Technical University of Crete

School of Electronic and Computer Engineering
Telecommunications Division

Analysis of LDPC and Repeat-Accumulate Codes

by

Theodoros Georgiopoulos

A Thesis submitted in partial fulfilment of the requirements
for the Electronic and Computer Engineering diploma degree.

July 2016

THESIS COMMITTEE

Professor Athanasios P. Liavas, *Thesis Supervisor*
Associate Professor Aggelos Bletsas
Associate Professor George Karystinos

Abstract

Information Theory was created in 1948, by Claude Shannon. Besides other contributions, Shannon defined the channel capacity and proved that it can be achieved arbitrarily well via channel coding. Since then, both theory and practice for point-to-point communications have been constantly developed, up to the point that, nowadays, techniques for practically attaining channel capacity exist. This has been made possible by the invention of powerful coding methods, such as turbo codes and low-density parity-check (LDPC) codes. In this thesis, we focus on LDPC codes, which were invented by Robert Gallager in the early Sixties. At that time, the limited available computational power made the use of LDPC codes impractical and prevented scientists from fully understanding their potential. After the introduction of irregular LDPC codes and of practical performance analysis tools in the late nineties, LDPC codes became the most powerful error correcting codes, enabling reliable transmissions at rates close to the channel capacity for a number of communication channels.

Acknowledgements

First and foremost, I would like to thank my family and dedicate this work to them. Without their continuous and selfless support, the fulfillment of this thesis would not have been possible.

Of course, I would like to thank my teacher and advisor, Athanasios P. Liavas, for his valuable assistance, collaboration, and guidance through every step of this thesis.

Finally, I would like to thank my good friends at the Technical University of Crete for the wonderful expreriences we shared the last years.

Contents

List of Figures

Chapter 1

Coding, channels, and capacity

1.1 Introduction

In 1948, Shannon demonstrated that, by proper encoding of the information, errors created by a noisy channel can be reduced to any desired level without sacrificing the rate of information transmission, as long as the information rate is less than the capacity of the channel [1]. Since then, much effort has been spent on the problem of efficient encoding and decoding methods for error control in a noisy communication environment. A typical transmission system can be represented by the block diagram shown in the figure below [2, p. 2].

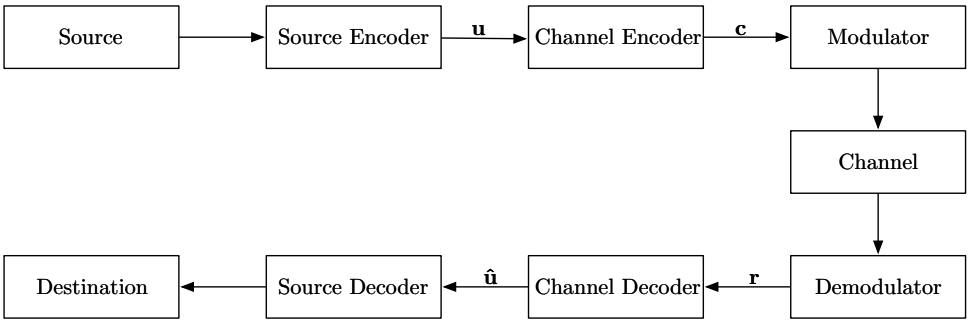

FIGURE 1.1: A typical communication system.

The *information source* can be either a continuous waveform or a sequence of discrete symbols. The *source encoder* transforms the source output into a sequence of binary digits (bits) called *information word* ($\mathbf{u}$). The source encoder is ideally designed so that the length of the bit sequence is minimized and it can be recovered without loss of information. The *channel encoder* adds redundancy to the information word and creates a discretely encoded sequence called *codeword* ($\mathbf{c}$), which, in many instances, is also binary. The design of channel encoder is based on the size and the kind of channel noise.

The channel encoder output symbol stream is not suitable for transmission over a physical channel. The *modulator* transforms this stream into a waveform that is suitable for transmission. This waveform enters the channel and is corrupted by noise. The *demodulator* processes each received waveform and produces either a discrete (quantized) or a continuous (unquantized) output. The sequence of demodulator outputs corresponding to the encoded sequence, $\mathbf{c}$, is called the *received sequence*, $\mathbf{r}$.

The *channel decoder* transforms the received sequence $\mathbf{r}$ into a binary sequence $\hat{\mathbf{u}}$ called the *estimated information sequence*. Ideally, $\hat{\mathbf{u}}$ is the same as the information sequence $\mathbf{u}$, but the noise usually causes decoding errors. The *source decoder* transforms the estimated information sequence $\hat{\mathbf{u}}$ into an estimate of the source output and delivers this estimate to the *destination*. In a well-designed system, the estimate is a faithful reproduction of the source output.

1.2 Block codes

The most commonly used codes nowadays are the *block codes* [3, pp. 66–71]. The encoder for a block code divides the information sequence into message blocks of k information bits each. A message block is represented by the binary k-tuple $\mathbf{u} = (u_1, u_2, \ldots, u_k)$, called a *message*. There are a total of 2^k different possible messages. The encoder transforms each message $\mathbf{u}$ independently into an n-tuple $\mathbf{c} = (c_1, c_2, \ldots, c_n)$ of discrete symbols, called a codeword. Therefore, corresponding to the 2^k different possible messages, there are 2^k different possible codewords at the encoder output. This set of codewords of length n is called an (n, k) *block code*. The ratio $R = k/n$ is called the *code rate*, which is the ratio of the number of bits that enter the channel encoder to the

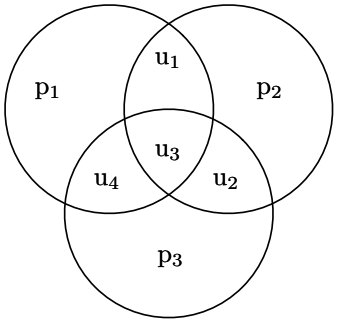

FIGURE 1.2: Venn diagram representation of $(7, 4)$ Hamming-code encoding and decoding rules.

number of bits that depart from it, with $0 < R \leq 1$. For example, if a 1000-bit codeword is assigned to each 500-bit information word, then $R = 1/2$ and there are 500 redundant bits in each codeword.

If the 2^k codewords of a block code of length n form a k-dimensional linear subspace of the vector space of all binary n-tuples, then the block code is called *linear*. In simpler terms, a block code is linear, if any linear combination of codewords is also a codeword.

A very popular linear block code is the $(7, 4)$ *Hamming code* [2, pp. 4–7]. The codeword length is $n = 7$ and the information word length is $k = 4$, so the code rate is $R = 4/7$. The Hamming code is easily described by the Venn diagram in Figure 1.2, in which $\mathbf{u} = (u_1, u_2, u_3, u_4)$ is the information word and $\mathbf{p} = (p_1, p_2, p_3)$ are the parity bits. The concatenation of parity bits and information bits gives the codeword:

$$\mathbf{c} = (\mathbf{p}\ \mathbf{u}) = (p_1, p_2, p_3, u_1, u_2, u_3, u_4) = (c_1, c_2, c_3, c_4, c_5, c_6, c_7).$$

For encoding, the parity bits are chosen so that each circle in Figure 1.2 has an even number of 1s. So, the parity equations are:

$$p_1 = u_1 \oplus u_3 \oplus u_4,$$

$$p_2 = u_1 \oplus u_2 \oplus u_3,$$

$$p_3 = u_2 \oplus u_3 \oplus u_4,$$

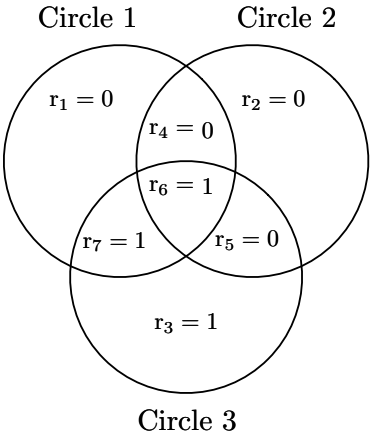

FIGURE 1.3: Venn diagram setup for the Hamming decoding example.

and the 16 codewords are:

$$
\begin{array}{llll}
(01) \to 000\ 0000 & (02) \to 110\ 1000 & (03) \to 011\ 0100 & (04) \to 001\ 1010 \\
(05) \to 000\ 1101 & (06) \to 100\ 0110 & (07) \to 010\ 0011 & (08) \to 101\ 0001 \\
(09) \to 111\ 0010 & (10) \to 011\ 1001 & (11) \to 101\ 1100 & (12) \to 010\ 1110 \\
(13) \to 001\ 0111 & (14) \to 100\ 1011 & (15) \to 110\ 0101 & (16) \to 111\ 1111.
\end{array}
$$

We can see that the modulo-2 sum of 2 codewords yields a codeword. So, the code is linear.

Suppose now that $\mathbf{c} = (\mathbf{p}\ \mathbf{u}) = (001\ 0111)$ is transmitted, but $\mathbf{r} = (001\ 0011)$ is received. The symbol c_5 has been changed, so we have an error. We can use the Venn diagram of Figure 1.3 for decoding $\mathbf{r}$ and error correction. According to this Venn diagram, the first cycle has an even number of 1s, because $r_1 = r_4 = 0, r_6 = r_7 = 1$. The second circle has an odd number of 1s, because $r_2 = r_4 = r_5 = 0, r_6 = 1$. The r_4 and r_6 are part of the first circle which is correct. So, the error must be for r_2 or r_5. In the same manner for third circle, the error must be for r_3 or r_5. Consequently, the error is for r_5 and correcting it, we have $r_5 = 1$.

