Chapter 6
Stochastic Decoders for LDPC Codes

François Leduc-Primeau, Vincent C. Gaudet, and Warren J. Gross

6.1 Introduction

Low density parity check (LDPC) codes have now been established as one of the leading channel codes for approaching the channel capacity in data storage and communication systems. Compared to other codes, they stand out by their ability of being decoded with a message-passing decoding algorithm that offers a large degree of parallelism, which offers interesting possibilities for simultaneously achieving a large coding gain and a high data throughput.

Exploiting all the available parallelism in message-passing decoding is difficult because of the logic area required for replicating processing circuits, but also because of the large number of wires required for exchanging messages. The use of stochastic computing was proposed as a way of achieving highly parallel LDPC decoders with a smaller logic and wiring complexity.

Current decoding algorithms based on stochastic computing do not outperform standard algorithms on all fronts, but they generally offer a significant advantage in average processing throughput normalized to circuit area.

We start this Chapter by providing an overview of LDPC codes. Readers familiar with the topic can safely skip to Sect. 6.3, where we present the stochastic number representation and the concept of stochastic computing. Section 6.4 then describes several LDPC decoding algorithms that perform all their computations using the stochastic representation. It is also possible to use the stochastic representation

F. Leduc-Primeau • W.J. Gross (✉)
McGill University, Montreal, QC, Canada
e-mail: francois.leduc-primeau@mail.mcgill.ca; warren.gross@mcgill.ca

V.C. Gaudet
University of Waterloo, Waterloo, ON, Canada
e-mail: vcgaudet@uwaterloo.ca

C. Chavet, P. Coussy (eds.), *Advanced Hardware Design for Error Correcting Codes*, DOI 10.1007/978-3-319-10569-7_6

for only a subset of the computations. Section 6.5 presents an algorithm that uses that approach and presents methods for efficiently converting numbers between a conventional (deterministic) representation and a stochastic representation. Finally, Sect. 6.6 presents a stochastic approach for decoding non-binary LDPC codes.

6.2 Overview of LDPC Codes

In this section, we review the aspects of LDPC codes that are required to explain the stochastic decoding algorithms. We start in Sect. 6.2.1 by describing the structure of the codes, and Sect. 6.2.2 then presents the standard decoding algorithm, called the Sum–Product algorithm (SPA).

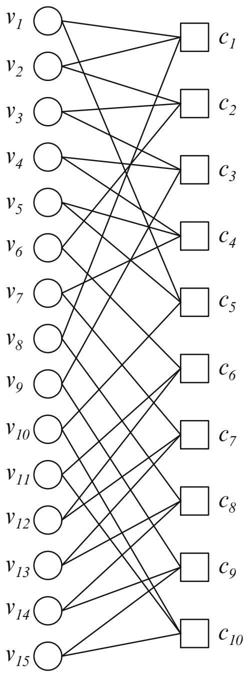
6.2.1 Structure

LDPC codes are part of the family of linear block codes, which are commonly defined using a parity-check matrix H of size $m \times n$. The codewords corresponding to H are the column vectors $\mathbf{x}$ of length n for which $H \cdot \mathbf{x} = \mathbf{0}$, where $\mathbf{0}$ is the zero vector. LDPC codes can be binary or non-binary. For a binary code, the elements of H and $\mathbf{x}$ are from the Galois Field of order 2, or equivalently $H \in \{0,1\}^{m \times n}$ and $\mathbf{x} \in \{0,1\}^n$. Non-binary LDPC codes are defined similarly, but the elements of H and $\mathbf{x}$ are taken from higher order Galois Fields. The rate r of a code expresses the number of information bits contained in the codeword divided by the code length. Assuming H is full rank, we have $r = 1 - \frac{m}{n}$.

A block code can also be equivalently represented as a bipartite graph. We call a node of the first type a *variable* node (VN), and a node of the second type a *check* node (CN). Every row i of H corresponds to a check node c_i, and every column j of H corresponds to a variable node j. An edge exists between c_i and v_j if $H(i,j)$ is non-zero. The two equivalent representations are illustrated in Fig. 6.1.

The key property that distinguishes LDPC codes from other linear block codes is that their parity-check matrix is sparse (or "low density"), in the sense that each row and each column contains a small number of non-zero elements. Furthermore this number does not depend on n. In other words, increasing the code size n also increases the sparsity of H. The number of non-zero elements in a row of H is equal to the number of edges incident on the corresponding check node and is called the check node degree, denoted d_c. Similarly the number of non-zero elements in a column is called the variable node degree and denoted d_v.

When all the rows of H have the same degree d_c and all the columns have the same degree d_v, we say that the code is part of the family of regular (d_v, d_c) codes. Code families are important for analyzing the error correction performance of an LDPC code. When n is sufficiently large, two codes taken from the same family will have the same error correction performance with high probability when decoded



$$\begin{array}{c|ccccccccccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1
\end{array}$$

Fig. 6.1 *Left*: Graphical representation of a short binary LDPC code with $n = 15$ and $m = 10$. *Circles* represent variable nodes and *squares* represent check nodes. *Right*: Parity-check matrix representation of the same code

with the SPA. For a large n it is therefore possible to predict the performance of a code by instead analyzing the average performance of the corresponding family of codes, which is a much simpler task. Families of irregular codes can also be defined by specifying a VN degree distribution and a CN degree distribution. For example, we can define a family of irregular codes with rate $\frac{1}{2}$ by specifying that 40 % of the VNs have degree 3 and 60 % have degree 4, while 20 % of the CNs have degree 12 and 80 % have degree 6. The reader is advised to consult a reference such as [15] for more information on the construction and analysis of LDPC codes.

6.2.2 *Decoding*

LDPC codes can be decoded using a variety of message-passing algorithms that operate by passing messages on the edges of the code graph. These algorithms are interesting because they have a low complexity per bit while also offering a high level of parallelism. If the graph contains no cycles, there exists a message-passing algorithm that yields the maximum-likelihood estimate of each transmitted bit, called the SPA. In practice, all good LDPC codes contain cycles, and in that case the SPA is not guaranteed to generate the optimal estimate of each symbol. Despite this fact, the SPA usually performs very well on graphs with cycles, and experiments have shown that an LDPC code decoded with the SPA can still be used to approach the channel capacity [4]. The SPA can be defined in terms of various likelihood metrics, but when decoding binary codes, the log likelihood ratio (LLR) is preferred because it is better suited to a fixed-point representation and removes the need to perform multiplications. Suppose that p is the probability that the transmitted bit is a 1 (and $1-p$ the probability that it is a 0). The LLR metric Λ_i is defined as

$$\Lambda_i = \ln\left(\frac{1-p}{p}\right).$$

Algorithm 1 describes the SPA for binary codes (the SPA for non-binary codes is described in Sect. 6.6.1). The algorithm takes LLR priors as inputs and outputs an estimate of each codeword bit. If the modulated bits are represented as $x_i \in \{-1,1\}$ and transmitted over the additive white Gaussian noise channel, the LLR priors Λ_i corresponding to each codeword bit $i \in \{1,2,\ldots,n\}$ are obtained from the channel output y_i using

$$\Lambda_i = \frac{-2y_i}{\sigma^2},$$

where σ^2 is the noise variance. The algorithm operates by passing messages on the code graph. We denote a message passed from a variable node i to a check node j as $\eta_{i,j}$, and from a check node j to a variable node i as $\theta_{j,i}$. Furthermore, for each variable node v_i we define a set V_i that contains all the check node neighbors of v_i, and similarly for each check node c_j, we define a set C_j that contains the variable node neighbors of c_j. The computations can be described by two functions: a variable node function $\textsc{var}(S)$ and a check node function $\text{CHK}(S)$, where S is a set containing the function's inputs. If we let $S = \{\Lambda_1, \Lambda_2, \ldots, \Lambda_d\}$, the functions are defined as follows:

$$\textsc{var}(S) = \sum_{i=1}^{d} \Lambda_i \tag{6.1}$$

$$\text{CHK}(S) = \operatorname{arctanh}\left(\prod_{i=1}^{d} \tanh(\Lambda_i)\right). \tag{6.2}$$

