда складываемые величины совпадают, в противном случае она равна единице: $00 \to 0$, $11 \to 0$, $01 \to 1$, $10 \to 1$. Почему именно сумма по модулю два? Потому что это линейная операция, в отличие от OR, AND и NOT. Напомним, что изучаемые здесь LDPC коды являются линейными. Так вот, рассматривая выход блока XOR как квантованный (бинарный) выход, можно заключить, что бит 0 на выходе возможен лишь тогда, когда на первом входе 0 и на втором входе 0, или на первом 1 и на втором 1. Мы только что сформулировали некоторое логическое условие, и теперь, приписывая нулям и единицам некоторые заранее неизвестные вероятности $p(0)$ и $p(1)$ (меру доверия), а также используя теоремы сложения и умножения вероятностей, можно составить элементарное уравнение относительно выходного бита 0

$$p_c(0) = p_a(0)p_b(0) + p_a(1)p_b(1).$$

Здесь произведение вероятностей соответствует логике И, сложение – логике ИЛИ. Аналогично запишем уравнение относительно бита 1

$$p_c(1) = p_a(0)p_b(1) + p_a(1)p_b(0).$$

Можно показать, что для любых входных вероятностей сумма выходных равна единице, поэтому нормировка здесь не требуется. Таким образом, битовая логика XOR была расширена на операции с вероятностями – вещественными числами. Получили своего рода блок XOR, но с двумя вещественными входами и одним вещественным выходом. Такое расширение является основой алгоритма Sum-Product, но не только – это довольно-таки общий конструктивный подход.

Постепенный переход от вещественного числа к биту – это, фактически, логика работы любого приемника некоторой системы передачи информации, а постепенный переход от бита к вещественному числу – это логика работы любого передатчика той же системы. Бит в данном случае является передаваемой информацией, а вещественное число – сигналом, с помощью которого эта информация передается.

Известно, что сигналы $s(t)$ можно охарактеризовать как зависимость напряжения (или тока) от времени, так вот это напряжение $s_0 = s(t_0)$ и является тем самым вещественным числом. Так уж вышло, что напрямую биты передавать нельзя, биты – это воображаемая абстракция, всегда существующая в виде сигналов. Когда бит хранится в файловом хранилище, этому соответствуют одни сигналы, когда бит передается по линии связи – другие, и т.д.

Упомянутое выше слово «постепенный» очень важно – чем современнее модем (приемопередатчик), тем больше в нем ступеней при преобразовании «вещественное число – бит». Разумно введенная плавность обеспечивает меньшую вероятность ошибки при приеме, при этом возросшие вычислительные затраты должны компенсироваться наличием производительного аппаратного обеспечения (hardware) – в противном случае придется снижать скорость передачи информации, либо повышать мощность и/или расширять диапазон используемых частот.

Связь с сигналами и шумами

Для того, чтобы глубже осознать алгоритм Sum-Product, следует мысленно держать связь с сигналами и шумами. Понимание сигналов позволяет увидеть (осознать) то вещественное число, которое будет

преобразовано в выдаваемый приемником бит, а шум позволяет увидеть то число, которое будет мешать принятию приемником верного решения – с этим шумовым числом и должен бороться всякий корректирующий код, а в особенности LDPC, потому что эти коды предназначены для непосредственного исправления ошибок (Forward Error Correction). Есть коды, которые предназначены для обнаружения ошибок (например, циклические коды CRC).

Проще всего рассмотреть аддитивное влияние шума $n(t)$ на сигнал заранее известной формы $s_i(t)$

$$v(t) = s_i(t) + n(t).$$

В системах передачи информации существует такое понятие как канал. Это довольно-таки абстрактная вещь, которая обозначает некий черный ящик с входом и выходом, причем существует описание процесса обработки (преобразования) входного сигнала. Так вот, введенное только что преобразование выполняется каналом, который называют аддитивным каналом с шумом, подчеркивая, что шум аддитивно складывается с сигналом.

Считается, что задержка в канале и коэффициент усиления сигнала постоянные и потому известные – это канал с постоянными параметрами, поэтому для простоты задержку можно положить равной нулю, а коэффициент усиления приравнять к единице. Индекс i здесь обозначает передаваемый бит. Битам 0 и 1 соответствуют отличающиеся сигналы (выбранные заранее).

Шум в некоторый момент времени $n(t_0)$ является случайной величиной с некоторым законом распределения вероятностей (плотностей вероятностей). Проще всего рассмотреть случайную величину с гауссовским распределением, тем более доказано, что такая величина (гауссовский шум) больше всех вредит правильному приему (при аддитивном наложении помехи). Вы возразите, что на практике больше всего вредит негауссовская помеха – да, это так, но не потому, что она негауссовская, а потому, что параметры этой помехи зависят от времени и приемник должен постоянно подстраиваться под меняющуюся помеховую обстановку (как правило, такое характерно для радиоканалов). Мы

рассматриваем аддитивный канал с постоянными параметрами – в этом случае наиболее вредной помехой является гауссовский шум. Такой канал называют стационарным аддитивным гауссовским каналом (можно было еще добавить прилагательное «непрерывным»).

Любой шум, понятное дело, зависит от времени. Это говорит о необходимости указания степени быстроты изменения шума во времени или, что точнее, частотного состава этого шума: каковы, в среднем, мощности интересующих нас частотных компонент. Проще всего рассмотреть так называемый белый шум, частотные компоненты которого имеют одинаковую среднюю мощность. Говорят, что вся мощность такого шума равномерно распределена по всему частотному диапазону; полная же мощность белого шума, естественно, равна бесконечности, а вот плотность мощности N_0, которая измеряется в ваттах-на-герц, Вт/Гц, всегда конечна, и именно она является характеристикой такого шума.

В итоге, определенный выше канал называют аддитивным каналом с белым гауссовским шумом (АБГШ или AWGN). Иногда добавляют слово «стационарный», подразумевая постоянство параметров, хотя зачастую это слово опускают и принимают это по умолчанию.

Задача приемника – пронаблюдать принятую реализацию $v(t)$ и принять решение о переданном бите i. Конечно, приемник заранее знает формы сигналов $s_i(t)$ и параметры шума $n(t)$: что шум аддитивный, белый и гауссовский, но не знает конкретной реализации шума, выпавшей в текущий момент приема, что не позволяет ее вычесть из наблюдаемой реализации и, тем самым, вынести правильное решение о переданном бите.

К счастью, несмотря на то что гауссовский шум (при аддитивном его наложении) самый вредный, существует очень простой (линейный) способ оптимального приема (обработки) аддитивной смеси сигнала с белым гауссовским шумом[2]. Оптимальность здесь понимается в плане минимизации

[2] Это восходит к работам академика В.А. Котельникова

вероятности ошибки, однако, одновременно, благодаря сочетанию «белости» и «гауссовости» шума, обеспечивается и максимальное отношение сигнал-шум. Для такого «рафинированного» AWGN шума два критерия оптимальности являются эквивалентными. Оптимальный способ обработки определяется корреляционным интегралом

$$v = \int_0^T v(t)[s_1(t) - s_0(t)]dt.$$

Устройство, которое вычисляет корреляционный интеграл, называется коррелятором[3]. Некоторый интеграл, как известно, это число, поэтому здесь мы наблюдаем преобразование набора (теоретически бесконечного) вещественных чисел в одно вещественное число – в некую эссенцию, содержащую наибольшее количество информации о переданном бите. Здесь параметр T – это длительность наблюдаемой реализации. Сигнал $w(t) = s_1(t) - s_0(t)$ называют разностным сигналом. Ясно, что если он будет равен нулю, то правильный прием будет невозможен в принципе. Подставляя в корреляционный интеграл наблюдаемую реализацию, получим представление числа v в виде суммы сигнальной и шумовой компоненты

$$v = v_c(i) + v_{\text{ш}} = \int_0^T s_i(t)[s_1(t) - s_0(t)]dt + \int_0^T n(t)[s_1(t) - s_0(t)]dt.$$

Величина v является случайной величиной, причем гауссовской, так как интегрирование является линейной операцией, а шум – белым гауссовским. Известно, что гауссовская случайная величина имеет два параметра – среднее значение m и дисперсию D

$$m_v = \bar{v} = v_c(i),$$

$$D_v = \overline{(v - \bar{v})^2} = \frac{N_0}{2}\int_0^T w^2(t)dt = \frac{E_w N_0}{2}.$$

[3] Модные сейчас нейронные сети, на самом деле, недалеко ушли от корреляционного приема, если вообще ушли…

Здесь $N_0/2$ – двусторонняя спектральная плотность мощности шума, Вт/Гц, E_w – энергия разностного сигнала. Среднее значение шума $n(t)$ равно нулю.