1.3 The minimum distance of a block code

An important parameter for error detection and correction of a block code is the *minimum distance* [3, pp. 76–77]. Let $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ be a binary n-tuple. The *Hamming weight* of $\mathbf{v}$, denoted by $w(\mathbf{v})$, is defined as the number of nonzero components of $\mathbf{v}$. For example, the Hamming weight of $\mathbf{v} = (1\ 1\ 1\ 0\ 0\ 1\ 1)$ is 5. Let $\mathbf{v}$ and $\mathbf{w}$ be two n-tuples. The *Hamming distance* between $\mathbf{v}$ and $\mathbf{w}$, denoted $d(\mathbf{v}, \mathbf{w})$, is defined as the number of places where they differ. For example, the Hamming distance between $\mathbf{v} = (1\ 1\ 1\ 0\ 0\ 1\ 1)$ and $\mathbf{w} = (0\ 1\ 0\ 0\ 0\ 0\ 1)$ is 3, because they differ in first, third, and sixth bit. The Hamming distance is a metric function that satisfies the *triangle inequality*. Let $\mathbf{v}$, $\mathbf{w}$, and $\mathbf{x}$ be three n-tuples. Then,

$$d(\mathbf{v}, \mathbf{w}) + d(\mathbf{w}, \mathbf{x}) \geq d(\mathbf{v}, \mathbf{x}). \tag{1.1}$$

Moreover, it can be shown that the Hamming distance between two n-tuples $\mathbf{v}$ and $\mathbf{w}$ is equal to the Hamming weight of the modulo-2 sum of $\mathbf{v}$ and $\mathbf{w}$. So,

$$d(\mathbf{v}, \mathbf{w}) = w(\mathbf{v} \oplus \mathbf{w}). \tag{1.2}$$

For example, the Hamming distance between $\mathbf{v} = (1\ 1\ 1\ 0\ 0\ 1\ 1)$ and $\mathbf{w} = (0\ 1\ 0\ 0\ 0\ 0\ 1)$ is 3, and the weight of $\mathbf{v} \oplus \mathbf{w} = (1\ 0\ 1\ 0\ 0\ 1\ 0)$ is also 3.

Given a block code C, one can compute the Hamming distance between any two distinct codewords. The *minimum distance* of C, denoted by d_{min}, is defined as

$$d_{min} \triangleq \min\{d(\mathbf{v}, \mathbf{w}) : \mathbf{v}, \mathbf{w} \in C, \mathbf{v} \neq \mathbf{w}\}. \tag{1.3}$$

From the definition of linear block codes, we know that the sum of two codewords is also a codeword. So, if C is a linear block code, from (1.2) we have that the Hamming distance between two codewords in C is equal to the Hamming weight of a third codeword in C. Then, it follows from (1.3) that

$$\begin{aligned} d_{min} &= \min\{w(\mathbf{v} \oplus \mathbf{w}) : \mathbf{v}, \mathbf{w} \in C, \mathbf{v} \neq \mathbf{w}\} \\ &= \min\{w(\mathbf{x}) : \mathbf{x} \in C, \mathbf{x} \neq 0\} \\ &\triangleq w_{min}. \end{aligned} \tag{1.4}$$

The parameter $w_{min} \triangleq \min\{w(\mathbf{x}) : \mathbf{x} \in C, \mathbf{x} \neq 0\}$ is called the *minimum weight* of the linear code C. So, the minimum distance of a linear block code is equal to the minimum weight of its nonzero codewords.

1.4 Error-detection and error-correction capabilities of a block code

When a codeword $\mathbf{c}$ of a block code C is transmitted over a noisy channel, an error pattern of l errors will result in a received vector $\mathbf{r}$ that differs from the transmitted codeword $\mathbf{c}$ in l places. So, we have $d(\mathbf{c}, \mathbf{r}) = l$. If the minimum distance of code C is d_{min}, any two distinct codewords of C differ in at least d_{min} places. For this code, no error pattern of $d_{min} - 1$ or fewer errors can change one codeword into another. Therefore, any error pattern of $d_{min} - 1$ or fewer errors will result in a received vector $\mathbf{r}$ that is not a codeword in C. Hence, a block code with minimum distance d_{min} is capable of detecting all the error patterns of $d_{min} - 1$ or fewer errors. However, it cannot detect all the error patterns of d_{min} errors because there exists at least one pair of codewords that differ in d_{min} places, and there is an error pattern of d_{min} errors that will carry one into the other. For example, for the $(7, 4)$ Hamming code, if we receive $\mathbf{r} = (111\ 0010)$, this is a codeword and there is no any error detection. However, there is a possibility that the transmitter sent $\mathbf{u} = (111\ 1111)$ and the channel noise created $d_{min} = 3$ errors. The same argument applies to error patterns of more than d_{min} errors. For this reason, we say that the error-detection capability of a block code with minimum distance d_{min} is $d_{min} - 1$ [3, p. 78].

If a block code C with minimum distance d_{min} is used for error correction, one would like to know how many errors the code is able to correct. Suppose that the code can correct up to t errors, with $t \leq (d_{min} - 1)/2$. Let $\mathbf{c}$ be the transmitted codeword and $\mathbf{r}$ be the received vector with t or fewer errors, so that $d(\mathbf{c}, \mathbf{r}) \leq t$. To see that C can correct these errors, note that, if $\mathbf{w}$ is a codeword other than $\mathbf{c}$, then $d(\mathbf{r}, \mathbf{w}) \geq t + 1$. To see this note that if $d(\mathbf{r}, \mathbf{w}) \leq t$, then by the triangle inequality $d(\mathbf{c}, \mathbf{w}) \leq d(\mathbf{c}, \mathbf{r}) + d(\mathbf{r}, \mathbf{w}) \leq t + t \leq 2t$, contradicting the assumption that $t \leq (d_{min} - 1)/2$. In summary, a block code with minimum distance d_{min} guarantees correction of all the error patterns of

$t = \lfloor (d_{min} - 1)/2 \rfloor$ or fewer errors. The parameter $t = \lfloor (d_{min} - 1)/2 \rfloor$ is called the *error-correction* capability of the code [3, p. 81].

1.5 Channels and capacity

We consider a channel with input X, output Y, and transfer function $p_{Y|X}(y|x)$. Input X has probability mass function $p_X(x)$ and output Y has probability mass function $p_Y(y)$. The *mutual information* of X and Y is defined as [2, p. 10]

$$I(X;Y) = H(Y) - H(Y|X), \tag{1.5}$$

where $H(Y)$ is the *entropy* of the channel output,

$$H(Y) = -\sum_y p_Y(y) \log_2(p_Y(y)),$$

and $H(Y|X)$ is the *conditional entropy* of Y given X,

$$\begin{aligned} H(Y|X) &= -\sum_x p_X(x) H(Y|X=x) \\ &= -\sum_x \sum_y p_{X,Y}(x,y) \log_2(p_{Y|X}(y|x)) \\ &= -\sum_x \sum_y p_X(x) p_{Y|X}(y|x) \log_2(p_{Y|X}(y|x)). \end{aligned}$$

The mutual information can also be expressed as $I(X;Y) = H(X) - H(X|Y)$, which is sometimes useful. The capacity of a channel is defined as [2, p. 11]

$$C = \max_{p_X(x)} I(X;Y), \tag{1.6}$$

in which we see that the capacity is the maximum mutual information, where the maximization is over the channel input probability distribution $\{p_X(x)\}$.

The three most popular channels are the binary erasure channel (BEC), the binary symmetric channel (BSC), and the binary-input additive white-Gaussian-noise channel (BI-AWGNC). All these channels are named memoryless because their output, at any time instant, depends only on their input at that time instant. More precisely, for a sequence

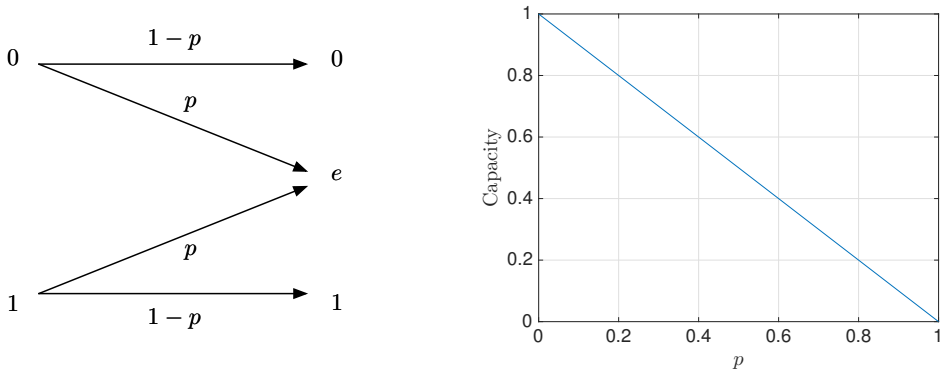

FIGURE 1.4: The BEC channel and its capacity.

of transmitted symbols $\mathbf{x} = [x_1, x_2, \ldots, x_N]$ and received symbols $\mathbf{y} = [y_1, y_2, \ldots, y_N]$:

$$p_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x}) = \prod_{i=1}^{N} p_{Y|X}(y_i|x_i). \tag{1.7}$$

A memoryless channel is therefore completely described by its input and output alphabets and the conditional probability distribution $p_{Y|X}(y|x)$ for each input-output symbol pair.

Specifically, a *binary erasure channel* with erasure probability p, denoted as BEC(p), is a channel with binary input, ternary output, and probability of erasure p. That is, let X be the transmitted random variable with alphabet $\{0, 1\}$ and let Y be the received variable with alphabet $\{0, 1, e\}$, where e is the erasure symbol. Then, the channel is characterized by the following transition probabilities:

$$\begin{aligned} p_{Y|X}(0|0) &= p_{Y|X}(1|1) = 1 - p, \\ p_{Y|X}(e|0) &= p_{Y|X}(e|1) = p, \\ p_{Y|X}(1|0) &= p_{Y|X}(0|1) = 0. \end{aligned}$$

The capacity of BEC(p) is [2, p. 12]

$$C_{BEC}(p) = 1 - p,$$

and is plotted in Figure 1.4.