The algorithm performs up to L iterations, and stops as soon as the bit estimate vector $\hat{\mathbf{x}}$ forms a valid codeword, that is $H \cdot \hat{\mathbf{x}} = \mathbf{0}$.

```
input : $\{\Lambda_1, \Lambda_2, \ldots, \Lambda_n\}$
output: $\hat{\mathbf{x}} = [\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_n]$
begin
    $\theta_{j,i} \leftarrow 0, \forall i, j$
    for $t \leftarrow 1$ to $L$ do
        for $i \leftarrow 1$ to $n$ do                                  // VN to CN messages
            foreach $j \in V_i$ do
                $\eta_{i,j} \leftarrow \text{VAR}(\{\Lambda_i\} \cup \{\theta_{a,i} : a \in V_i\} \setminus \{\theta_{j,i}\})$
        for $j \leftarrow 1$ to $m$ do                                  // CN to VN messages
            foreach $i \in C_j$ do
                $\theta_{j,i} \leftarrow \text{CHK}(\{\eta_{a,j} : a \in C_j\} \setminus \{\eta_{i,j}\})$
        for $i \leftarrow 1$ to $n$ do                       // Compute the decision vector
            $\Lambda'_i \leftarrow \text{VAR}(\{\Lambda_i\} \cup \{\theta_{a,i} : a \in V_i\})$
            if $\Lambda'_i \geq 0$ then $\hat{x}_i \leftarrow 0$
            else $\hat{x}_i \leftarrow 1$
        Terminate if $\hat{\mathbf{x}}$ is a valid codeword
    Declare a decoding failure
```

Algorithm 1: Sum–Product decoding of an LDPC code using LLR messages

6.3 Stochastic Computing

Stochastic computing was originally studied by Gaines [6] and Poppelbaum [12]. At the time, the motivation was to allow realizing digital circuit implementations of systems that were too complex to be implemented using a conventional number representation, by taking advantage of the fact that many computations can be performed on stochastic streams using very simple circuits. The downside of stochastic computing is that it has a limited precision, but the combination of low complexity and low precision is well suited to a number of applications. For example, stochastic computing has been proposed for the circuit implementation of neural networks [3], which can benefit from massive parallelism and can often tolerate low precision computation. Additionally, with the continuous shrinking of transistors in CMOS integrated circuits, process variations as well as random fluctuations in operating parameters have now become important limiting factors for the speed and energy efficiency of circuits. Because of its random nature, stochastic computing is naturally fault-tolerant, and there has been renewed interest in using it to achieve fault-tolerant circuit implementations [14]. A comprehensive survey of applications for which stochastic computing has been considered can be found in [1].

6.3.1 The Stochastic Stream Representation

To explain stochastic streams, it is useful to make a parallel with analog signals. Let us consider a discretized version of a real-valued analog signal that we denote $a[t]$, for $t \in \{1,2,3,\ldots\}$. A stochastic stream $x[t]$ can be thought of as a random sequence that is a coarse quantization of $a[t]$, but where the randomness is used to ensure that the quantization is unbiased, so that the expected value of $x[t]$ is always $a[t]$. If we denote the expected value of a random variable X as $\mathbb{E}[X]$, this can be written as $\mathbb{E}[x[t]] = a[t]$.

Since the motivation of stochastic computing is reducing the complexity of the computing circuits, stochastic streams are usually binary-valued. In that case they can be communicated using a single wire in the circuit, just like analog signals. A binary stochastic stream is defined as a sequence of independent random variables $x[t]$, distributed such that $\mathbb{E}[x[t]] = a[t]$. Note that since $x[t]$ is binary, its distribution is fully determined by its expected value. Unless otherwise mentioned, the term *stochastic stream* refers to a binary stochastic stream. If we label the two values of a binary stochastic stream as 0 and 1 we have $x[t] \in \{0,1\}$, and therefore the range of values that can be represented is $0 \le \mathbb{E}[x[t]] \le 1$. In order to represent analog signals that are valued over a different range, we can simply map $a[t]$ to a signal $a'[t]$ such that $a'[t] \in [0,1]$. If the mapping is one-to-one, it can be inverted at the output of the computation to restore the initial range. We call the sequence $\mathbb{E}[x[t]]$ the expectation sequence of the stochastic stream $x[t]$.

An important aspect of any number representation is its precision. Suppose that a real number a is represented as $\hat{a}$. The precision can be defined as the worst error of $\hat{a}$ with respect to a. As a point of comparison, we consider the precision of representing a as a fixed-point number, since this is the most common approach in conventional signal processing systems. Suppose that we want to represent a real number $a \in [0,1]$ as an n-bit fixed-point number $\hat{a}$. We obtain $\hat{a}$ using

$$\hat{a} = \frac{\textsc{round}(a \cdot 2^n)}{2^n},$$

where the rounding is to the nearest integer. Since $|\hat{a} - a| \le 1/2^n$, we say that the precision is $1/2^n$.

We can now compare this with the precision of a stochastic stream $x[t]$ where t runs from 1 to n. Let us first consider a simple case where the expectation sequence $a[t]$ does not vary in time, that is $\mathbb{E}[x[t]] = a[t] = c$, with $c \in [0,1]$. The sequence $a[t]$ can be reconstructed from $x[t]$ in an optimal way by using the mean estimator

$$\hat{a}[t] = \frac{1}{t}\sum_{i=1}^{t} x[i]. \tag{6.3}$$

Since the stochastic stream is generated randomly, the precision of a sequence of length n is also random. To simplify, let us assume that the length of the sequence is known in advance, and that the sequence is generated deterministically to get the best precision possible, which is achieved by choosing the sequence such that $\sum_{i=1}^{n} x[i] = \textsc{round}(c \cdot n)$. In this case we have

$$|\hat{a}[t] - a[t]| = \left| \frac{\textsc{round}(c \cdot t)}{t} - c \right| \le \frac{1}{2t}. \tag{6.4}$$

The precision of the stochastic stream improves with time, since the estimation error $|\hat{a}[t] - a[t]|$ goes to 0 for all c, as t goes to infinity. However, to obtain the same precision as a fixed-point representation with n bits, we must wait until $t = 2^{n-1}$. If $n \leq 2$, the two representations have equivalent precision, but as n increases, the stochastic representation needs exponentially more bits to achieve the same precision. Why then use stochastic streams? The hope is that the gains made in the complexity of the computation circuits are more important than the loss in precision. Also, a nice consequence of the randomness of stochastic streams is that the desired precision does not need to be established in advance, but instead can be chosen dynamically by running the computation for a shorter or longer time. Nonetheless, it is clear that stochastic streams are only suitable for computations that require relatively low precision.

If $a[t]$ varies in time, the optimal estimator will be different from (6.3), but it is easy to see that in that case the estimation error depends not only on time but also on the rate of change of $a[t]$ (the smallest error being obtained when $a[t]$ is constant).