Правило принятия решения приемником элементарно

$$v > v_{threshold} \to 1,$$

$$v < v_{threshold} \to 0.$$

В случае равновероятно передаваемых битов величина

$$v_{threshold} = \frac{v_c(0) + v_c(1)}{2} = \frac{E_1 - E_0}{2}$$

является оптимальным пороговым уровнем, обеспечивающим минимальную вероятность ошибки. Если мы запишем выражение для этой вероятности

$$p_{\text{ош}} = \int_{v_{threshold}}^{\infty} \frac{1}{\sqrt{2\pi D_v}} e^{-\frac{(x - v_c(0))^2}{2D_v}} dx,$$

и представим результат через стандартную функцию ошибок нормального распределения

$$p_{\text{ош}} = \frac{1}{2} \operatorname{erfc} \frac{v_c(1) - v_c(0)}{2\sqrt{2D_v}},$$

то заметим, что в формулу входит некое отношение

$$q_v = \frac{v_c(1) - v_c(0)}{\sqrt{D_v}},$$

называемое отношением сигнал-шум после обработки. Примечательно, что разность сигнальных компонент равна энергии разностного сигнала, поэтому

$$q_v = \sqrt{\frac{2E_w}{N_0}}.$$

Какова бы ни была энергия информационных сигналов $s_i(t)$, полезно ввести среднюю энергию на один бит

$$E_b = p(0)E_0 + p(1)E_1.$$

При этом если $E_0 = E_1$, то $E_b = E_0 = E_1$. В этом случае энергия разностного сигнала легко[4] выражается через среднюю энергию на один бит

$$E_w = 2E_b(1 - r),$$

где r – коэффициент корреляции двух сигналов

$$r = \frac{1}{\sqrt{E_0 E_1}} \int_0^T s_0(t) s_1(t) dt,$$

поэтому

$$q_v^2 = \frac{4E_b(1 - r)}{N_0}.$$

Отношение сигнал-шум мы возвели в квадрат, после чего оно стало отношением сигнал-шум по мощности; до этого было по напряжению, потому что q_v пропорционально напряжению v на выходе коррелятора.

Отношение правдоподобия (LLR)

Вернемся к LDPC кодам. Выданное коррелятором число v при известных параметрах $v_c(i)$ и D_v определяет меру доверия для бита 0 или бита 1: чем выше v, тем больше вероятность того, что передавался бит 1; чем ниже v, тем выше вероятность того, что передавался бит 0. Напомним, что биты передаются равновероятно, но то – совершенно другие вероятности: доопытные, априорные. Понятное дело, что если $v = v_{threshold}$, то вероятности для обоих битов равны по $1/2$ – самая неопределенная ситуация при приеме.

Отношение мер доверия для битов 0 и 1 называют отношением правдоподобия γ, которое сравнивается с единицей. Так как мы знаем закон распределения $w_i(v)$ величины v, то можем найти выражение для отношения правдоподобия в явном виде[5]

$$\gamma = \frac{p(1)}{p(0)} = \frac{w_1(v)}{w_0(v)} = e^{\frac{E_w}{D_v}(v - v_{threshold})}.$$

---

[4] Правда, осознать, что полученное в итоге выражение для q_v^2 работает для любых E_0 и E_1 является еще той головоломкой…

[5] Здесь полезно помнить о пропорциональности некоторой плотности вероятности $w(x)$ величине вероятности $p(x_0) \approx w(x_0)dx$ попадания в узкий (бесконечно малый) интервал dx

Для удобства вычислений берут логарифм отношения правдоподобия (LLR, Log-Likelihood Ratio)

$$L = \ln\gamma = \frac{E_w}{D_v}(v - v_{threshold}) = q_v^2 \frac{v - v_{threshold}}{v_c(1) - v_c(0)}.$$

Так как сумма вероятностей должна быть равна единице, то в итоге

$$p(0) = \frac{1}{e^L + 1},$$

$$p(1) = \frac{e^L}{e^L + 1}.$$

Данные вероятности удобно выразить через гиперболический тангенс

$$p(0) = \frac{1 - \mathrm{th}\frac{L}{2}}{2},$$

$$p(1) = \frac{1 + \mathrm{th}\frac{L}{2}}{2}.$$

Таким образом, мы получили выражения[6], которые по принятому сигналу $v(t)$ позволяют определить число v, а затем и вектор из двух вероятностей

$$\vec{p} = \begin{pmatrix} p(0) \\ p(1) \end{pmatrix},$$

который пойдет на вход LDPC декодера, работающего по алгоритму Sum-Product. Естественно, на вход поступит n векторов – столько, сколько столбцов в проверочной матрице, при этом итоговым выходом декодера будут $k < n$ информационных битов. Заметим, что расчет вероятностей (как часть процесса декодирования) зависит от отношения сигнал-шум q_v, которое в процессе работы реального модема может изменяться со временем, поэтому для данного алгоритма декодирования требуется его постоянная (real-time) оценка, что не является большой проблемой, но добавляет плюс в копилку вычислительных затрат.

[6] Надо сказать, идеальные. При практической реализации приемника их модифицируют из-за стремления снизить вычислительные затраты на гиперболический тангенс – такое вот соотношение между требуемой скоростью передачи и существующим hardware (конечно, влияет также ограничение стоимости hardware для устройств связи массового сегмента рынка)

Основная часть

Суть алгоритма декодирования проще понять на конкретной проверочной матрице (n, k)-кода небольшой длины n

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}.$$

Это проверочная матрица кода Хемминга $(7, 4)$. Она содержит три строки, которые соответствуют трем правилам проверки на четность принятого вектора $\vec{v}$

$$\vec{c} = \vec{v}H^T.$$

Вектор, который состоит из результатов всех проверок на четность, называется синдромом. Если все элементы синдрома равны нулю, то принятый вектор $\vec{v}$ принадлежит линейному коду C. Если вектор принадлежит коду, то либо вектор принят без ошибок, либо ошибка такая, что не может быть обнаружена: любой линейный (n, k)-код ограничен по количеству обнаруживаемых ошибок величиной $2^n - 2^k$, что довольно-таки неплохо при больших n и не слишком больших k.

Для каждой строки проверочной матрицы строится узел (блок), входом которого является набор проверяемых символов с индексами, соответствующими позициям единиц текущей строки, а выходом – результат проверки на четность (элемент вектора синдрома), Рисунок 1.