The second channel which we consider is the *binary symmetric channel* with crossover probability p, denoted as $BSC(p)$. This is a channel with binary input, binary output,

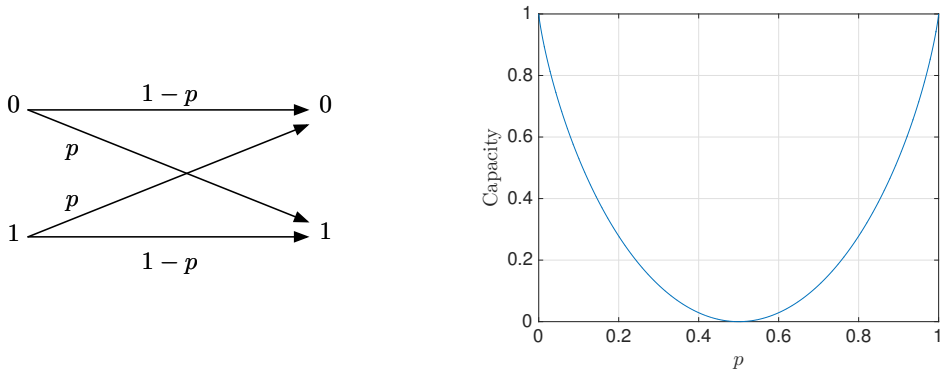

FIGURE 1.5: The BSC channel and its capacity.

and probability of error p. That is, if X is the channel input and Y the channel output, then

$$p_{Y|X}(0|0) = p_{Y|X}(1|1) = 1 - p,$$
$$p_{Y|X}(1|0) = p_{Y|X}(0|1) = p.$$

The capacity of BSC(p) is [2, p. 12]

$$C_{BSC}(p) = 1 - H(p),$$

and it is plotted in Figure 1.5.

The last channel which we consider is the *binary-input additive white-Gaussian-noise* channel, denoted as $BI\text{-}AWGN$, and depicted in Figure 1.6. If X is the channel input, Y the channel output, and Z the AWGN, then the channel can be described by the equation

$$Y = X + Z,$$

where $X \in \{\pm 1\}$ is the transmitted symbol, Y is the received symbol and Z is a real-valued additive white Gaussian noise (AWGN) sample with mean 0 and variance σ^2, i.e., $Z \sim \mathcal{N}(0, \sigma^2)$. The probability density function of Z is

$$p_Z(z) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{z^2}{2\sigma^2}}. \tag{1.8}$$

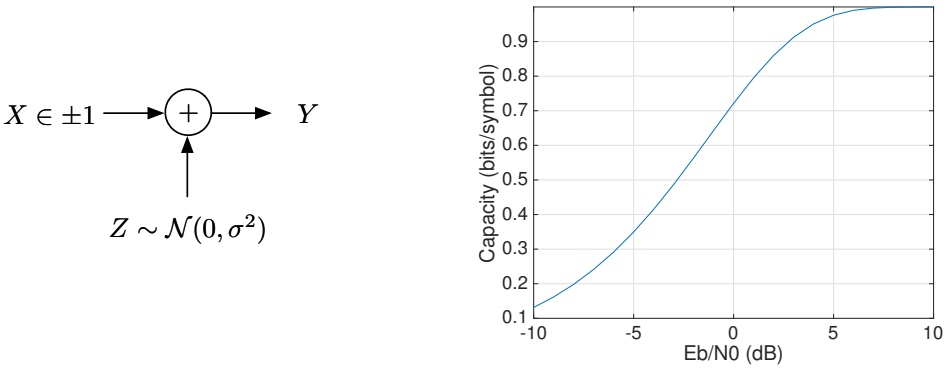

FIGURE 1.6: The binary AWGN channel and its capacity.

The capacity of the binary AWGN channel is [2, p. 14]:

$$C_{BI-AWGN} = -0.5 \log_2(2\pi e\sigma^2) - \int p_Y(y) \log_2(p_Y(y))dy, \tag{1.9}$$

where, assuming equiprobable source,

$$\begin{aligned} p_Y(y) &= p_{Y|X}(y|+1)p_X(+1) + p_{Y|X}(y|-1)p_X(-1) \\ &= \frac{1}{2}(p_{Y|X}(y|+1) + p_{Y|X}(y|-1)) \end{aligned}$$

and

$$p_{Y|X}(y|x = \pm 1) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{(-(y\mp 1)^2/2\sigma^2)}.$$

The capacity of the BI-AWGN channel is plotted in Figure 1.6 as a function of signal-to-noise-radio (SNR) measure E_b/N_0, where E_b is the average energy per information bit and $N_0 = 2\sigma^2$ is the two-sided power spectral density.

1.6 Maximum likelihood decoding

The most common error correction decoder is the *maximum likelihood* decoder [4, p. 19], as it will always choose the codeword that is most likely to have produced $\mathbf{y}$. Specifically, given a received vector $\mathbf{y}$, the ML decoder will choose the codeword $\mathbf{c}$ that maximizes the probability $p_{\mathbf{Y}|\mathbf{C}}(\mathbf{y}|\mathbf{c})$. The ML decoder returns the decoded codeword $\hat{\mathbf{c}}$ according

to the rule

$$\hat{\mathbf{c}} = \underset{\mathbf{c}\in\mathbf{C}}{\operatorname{argmax}}\ p_{\mathbf{Y}|\mathbf{C}}(\mathbf{y}|\mathbf{c}),$$

where assuming a memoryless channel,

$$p_{\mathbf{Y}|\mathbf{C}}(\mathbf{y}|\mathbf{c}) = \prod_{i=1}^{N} p_{Y_i|C_i}(y_i|c_i).$$

For a binary symmetric channel with crossover probability less than 0.5, the most likely codeword is the one that requires the fewest number of flipped bits to produce $\mathbf{y}$, since a bit is more likely to be received correctly than flipped. Then, the ML decoder is equivalent to choosing the codeword closest in Hamming distance to $\mathbf{y}$.

Example 1.6.1. We have the following codeword set

$$C = \{000000, 000111, 011100, 011011, 110001, 110110, 101010, 101101\}. \tag{1.10}$$

We transmit $\mathbf{c} = [011011]$ over a binary symmetric channel and we receive the vector $\mathbf{y} = [001011]$, which is not a codeword. So, the ML decoder will choose $\mathbf{c} = [011011]$ as the closest codeword, because it is the only codeword with hamming distance 1 from $\mathbf{y}$. If we transmit the same vector and the received vector is $\mathbf{y} = [011010]$, we have 2 bits flipped and the ML decoder will choose $\mathbf{c} = 011011$, which is wrong. This happens because the minimum distance of the code is 3 and the error-correcting capability of the code is $t = \lfloor (d_{min} - 1)/2 \rfloor = 1$.

When the channel has not binary output, the Hamming distance is replaced by the Euclidean distance described as

$$\|\mathbf{y} - \mathbf{x}\| = \sqrt{\sum_i |y_i - x_i|^2}.$$

Example 1.6.2. We assume the codeword set of (1.10) and suppose that now we transmit over a BI-AWGN channel. The codewords $\mathbf{c} = [c_1 \dots c_n]$ are transmitted by mapping

$\mathbf{c}$	$\mathbf{x}$	$\lvert\mathbf{y}-\mathbf{x}\rvert$
[0 0 0 0 0 0]	$[+1+1+1+1+1+1]$	6.1890
[0 0 0 1 1 1]	$[+1+1+1-1-1-1]$	5.5570
[0 1 1 1 0 0]	$[+1-1-1-1+1+1]$	7.2910
[0 1 1 0 1 1]	$[+1-1-1+1-1-1]$	7.1030
[1 1 0 0 0 1]	$[-1-1+1+1+1-1]$	5.9330
[1 1 0 1 1 0]	$[-1-1+1-1-1+1]$	4.5410
[1 0 1 0 1 0]	$[-1+1-1+1-1+1]$	5.2190
[1 0 1 1 0 1]	$[-1+1-1-1+1-1]$	6.1670

TABLE 1.1: Euclidean distances from the vector $\mathbf{y} = [-0.535 \ +0.217 \ +0.445 \ -0.111 \ -0.395 \ +0.190]$

the codeword bits $c_i \in \{0, 1\}$ to the symbols $x_i \in \{+1, -1\}$

$$0 \to +1$$

$$1 \to -1.$$

The X_i are then transmitted over the BI-AWGN channel which can be described as

$$Y_i = X_i + Z_i,$$

where $Z_i \sim \mathcal{N}(0, \sigma^2)$. Suppose that a codeword $\mathbf{c}$ is transmitted ($\mathbf{x} = -2\mathbf{c} + \mathbf{1}$) and that we receive the vector $\mathbf{y} = [-0.535 \ +0.217 \ +0.445 \ -0.111 \ -0.395 \ +0.190]$. Then, we can find the Euclidean distance of $\mathbf{y}$ from every vector $\mathbf{x}$ of the code. As wee see in the Table 1.1, the ML decoder will choose the codeword $\mathbf{c} = [1\ 1\ 0\ 1\ 1\ 0]$.

The ML decoder we used in example (1.6.2) is named *soft decision* decoder. If we would like to use *hard decision* decoder and we have received the same vector $\mathbf{y} = [-0.535 \ +0.217 \ +0.445 \ -0.111 \ -0.395 \ +0.190]$, we first convert it to [1 0 0 1 1 0] and find the Hamming distance from any codeword. In this case, the hard decision ML decoder would have chosen $\mathbf{c} = [1\ 1\ 0\ 1\ 1\ 0]$, as the codeword with minimum distance of the received vector. The same process we did in example (1.6.1) for the binary symmetric channel. In our example, soft and hard decisions gave us the same result, but generally the soft decision decoder is better because it uses more information.

1.7 Maximum a posteriori (MAP) decoding

A *maximum a posteriori (MAP) decoder* [4, p. 22] chooses the codeword $\mathbf{c}$ that maximizes $p_{\mathbf{C}|\mathbf{Y}}(\mathbf{c}|\mathbf{y})$, so $\hat{c}$ is chose according to

$$\hat{\mathbf{c}} = \underset{\mathbf{c}\in C}{\operatorname{argmax}}\, p_{\mathbf{C}|\mathbf{Y}}(\mathbf{c}|\mathbf{y}).$$

The probability $p_{C|Y}(\mathbf{c}|\mathbf{y})$ is called the *a posteriori probability*. We can continue according to Bayes and find that

$$\begin{aligned}\hat{\mathbf{c}} &= \underset{\mathbf{c}\in C}{\operatorname{argmax}}\, p_{\mathbf{C}|\mathbf{Y}}(\mathbf{c}|\mathbf{y}) \\ &= \underset{\mathbf{c}\in C}{\operatorname{argmax}}\, \frac{p_{\mathbf{Y}|\mathbf{C}}(\mathbf{y}|\mathbf{c})p_{\mathbf{C}}(\mathbf{c})}{p_{\mathbf{Y}}(\mathbf{y})}.\end{aligned}$$

Because the $p_{\mathbf{Y}}(\mathbf{y})$ is common for all the probabilities we have to compute, we can remove it. So, the final rule for MAP decoder is

$$\hat{\mathbf{c}} = \underset{\mathbf{c}\in C}{\operatorname{argmax}}\, p_{\mathbf{Y}|\mathbf{C}}(\mathbf{y}|\mathbf{c})p_{\mathbf{C}}(\mathbf{c}).$$

If each codeword is equally likely to have been sent, then the MAP and ML decoder give identical result. However, because MAP decoder takes into account the a priori information instead of ML decoder, we conclude that the MAP decoder is optimal.