6.3.2 Computation Circuits

A stochastic stream $x[t]$ can be generated from a real sequence $a[t] \in [0,1]$ by comparing it with a random threshold T having a uniform distribution over $[0,1]$:

$$x[t] = \begin{cases} 0 & \text{if a[t]} < \text{T}, \\ 1 & \text{otherwise.} \end{cases} \tag{6.5}$$

In some cases, computations can be performed on stochastic streams using very simple circuits. Unless otherwise specified, circuits for stochastic computation are designed by assuming that the input streams are mutually independent, which is the case if they are generated using independent threshold variables. Figure 6.2 shows two basic computing circuits. The first circuit is an AND gate. Suppose that A and B are independent binary random variables with $\Pr(A = 1) = p_A$ and $\Pr(B = 1) = p_B$. Then $\Pr(C = 1) = p_A p_B$, and therefore the AND gate performs a multiplication. The second circuit is a multiplexer. Similarly, let $\Pr(D = 1) = p_D$, $\Pr(E = 1) = p_E$, $\Pr(Z = 1) = p_Z$, then $\Pr(F = 1) = (1 - p_Z)p_D + p_Z p_E$, and therefore the circuit performs a convex combination of streams D and E, with the weights determined by p_Z. Note that it is not possible to directly perform the addition of two streams D and E, since the domain of $p_D + p_E$ is $[0,2]$. However, a normalized addition can be implemented by using $p_Z = \frac{1}{2}$.

Several other circuits have been proposed to implement more complex functions (see [1] for more examples).

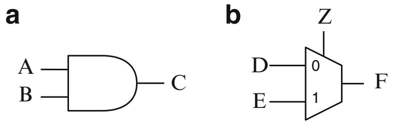


Fig. 6.2 Stochastic computing circuits for (**a**) multiplication and (**b**) convex combination

6.4 Fully Stochastic Decoders

This section discusses LDPC decoding algorithms in which all computations are performed in the stochastic domain, which we refer to as fully stochastic decoders. Other algorithms that only perform a subset of the computations in the stochastic domain are described in Sects. 6.5 and 6.6. Section 6.4.1 introduces the fundamental circuits used in a stochastic LDPC decoder. Following this, Sects. 6.4.2 and 6.4.3 describe two different extensions of the simple stochastic algorithm that both allow decoding practical LDPC codes.

6.4.1 A Simple Stochastic Decoder

LDPC decoders have the potential to achieve a high throughput because each of the n codeword bits can be decoded in parallel. However, the length of the codes used in practice is on the order of 10^3, going up to 10^5 or more. This makes it difficult to make use of all the available parallelism while still respecting circuit area constraints. One factor influencing area utilization is of course the complexity of the VAR and CHK functions to be implemented, but because of the nature of the message-passing algorithm, the wires that carry messages between processing nodes also have a large influence on the area, as was identified early on in one of the first circuit implementations of an SPA LDPC decoder [2].

The need to reduce both logic and wiring complexity suggests that stochastic computing could be a good approach. The use of stochastic computation for the message-passing decoding of block codes was first proposed by Gaudet and Rapley [7]. The idea was prompted by the realization that the two SPA functions VAR and CHK had very simple stochastic implementations when performed in the probability domain. Let us first consider the CHK function. In the LLR domain, the function is given by (6.2), which in the probability domain becomes

$$\mathrm{CHK}(p_1, p_2, \ldots, p_d) = \frac{1 - \prod_{i=1}^{d}(1 - 2p_i)}{2}. \tag{6.6}$$

The implementation of this function in the stochastic domain is simply an exclusive-OR (XOR) gate. That is, if we have independent binary random variables $X_1, X_2, \ldots, X_d$, each distributed such that $\Pr(X_i = 1) = p_i$, then taking

$$Y = X_1 + X_2 + \cdots + X_d \mod 2 \tag{6.7}$$

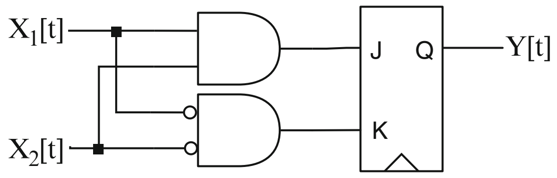


Fig. 6.3 Stochastic computation circuit for the two-input VAR function in the probability domain

yields $\Pr(Y=1) = \text{CHK}(p_1, p_2, \ldots, p_d)$. This result is not as surprising as it might seem. Indeed, the modulo-2 sum is exactly the constraint that must be satisfied by the codeword bits involved in this check node operation. Using stochastic streams instead of codeword bits is akin to performing a Monte-Carlo simulation to find the probability associated with an unknown bit connected to this check node.

A circuit for computing the VAR function with two inputs was also presented in [7]. In the probability domain, the LLR function of (6.1) with $d=2$ is given by

$$\text{VAR}(p_1, p_2) = \frac{p_1 p_2}{p_1 p_2 + (1-p_1)(1-p_2)}. \tag{6.8}$$

In the stochastic domain, this function can be computed approximately using the circuit shown in Fig. 6.3. The JK flip-flop becomes 1 if its J input is 1, and 0 if its K input is 1. Otherwise, it retains its previous value. This implementation is different in nature from the one used for the CHK function, since it contains a memory. The behavior of the circuit can be analyzed by modeling the output Y as a Markov chain with states $Y=0$ and $Y=1$. Suppose that the stochastic streams $X_1[t]$ and $X_2[t]$ are generated according to the expectation sequences $p_1[t]$ and $p_2[t]$, respectively, and let the initial state be $Y[0] = s_o$. Then, at time $t=1$, we have

$$\mathbb{E}[Y[1]] = \Pr(Y[1]=1) = \begin{cases} p_1[1]p_2[1] & \text{if } \mathrm{s_o} = 0, \\ p_1[1] + p_2[1] - p_1[1]p_2[1] & \text{if } \mathrm{s_o} = 1. \end{cases}$$

None of the expressions above are equal to $\text{VAR}(p_1[1], p_2[1])$, and therefore the expected value of the first output of the circuit is incorrect, irrespective of the starting state. However, if we assume that the input streams are independent and identically distributed (i.i.d.) with $p_1[t] = p_1$ and $p_2[t] = p_2$, it is easy to show that the Markov chain converges to a steady-state such that

$$\lim_{t\to\infty} \mathbb{E}[Y[t]] = \text{VAR}(p_1, p_2). \tag{6.9}$$

To build a circuit that will compute the VAR function for more than two inputs, we can make use of the fact that the VAR function can be distributed arbitrarily, which can easily be seen by considering the equivalent LLR-domain formulation in (6.1). For example we have $\text{VAR}(p_1, p_2, p_3) = \text{VAR}(\text{VAR}(p_1, p_2), p_3)$.

Stochastic decoders built using these circuits were demonstrated for very small codes, but they are unable to decode realistic LDPC codes. The reason is that (6.9) is not sufficient to guarantee the accuracy of the variable node computation, since we do not know that the input streams are stationary or close to stationary. In graphs with cycles, low precision messages can create many fixed points in the decoder's iterative dynamics that would not be there otherwise. This was noted in [21], and the authors proposed to resolve the precision issue by adding an element called a *supernode*, which takes one stochastic stream as input and outputs another stochastic stream. It interrupts the feedback path by using a constant expectation parameter to generate the output stochastic stream. Simultaneously, it estimates the mean of the incoming stochastic stream. The decoding is performed in several iterations, and an iteration is completed once a stochastic stream of length ℓ has been transmitted on every edge of the graph. Once the iteration completes, the expectation parameter in each supernode is updated with the output of the mean estimator. Because the expectation parameter is kept constant while the stochastic streams are being generated, the precision can be increased by increasing ℓ.