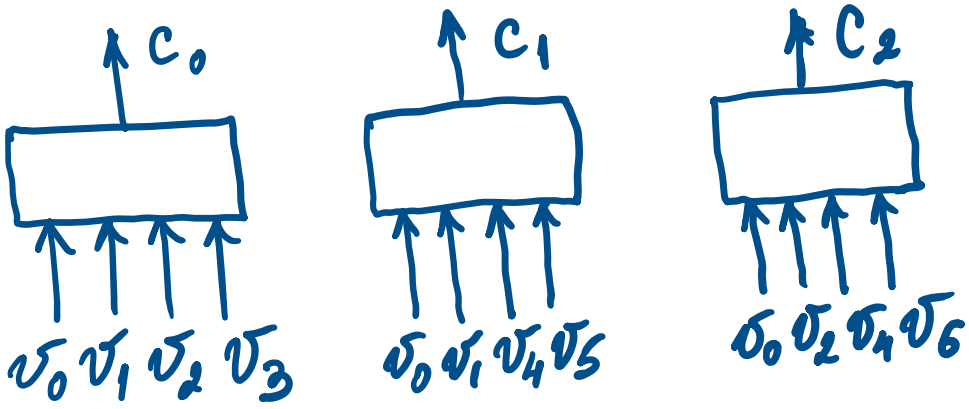

Рисунок 1 Проверочные узлы для кода Хемминга $(7, 4)$

Данным узлам логически соответствует блок «Исключающее ИЛИ» (XOR)

$$c_0 = v_0 + v_1 + v_2 + v_3,$$
$$c_1 = v_0 + v_1 + v_4 + v_5,$$
$$c_2 = v_0 + v_2 + v_4 + v_6.$$

Здесь и далее вместо значка $\oplus$ будем использовать знак обычной суммы, подразумевая, что если входы и выход – двоичные, то этому соответствует операция XOR (сумма по модулю два).

Далее, входные элементы (биты вектора $\vec{v}$) упорядочиваются по индексу и отображаются в виде узлов, но уже, естественно, не узлов XOR, а узлов типа «Равно», которым соответствует операция копирования, в результате чего формируется так называемый проверочный граф, Рисунок 2.

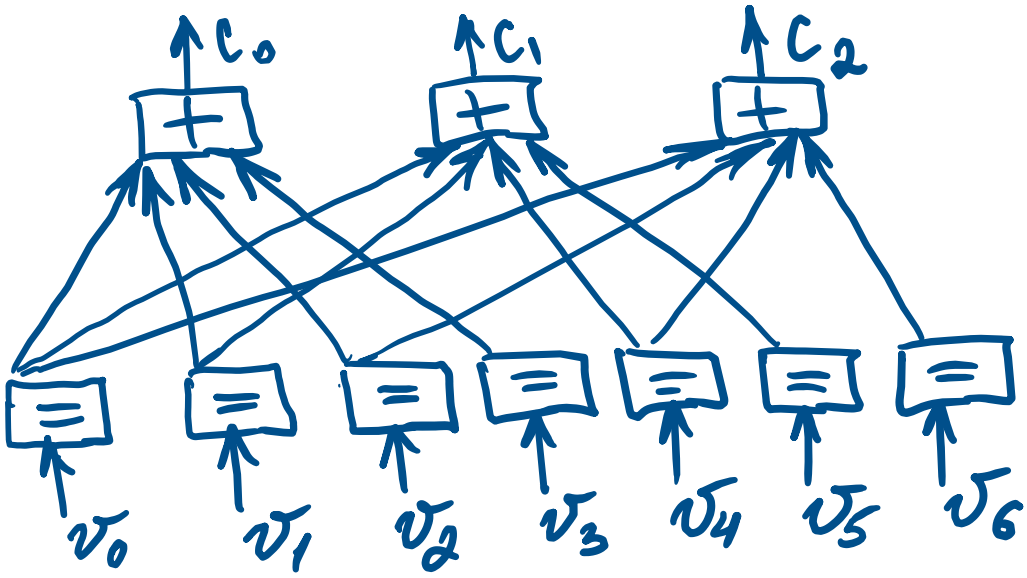

Рисунок 2 Проверочный граф для кода Хемминга (7, 4)

Сопоставляя каждому элементу вектора $\vec{v}$ введенный ранее вектор вероятностей $\vec{p}$ и используя введенное ранее расширение операции XOR на вероятности, можно организовать вычислительную процедуру. При этом блоки «Равно» в случае $1{:}\,N$ работают как копировальщики, а в случае $N{:}\,1$ – как «Логическое И», потому что, например, рассматривая случай $2{:}\,1$, можно составить следующую таблицу истинности, ориентируясь исключительно на логику «Равно»

$$00 \to 0.$$
$$11 \to 1.$$

Заметим, что комбинации 01 и 10 на входе блока «Равно» запрещены по смыслу равенства входов, поэтому логика ИЛИ здесь исключена. Тогда выходные вероятности для логики $c = a = b$ определятся следующим образом

$$p_c(0) = \frac{p_a(0)p_b(0)}{p_a(0)p_b(0) + p_a(1)p_b(1)},$$

$$p_c(1) = \frac{p_a(1)p_b(1)}{p_a(0)p_b(0) + p_a(1)p_b(1)}.$$

Конечно, здесь была введена нормировка, так что $p_c(0) + p_c(1) = 1$.

Организовывая вычислительную процедуру, следует помнить, что вероятности для всех элементов синдрома должны соответствовать идеальным нулям (так, как требует любой линейный код)

$$\vec{p}_{cyndrom} = \binom{1}{0}.$$

Это и есть требование, благодаря которому вычислительная процедура позволяет исправлять ошибки. Фактически, вероятности на выходах блоков XOR зафиксированы и не подлежат изменению, поэтому чтобы лучше понять суть вычислительной процедуры, следует прибегнуть к небольшой хитрости, ведь если выходные значения известны, то что вычислять и где, собственно, входы и где выход? Вычислительная процедура по своей природе итеративная, поэтому входы и выходы от итерации к итерации меняются местами, являясь по факту динамическими, и значит ориентированность графа стрелочками (Рисунок 2) не до конца отражает суть декодирования. На самом деле проверочный граф рисуют без стрелочек, а точки входа-выхода именуют портами.

Ориентируясь на Рисунок 2, опишем вычислительную процедуру декодирования. Сначала входами являются входы блоков «Равно», на выходах которых путем простого копирования формируются векторы вероятностей (идем снизу-вверх). Далее в качестве выходного выбирается один порт блока XOR (например, первый), а под входы определяют оставшиеся порты, причем синдромный порт всегда будет входным. Определив все входы и один выход вычисляют выходной вектор вероятностей, при этом независимо

(параллельно) выполняют ту же процедуру для всех других портов (на рисунке – нижних) текущего блока XOR. Данные «портовые» операции выполняются для всех блоков XOR независимо (параллельно). Таким образом, в результате формируются «обратные» векторы вероятностей, которые можно назвать «отраженными» от блоков XOR. Обратные векторы пропускаются через блоки «Равно» (идем сверху-вниз), используя поэлементное умножение входных вероятностей и последующую нормировку результата, после чего итоговые вероятности как бы встречаются с теми, которые были на этих (самых нижних) портах до выполнения вычислительной итерации. В этих точках схода пары векторов объединяются в один, где первый вектор является доопытным (априорным), второй – послеопытным (апостериорным), при этом опыт заключается в выполнении одной итерации. Объединение векторов осуществляется путем поэлементного перемножения вероятностей и последующей нормировки результата – аналогично логике блока «Равно». Наконец, к итоговым векторам вероятностей применяется скалярная функция, возвращающая индекс элемента с максимальной вероятностью, argmax(), в результате чего каждый вектор превращается в бит, что и является результатом декодирования (точнее, результатом коррекции принятого вектора, ведь скорректированный вектор необходимо декодировать в информационный, см. далее).