Chapter 2

Factor Graphs

2.1 Introduction

A large number of computational problems have to deal with complicated global functions of many variables. These functions can be factorized into a product of simpler local functions to reduce the time complexity of many computational problems. Such a factorization can be visualized using a *factor graph*, that expresses which variables are arguments of which local functions.

2.2 Marginal functions

Let $x_1, x_2, \ldots, x_n$ be a collection of variables. For every function $f(x_1, \ldots, x_n)$, there are n *marginal* functions $f_i(x_i)$. Let $f(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$ be a function of seven variables. We denote the *marginal* of f with respect to x_1, as the summation over all variables contained in function, except x_1

$$f(x_1) = \sum_{x_2,x_3,x_4,x_5,x_6,x_7} f(x_1, x_2, x_3, x_4, x_5, x_6, x_7) = \sum_{\sim x_1} f(x_1, x_2, x_3, x_4, x_5, x_6, x_7).$$

Assuming that all variables take values in a finite alphabet X, the time complexity of determining $f(x_1)$ for all values of x_1 by brute force is $\Theta(|X|^7)$. Suppose that f can be expressed as a product of five factors.

$$f(x_1, \ldots, x_7) = f_1(x_1, x_2, x_3) f_2(x_1, x_4, x_5) f_3(x_4) f_4(x_5, x_6) f_5(x_5, x_7). \tag{2.1}$$

This product can be visualized as a *factor graph*. A factor graph is a *bipartite graph*, which has two types of nodes, variable nodes and factor nodes, and edges that connect only different types of nodes. We see that the factor graph for our particular example

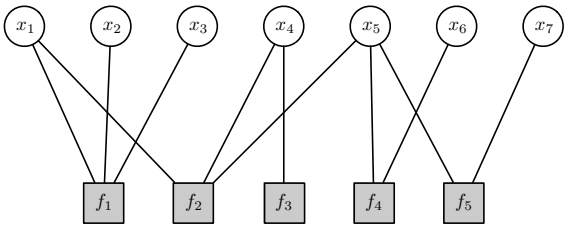

FIGURE 2.1: A factor graph of the right-handside of (2.1).

has no cycles. This means that there is one and only one path between each pair of nodes. Such factor graphs are called and can be visualized as *trees*. Generally, suppose

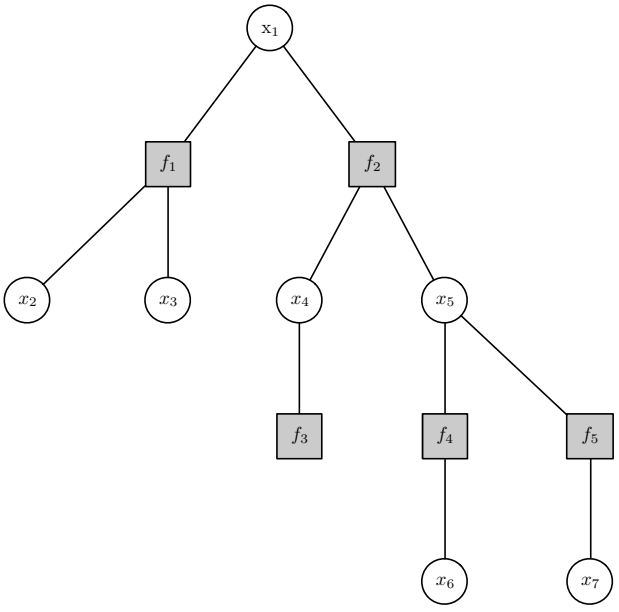

FIGURE 2.2: A tree representation of (2.1).

that we have a function g and we are interested in marginalizing g with respect to the variable z. If the factor graph of g is a bipartite tree, g can be factorized as

$$g(z, \dots) = \prod_{k=1}^{K} [g_k(z, \dots)] \tag{2.2}$$

for some integer K with the property that z appears in every factor g_k, but all other variables appear in *only one* factor.

From the marginal definition, we have

$$g(z) = \sum_{\sim z} g(z, \dots). \tag{2.3}$$

If we use (2.2) in (2.3), we have

$$\sum_{\sim z} g(z, \dots) = \underbrace{\sum_{\sim z} \prod_{k=1}^{K} [g_k(z, \dots)]}_{\text{marginal of product}} = \underbrace{\prod_{k=1}^{K} \left[\sum_{\sim z} g_k(z, \dots) \right]}_{\text{product of marginals}}. \tag{2.4}$$

In function f of (2.1), $K = 2$, and the marginal of product is

$$f(x_1) = \sum_{\sim x_1} [f_1(x_1, x_2, x_3)][f_2(x_1, x_4, x_5) f_3(x_4) f_4(x_5, x_6) f_5(x_5, x_7)]$$

which has $\Theta(|X|^7)$ time complexity, and the product of marginals is

$$f(x_1) = \left[\sum_{x_2, x_3} f_1(x_1, x_2, x_3) \right] \left[\sum_{x_4, x_5, x_6, x_7} f_2(x_1, x_4, x_5) f_3(x_4) f_4(x_5, x_6) f_5(x_5, x_7) \right]$$

which has $\Theta(|X|^5)$ time complexity. The complexity has been reduced, but we can do better if every marginal of product becomes product of marginals.

In general, each g_k is a product of factors. In our example, the one product of factors is $f_1(x_1, x_2, x_3)$ and the other is $f_2(x_1, x_4, x_5) f_3(x_4) f_4(x_5, x_6) f_5(x_5, x_7)$. Since the factor graph is a bipartite tree, g_k must in turn have a generic factorization of the form

$$g_k(z, \dots) = \underbrace{h(z, z_1, \dots, z_J)}_{\text{kernel}} \prod_{j=1}^{J} \underbrace{[h_j(z_j, \dots)]}_{\text{factors}}, \tag{2.5}$$

where z appears only in the "kernel" and each of the z_j appears *at most twice*, possibly in the kernel and in at most one of the "factors". For our running example, we have

$$f_1(x_1, x_2, x_3) = \underbrace{f_1(x_1, x_2, x_3)}_{\text{kernel}} \underbrace{[1]}_{x_2} \underbrace{[1]}_{x_3} \tag{2.6}$$

and

$$f_2(x_1, x_4, x_5) f_3(x_4) f_4(x_5, x_6) f_5(x_5, x_7) = \underbrace{f_2(x_1, x_4, x_5)}_{\text{kernel}} \underbrace{[f_3(x_4)]}_{x_4} \underbrace{[f_4(x_5, x_6) f_5(x_5, x_7)]}_{x_5}$$

We can go further and say that

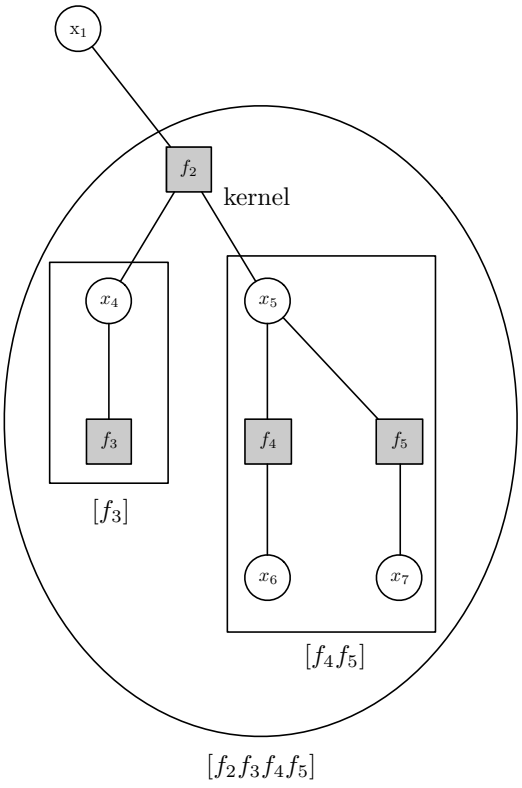

FIGURE 2.3: The particular instance of f_2 factorization.

$$f_4(x_5, x_6) = \underbrace{f_4(x_5, x_6)}_{\text{kernel}} \underbrace{[1]}_{x_6}$$

and

$$f_5(x_5, x_7) = \underbrace{f_5(x_5, x_7)}_{\text{kernel}} \underbrace{[1]}_{x_7}.$$

From equation (2.5), we have

$$\begin{aligned} \sum_{\sim z} g_k(z, \dots) &= \sum_{\sim z} \left[h(z, z_1, \dots, z_J) \prod_{j=1}^{J} [h_j(z_j, \dots)] \right] \\ &= \sum_{\sim z} \left[h(z, z_1, \dots, z_J) \underbrace{\prod_{j=1}^{J} \left(\sum_{\sim z_j} h_j(z_j, \dots) \right)}_{\text{product of marginals}} \right], \end{aligned} \tag{2.7}$$

which says that the desired marginal $g_k(z)$ can be computed by multiplying the kernel h with the individual marginals and summing all the remaining variables other than z. So, we continue to analyze every factor until we reach the leaves of the tree. The calculation of the marginal then follows the recursive splitting in reverse. In general, nodes in the graph compute marginals and send them to the next level. This is the idea of *message passing*.

2.3 Message passing

The algorithm proceeds by sending messages along the edges of the tree. Message passing starts at the leaf nodes. If the leaf node is variable node, we have from (2.6) that node sends the constant function 1 to its parent node. If the leaf node is a function node, then it has the generic form $g_k(z)$, so that $\sum_{\sim z} g_k(z) = g_k(z)$ and it sends the function itself.

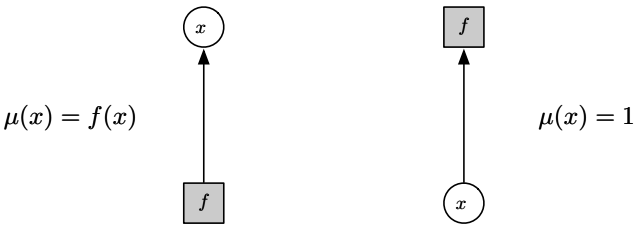

FIGURE 2.4: Initialization at leaf nodes.