While the supernode approach works, it requires large values of ℓ to achieve sufficient precision, and therefore a lot of time for transmitting the ℓ bits in each iteration. However, it is not necessary for the expectation parameter of a stochastic stream to be constant. Any method that can control the rate of change of the expectation sequences will allow avoiding fixed points in the decoding algorithm created by insufficient precision. In particular, this can be achieved by using low-pass filters, some of which are described in Sect. 6.4.2.

6.4.2 *Stochastic Decoders Using Successive Relaxation*

Now that we have explained the basic concepts used to build stochastic decoders, we are ready to present stochastic decoding algorithms that are able to decode practical LDPC codes. Most such algorithms make use of a smoothing mechanism called Successive Relaxation. In Sect. 6.4.2.1, we explain how Successive Relaxation was introduced into the domain of LDPC decoding algorithms, and how it is used in stochastic decoders. Then, in Sect. 6.4.2.2, we present circuit implementations for the stochastic computation of the variable node function. Using stochastic streams to represent likelihood information can sometimes cause precision issues. The main problem lies in the inability of accurately representing probability values very close to 0 or 1. Section 6.4.2.3 discusses this range problem and how it can be mitigated. Finally, Sect. 6.4.2.4 summarizes the performance of a fully stochastic decoder when decoding the LDPC code standardized by the IEEE 802.3an standard for 10 Gbps Ethernet networking.

6.4.2.1 The Role of Successive Relaxation

Message-passing LDPC decoders are iterative algorithms. We can express their iterative progress by defining a vector $\mathbf{x}_o$ of length n containing the information received from the channel, and a second vector $\mathbf{x}[t]$ of length n_e containing the messages sent from each variable node to its neighboring check nodes at iteration t, where n_e is the number of edges in the graph. The standard SPA decoder for an LDPC code is an iterative algorithm that is memoryless, by which we mean that the messages sent on the graph edges at one iteration only depend on the initial condition, and on the messages sent at the previous iteration. As a result, the decoder's progress can be represented as follows:

$$\mathbf{x}[t] = h(\mathbf{x}[t-1], \mathbf{x}_o),$$

where $h()$ is a function that performs the check node and variable node message updates, as described in Algorithm 1.

In the past, analog circuit implementations of SPA decoders have been considered for essentially the same reasons that motivated the research into stochastic decoders. Since these decoders operate in continuous time, a different approach was needed to simulate their decoding performance. The authors of [8] proposed to simulate continuous-time SPA by using a method called *successive relaxation* (SR). Under SR, the iterative progress of the algorithm becomes

$$\mathbf{x}[t] = (1-\beta)\cdot\mathbf{x}[t-1] + \beta\cdot h(\mathbf{x}[t-1], \mathbf{x}_o), \tag{6.10}$$

where $0 < \beta \leq 1$ is known as the relaxation factor. As $\beta \to 0$, the simulated progress of the decoder approaches a continuous-time (analog) decoder. However, the most interesting aspect of this method is that it can be used not only as a simulator, but also as a decoding algorithm in its own right, usually referred to as Relaxed SPA. Under certain conditions, Relaxed SPA can provide significantly better decoding performance than the standard SPA.

In stochastic decoders, SR cannot be applied directly because the vector of messages $\mathbf{x}[t]$ is a binary vector, while $\mathbf{x}[t]$ obtained using (6.10) is not if $\beta < 1$. However, if we want to add low-pass filters to a stochastic decoder, we must add memories that can represent the expectation domain of the stochastic streams. Suppose that we associate a state memory with each edge, and group these memories in a vector $\mathbf{s}[t]$ of length n_e. Since the expectation domain is the probability domain, the elements of $\mathbf{s}[t]$ are in the interval $[0,1]$. Stochastic messages can be generated from the edge states by comparing each edge state to a random threshold, as described in (6.5). We can then rewrite (6.10) as a mean tracking filter, where $\mathbf{s}[t]$ is the vector of estimated means after iteration t, and $\mathbf{x}[t]$, $\mathbf{x}_o[t]$ are vectors of stochastic bits:

$$\mathbf{s}[t] = (1-\beta)\cdot\mathbf{s}[t-1] + \beta\cdot h(\mathbf{x}[t-1], \mathbf{x}_o[t-1]). \tag{6.11}$$

The value of β controls the rate at which the decoder state can change, and since $\mathbb{E}[\mathbf{x}[t]] = \mathbf{s}[t]$, it also controls the precision of the stochastic representation.

6.4.2.2 Circuit Implementations of the VN Function

We will first consider stochastic variable node circuits with two inputs $X_1[t]$ and $X_2[t]$. As previously, we denote by $p_1[t]$ and $p_2[t]$ the expectation sequences associated with each input stream. Let E be the event that $X_1[t] = X_2[t]$. We have that

$$\mathbb{E}[X_1[t] \,|\, E] = \mathbb{E}[X_2[t] \,|\, E] = \textsc{var}(p_1[t], p_2[t]),$$

where $\textsc{var}()$ is defined in (6.8). Therefore, one way to implement the variable node function for stochastic streams is to track the mean of the streams at the time instants when they are equal. As long as $\Pr(E) > 0$, a mean tracker can be as close as desired to $\textsc{var}(p_1[t], p_2[t])$ if the rate of change of $p_1[t]$, $p_2[t]$ is appropriately limited. If the mean tracker takes the form of (6.11), this corresponds to choosing a sufficiently small β.

The first use of relaxation in the form of (6.11) was proposed in [17], where the relaxation (or mean tracking) step is performed in the variable node, by using a variable node circuit that is an extension of the original simple circuit shown in Fig. 6.3. In the original VN circuit, each graph edge had a corresponding 1-bit flip-flop. This flip-flop can be extended to an ℓ-bit shift-register, in which a "1" is shifted in if both inputs $X_1[t]$ and $X_2[t]$ are equal to 1, and a "0" is shifted in if both inputs are equal to 0. When a new bit is shifted in, the oldest bit is discarded.

Let us denote the number of "1" bits in the shift-register by $w[t]$, and define the current mean estimate as $s[t] = w[t]/\ell$. If we make the simplifying assumptions that the bits in the shift register are independent from $X_1[t]$ and $X_2[t]$, and that when a bit is added to the shift register, the bit to be discarded is chosen at random, then it is easy to show that the shift-register implements the successive relaxation rule of (6.10) *in distribution*, with $\beta = \Pr(E)/\ell$, in the sense that

$$\mathbb{E}[s[t]] = \left(1 - \frac{\Pr(E)}{\ell}\right) \cdot s[t-1] + \frac{\Pr(E)}{\ell} \cdot \textsc{var}(p_1[t-1], p_2[t-1]).$$

When the variable node degree is large, it was suggested in [18] to implement the variable node function using a computation tree with two levels. Let us denote the computation performed by the first level circuit as VARST_1 and by the second level circuit as VARST_2. For example, the circuit for a degree-6 VN can be implemented as

$$\mathrm{VARST}(x_o, x_1, \ldots, x_5) = \mathrm{VARST}_2(\mathrm{VARST}_1(x_1, x_2, x_3), \mathrm{VARST}_1(x_0, x_4, x_5)),$$

where x_o is the current stochastic bit corresponding to the channel information, and $x_1, x_2, \ldots, x_5$ are the stochastic bits received from the neighboring check nodes. The corresponding circuit is shown in Fig. 6.4. When using such a two-level implementation, it is proposed in [18] to use small shift-registers for the first level, and a large one for the second level.