Заметим, что мы описали одну итерацию алгоритма Sum-Product. В действительности итераций может быть несколько, когда обновленные вектора вероятностей поступают на входы блоков «Равно» и проходят весь вычислительный цикл (вверх-вниз), и так далее. Моделирование показывает, что, в целом, увеличение количества итераций идет на пользу достоверности приема, однако, увеличивает время декодирования. Одним из способов определения момента останова итераций является равенство синдрома нулю – в этом случае сделанное количество итераций будет «плавать» от раза к разу (variable iteration number), зато выданный вектор всегда будет принадлежать коду, но, правда, в том случае, если декодер дождется-таки нулевого синдрома

– вероятно, в некоторых случаях возможен вечный цикл и тогда следует ограничить количество итераций некоторым заранее определенным (условно большим) числом. Если выставить фиксированное число итераций (fixed iteration number), то иногда будет выдаваться вектор, не принадлежащий коду, что потребует вычисления синдрома после сделанных итераций – если синдром будет отличен от нуля, то следует отказаться от декодирования, ибо обнаруженная ошибка не может быть исправлена данным алгоритмом.

Алгоритм Sum-Product для двоичных линейных (n, k)-кодов может быть формализован так:

1. С выхода коррелятора последовательно снимаются n вещественных отсчетов v_i, которые соответствуют (синхронно по времени) переданному кодовому вектору $\vec{s}$, состоящему из n битов
2. Отсчеты v_i пересчитываются в n векторов вероятностей $\vec{p}_i$
3. Векторы вероятностей $\vec{p}_i$ пропускаются через блоки «Равно», формируя тем самым векторы $\vec{q}_i$, снизу-вверх
4. По векторам вероятностей $\vec{q}_i$ рассчитываются отраженные векторы вероятностей $\vec{q}_{\text{отр}\,i}$, отражение
5. Отраженные векторы $\vec{q}_{\text{отр}\,i}$ пропускаются через блоки «Равно», формируя векторы $\vec{p}_{\text{отр}\,i}$, сверху-вниз
6. Векторы вероятностей $\vec{p}_{\text{отр}\,i}$ объединяются с первоначальными векторами вероятностей $\vec{p}_i \leftarrow \text{Eq}(\vec{p}_{\text{отр}\,i}, \vec{p}_i)$, виртуальные блоки «Равно» в режиме $2{:}\,1$
7. Итоговые векторы вероятностей $\vec{p}_i$ пропускаются через скалярную функцию, возвращающую индекс наибольшего элемента, формируя вектор $\hat{\vec{s}} = \text{argmax}\,\vec{p}_i$, детектирование битов
8. В соответствии с выбранным критерием либо выполняется еще одна итерация (переход к п.3), либо процесс останавливается и декодер выдает вектор $\hat{\vec{s}}$, являющийся оценкой переданного (истинного) кодового вектора $\vec{s}$.

Таким образом, можно заметить, что ядро процесса декодирования содержит два блока (устройства): многовходовый блок XOR и многовходовый блок «Равно». Используя англоязычную аббревиатуру, к блоку «Равно» выгодно применить обозначение EQ. Что касается блока пересчета уровней с выхода коррелятора в вектор вероятностей, а также детектора битов argmax(), то их логично отнести к интерфейсной части декодера: входной и выходной соответственно, Рисунок 3.

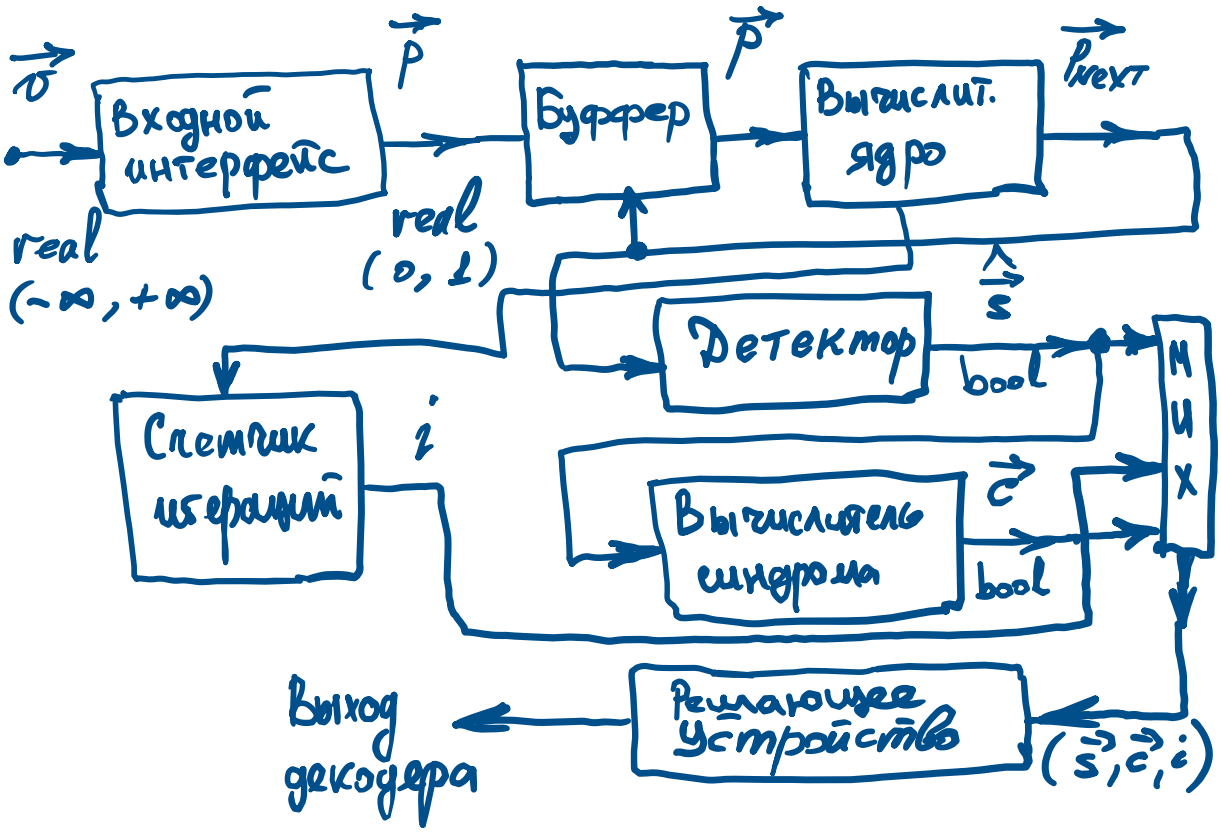

Рисунок 3 Структурная схема итеративного декодера LDPC кода с вычислительным ядром Sum-Product

Выходной интерфейс состоит из детектора, вычислителя синдрома, мультиплексора MUX, который объединяет три компонента в один кортеж (набор), а также из решающего устройства, которое принимает решение о возможности декодирования и выдает оценку кодового вектора $\hat{\vec{s}}$. Если синдром равен нулю, то декодирование возможно, в противном случае идет отказ от декодирования – ошибка не исправлена. Преобразование вектора $\hat{\vec{s}}$ в исходный информационный вектор $\vec{a}$ может быть осуществлено любым

способом, основанным на общем правиле кодирования для линейных (n, k)-кодов

$$\vec{s} = \vec{a}G.$$

Например, в порождающей матрице G можно заранее найти k линейно независимых столбцов и сформировать из них квадратную матрицу, соответственно укоротив при этом кодовый вектор $\hat{\vec{s}}$. Правило укорочения следует запомнить. После чего следует отыскать (путем компьютерного моделирования, используя датчик случайных чисел) и запомнить такой набор эквивалентных преобразований этой квадратной матрицы, что она обращается в единичную матрицу. В итоге, в декодере к укороченному вектору $\hat{\vec{s}}$ следует последовательно применить правило укорочения и набор преобразований (попарный XOR элементов, плюс итоговая их перестановка), в результате чего сформируется искомый вектор $\vec{a}$.