Now, we have to find the message passing rules after the initialization. Suppose a variable node has received messages from all its children. The message that will be sent is the product of its incoming messages because, according to (2.4), the marginal of products is the product of marginals. Suppose now that a factor node has received messages from all its children. The message that will be sent, according to (2.7), is the product of the incoming messages times the kernel of this function f, after summing all variable nodes except the node that the message will be sent.

Now, it is easy to perform an example of marginalization via message passing. Suppose we have the function f of (2.1) and we want to marginalize f with respect to the variable x_1. We have the tree from Figure 2.2.

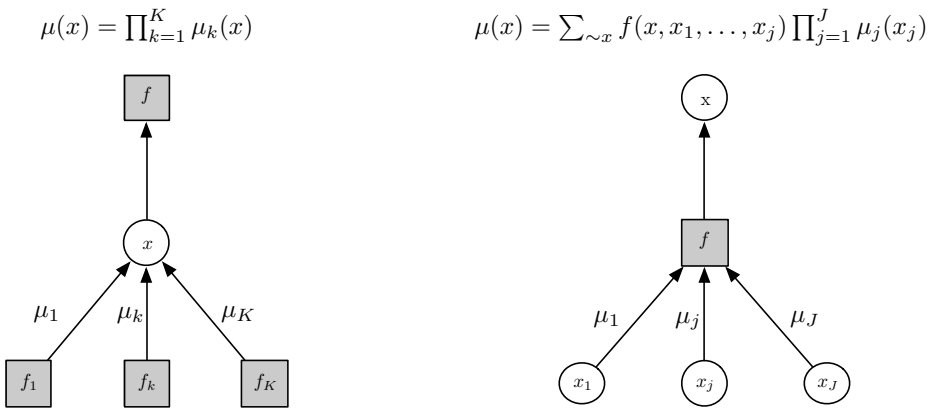

FIGURE 2.5: Variable/Function node processing.

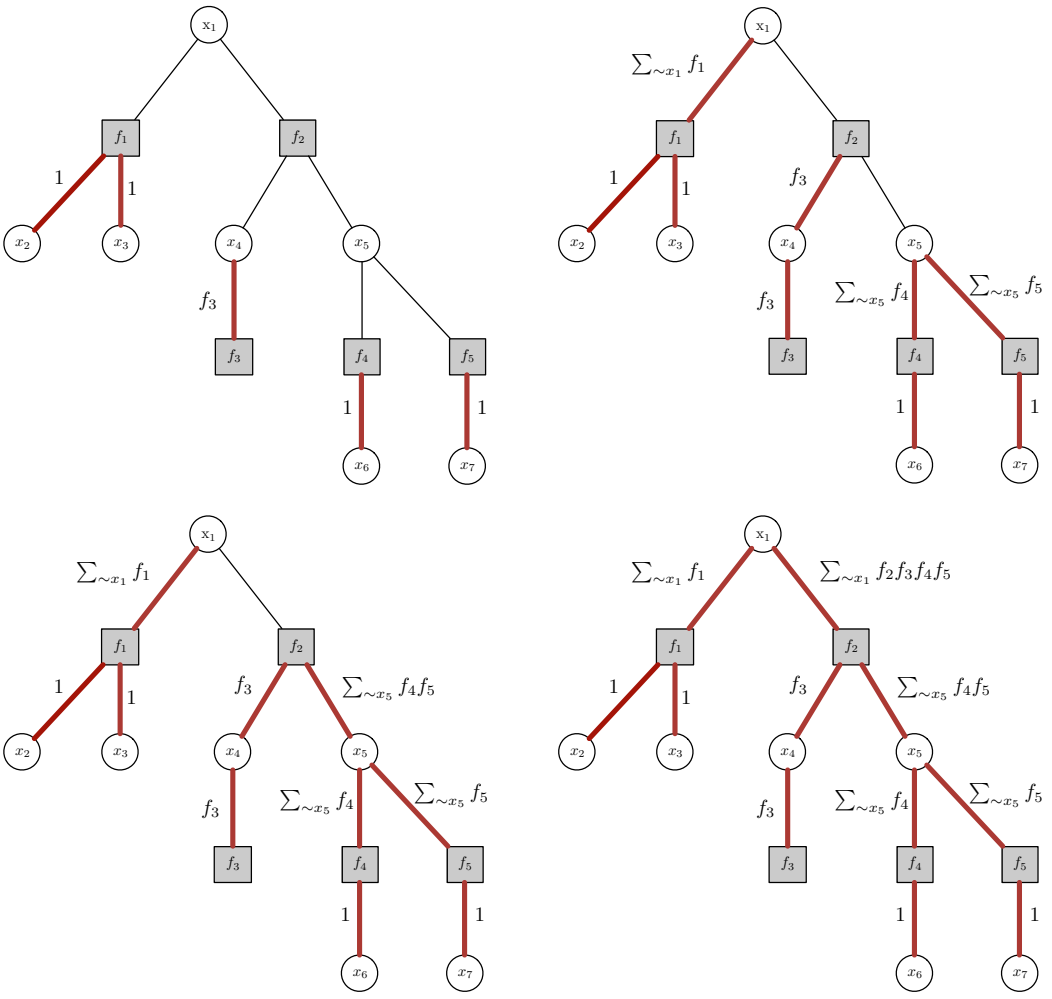

FIGURE 2.6: The 4 steps of marginalization of function f of (2.1) via message passing.

The messages are generated as follows

Step 1

$$\mu_{x_2 \to f_1}(x_2) = 1$$

$$\mu_{x_3 \to f_1}(x_3) = 1$$

$$\mu_{f_3 \to x_4}(x_4) = f_3(x_4)$$

$$\mu_{x_6 \to f_4}(x_6) = 1$$

$$\mu_{x_7 \to f_5}(x_7) = 1$$

Step 2

$$\mu_{f_1 \to x_1}(x_1) = \sum_{x_2, x_3} f_1(x_1, x_2, x_3)[1][1] = \sum_{x_2, x_3} f_1(x_1, x_2, x_3)$$

$$\mu_{x_4 \to f_2}(x_4) = f_3(x_4)$$

$$\mu_{f_4 \to x_5}(x_5) = \sum_{x_6} f_4(x_5, x_6)[1] = \sum_{x_6} f_4(x_5, x_6)$$

$$\mu_{f_5 \to x_5}(x_5) = \sum_{x_7} f_5(x_5, x_7)[1] = \sum_{x_7} f_5(x_5, x_7)$$

Step 3

$$\mu_{x_5 \to f_2}(x_5) = \sum_{x_6} f_4(x_5, x_6) \sum_{x_7} f_5(x_5, x_7)$$

Step 4

$$\mu_{f_2 \to x_1}(f_2) = \sum_{x_4, x_5} \left[f_2(x_1, x_4, x_5) f_3(x_4) \sum_{x_6} f_4(x_5, x_6) \sum_{x_7} f_5(x_5, x_7) \right]$$

Termination

$$f(x_1) = \left[\sum_{x_2, x_3} f_1(x_1, x_2, x_3) \right] \left[\sum_{x_4, x_5} \left(f_2(x_1, x_4, x_5) f_3(x_4) \sum_{x_6} f_4(x_5, x_6) \sum_{x_7} f_5(x_5, x_7) \right) \right]$$

which is the marginal $\sum_{\sim x_1} f_1 f_2 f_3 f_4 f_5$. The first factor has $\Theta(|X|^2)$ complexity and the second factor has $\Theta(|X|^3)$. Since there are $|X|$ values for x_1, the overall task has complexity $\Theta(|X|^4)$. This compares favorably to the complexities we found before for the marginal computations.

We saw how we can marginalize a function with respect to a *single* variable. For decoding, the case of interest is the marginalization over all variables. Such a computation might be accomplished by drawing a tree rooted in this variable and applying the algorithm on each tree. However, this approach is not efficient because some subcomputations will be the same. So, we perform the algorithm simultaneously on a single tree. We start at all leaf nodes and for every edge we compute the outgoing message as soon as we have received the incoming messages along all other edges that connect to the given node. We continue this until a message has been sent in both directions along every edge. Let's see an example on our previous function f of 2.1.

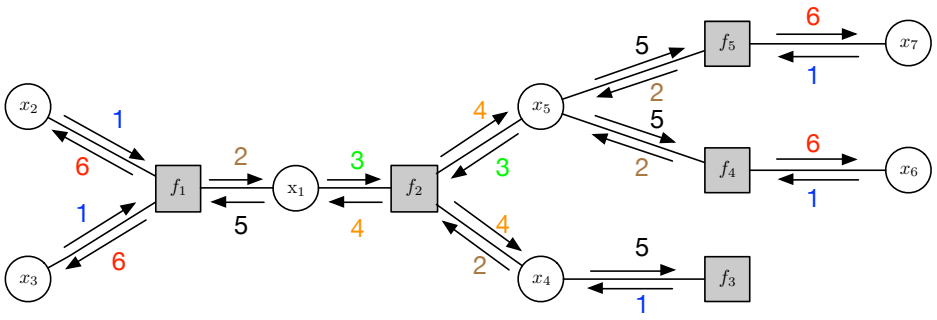

FIGURE 2.7: Messages generated in each step of the message passing algorithm.