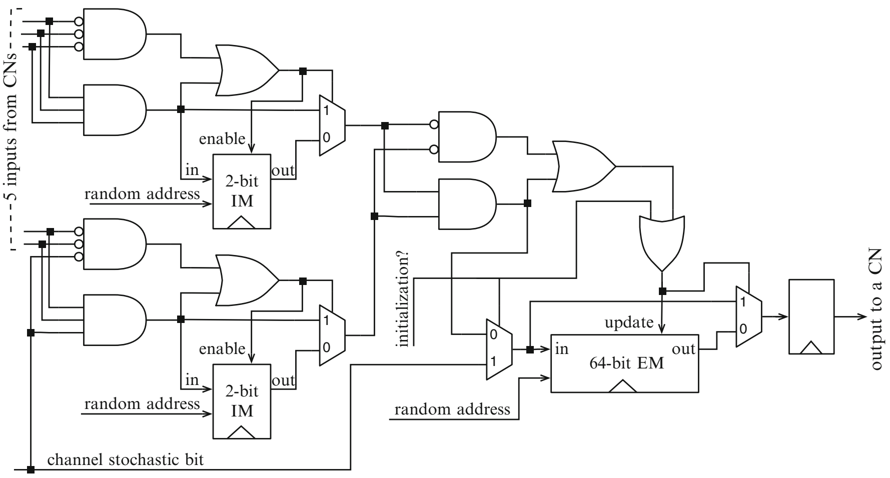


Fig. 6.4 Degree-6 stochastic VN circuit corresponding to one output [18]

Using a shift register is interesting from a pedagogical point of view because it uses a stochastic representation for the mean estimate. However, there are several reasons that discourage its use in decoders. First, the choice of relaxation factor β is tied to the precision of the mean estimate, which prevents from freely optimizing the value of β. Second, because the mean is represented stochastically, storing a high precision estimate requires a lot of register bits, which are costly circuit components. Lastly, the relaxation factor β is not constant, since $\beta = \Pr(E)/\ell$. This can complicate the analysis and design of the decoder.

For these reasons, a better approach to performing the mean tracking, proposed in [19], is to directly represent the mean estimate $s[t]$ as a fixed-point number. The VN computation with inputs X_1, X_2 can now be implemented as

$$s[t+1] = \begin{cases} (1-\beta)\cdot s[t] + \beta & \text{if } \mathrm{X}_1 = \mathrm{X}_2 = 1, \\ (1-\beta)\cdot s[t] & \text{if } \mathrm{X}_1 = \mathrm{X}_2 = 0, \\ s[t] & \text{otherwise.} \end{cases}$$

If β is chosen as a negative power of 2, the update can be implemented with only bit shifts and additions.

6.4.2.3 Tweaking the Probability Domain Representation

The stochastic computations used in decoding LDPC codes are expressed in the probability domain because that greatly simplifies the computation of the check node function, which becomes a simple XOR gate, as stated in (6.7). It was found that when running SPA in the LLR domain, the performance is adversely affected if the representable range of LLR values is too limited. The probability domain representation requires a large number of bits to represent large LLR magnitudes, and representing the probability as a stochastic stream requires a further exponential increase in the number of bits used, as was discussed in Sect. 6.3.1. To illustrate this, consider a stationary stochastic stream of length t. The smallest non-zero probability value that it can represent is $1/t$, and therefore the largest LLR value $\Lambda_{\max}$ less than infinity that it can represent is

$$\Lambda_{\max} = \ln\left(\frac{1-\frac{1}{t}}{\frac{1}{t}}\right) = \ln(t-1).$$

For example, the ability to distinguish an LLR value of 2 from larger values requires a stationary stream of length $t \geq 9$, while distinguishing an LLR of 4 from larger values requires $t \geq 56$.

One way to improve the representable range is to note that when decoding an LDPC code, we are not interested in approximating the true a-posteriori probabilities of each transmitted bit, but simply of identifying which codeword is more likely to have been transmitted. Therefore, we can try scaling down the prior LLR values to increase the representation precision for large LLR magnitudes (at the expense of decreasing the representation precision for values near zero). Suppose that we apply a scaling factor α to all LLR values. The minimal length of a stationary stream that will distinguish $\Lambda_{\max}$ from larger values becomes

$$t \geq \lceil e^{\alpha\Lambda_{\max}} + 1 \rceil$$

Continuing the previous example, if we set $\alpha = \frac{1}{2}$, distinguishing an LLR value of $\Lambda_{\max} = 2$ now requires only $t \geq 4$, and $t \geq 9$ for $\Lambda_{\max} = 4$. This LLR scaling approach was initially proposed in [17] under the name Noise Dependent Scaling, where it was shown that it is necessary to choose the right scaling factor in order to achieve a low error rate when the channel signal-to-noise ratio is large.

6.4.2.4 Benchmark Using the IEEE 802.3an Standard

The IEEE 802.3an 10GBASE-T standard (10 Gbps Ethernet networking over CAT-6 copper cables) uses an LDPC code of length 2,048 and rate 0.841 for error correction. Because of its industrial relevance and high throughput requirements, this code became a popular benchmark for LDPC decoders. A stochastic decoder for this code that uses the techniques presented in Sects. 6.4.2.1–6.4.2.3 was described in [20]. In addition, this decoder uses a heuristic circuit optimization called “MTFM” where a single relaxation filter is used simultaneously by the d_v outputs of a variable node circuit.

The main strength of this stochastic decoder compared to other decoder implementations is that it achieves a high average decoding throughput, especially when normalizing for circuit area. Its main inconvenient is that its worst-case decoding time is significantly larger. This worst case can be dealt with by adding buffering at the input, but many applications have stringent latency requirements that prevent such buffering.

6.4.3 The "Delayed" Stochastic Decoder

The simple stochastic decoder discussed in Sect. 6.4.1 is interesting for the very low complexity of the circuits required, but unfortunately it is unable to decode practical codes because the precision of the messages exchange is in most cases insufficient to ensure convergence. The relaxation-based approach presented in Sect. 6.4.2 resolves this issue by controlling the rate of change of the expectation sequences associated with each stochastic stream, allowing the decoder to converge. The "delayed" stochastic (DS) decoder introduced in [11] takes a different approach to resolving the convergence problem, with the objective of eliminating a large portion of the register circuits required in relaxation-based decoders.

Instead of controlling the precision of the stochastic streams, the DS decoder propagates "freeze" flags that indicate whether a stochastic message is reliable or not. If a check node receives a freeze flag, it propagates the flag to its neighbors and the messages sent by this check node are ignored in the variable nodes.

6.4.3.1 Simple DS Decoder

In its simplest form, the DS decoder uses the same variable node function as the simple stochastic decoder of Sect. 6.4.1. In the first iteration, the variable nodes send a stochastic bit generated from the channel probability. In following iterations, the variable nodes receive messages from the check nodes, and the reliability of each output of a VN is determined by the presence of an agreement among input bits. If there is no agreement for generating a given VN output, a "freeze" flag is propagated on that output.

6.4.3.2 Some Heuristics for Improved Convergence

In the simple DS algorithm, the proportion of messages that are deemed reliable was observed to be insufficient in many cases, resulting in very slow convergence. Two modifications to the algorithm were proposed to improve the decoding time. First, small memories called "internal memories" (IM) are added to the variable node circuit to provide some averaging of the stochastic streams. The resulting VN circuit used to generate one output then becomes identical to the one in Fig. 6.4, but without the 64-bit edge-memory (EM). Instead of sampling from the EM, the DS circuit outputs a "freeze" flag. Second, a special condition is added to further reduce the number of "freeze" flags being propagated. When a majority of a VN's outputs would be sending "freeze" flags, the VN circuit instead enters a special state where no "freeze" flags are sent, and the output messages are sampled from the IMs.