Реализация мягкого декодера на примере NumPy/Python

Все основные внутренние операции декодера будем делать в формате числа $float64$ библиотеки NumPy, в том числе и операции с битами.

Вспомогательные функции

Рассмотрим реализацию блока EQ. На его вход поступает вектор из m вероятностей $\vec{p}_i$, на выходе формируется вектор по правилу

$$\vec{p}_{EQ} = \frac{1}{\prod_{i=0}^{m-1} p_i(0) + \prod_{i=0}^{m-1} p_i(1)} \begin{pmatrix} \prod_{i=0}^{m-1} p_i(0) \\ \prod_{i=0}^{m-1} p_i(1) \end{pmatrix}.$$

Набор из m векторов $\vec{p}_i$ организуем в виде массива NumPy размером $shape = (m, 2)$, то есть матрицы из m строк и двух столбцов. Правило обработки оформим в виде функции

```
def eq(p):
    out = p.prod(axis=0, keepdims=False)
    return out / np.sum(out)
```

Здесь функция prod() отвечает за поэлементное произведение векторов-строк матрицы. Параметр $axis = 0$ говорит о взятии произведения по первой размерности матрицы (у нас матрица двухмерная), а параметр $keepdims = False$ позволяет понизить размерность произведения, в нашем случае с двух до одного – на выходе формируется вектор из двух элементов, то есть размером $shape = (2,)$

```
eq(np.array([[0.2, 0.8], [0.4, 0.6], [0.7, 0.3]]))
array([0.28, 0.72])
```

Видим, что сумма выходных вероятностей равна единице.

С реализацией блока XOR намного интереснее, ведь требуется вычислить сумму всех таких произведений, в которые вероятность для бита 1 входит четное число раз – так мы вычисляем выходную вероятность для бита 0, ведь 0 при логической операции XOR будет только при четном числе единиц на входе

$$p_{XOR}(0) = \sum_{\vec{b}} \prod_{i} p_i(b_i),$$

$$b_i = \{0, 1\},$$

$$\left(\sum_{i} b_i\right) \bmod 2 = 0.$$

Поэтому организуем сначала генератор всех четных двоичных слов (имеется в виду с четным весом Хемминга) заданного размера m. Генерацию таких слов можно осуществить с помощью генераторной матрицы простейшего кода с одной проверкой на четность

$$G = (I^{(m-1)}, \quad \mathbf{1}),$$

где I^m – единичная матрица размером $m \times m$, $\mathbf{1}$ – вектор-столбец из одних единиц

```
def init_gen(m):
    return np.hstack((np.eye(m-1), np.ones((m-1,1))))
```

Здесь функция eye() возвращает единичную матрицу заданного размера, а функция ones() – единичный вектор-столбец заданного размера. Функцией же hstack(,) они объединяются в одну матрицу («стек» по горизонтали). Сам блок XOR можно реализовать следующим образом

```
def xor(p, g):
    m = p.shape[0]
    out = np.array([0., 0.])
    for i in range(2**(m-1)):
        a = int_to_bit_array(i, m)
        s = vm_xor(a, g) # слово с четным весом Хемминга
        out[0] += product_selective(p, s)
    out[1] = 1 - out[0]
    return out
```

Здесь функция

```
def int_to_bit_array(x, n):
    return np.fromiter(format(x, "b").zfill(n), dtype = np.float64)
```

конвертирует целое положительное число x в битовый вектор размером n

```
int_to_bit_array(11, 6)
array([0., 0., 1., 0., 1., 1.])
```

Функция

```
def vm_xor(a, g):
    return np.round(np.sum(a[:, np.newaxis] * g, axis=0, keepdims=True)) % 2
```

перемножает вектор a на матрицу g и приводит результат «по модулю два»

```
a = array([0., 0., 1., 0., 1., 1.])
g = array([[1., 0., 0., 0., 0., 0., 1.],
       [0., 1., 0., 0., 0., 0., 1.],
       [0., 0., 1., 0., 0., 0., 1.],
       [0., 0., 0., 1., 0., 0., 1.],
       [0., 0., 0., 0., 1., 0., 1.],
       [0., 0., 0., 0., 0., 1., 1.]])
vm_xor(a, g)
array([[0., 0., 1., 0., 1., 1., 1.]])
```

причем сохраняя размерность матрицы g, в нашем случае – два. Фактически, результат такого умножения – это XOR тех строк матрицы, которым соответствуют единицы вектора a (в нашем случае это 3-я, 5-я и 6-я строки).

Интересной является функция

```
def product_selective(p, s):
    return np.prod(np.sum(p * np.vstack((1-s,s)).T, axis=1))
```

которая из каждой строки матрицы p выбирает $p(0)$ или $p(1)$ в зависимости от соответствующего значения флага s и перемножает результат выбора

```
product_selective( np.array([[0.2, 0.8], [0.4, 0.6], [0.5, 0.5], [0.7,
0.3]]), np.array([[1, 0, 0, 1]]) )
0.04800000000000001

0.8*0.4*0.5*0.3
0.04800000000000001
```

Здесь поэлементно перемножаются две матрицы размером $(m, 2)$, затем вычисляется сумма результирующих столбцов (на что указывает $axis = 1$), и, наконец, вычисляется произведение элементов столбца.

Пример работы функции XOR

```
xor(np.array([[0.2, 0.8], [0.1, 0.9], [0.1, 0.9], [0.7, 0.3]]), G)
array([0.4232, 0.5768])
```

для матрицы

```
G = array([[1., 0., 0., 1.],
       [0., 1., 0., 1.],
       [0., 0., 1., 1.]])
```

Вычисление вероятности нуля для приведенного примера выглядит следующим образом

$$p(0) = 0.2 \cdot 0.1 \cdot 0.1 \cdot 0.7 + 0.2 \cdot 0.1 \cdot 0.9 \cdot 0.3 + 0.2 \cdot 0.9 \cdot 0.1 \cdot 0.3 + 0.2 \cdot 0.9 \cdot 0.9 \cdot 0.7 + 0.8 \cdot 0.9 \cdot 0.1 \cdot 0.7 + 0.8 \cdot 0.9 \cdot 0.9 \cdot 0.3 + 0.8 \cdot 0.1 \cdot 0.1 \cdot 0.3 + 0.8 \cdot 0.1 \cdot 0.9 \cdot 0.7 = 0.4232.$$

Таким образом, видно, что алгоритм Sum-Product вычислительно трудоемок: $m \cdot 2^{m-1} - 1$ операций, где m – число единиц в строке проверочной матрицы H кода (сложение и умножение считаем эквивалентными).

Наконец, введем функцию детектора битов по матрице вероятностей p

```
def detect(p):
    return (p > 0.5).astype(np.float64)[:, 1]
```

Для двоичных кодов детектор символов можно реализовать как пороговую функцию с порогом ½

```
detect(np.array([[0.3, 0.7], [0.8, 0.2]]))
array([1., 0.])
```

Здесь в первом векторе побеждает нижняя вероятность, 0.7, во втором – верхняя, 0.8, поэтому детектированные биты – это 1 и 0.