The messages are generated as follows

Step 1

$\mu_{x_2 \to f_1}(x_2) = 1$

$\mu_{x_3 \to f_1}(x_3) = 1$

$\mu_{x_7 \to f_5}(x_7) = 1$

$\mu_{x_6 \to f_4}(x_6) = 1$

$\mu_{f_3 \to x_4}(x_4) = f_3(x_4)$

Step 2

$$\mu_{f_1 \to x_1}(x_1) = \sum_{x_2, x_3} f_1(x_1, x_2, x_3)$$

$$\mu_{f_5 \to x_5}(x_5) = \sum_{x_7} f_5(x_5, x_7)$$

$$\mu_{f_4 \to x_5}(x_5) = \sum_{x_6} f_4(x_5, x_6)$$

$$\mu_{x_4 \to f_2}(x_4) = f_3(x_4)$$

Step 3

$$\mu_{x_1 \to f_2}(x_1) = \mu_{f_1 \to x_1}(x_1) = \sum_{x_2, x_3} f_1(x_1, x_2, x_3)$$

$$\mu_{x_5 \to f_2}(x_5) = \mu_{f_4 \to x_5}(x_5)\mu_{f_5 \to x_5}(x_5) = \sum_{x_6} f_4(x_5, x_6) \sum_{x_7} f_5(x_5, x_7)$$

Step 4

$$\begin{aligned}
\mu_{f_2 \to x_5}(x_5) &= \sum_{x_1, x_4} [f_2(x_1, x_4, x_5)\mu_{x_4 \to f_2}(x_4)\mu_{x_1 \to f_2}(x_1)] \\
&= \sum_{x_1, x_4} \left[f_2(x_1, x_4, x_5) f_3(x_4) \sum_{x_2, x_3} f_1(x_1, x_2, x_3) \right] \\
\mu_{f_2 \to x_4}(x_4) &= \sum_{x_1, x_5} [f_2(x_1, x_4, x_5)\mu_{x_1 \to f_2}(x_1)\mu_{x_5 \to f_2}(x_5)] \\
&= \sum_{x_1, x_5} \left[f_2(x_1, x_4, x_5) \sum_{x_2, x_3} f_1(x_1, x_2, x_3) \sum_{x_6} f_4(x_5, x_6) \sum_{x_7} f_5(x_5, x_7) \right] \\
\mu_{f_2 \to x_1}(x_1) &= \sum_{x_4, x_5} [f_2(x_1, x_4, x_5)\mu_{x_4 \to f_2}(x_4)\mu_{x_5 \to f_2}(x_5)] \\
&= \sum_{x_4, x_5} \left[f_2(x_1, x_4, x_5) f_3(x_4) \sum_{x_6} f_4(x_5, x_6) \sum_{x_7} f_5(x_5, x_7) \right]
\end{aligned}$$

Step 5

$$\mu_{x_1\to f_1}(x_1) = \mu_{f_2\to x_1}(x_1)$$
$$= \sum_{x_4,x_5}\left[f_2(x_1,x_4,x_5)f_3(x_4)\sum_{x_6}f_4(x_5,x_6)\sum_{x_7}f_5(x_5,x_7)\right]$$
$$\mu_{x_5\to f_5}(x_5) = \mu_{f_2\to x_5}(x_5)\mu_{f_4\to x_5}(x_5)$$
$$= \sum_{x_6}f_4(x_5,x_6)\sum_{x_1,x_4}\left[f_2(x_1,x_4,x_5)f_3(x_4)\sum_{x_2,x_3}f_1(x_1,x_2,x_3)\right]$$
$$\mu_{x_5\to f_4}(x_5) = \mu_{f_5\to x_5}(x_5)\mu_{f_2\to x_5}(x_5)$$
$$= \sum_{x_7}f_5(x_5,x_7)\sum_{x_1,x_4}\left[f_2(x_1,x_4,x_5)f_3(x_4)\sum_{x_2,x_3}f_1(x_1,x_2,x_3)\right]$$
$$\mu_{x_4\to f_3}(x_4) = \mu_{f_2\to x_4}(x_4)$$
$$= \sum_{x_1,x_5}\left[f_2(x_1,x_4,x_5)\sum_{x_2,x_3}f_1(x_1,x_2,x_3)\sum_{x_6}f_4(x_5,x_6)\sum_{x_7}f_5(x_5,x_7)\right]$$

Step 6

$$\mu_{f_5\to x_7}(x_7) = \sum_{x_5}[f_5(x_5,x_7)\mu_{x_5\to f_5}(x_5)]$$
$$= \sum_{x_5}\left[f_5(x_5,x_7)\sum_{x_6}f_4(x_5,x_6)\sum_{x_1,x_4}\left(f_2(x_1,x_4,x_5)f_3(x_4)\sum_{x_2,x_3}f_1(x_1,x_2,x_3)\right)\right]$$
$$\mu_{f_4\to x_6}(x_6) = \sum_{x_5}[f_4(x_5,x_6)\mu_{x_5\to f_4}(x_5)]$$
$$= \sum_{x_5}\left[f_4(x_5,x_6)\sum_{x_7}f_5(x_5,x_7)\sum_{x_1,x_4}\left(f_2(x_1,x_4,x_5)f_3(x_4)\sum_{x_2,x_3}f_1(x_1,x_2,x_3)\right)\right]$$
$$\mu_{f_1\to x_2}(x_2) = \sum_{x_1}[f_1(x_1,x_2,x_3)\mu_{x_1\to f_1}(x_1)\mu_{x_3\to f_1}(x_3)]$$
$$= \sum_{x_1}\left[f_1(x_1,x_2,x_3)\sum_{x_4,x_5}\left(f_2(x_1,x_4,x_5)f_3(x_4)\sum_{x_6}f_4(x_5,x_6)\sum_{x_7}f_5(x_5,x_7)\right)\right]$$
$$\mu_{f_1\to x_3}(x_3) = \sum_{x_1}[f_1(x_1,x_2,x_3)\mu_{x_2\to f_1}(x_2)\mu_{x_1\to f_1}(x_1)]$$
$$= \sum_{x_1}\left[f_1(x_1,x_2,x_3)\sum_{x_4,x_5}\left(f_2(x_1,x_4,x_5)f_3(x_4)\sum_{x_6}f_4(x_5,x_6)\sum_{x_7}f_5(x_5,x_7)\right)\right]$$

Termination

$$
\begin{aligned}
f(x_1) &= \mu_{f_1 \to x_1}(x_1)\mu_{f_2 \to x_1}(x_1) \\
&= \left[\sum_{x_2,x_3} f_1(x_1,x_2,x_3)\right]\left[\sum_{x_4,x_5}\left(f_2(x_1,x_4,x_5)f_3(x_4)\sum_{x_6} f_4(x_5,x_6)\sum_{x_7} f_5(x_5,x_7)\right)\right] \\
f(x_2) &= \mu_{f_1 \to x_2}(x_2) \\
&= \sum_{x_1}\left[f_1(x_1,x_2,x_3)\sum_{x_4,x_5}\left(f_2(x_1,x_4,x_5)f_3(x_4)\sum_{x_6} f_4(x_5,x_6)\sum_{x_7} f_5(x_5,x_7)\right)\right] \\
f(x_3) &= \mu_{f_1 \to x_3}(x_3) \\
&= \sum_{x_1}\left[f_1(x_1,x_2,x_3)\sum_{x_4,x_5}\left(f_2(x_1,x_4,x_5)f_3(x_4)\sum_{x_6} f_4(x_5,x_6)\sum_{x_7} f_5(x_5,x_7)\right)\right] \\
f(x_4) &= \mu_{f_3 \to x_4}(x_4)\mu_{f_2 \to x_4}(x_4) \\
&= f_3(x_4)\sum_{x_1,x_5}\left[f_2(x_1,x_4,x_5)\sum_{x_2,x_3} f_1(x_1,x_2,x_3)\sum_{x_6} f_4(x_5,x_6)\sum_{x_7} f_5(x_5,x_7)\right] \\
f(x_5) &= \mu_{f_2 \to x_5}(x_5)\mu_{f_5 \to x_5}(x_5)\mu_{f_4 \to x_5}(x_5) \\
&= \left[\sum_{x_7} f_5(x_5,x_7)\right]\left[\sum_{x_6} f_4(x_5,x_6)\right]\left[\sum_{x_1,x_4}\left(f_2(x_1,x_4,x_5)f_3(x_4)\sum_{x_2,x_3} f_1(x_1,x_2,x_3)\right)\right] \\
f(x_6) &= \mu_{f_4 \to x_6}(x_6) \\
&= \sum_{x_5}\left[f_4(x_5,x_6)\sum_{x_7} f_5(x_5,x_7)\sum_{x_1,x_4}\left(f_2(x_1,x_4,x_5)f_3(x_4)\sum_{x_2,x_3} f_1(x_1,x_2,x_3)\right)\right] \\
f(x_7) &= \mu_{f_5 \to x_7}(x_7) \\
&= \sum_{x_5}\left[f_5(x_5,x_7)\sum_{x_6} f_4(x_5,x_6)\sum_{x_1,x_4}\left(f_2(x_1,x_4,x_5)f_3(x_4)\sum_{x_2,x_3} f_1(x_1,x_2,x_3)\right)\right]
\end{aligned}
$$

We saw how we compute marginals over all variables. Since the messages represent probabilities or *beliefs*, the algorithm is also known as the *belief propagation* (BP) algorithm.

2.4 Sum product decoding with example

Assume we transmit over a binary-input ($X_i \in \{\pm 1\}$) memoryless ($p_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x}) = \prod_{i=1}^{n} p_{Y_i|X_i}(y_i|x_i)$) channel using a linear code C defined by its parity-check matrix H and assume that codewords are chosen uniformly at random. We saw that that the rule for the MAP decoder is

$$\begin{aligned}\hat{\mathbf{x}}^{MAP}(\mathbf{y}) &= \arg\max_{x_i \in \pm 1} \sum_{\sim x_i} p_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x}) p_{\mathbf{X}}(\mathbf{x}) \\ &= \arg\max_{x_i \in \pm 1} \sum_{\sim x_i} \left(\prod_j p_{Y_j|X_j}(y_j|x_j) \right) \mathbb{1}_{\{\mathbf{x} \in C\}},\end{aligned}$$

where in the last step we have used the fact that the channel is memoryless and that codewords have uniform prior. Each term $\mathbb{1}_{\{.\}}$ is an *indicator function*: it is 1 if the condition inside the braces is fulfilled and 0 otherwise.

A message $\mu(x)$ can be though of as a real-valued vector of length 2, $(\mu(1), \mu(-1))$. The initial such message sent from the factor leaf node representing the i-th channel realization to the variable node i is $(p_{Y|X}(y_i|1)), p_{Y|X}(y_i|-1))$. Recall that, at a variable node of degree $K+1$, the message passing rule calls for a pointwise multiplication

$$\mu(1) = \prod_{k=1}^{K} \mu_k(1), \qquad \mu(-1) = \prod_{k=1}^{K} \mu_k(-1).$$

Introduce the ratio $r_k = \frac{\mu_k(1)}{\mu_k(-1)}$. These ratios are the likelihood ratios associated with the channel observations. We have

$$r = \frac{\mu(1)}{\mu(-1)} = \frac{\prod_{k=1}^{K} \mu_k(1)}{\prod_{k=1}^{K} \mu_k(-1)} = \prod_{k=1}^{K} r_k, \tag{2.8}$$

so the ratio of the outgoing message at a variable node is the product of the incoming ratios. If we define the log-likelihood ratio $l_k = \ln(r_k)$, then the processing rule is simplified to $l = \sum_{k=1}^{K} l_k$, which is widely used as processing rule for the **variable nodes**.