6.4.3.3 Benchmark Using the IEEE 802.3an Standard

A hardware decoder for the code specified in the IEEE 802.3an standard is reported in [11]. The results show that the decoding latency of the DS algorithm is longer than for the relaxation-based decoder presented in Sect. 6.4.2. The latency can be made equivalent at the cost of reducing the coding gain by 0.2 dB. However, the DS decoder occupies a much smaller area: 3.93 mm^2 instead of 6.38 mm^2 (in a 90 nm technology). The average throughput of the DS decoder is also 2.8 times higher.

These hardware results show that it is possible for a stochastic decoder to converge on practical LDPC codes even when the circuit contains a very small amount of memory. However more work must be done to improve the worst-case convergence time and the coding gain.

6.5 Mixing Stochastic and Conventional Computations

It is a common occurrence in computer architecture that the complexity of a computation can be reduced by changing the representation domain of the data. For example, applying a Fourier transform can simplify a complex convolution operation into a simpler multiplication, while operating with a logarithmic representation simplifies multiplications into additions. Besides the issue of numerical precision which must also be considered, the decision to use an alternate representation in an implementation must take into account the other computations that are required by the algorithm, and the cost of converting one representation domain to another.

This situation is found in the SPA. As was presented in Sect. 6.4.1, performing the check node computation in the stochastic domain results in an exact implementation (in distribution) of the SPA check node function that has a very low complexity. On the other hand, the stochastic variable node computation can only become exact if the input streams are stationary, which is a case of limited interest since it implies that the decoder is not making any progress. Among exact implementations of the variable node computation, the simplest is obtained by using the LLR domain, for which the computation is simply an addition.

In this section, we present an algorithm called the Relaxed Half-Stochastic (RHS) algorithm [10] that uses the stochastic domain for the check node computation and the LLR domain for the variable node computation, and which includes mechanisms to efficiently convert the values between the two domains.

The stochastic domain has the potential of greatly simplifying some types of computations, but there are also situations where it is more likely to increase the complexity, for example when a computation requires a high precision. The approach presented here could therefore also provide opportunities for other algorithms that would benefit from using stochastic computation in only a portion of the algorithm.

6.5.1 The RHS Algorithm

The structure of the RHS algorithm, like all other algorithms presented in this chapter, is very similar to that of the SPA, described in Algorithm 1. In fact, part of the variable node computation is identical to the SPA, since it operates in the LLR domain. When describing the algorithm we assume that messages are exchanged according to the so-called *flooding* schedules. The steps described can be trivially modified to use other schedules.

Like the SPA, the RHS algorithm takes as input a set of n LLR values $\{\Lambda_1, \Lambda_2, \ldots, \Lambda_n\}$ that correspond to the n bits received from the channel. However, it is easier to describe the algorithm from the perspective of individual variable nodes and check nodes. Still following the SPA described in Algorithm 1, the first iteration begins by generating the LLR values $\eta_{i,j}$, where the index i runs over the variable nodes, while j runs over the neighboring check nodes of the variable node. When considering a particular variable node i we can simplify the notation to η_j. In RHS, we call the η_j values *intermediate VN outputs*. These intermediate VN outputs are then converted to stochastic messages and sent to neighboring check nodes. The check node computation is performed in the stochastic domain, and stochastic messages are sent back from check nodes to variable nodes, which completes the iteration.

The main interest of the algorithm lies in the mechanism used to convert to and from the stochastic domain. This is described in Sects. 6.5.2 and 6.5.3.

6.5.2 Domain Conversion: LLR to Stochastic

The RHS algorithm uses an extended version of stochastic streams where each stream element consists of a binary vector of length k. Therefore, when a variable node sends a message to a check node, it transmits k bits, either serially or in parallel. Let us denote such a message as $\mathbf{X} = [X_1, X_2, \ldots, X_k]$, where each X_i is an independent binary random variable. In order to use the simple check node function common to all stochastic LDPC decoders, we define the expectation parameter in the probability domain. The fact that the k bits in a message vector are generated simultaneously allows exploring various generation rules, but one way to define $\mathbf{X}$ is to have independent and identically distributed bits, such that $\mathbb{E}[X_1] = \mathbb{E}[X_2] = \cdots = \mathbb{E}[X_k] = p_j$. The expectation parameter p_j is simply the LLR-domain intermediate output η_j converted to the probability domain:

$$p_j = \frac{1}{e^{\eta_j} + 1}.$$

One way to generate the stochastic bits is to compare p_j to a random threshold, uniformly distributed over the probability domain. However, the computation of p_j can be avoided by instead converting the threshold to the LLR domain. Let Z be a uniform random variable over the open interval $(0,1)$. An LLR threshold T is obtained using $T = \ln((1-Z)/Z)$. We then generate a stochastic bit for each $i \in [1,k]$ by comparing the intermediate output with a threshold sampled from T:

$$X_i = \begin{cases} 0 & \text{if } \eta_i > T, \\ 1 & \text{otherwise.} \end{cases} \tag{6.12}$$

In the circuit implementation, a realization of Z can be generated easily using a linear feedback shift-register (LFSR) circuit, and the conversion to the LLR domain can be approximated accurately using a priority encoder circuit and a few additional logic gates.[1]

As discussed in Sect. 6.4.1, the stochastic check node circuit composed of XOR gates implements the SPA check node function provided that the stochastic inputs are independent. This requirement determines which thresholds in the decoder must be independent. Any two messages that are sent to different check nodes can be generated with the same threshold. On the other hand, each of the k bits in the message vector must be independent. Therefore, $k \cdot d_c$ independent thresholds are required in a complete decoder, where d_c is the check node degree. Since $kd_c \ll n$, the complexity of the random threshold generator circuit does not have a large impact on the complexity of the decoder.

6.5.3 Domain Conversion: Stochastic to LLR

A check node is connected to d_c variable nodes, and therefore at every decoding iteration it receives d_c messages and outputs d_c messages. However, each output message is generated from only $d_c - 1$ messages, following the extrinsic information principle at the center of the SPA.

The RHS check node function therefore takes $d_c - 1$ stochastic vectors $\mathbf{X}_i$ as inputs, each of length k. Let $i \in [1, d_c - 1]$ be an index running over the check node inputs, and $j \in [1,k]$ an index running over the bits in each vector. Therefore we can denote an individual bit inside a vector as $X_{i,j}$. To simplify the presentation, we consider the generation of one of the d_c check node output messages. Using XOR gates, the check node function computes an output vector $\mathbf{Y} = [Y_1, \ldots, Y_k]$ as

[1]A detailed example of a circuit implementation can be found in [9].

$$Y_j = \sum_{i=1}^{d_c-1} X_{i,j} \mod 2,$$

for $j \in [1,k]$.

This output vector $\mathbf{Y}$ is then transmitted back to a neighboring variable node. At this point, we need to use the knowledge of how the vectors $\{\mathbf{X}_1,\dots,\mathbf{X}_{d_c}\}$ were generated to interpret $\mathbf{Y}$. Using the generation rule described in 6.5.2, the expectation of $\mathbf{Y}$ lies in the probability domain, and since the bits are i.i.d., the maximum-likelihood estimate $\hat{m}$ of the probability represented by $\mathbf{Y}$ is

$$\hat{m} = \frac{1}{k}\sum_{j=1}^{k} Y_j. \tag{6.13}$$

We define a set $\mathscr{M}$ such that $\hat{m} \in \mathscr{M}$. The set $\mathscr{M}$ contains all the possible messages that can be received by a variable node in a single iteration. Under (6.13), we have $\mathscr{M} = \{0, \frac{1}{k}, \frac{2}{k}, \dots, 1\}$, for a total of $k+1$ distinct messages.