Инициализация декодера

Код инициализации выглядит следующим образом

```
n = 7 # Длина кода
k = 4 # Количество информационных битов
r = n - k # Количество проверочных битов
snr = 5. # Отношение сигнал-шум после обработки (на выходе коррелятора), дБ
m0 = -1. # Уровень для бита 0 (на выходе коррелятора)
m1 = 1. # Уровень для бита 1 (на выходе коррелятора)
v_thr = (m0 + m1) / 2 # Пороговый уровень
rms = np.abs(m1 - m0) / np.power(10, 0.05*snr)
m = 4 # Число входов сумматора XOR
G = init_gen(m)
```

Здесь rms – среднеквадратическое отклонение шума.

Интерфейсная часть декодера

Источник битов и интерфейсную часть между гауссовским каналом с непрерывным выходом и входом декодера реализуем так

```
b_transmit = np.array([1]*n, dtype=np.float64) # Все единицы или все нули принадлежат кодам Хемминга
v = np.random.normal((m1-m0)*b_transmit + m0, rms, n) # Мягкие биты с выхода коррелятора
x = np.power(10, 0.1*snr) * (v - v_thr) / (m1 - m0)
p = np.column_stack((0.5 - 0.5*np.tanh(x/2), 0.5 + 0.5*np.tanh(x/2)))
```

Здесь передается кодовый вектор из одних единиц – такая комбинация всегда принадлежит кодам Хемминга. Для линейных кодов, к которым относится код Хемминга, нет разницы какой кодовый вектор был передан – вероятность ошибки после декодирования не зависит от переданного вектора. Однако, в практических схемах, особенно при их отладке, следует передавать разные кодовые векторы – для этого необходима порождающая матрица кода.

Вычислительное ядро Sum-Product

Реализуем основную часть декодера: вычислительное ядро для рассматриваемого кода Хемминга. Итерации организованы циклом *while*

```
pp = [np.zeros((m,2)) for _ in range(r)]
ppr = [np.zeros((m,2)) for _ in range(r)]
buff = [np.zeros((m,2)) for _ in range(r)]
pz = np.array([1., 0.])
t = [(0,1,2,3), (0,1,4,5), (0,2,4,6)]
q = [[(0,0),(1,0),(2,0)], [(0,1),(1,1)], [(0,2),(2,1)], [(0,3)],
[(1,2),(2,2)], [(1,3)], [(2,3)]]
iter_count = 0
while True:
    iter_count += 1
    for i in range(r):
        np.take(p, t[i], axis=0, out=pp[i])
        np.copyto(buff[i], pp[i])
        for j in range(m):
            buff[i][j] = pz
            ppr[i][j] = xor(buff[i], G)
            buff[i][j] = pp[i][j]
    for i in range(n):
        tmp = [p[i]]
        for j,k in q[i]:
            tmp.append(ppr[j][k])
        p[i] = eq(np.vstack(tmp))
    b = detect(p)
    cyndrom = 0
    for i in range(r):
        cyndrom += sum(b[list(t[i])]) % 2
    if not cyndrom:
        break
```

Условием выхода из цикла является равенство синдрома нулю – при этом кодовый вектор b будет принадлежать рассматриваемому коду. Здесь для упрощения выбран синдром-число (вообще, синдром – это вектор, но для проверки его равенства на нуль элементы вектора можно сложить). Вызов функции $\mathrm{take}(,,)$ копирует строки матрицы p в матрицу pp – те строки, на которые указывает вектор t. Здесь параметр r – это число блоков XOR, и для удобства организован цикл for по узлам XOR. Функция $\mathrm{copyto}(,)$ необходима для копирования значений матрицы pp в матрицу-буфер $buff$. Внутренний цикл for по переменной j позволяет вставлять в буфер константу pz и проводить вычисления блоком XOR, формируя матрицу «отраженных» вероятностей ppr. Заметим, что константа pz в буфере всегда в одном экземпляре, при перемещении константы с одной строки на другую значение буфера в строке восстанавливается (см. третью строку в цикле). В следующем цикле for (по i n раз) для каждого i формируется набор из «отраженных» вероятностей по правилу, которое указано в списке $q[i]$. В кортежах (j,k) указаны индекс j узла XOR и индекс порта k выбранного узла XOR. Блоков XOR, напомним, три, а число портов – четыре на каждый блок, Рисунок 2. Количество кортежей в списке $q[i]$ определяется количеством линий (отводов) в i-м блоке «Равно». Отобранные вероятности для текущего блока «Равно» вместе с предыдущей (входной) вероятностью $p[i]$ обрабатываются этим блоком, перезаписывая вероятность $p[i]$ новым (обновленным) значением – тем самым эта вероятность при необходимости пойдет на следующую итерацию.

Заметим, что список q полностью определяется списком t, который полностью определяется проверочной матрицей кода, однако, для удобства оба списка введены в код независимо. Отличным упражнением будет осознание логики построения q по известному t.

Жесткий вариант алгоритма Sum-Product

Изучив реализацию мягкого (как более общего) алгоритма Sum-Product, приступим к его упрощению до жесткого варианта, когда входные вероятности округляются до вещественных 0 или 1. Базовым для нас является блок XOR, поэтому на примере двухвходового блока XOR посмотрим на его реакцию на идеальные (краевые) вероятности. Для наглядности операцию XOR обозначим как положено, а вещественность чисел подчеркнем точкой

$$\binom{0.}{1.} \oplus \binom{0.}{1.} = \binom{1.}{0.},$$

$$\binom{1.}{0.} \oplus \binom{1.}{0.} = \binom{1.}{0.},$$

$$\binom{1.}{0.} \oplus \binom{0.}{1.} = \binom{0.}{1.},$$

$$\binom{0.}{1.} \oplus \binom{1.}{0.} = \binom{0.}{1.}.$$

Все как и положено для двоичной операции XOR, однако работа двухвходового блока «Равно» не так очевидна (операцию «равно» обозначим символом ≡)

$$\binom{0.}{1.} \equiv \binom{0.}{1.} = \binom{0.}{1.},$$

$$\binom{1.}{0.} \equiv \binom{1.}{0.} = \binom{1.}{0.},$$

$$\binom{1.}{0.} \equiv \binom{0.}{1.} = \binom{0.5}{0.5},$$

$$\binom{0.}{1.} \equiv \binom{1.}{0.} = \binom{0.5}{0.5}.$$

Два последних выражения говорят о неопределенной ситуации, когда на входы блока «Равно» поступают противоположные биты – выходные вероятности при этом равны по $1/2$. Рассмотрим, однако, работу трехвходового блока «Равно»

$$\binom{0.}{1.} \equiv \binom{0.}{1.} \equiv \binom{1.}{0.} = \binom{0.}{1.},$$

$$\binom{0.}{1.} \equiv \binom{1.}{0.} \equiv \binom{1.}{0.} = \binom{1.}{0.}.$$

Заметим, что нет разницы в каком порядке следуют аргументы, потому что этой операции соответствует поэлементное перемножение вероятностей, результат которого не зависит от порядка следования сомножителей. Если мы напрямую перемножим вероятности, то результат будет нулевой. Чтобы этого избежать, на вход полноценного блока «Равно» следует подать вероятности, немного отличающиеся от идеальных

```
eq(np.array([[0.01, 0.99], [0.99, 0.01], [0.99, 0.01]]))
array([0.99, 0.01])
```

после чего результат следует округлить обратно. Комбинируя таким способом разные варианты входов, можно убедиться, что блок «Равно» в жестком варианте работает как мажорирующий блок, работающий по правилу большинства: больше нулей на входе, выдаем ноль, больше единиц – единицу. Самое интересное, когда число нулей совпадает с числом единиц, то результат получается неопределенным, и кодируется следующим вектором вероятностей

$$\binom{0.5}{0.5},$$

с чем мы уже сталкивались выше – такой исход говорит о необходимости либо отказа от декодирования, либо инвертирования соответствующего канального бита и проверке синдрома на равенство нулю. Если после инвертирования синдром окажется отличен от нуля, то обнаруженная ошибка должна считаться неисправимой и должен следовать отказ от декодирования.