At the check nodes, the processing rule is slightly more complex. Consider the ratio of an outgoing message at a check node which has degree $J+1$. The associated kernel will

be

$$f(x, x_1, \ldots, x_J) = \mathbb{1}_{[\prod_{j=1}^J x_j = x]}.$$

We have a product instead of a modulo-2 sum since we have assumed that X_i takes values in $\{\pm 1\}$ and not in $\{0, 1\}$. For the outgoing message of a check node we have

$$\begin{aligned} r = \frac{\mu(1)}{\mu(-1)} &= \frac{\sum_{\sim x} f(1, x_1, \ldots, x_J) \prod_{j=1}^J \mu_j(x_j)}{\sum_{\sim x} f(-1, x_1, \ldots, x_J) \prod_{j=1}^J \mu_j(x_j)} \\ &= \frac{\sum_{x_1,\ldots,x_J : \prod_{j=1}^J x_j = 1} \prod_{j=1}^J \mu_j(x_j)}{\sum_{x_1,\ldots,x_J : \prod_{j=1}^J x_j = -1} \prod_{j=1}^J \mu_j(x_j)} \\ &= \frac{\sum_{x_1,\ldots,x_J : \prod_{j=1}^J x_j = 1} \prod_{j=1}^J \frac{\mu_j(x_j)}{\mu_j(-1)}}{\sum_{x_1,\ldots,x_J : \prod_{j=1}^J x_j = -1} \prod_{j=1}^J \frac{\mu_j(x_j)}{\mu_j(-1)}} \\ &= \frac{\sum_{x_1,\ldots,x_J : \prod_{j=1}^J x_j = 1} \prod_{j=1}^J r_j^{(1+x_j)/2}}{\sum_{x_1,\ldots,x_J : \prod_{j=1}^J x_j = -1} \prod_{j=1}^J r_j^{(1+x_j)/2}} \\ &= \frac{\prod_{j=1}^J (r_j + 1) + \prod_{j=1}^J (r_j - 1)}{\prod_{j=1}^J (r_j + 1) - \prod_{j=1}^J (r_j - 1)}. \end{aligned}$$

For the last step, we use the fact that

$$\prod_{j=1}^J (r_j + 1) + \prod_{j=1}^J (r_j - 1) = 2 \prod_{x_1,\ldots,x_J : \prod_{j=1}^J x_j = 1} r_j^{(1+x_j)/2},$$

which can be applied to both the numerator and the denominator in order to get the last equation.

If we divide both numerator and denominator by $\prod_{j=1}^J (r_j + 1)$, we see that

$$r = \frac{1 + \prod_{j=1}^J \frac{r_j - 1}{r_j + 1}}{1 - \prod_{j=1}^J \frac{r_j - 1}{r_j + 1}},$$

which in turns implies that

$$\frac{r - 1}{r + 1} = \prod_{j=1}^J \frac{(r_j - 1)}{r_j + 1}.$$

Recall that l denotes the message from a variable node in the log-likelihood form. Then, $r = e^l$ and we see that

$$\frac{r-1}{r+1} = \tanh(l/2).$$

Combining these two statements, we have

$$\tanh(l/2) = \frac{r-1}{r+1} = \prod_{j=1}^{J} \frac{r_j - 1}{r_j + 1} = \prod_{j=1}^{J} \tanh(l_j/2),$$

so that the final processing rule for the **check node** is

$$l = 2\tanh^{-1}\left(\prod_{j=1}^{J} \tanh(l_j/2)\right). \tag{2.9}$$

At the end of each decoding iteration, the overall LLR is calculated for each variable node and its estimated value is changed accordingly. If, at any point, the estimated vector is found to be a codeword, decoding stops declaring a success.

Example 2.4.1. We have a code C described by the parity check matrix

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix},$$

and we transmit the codeword

$$\mathbf{c} = [0\ 0\ 1\ 0\ 1\ 1].$$

The vector $\mathbf{c}$ is sent through a binary AWGN channel where the noise has mean 0 and variance $\sigma^2 = 0.5$ and the received signal is

$$\mathbf{y} = [-0.83\ +0.72\ -1.35\ +1.1\ -0.98\ -1.59].$$

The log-likelihood ratios will be

$$\begin{aligned} l(y) &= \log \frac{p_{Y|X}(y|+1)}{p_{Y|X}(y|-1)} \\ &= \log \frac{\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y-1)^2}{2\sigma^2}\right]}{\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y+1)^2}{2\sigma^2}\right]} \\ &= \log \exp\left[-\frac{(y-1)^2}{2\sigma^2} + \frac{(y+1)^2}{2\sigma^2}\right] \\ &= \frac{2}{\sigma^2} y. \end{aligned}$$

Then

$$\mathbf{R} = [-6.64 \ +5.76 \ -10.80 \ +8.80 \ -7.84 \ -12.72].$$

To begin decoding, we set the maximum number of iterations to 2. At initialization,

$$M_{j,i} = L_i.$$

The first variable node is connected only to the first and third check node. So, the message L_1 will be sent only to these check nodes:

$$M_{1,1} = L_1 = -6.64 \quad \text{and} \quad M_{3,1} = L_1 = -6.64.$$

Repeating this for the remaining variable nodes gives

$$\begin{aligned} &\text{for } i = 2, \quad M_{1,2} = R_2 = +5,76, \quad M_{2,2} = R_2 = +5.76 \\ &\text{for } i = 3, \quad M_{2,3} = R_3 = -10.8, \quad M_{4,3} = R_3 = -10.8 \\ &\text{for } i = 4, \quad M_{1,4} = R_4 = +8.8, \quad M_{4,4} = R_4 = +8.8 \\ &\text{for } i = 5, \quad M_{2,5} = R_5 = -7.84, \quad M_{3,5} = R_5 = -7.84 \\ &\text{for } i = 6, \quad M_{3,6} = R_6 = -12.72, \quad M_{4,6} = R_6 = -12.72. \end{aligned}$$

Now, the extrinsic probabilities are calculated for the check to variable node messages. The first parity check node is connected to the first, second and fourth variable nodes. So, according to rule (2.9), the probability from the first check node to the first variable

node depends on the probabilities of the second and fourth variable nodes

$$\begin{aligned} E_{1,1} &= 2\tanh^{-1}\Big(\tanh(M_{1,2}/2)\tanh(M_{1,4}/2)\Big) \\ &= 2\tanh^{-1}\Big(\tanh(5.76/2)\tanh(8.8/2)\Big) \\ &= 5.7133. \end{aligned}$$

Similarly, the extrinsic probability from the first check node to the second variable node depends on the probabilities of the first and fourth variable nodes

$$\begin{aligned} E_{1,2} &= 2\tanh^{-1}\Big(\tanh(M_{1,1}/2)\tanh(M_{1,4}/2)\Big) \\ &= -6.5309, \end{aligned}$$

and the extrinsic probability from the first check node to the fourth variable node depends on the LLRs sent from the first and second variable nodes to the first check node

$$\begin{aligned} E_{1,4} &= 2\tanh^{-1}\Big(\tanh(M_{1,1}/2)\tanh(M_{1,2}/2)\Big) \\ &= -5.413 \end{aligned}$$

Repeating for all check nodes gives the extrinsic LLRs

$$E = \begin{bmatrix} 5.1733 & -6.5309 & . & -5.143 & . & . \\ . & 7.7895 & -5.6423 & . & -5.7535 & . \\ -7.8324 & . & . & . & 6.6377 & 6.3767 \\ . & . & -8.7804 & 10.6632 & . & -8.6731 \end{bmatrix}$$

To save space, the extrinsic LLRs are given in matrix form, where the (j,i)-th entry of E holds $E_{j,i}$, which is the message that is going from the jth check node to the ith variable node. A dot entry indicates that an LLR does not exist for that i and j.

After the messages pass from the variable nodes to check nodes and return back to variable nodes, one iteration has been completed. After every iteration, the message-passing algorithm calculates the overall LLR for every variable node, makes a hard decision and checks if the vector forms a codeword. The first variable node has extrinsic LLRs from the first and third checks and an intrinsic LLR from the channel. The total

LLR is their sum

$$L_1 = R_1 + E_{1,1} + E_{3,1} = -6.64 + 5.7133 - 7.8324 = -8.7591.$$

Repeating for the second to sixth variable nodes gives

$$\begin{aligned} L_2 &= R_2 + E_{1,2} + E_{2,2} = +7.0186 \\ L_3 &= R_3 + E_{2,3} + E_{4,3} = -25.2227 \\ L_4 &= R_4 + E_{1,4} + E_{4,4} = +14.3202 \\ L_5 &= R_5 + E_{2,5} + E_{3,5} = -6.9558 \\ L_6 &= R_6 + E_{3,6} + E_{4,6} = -15.0164. \end{aligned}$$

The hard decision on the received bits is simply given by the signs of the LLRs

$$\hat{\mathbf{c}} = [101011].$$

For $\hat{\mathbf{c}}$ being a valid codeword, we must have

$$\hat{\mathbf{c}} H^T = \mathbf{0},$$

which is not true for the particular $\hat{\mathbf{c}}$, so the decoding will continue with the second iteration and so on until either $\hat{\mathbf{c}}$ is a valid codeword either the total iterations reached.