For small k values, the precision of the message received is too crude for the iterative decoder to converge, and therefore we are interested in tracking the mean of the messages received over the iterations. Let $\hat{m}[t]$ be a message received on an input of a variable node at iteration t, and let $s[t]$ be the mean probability estimate at the end of iteration t, with $s[0] = \frac{1}{2}$. For $t \geq 1$, we can apply the relaxation filter to generate an updated estimate $s[t]$:

$$s[t] = (1-\beta)s[t-1] + \beta\hat{m}[t]. \tag{6.14}$$

However, this estimate $s[t]$ is in the probability domain, whereas we are interested in converting the stochastic message to the LLR domain. For small values of k, we can design a tracking filter operating in the LLR domain by studying the transfer function corresponding to each message in $\mathscr{M}$. The LLR domain tracking then simply amounts to choosing the right transfer function according to the received message $\hat{m}$:

$$\Lambda[t] = f(\Lambda[t-1]; \hat{m}[t]),$$

where $\Lambda[t]$ is the LLR estimate after iteration t, and $f(\Lambda;\hat{m})$ is the transfer function corresponding to message $\hat{m}$. We obtain the *ideal* transfer functions by expressing (6.14) in the LLR domain, which yields

$$\Lambda[t] = \ln\left(\frac{e^{\Lambda[t-1]} + \beta(1 - \hat{m}[t](e^{\Lambda[t-1]}+1))}{1 - \beta(1 - \hat{m}[t](e^{\Lambda[t-1]}+1))}\right).$$

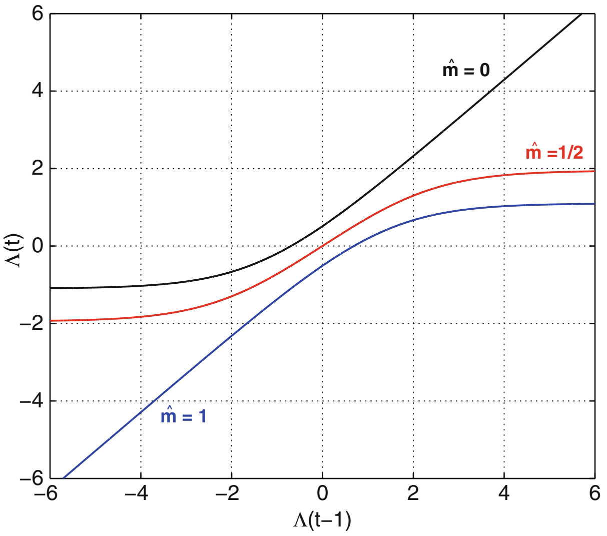


Fig. 6.5 LLR tracker transfer functions for $\mathscr{M} = \{0, \frac{1}{2}, 1\}$ and $\beta = \frac{1}{4}$

However, using the ideal functions directly in the circuit implementation would result in overly complex circuits. Instead, we must approximate each transfer function using simple linear functions, or alternatively using look-up tables, leaving the logic synthesis tool to optimize the circuit. An implementation should also exploit the symmetry of the transfer functions. We have that $f(\Lambda;\mu_i) = -f(-\Lambda;\mu_{k+2-i})$, where μ_i is the i-th largest message in $\mathscr{M}$.

For example, Fig. 6.5 shows the transfer functions for $\mathscr{M} = \{0, \frac{1}{2}, 1\}$ (hence $k=2$) and $\beta = \frac{1}{4}$. These functions can be approximated with good accuracy by the following piecewise-linear functions:

$$f(\Lambda;0) \approx \begin{cases} \Lambda + b & \text{if } \Lambda \geq \text{d}, \\ d & \text{if } \Lambda < \text{d}, \end{cases} \tag{6.15}$$

$$f(\Lambda;\tfrac{1}{2}) \approx \begin{cases} a\Lambda & \text{if } -\text{c} \leq \Lambda \leq \text{c}, \\ -c & \text{if } \Lambda < -\text{c}, \\ c & \text{if } \Lambda > \text{c}, \end{cases} \tag{6.16}$$

where the parameters a,b,c,d are chosen based on the value of β. The $f(\Lambda;1)$ function is obtained using the symmetry rule. Using these approximations, it is possible to design a very simple circuit that will convert the stochastic messages into LLR values, while also implementing the relaxation filter.

6.5.4 Benchmark Using the IEEE 802.3an Standard

Hardware implementation results of an RHS decoder for the LDPC code included in the IEEE 802.3an standard were reported in [9]. Compared to the other stochastic decoders presented in this chapter, the decoder based on the RHS algorithm occupies a larger area. Implemented in a 65 nm CMOS technology, the RHS decoder uses 4.41 mm^2, while using a quadratic scaling law, the equivalent area for the relaxation-based fully stochastic decoder of Sect. 6.4.2 is 3.33 mm^2, and that of the delayed stochastic decoder of Sect. 6.4.3 is 2.05 mm^2. In return for the increased area, the RHS decoder reduces the worst-case latency by 3.6 times, while simultaneously providing an additional coding gain of 0.2 dB with respect to the relaxation-based stochastic decoder, and 0.4 dB with respect to the delayed stochastic decoder. Finally, its average throughput is 2.6 times larger than the relaxation-based decoder, and similar to that of the delayed stochastic decoder.

6.6 Stochastic Decoders for Non-Binary LDPC Codes

Non-binary LDPC codes were shown to outperform their binary counterpart at an equivalent bit length [5], and furthermore are particularly interesting for channels that exhibit bursty error patterns, which are prominent in applications such as data storage and wireless communication. Unfortunately, they are also difficult to decode. Stochastic computation is one approach that has been explored to reduce the complexity of the decoding algorithm.

6.6.1 Message-Passing Decoding

In a non-binary code, codeword symbols can take any value from the Galois Field (GF) of order q. The field order is usually chosen as a power of 2, and in that case we denote the power as p, that is $2^p = q$. The information received about a symbol can be expressed as a probability mass function (PMF) that, for each of the q possible values of this symbol, indicates the probability that it was transmitted, given the channel output. For a PMF U, we denote by $U[\gamma]$ the probability corresponding to symbol value $\gamma \in \mathrm{GF}(q)$. Decoding is achieved by passing messages representing PMFs on the graph representation of the code, as in message-passing decoding of binary codes. However, when describing the algorithm, it is convenient to add a third node type called a *permutation* node (PN), which handles part of the computation associated with the parity-check constraint. The permutation nodes are inserted on every edge in the graph, such that any message sent from a VN to a CN or from a CN to a VN passes through a permutation node, resulting in a tripartite graph.

At every decoding iteration, a variable node v receives d_v PMF messages from neighboring permutation nodes. A PMF message sent from v to a permutation node p at iteration t, denoted by $U_{vp}^{(t)}$, is given by

$$U_{vp}^{(t)} = \textsc{norm}\left(L_v \times \prod_{p' \neq p} U_{p'v}^{(t-1)}\right), \tag{6.17}$$

where L_v is the channel PMF, and $\textsc{norm}()$ is a function that normalizes the PMF so that all its probabilities sum to 1. A PN p receives a message $U_{vp}^{(t)}$ from a VN and generates a message to a CN c by performing

$$U_{pc}^{(t)}[\gamma h_p] = U_{vp}^{(t)}[\gamma], \;\; \forall \gamma \in \mathrm{GF}(q),$$

where h_p is the element of matrix H that corresponds to VN v (which specifies the column) and CN c (which specifies the row). A CN c receives d_c messages from permutation nodes and generates messages by performing

$$U_{cp}^{(t)} = \mathop{\bigstar}_{p' \neq p} U_{p'c}^{(t)}, \tag{6.18}$$

where $\bigstar$ is the convolution operator. Finally, a message sent by a CN also passes through a PN, but this time the PN performs the inverse operation, given by

$$U_{pv}^{(t)}[\gamma h_p^{-1}] = U_{cp}^{(t)}[\gamma], \;\; \forall \gamma \in \mathrm{GF}(q),$$

where h_p^{-1} is such that $h_p \times h_p^{-1} = 1$.