Рассмотрим процесс исправления однократной ошибки с помощью жесткого варианта алгоритма Sum-Product. Пусть принят вектор (1 1 1 1 1 0 1), тогда для первого узла XOR (биты с индексами 0, 1, 2 и 3), Рисунок 1, имеем следующие 4 операции (напомним, что синдромный «нолик» гуляет по диагонали слева-направо)

$$0 + 1 + 1 + 1 = 1,$$
$$1 + 0 + 1 + 1 = 1,$$
$$1 + 1 + 0 + 1 = 1,$$
$$1 + 1 + 1 + 0 = 1.$$

Для второго узла XOR (биты с индексами 0, 1, 4 и 5) следующие

$$0 + 1 + 1 + 0 = 0,$$

$$1 + 0 + 1 + 0 = 0,$$

$$1 + 1 + 0 + 0 = 0,$$

$$1 + 1 + 1 + 0 = 1.$$

Для третьего узла XOR (биты с индексами 0, 2, 4 и 6)

$$0 + 1 + 1 + 1 = 1,$$

$$1 + 0 + 1 + 1 = 1,$$

$$1 + 1 + 0 + 1 = 1,$$

$$1 + 1 + 1 + 0 = 1.$$

Далее по блокам «Равно» имеем 7 мажорирующих функций (напомним, что первые биты здесь являются канальными)

$$s_0 = \text{major}(1, 1, 0, 1) = 1$$

$$s_1 = \text{major}(1, 1, 0) = 1$$

$$s_2 = \text{major}(1, 1, 1) = 1$$

$$s_3 = \text{major}(1, 1) = 1$$

$$s_4 = \text{major}(1, 0, 1) = 1$$

$$s_5 = \text{major}(0, 1) = ? = \text{NOT}\, 0 = 1$$

$$s_6 = \text{major}(1, 1) = 1$$

Получили неопределенность (из канала пришел бит 0, а после итерации декодер говорит, что должен быть бит 1), и в данной ситуации мы меняем канальный бит на противоположный, после чего вычисляем синдром и сравниваем его с нулевым синдромом. Здесь при инвертировании бита (т. е. при коррекции ошибки) получим нулевой синдром и правильный кодовый вектор.

Пример работы мягкого варианта алгоритма Sum-Product

Продолжим рассматривать код Хемминга $(7, 4)$. Пусть передан кодовый вектор (1 1 1 1 1 1 1), а соотношение сигнал-шум составляет $+5$ дБ. Подберем

наиболее интересный случай, когда декодер успешно исправляет ошибку. На входе декодера формируются 7 пар вероятностей

```
input decoder: p = [[0.27748508 0.72251492]
 [0.01851341 0.98148659]
 [0.55883163 0.44116837]
 [0.32563095 0.67436905]
 [0.58052793 0.41947207]
 [0.15080836 0.84919164]
 [0.03952914 0.96047086]]
```

Если декодирование выключить, то данным вероятностям соответствует кодовый вектор (1 1 0 1 0 1 1), который отличается от переданного двумя битами; таким образом, при жесткой демодуляции имеем двукратную ошибку, и соответствующий декодер кода Хемминга ее не исправит, потому что коды Хемминга рассчитаны на исправление однократных ошибок.

Рассмотрим первую итерацию декодирования. Имеем 3 узла XOR и 4 так называемых стадии. Для каждого узла XOR из входных пар p выбирается определенная четверка пар вероятностей. Для каждой стадии вектор $\binom{1.}{0.}$ помещается в четверку пар на позицию, соответствующую индексу стадии. На каждой стадии вычисляется одна пара вероятностей (как XOR четырех пар), и, таким образом, формируются $4 \cdot 3 = 12$ пар вероятностей, которые затем распределяются по узлам «Равно», при этом в качестве первой пары на входах узлов «Равно» стоят исходные пары вероятностей p, поэтому общее число пар по всем узлам «Равно» составляет $12 + 7 = 19$. Сказанное иллюстрируется следующим выводом:

```
iteration 1
 XOR node 0: take from (0, 1, 2, 3) indexies: [[0.27748508 0.72251492]
[0.01851341 0.98148659]
[0.55883163 0.44116837]
[0.32563095 0.67436905]]
  stage 0: input XOR:  [[1.         0.        ]
[0.01851341 0.98148659]
[0.55883163 0.44116837]
[0.32563095 0.67436905]]
             output XOR: [0.51975716 0.48024284]
  stage 1: input XOR:  [[0.27748508 0.72251492]
[1.         0.        ]
[0.55883163 0.44116837]
[0.32563095 0.67436905]]
             output XOR: [0.5091306 0.4908694]
  stage 2: input XOR:  [[0.27748508 0.72251492]
[0.01851341 0.98148659]
```

```
[1.         0.        ]
[0.32563095 0.67436905]]
             output XOR: [0.42527383 0.57472617]
  stage 3: input XOR:  [[0.27748508 0.72251492]
[0.01851341 0.98148659]
[0.55883163 0.44116837]
[1.         0.        ]]
             output XOR: [0.5252124 0.4747876]
 XOR node 1: take from (0, 1, 4, 5) indexies: [[0.27748508 0.72251492]
[0.01851341 0.98148659]
[0.58052793 0.41947207]
[0.15080836 0.84919164]]
  stage 0: input XOR:  [[1.         0.        ]
[0.01851341 0.98148659]
[0.58052793 0.41947207]
[0.15080836 0.84919164]]
             output XOR: [0.554157 0.445843]
  stage 1: input XOR:  [[0.27748508 0.72251492]
[1.         0.        ]
[0.58052793 0.41947207]
[0.15080836 0.84919164]]
             output XOR: [0.52502819 0.47497181]
  stage 2: input XOR:  [[0.27748508 0.72251492]
[0.01851341 0.98148659]
[1.         0.        ]
[0.15080836 0.84919164]]
             output XOR: [0.3503533 0.6496467]
  stage 3: input XOR:  [[0.27748508 0.72251492]
[0.01851341 0.98148659]
[0.58052793 0.41947207]
[1.         0.        ]]
             output XOR: [0.53451039 0.46548961]
 XOR node 2: take from (0, 2, 4, 6) indexies: [[0.27748508 0.72251492]
[0.55883163 0.44116837]
[0.58052793 0.41947207]
[0.03952914 0.96047086]]
  stage 0: input XOR:  [[1.         0.        ]
[0.55883163 0.44116837]
[0.58052793 0.41947207]
[0.03952914 0.96047086]]
             output XOR: [0.49127391 0.50872609]
  stage 1: input XOR:  [[0.27748508 0.72251492]
[1.         0.        ]
[0.58052793 0.41947207]
[0.03952914 0.96047086]]
             output XOR: [0.53300409 0.46699591]
  stage 2: input XOR:  [[0.27748508 0.72251492]
[0.55883163 0.44116837]
[1.         0.        ]
[0.03952914 0.96047086]]
             output XOR: [0.52411194 0.47588806]
  stage 3: input XOR:  [[0.27748508 0.72251492]
[0.55883163 0.44116837]
[0.58052793 0.41947207]
[1.         0.        ]]
             output XOR: [0.49578326 0.50421674]
  equal node 0: inputs [[0.27748508 0.72251492]
[0.51975716 0.48024284]
[0.554157   0.445843  ]
```

```
[0.49127391 0.50872609]]
                outputs [0.33284892 0.66715108]
  equal node 1: inputs [[0.01851341 0.98148659]
[0.5091306  0.4908694 ]
[0.52502819 0.47497181]]
                outputs [0.0211684 0.9788316]
  equal node 2: inputs [[0.55883163 0.44116837]
[0.42527383 0.57472617]
[0.53300409 0.46699591]]
                outputs [0.51686106 0.48313894]
  equal node 3: inputs [[0.32563095 0.67436905]
[0.5252124  0.4747876 ]]
                outputs [0.34817347 0.65182653]
  equal node 4: inputs [[0.58052793 0.41947207]
[0.3503533  0.6496467 ]
[0.52411194 0.47588806]]
                outputs [0.45115057 0.54884943]
  equal node 5: inputs [[0.15080836 0.84919164]
[0.53451039 0.46548961]]
                outputs [0.16938198 0.83061802]
  equal node 6: inputs [[0.03952914 0.96047086]
[0.49578326 0.50421674]]
                outputs [0.03889369 0.96110631]
 detection: [1. 1. 0. 1. 1. 1. 1.]
 cyndrom 2.0
```