Among the computations described above, the multiplications required in (6.17) are costly to implement, but (6.18) has the highest complexity, since the number of operations required scales exponentially in the field order q and in the CN degree d_c.

6.6.2 Stochastic Decoding Algorithms

Several ways of applying stochastic computing to the decoding of non-binary LDPC codes are proposed in [16]. Among these, the only algorithm that can decode any code, without restrictions on q or d_v, is the non-binary version of the RHS algorithm. Just like in the binary version presented in Sect. 6.5, the non-binary RHS decoder uses the stochastic representation for the check node computation only.

A stochastic message sent to a check node consists of a single symbol $X \in \mathrm{GF}(q)$, which can be represented using p bits. In comparison, messages exchanged in the SPA consist of complete PMF vectors requiring $qQ = 2^pQ$ bits, where Q is the number of bits used to represent one probability value in the PMF.

Therefore, stochastic algorithms significantly reduce the routing complexity of a circuit implementation.

For binary codes, the stochastic check node function is an addition over GF(2), which requires $d_c(d_c-1)$ XOR gates for generating all d_c outputs. For non-binary codes, it is an addition over GF(q). Assuming that q is a power of two, the addition can still be implemented using only XOR gates, requiring $pd_c(d_c-1)$ gates to generate all outputs. The stochastic check node function is therefore much less complex than the SPA one. Furthermore, its complexity scales only logarithmically in q.

Unlike binary codes, non-binary regular LDPC codes with $d_v = 2$ can be good codes [13], and the low variable node degree has the advantage of reducing the decoder complexity. A specialized stochastic algorithm called "NoX" is proposed in [16] for $d_v = 2$ codes. It significantly reduces the decoder's complexity by extending the update method based on transfer functions used in the binary RHS algorithm. The binary RHS algorithm achieves a low complexity by combining the mean tracking of stochastic streams with the conversion to the LLR domain, and using low-complexity approximations of the resulting transfer functions (see Sect. 6.5.3). The NoX algorithm for $d_v = 2$ non-binary codes also includes the VN computation in these transfer functions. This is rendered manageable by the fact that for $d_v = 2$, the VN function in (6.17) simplifies to $U_{vp}^{(t)} = \textsc{norm}(L_v \times U_{p'v}^{(t-1)})$, where p and p' are the two PNs connected to the VN.

6.6.3 Results

When considering the average number of iterations required for decoding as well as the number of operations required to perform one iteration, the NoX algorithm has about the same complexity as a version of the SPA that uses the Fast Fourier Transform to reduce the complexity of the CN update. The main reason explaining that NoX does not outperform SPA is that the iterative progress of NoX is slower than SPA. This could be acceptable since the complexity of one iteration of NoX is also much less. However, a side effect of the slower progress is that larger memories are required for storing the current state of the VN circuits, which in turn increases the complexity of an update. It is likely that the complexity of NoX can be reduced by increasing the amount of information exchanged in one iteration, just like the parameter k of the binary RHS algorithm (Sect. 6.5) can be increased to transmit more information in each message, while still relying on the low-complexity stochastic CN function.

References

1. Alaghi A, Hayes JP (2013) Survey of stochastic computing. ACM Trans Embed Comput Syst 12(2s):92:1–92:19. doi:10.1145/2465787.2465794
2. Blanksby AJ, Howland CJ (2002) A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder. IEEE J Solid-State Circuits 37(3):404–412
3. Brown BD, Card HC (2001) Stochastic neural computation I: computational elements. IEEE Trans Comput 50(9):891–905
4. Chung SY, David Forney J, Richardson TJ, Urbanke R (2001) On the design of low-density parity-check codes within 0.0045dB of the Shannon limit. IEEE Commun Lett 5(2):58–60
5. Davey M, MacKay D (1998) Low-density parity check codes over gf(q). IEEE Commun Lett 2(6):165–167. doi:10.1109/4234.681360
6. Gaines BR (1967) Stochastic computing. In: Proceedings of the spring joint computer conference, 18–20 April 1967. ACM, New York, pp 149–156
7. Gaudet V, Rapley A (2003) Iterative decoding using stochastic computation. Electron Lett 39(3):299–301. doi:10.1049/el:20030217
8. Hemati S, Banihashemi A (2006) Dynamics and performance analysis of analog iterative decoding for low-density parity-check (LDPC) codes. IEEE Trans Commun 54(1):61–70. doi:10.1109/TCOMM.2005.861668
9. Leduc-Primeau F, Raymond AJ, Giard P, Cushon K, Thibeault C, Gross WJ (2012) High-throughput LDPC decoding using the RHS algorithm. In: Proceedings of 2012 conference on design & architectures for signal & image processing (DASIP)
10. Leduc-Primeau F, Hemati S, Mannor S, Gross WJ (2013) Relaxed half-stochastic belief propagation. IEEE Trans Commun 61(5):1648–1659. doi:10.1109/TCOMM.2013.021913.120149
11. Naderi A, Mannor S, Sawan M, Gross W (2011) Delayed stochastic decoding of LDPC codes. IEEE Trans Signal Process 59(11):5617–5626. doi:10.1109/TSP.2011.2163630
12. Poppelbaum WJ, Afuso C, Esch JW (1967) Stochastic computing elements and systems. In: Proceedings of the fall joint computer conference, AFIPS ’67 (Fall), 14–16 November 1967. ACM, New York, pp 635–644. doi:10.1145/1465611.1465696. http://doi.acm.org/10.1145/1465611.1465696
13. Poulliat C, Fossorier M, Declercq D (2008) Design of regular (2,d/sub c/)-ldpc codes over gf(q) using their binary images. IEEE Trans Commun 56(10):1626–1635. doi:10.1109/TCOMM.2008.060527
14. Qian W, Li X, Riedel M, Bazargan K, Lilja D (2011) An architecture for fault-tolerant computation with stochastic logic. IEEE Trans Comput 60(1):93–105. doi:10.1109/TC.2010.202
15. Richardson T, Urbanke R (2008) Modern coding theory. Cambridge University Press, Cambridge
16. Sarkis G, Hemati S, Mannor S, Gross W (2013) Stochastic decoding of LDPC codes over GF(q). IEEE Trans Commun 61(3):939–950. doi:10.1109/TCOMM.2013.012913.110340
17. Sharifi Tehrani S, Gross W, Mannor S (2006) Stochastic decoding of LDPC codes. IEEE Commun Lett 10(10):716–718
18. Sharifi Tehrani S, Mannor S, Gross W (2008) Fully parallel stochastic LDPC decoders. IEEE Trans Signal Process 56(11):5692–5703. doi:10.1109/TSP.2008.929671
19. Sharifi Tehrani S, Naderi A, Kamendje GA, Mannor S, Gross WJ (2009) Tracking forecast memories in stochastic decoders. In: Proceedings of IEEE international conference on acoustics, speech, and signal processing (ICASSP)
20. Sharifi Tehrani S, Naderi A, Kamendje GA, Hemati S, Mannor S, Gross WJ (2010) Majority-based tracking forecast memories for stochastic LDPC decoding. IEEE Trans Signal Process 58(9):4883–4896. doi:10.1109/TSP.2010.2051434
21. Winstead C, Gaudet VC, Rapley A, Schlegel CB (2005) Stochastic iterative decoders. In: International symposium on information theory, pp 1116–1120