После детектирования выходных (итоговых) пар вероятностей получаем вектор (1 1 0 1 1 1 1), в котором один ошибочный бит вместо двух, однако, синдром отличен от нуля, и поэтому выполняется еще одна итерация:

```
iteration 2
 XOR node 0: take from (0, 1, 2, 3) indexies: [[0.33284892 0.66715108]
[0.0211684  0.9788316 ]
[0.51686106 0.48313894]
[0.34817347 0.65182653]]
  stage 0: input XOR:  [[1.         0.        ]
[0.0211684  0.9788316 ]
[0.51686106 0.48313894]
[0.34817347 0.65182653]]
             output XOR: [0.50490315 0.49509685]
  stage 1: input XOR:  [[0.33284892 0.66715108]
[1.         0.        ]
[0.51686106 0.48313894]
[0.34817347 0.65182653]]
             output XOR: [0.5017116 0.4982884]
  stage 2: input XOR:  [[0.33284892 0.66715108]
[0.0211684  0.9788316 ]
[1.         0.        ]
[0.34817347 0.65182653]]
             output XOR: [0.45139291 0.54860709]
  stage 3: input XOR:  [[0.33284892 0.66715108]
[0.0211684  0.9788316 ]
[0.51686106 0.48313894]
[1.         0.        ]]
             output XOR: [0.50539805 0.49460195]
 XOR node 1: take from (0, 1, 4, 5) indexies: [[0.33284892 0.66715108]
[0.0211684  0.9788316 ]
[0.45115057 0.54884943]
```

```
[0.16938198 0.83061802]]
  stage 0: input XOR:  [[1.         0.        ]
[0.0211684  0.9788316 ]
[0.45115057 0.54884943]
[0.16938198 0.83061802]]
             output XOR: [0.46906651 0.53093349]
  stage 1: input XOR:  [[0.33284892 0.66715108]
[1.         0.        ]
[0.45115057 0.54884943]
[0.16938198 0.83061802]]
             output XOR: [0.4892017 0.5107983]
  stage 2: input XOR:  [[0.33284892 0.66715108]
[0.0211684  0.9788316 ]
[1.         0.        ]
[0.16938198 0.83061802]]
             output XOR: [0.39415301 0.60584699]
  stage 3: input XOR:  [[0.33284892 0.66715108]
[0.0211684  0.9788316 ]
[0.45115057 0.54884943]
[1.         0.        ]]
             output XOR: [0.48436091 0.51563909]
 XOR node 2: take from (0, 2, 4, 6) indexies: [[0.33284892 0.66715108]
[0.51686106 0.48313894]
[0.45115057 0.54884943]
[0.03889369 0.96110631]]
  stage 0: input XOR:  [[1.         0.        ]
[0.51686106 0.48313894]
[0.45115057 0.54884943]
[0.03889369 0.96110631]]
             output XOR: [0.50151917 0.49848083]
  stage 1: input XOR:  [[0.33284892 0.66715108]
[1.         0.        ]
[0.45115057 0.54884943]
[0.03889369 0.96110631]]
             output XOR: [0.48493983 0.51506017]
  stage 2: input XOR:  [[0.33284892 0.66715108]
[0.51686106 0.48313894]
[1.         0.        ]
[0.03889369 0.96110631]]
             output XOR: [0.50519823 0.49480177]
  stage 3: input XOR:  [[0.33284892 0.66715108]
[0.51686106 0.48313894]
[0.45115057 0.54884943]
[1.         0.        ]]
             output XOR: [0.5005507 0.4994493]
  equal node 2: inputs [[0.33284892 0.66715108]
[0.50490315 0.49509685]
[0.46906651 0.53093349]
[0.50151917 0.49848083]]
                outputs [0.3114112 0.6885888]
  equal node 1: inputs [[0.0211684 0.9788316]
[0.5017116 0.4982884]
[0.4892017 0.5107983]]
                outputs [0.02042812 0.97957188]
  equal node 2: inputs [[0.51686106 0.48313894]
[0.45139291 0.54860709]
[0.48493983 0.51506017]]
                outputs [0.45317922 0.54682078]
  equal node 0: inputs [[0.34817347 0.65182653]
```

```
[0.50539805 0.49460195]]
                outputs [0.35308991 0.64691009]
  equal node 2: inputs [[0.45115057 0.54884943]
[0.39415301 0.60584699]
[0.50519823 0.49480177]]
                outputs [0.35317374 0.64682626]
  equal node 1: inputs [[0.16938198 0.83061802]
[0.48436091 0.51563909]]
                outputs [0.16075916 0.83924084]
  equal node 2: inputs [[0.03889369 0.96110631]
[0.5005507  0.4994493 ]]
                outputs [0.03897612 0.96102388]
 detection: [1. 1. 1. 1. 1. 1. 1.]
 cyndrom 0.0
```

После второй итерации синдром равен нулю, значит полученный вектор из 7 битов принадлежит коду (т. е. является разрешенным) и декодирование заканчивается (точнее, заканчивается коррекция кодового вектора, за которой должна идти процедура восстановления информационного вектора). В рассмотренном случае мягкий декодер исправил двукратную ошибку, однако, эксперимент показывает, что не всегда он исправляет двукратные ошибки, к тому же для мягкого декодера нет понятия кратности ошибки, т. к. кратность ошибки – это интерпретация степени искажения на выходе детектора битов.

Если для выбранного отношения сигнал-шум передать достаточно большое число кодовых векторов[7] (в данном случае 10000), то вероятность ошибки после декодирования будет равна 0.157 ± 0.002. Вероятность ошибки до декодирования при этом равна 0.189. Результат, конечно, не впечатляет, но что требовать от короткого кода, да и еще при небольшом отношении сигнал-шум (5 дБ)? Увеличив сигнал-шум до 8 дБ будем иметь вероятности 0.1 и 0.05 соответственно, значит рассматриваемый декодер в текущих условиях понижает частоту ошибок в два раза. Число итераций для последнего случая находится в диапазоне от 1 до 19, причем в 95% случаев число итераций менее 5. Число же итераций для первого случая – от 1 до 56, а 0.95-квантиль составляет 8. Таким образом, при ухудшении качества канала, помимо

[7] Вообще, для оценки некоторой вероятности p с точностью ε (среднеквадратичным отклонением) требуется передать

$$N = \frac{p(1-p)}{n\varepsilon^2}$$

кодовых векторов длиной n бит. Это равенство основано на величине дисперсии биномиального распределения.

снижения достоверности приема, увеличиваются вычислительные затраты на декодирование, и рано или поздно наступает такой момент, когда выбранный код и алгоритм декодирования становятся неэффективными, к тому же в реальном декодере количество итераций, хоть и величина переменная, но ограниченная некоторым числом, задающим допуск на время декодирования и, соответственно, требуемую скорость передачи информации.