Chapter 3

LDPC

3.1 Introduction

Low-density parity-check (LDPC) codes form a class of Shannon limit (or channel capacity) approaching codes. LDPC codes were invented by Gallager in the early 1960s [5]. Unfortunately, Gallager's remarkable discovery was mostly ignored by coding researchers for almost 20 years, until Tanner's work in 1981 [6], in which he provided a new interpretation of LDPC codes from a graphical point of view. Tanner's work was also ignored by coding theorists for another 14 years, until the late 1990s, where MacKay and other coding researchers began to investigate codes on graphs and iterative decoding [7, 8]. Their research resulted in the rediscovery of Gallager's LDPC codes and further generalizations. Long LDPC codes have been shown to achieve error performance near the Shannon limit [9, 10]. This makes LDPC codes important and popular in communication systems. Some applications of LDPC codes include

- DVB-S2 standard for the satellite transmission of digital television,
- 10GBase-T Ethernet, which sends data at 10 gigabits per second over twisted-pair cables,
- Wi-Fi 802.11 standard as an optional part of 802.11n and 802.11ac, in the High Throughput (HT) PHY specification,
- forward error correction (FEC) system for the ITU-T G.hn standard,

- WiMAX IEEE 802.16 for microwave communications,
- WRAN IEEE 802.22 for wireless broadband access that uses the so-called white spaces between occupied channels in the TV frequency spectrum.

Although Gallager proposed LDPC codes for error control, he did not provide a specific method for constructing good LDPC codes. However, he proposed a method for constructing a class of pseudorandom LDPC codes.

3.2 Matrix representation

A linear block code C of length n is uniquely specified by either a generator matrix $\mathbb{G}$ or a parity-check matrix $\mathbb{H}$. Suppose that the parity-check matrix $\mathbb{H}$ has m rows and n columns. For the $m \times n$ parity-check matrix $\mathbb{H}$, the code C is simply the null space of $\mathbb{H}$. An n-tuple $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$ over $GF(2)$ is a codeword if and only if $\mathbf{c}\mathbb{H}^T = \mathbf{0}$. This means that the bits of a codeword must satisfy a set of parity-check equations specified by the rows of $\mathbb{H}$. If the parity-check matrix $\mathbb{H}$ has low density of 1s, $\mathbb{H}$ is said to be a $low - density\ parity - check$ matrix and the code specified by $\mathbb{H}$ is hence called an LDPC code. The parity-check matrix $\mathbb{H}$ of a *regular* LDPC code has column weight g and row weight r, where $r = g(n/m)$ and $g \ll m$. If $\mathbb{H}$ has low density, but its row and column weight are not both constant, then the code is an *irregular* LDPC code. In general, low-density means that there are 10% or fewer 1s in the parity-check matrix.

3.3 Graphical representation

A *Tanner Graph* is a bipartite graph which includes two types of nodes and edges connecting only different types [6]. The two types of nodes are the *variable nodes*, which are denoted by VNs and the *check nodes*, which are denoted by CNs. For a $m \times n$ parity-check matrix, there are m CNs in its Tanner graph, one of each equation, and n VNs, one for each code bit. The Tanner graph of a code is drawn simply connecting CN i to VN j if and only if $H_{i,j} = 1$.

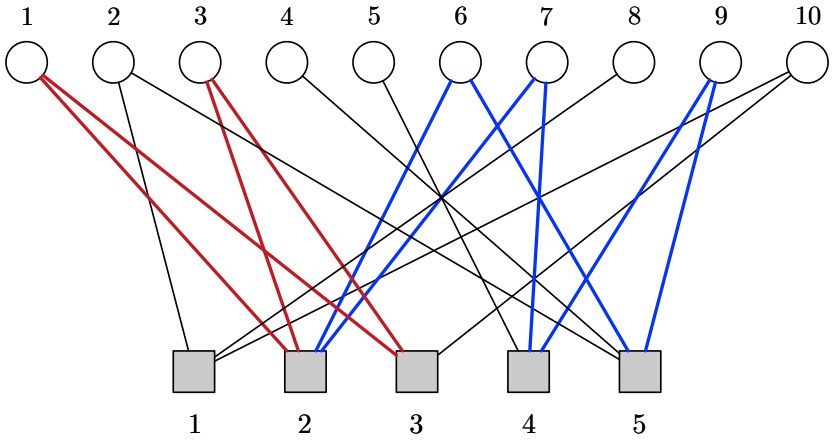

FIGURE 3.1: Tanner graph of H given in (3.1).

Consider the parity-check matrix

$$H = \left[\begin{array}{cccccccccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{array}\right]. \tag{3.1}$$

The Tanner graph corresponding to $\mathbb{H}$ is depicted in Figure 3.1.

Observe that VNs 2, 8, and 10 are connected to CN 1 because, in the first row of the parity check matrix, $H_{1,2} = H_{1,8} = H_{1,10} = 1$. The same applies to the check nodes connecting to variable nodes. For example, CNs 2 and 3 are connected to VN 1 because, in the first column of the parity-check matrix, we have $H_{2,1} = H_{3,1} = 1$. Moreover, if $\mathbf{v}H^T = \mathbf{0}$, which means that the vector $\mathbf{v}$ is a valid codeword, we have that all the bits of $\mathbf{v}$ that are connected to the same check node must sum to zero (mod 2). In Figure 3.1, we can see that there are four red and six blue edges. The red edges complete a length-4 cycle and the blue edges a length-6 cycle. In general, a *cycle* in a Tanner graph is a sequence of connected nodes which start and end at the same node and contains other nodes no more than once. The length of a cycle is the number of edges it contains and the *girth* of a graph is the size of its smallest size. We will see later that small cycles degrade the code performance. So we want a small number of cycles and the girth of a code to be as large as possible.

As we see in Figure 3.1, every check and variable node has not constant number of edges. This means that the code is irregular and for convenience we introduce some

degree distribution polynomials. Assume that an LDPC code has length n and that the number of variable nodes of degree i is Λ_i, so that $\sum_i \Lambda_i = n$. Moreover, denote the number of check nodes of degree i by P_i, so that $\sum_i P_i = n\overline{r}$, where r is the design rate of the code and $\overline{r}$ is a shorthand for $1 - r$. The number of edges from check nodes to variable nodes is equal to the number of edges from the variable nodes to check nodes because they match up. So, we have $\sum_i i\Lambda_i = \sum_i iP_i$. Furthermore, we introduce the following notation

$$\Lambda(x) = \sum_{i=1}^{l_{max}} \Lambda_i x^i, \qquad P(x) = \sum_{i=1}^{r_{max}} P_i x^i. \tag{3.2}$$

$\Lambda(x)$ and $P(x)$ are polynomials whose coefficients are equal to the number of nodes of various degrees. So, we call $\Lambda(x)$ and $P(x)$ the variable and check degree distributions from a *node perspective*. From these definitions, we can see the following relationships

$$\Lambda(1) = n, \qquad P(1) = n\overline{r}, \qquad r(\Lambda, P) = 1 - \frac{P(1)}{\Lambda(1)}, \qquad \Lambda'(1) = P'(1).$$

Sometimes, we normalize distributions Λ and P as:

$$L(x) = \frac{\Lambda(x)}{\Lambda(1)}, \qquad R(x) = \frac{P(x)}{P(1)}.$$

Let us see an example. We have

$$\Lambda(x) = 2x^3 + 3x^2 + 5x, \qquad P(x) = 3x^4 + x^5. \tag{3.3}$$

We can find the length of the code, $n = \Lambda(1) = 10$, and the rate of the code, $r(\Lambda, P) = 1 - \frac{4}{10} = 0.6$. Moreover, we can find the total number of the edges $\Lambda'(1) = P'(1) = 17$, and the normalized degree distributions from node perspective, $L(x) = \frac{1}{5}x^3 + \frac{3}{10}x^2 + \frac{1}{2}x$ and $R(x) = \frac{3}{4}x^4 + \frac{1}{4}x^5$. From any particular $\big(\Lambda(x), P(x)\big)$ pair, we can construct many different codes. From our $\big(\Lambda(x), P(x)\big)$ pair (3.7) , one possible parity check matrix and its respective Tanner is depicted in Figure 3.2.

In addition to the $\Lambda(x)$ and $P(x)$, which are degree distributions from node perspective, we have $\lambda(x)$ and $\rho(x)$, which are variable and check node degree distributions from an

$$\mathrm{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

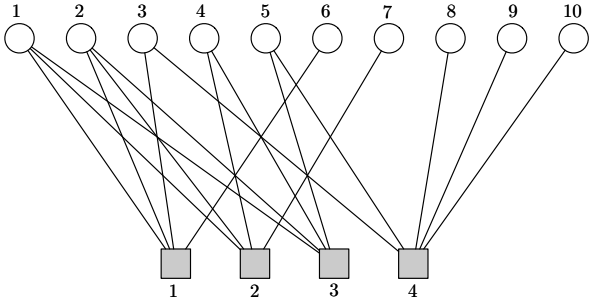

FIGURE 3.2: A parity check matrix from (3.7) $\Lambda(x), P(x)$ pair and its particular Tanner graph.

edge perspective

$$\lambda(x) = \sum_i \lambda_i x^{i-1} = \frac{\Lambda'(x)}{\Lambda'(1)} = \frac{L'(x)}{L'(1)}, \qquad \rho(x) = \sum_i \rho_i x^{i-1} = \frac{P'(x)}{P'(1)} = \frac{R'(x)}{R'(1)}. \quad (3.4)$$

We can see that λ_i is equal to the *fraction of edges* that connect to variable nodes of degree i and ρ_i is equal to the *fraction of edges* that connect to check nodes of degree i. The inverse relationships are

$$\frac{\Lambda(x)}{n} = L(x) = \frac{\int_0^x \lambda(z)\mathrm{d}z}{\int_0^1 \lambda(z)\mathrm{d}z}, \qquad \frac{P(x)}{n\overline{r}} = R(x) = \frac{\int_0^x \rho(z)\mathrm{d}z}{\int_0^1 \rho(z)\mathrm{d}z} \quad (3.5)$$

and the *design rate* is given by

$$r(\lambda, \rho) = 1 - \frac{L'(1)}{R'(1)} = 1 - \frac{\int_0^1 \rho(x)\mathrm{d}x}{\int_0^1 \lambda(x)\mathrm{d}x}. \quad (3.6)$$

We can convert now the (Λ, P) pair from (3.7) and find the particular degree distributions from edge perspective, which are

$$\lambda(x) = \frac{6}{17}x^2 + \frac{6}{17}x + \frac{5}{17}, \qquad \rho(x) = \frac{5}{17}x^4 + \frac{12}{17}x^3. \quad (3.7)$$

3.4 Introduction to encoding

We have a code C which consists of all length-8 vectors

$$\mathbf{c} = [c_1\ c_2\ c_3\ c_4\ c_5\ c_6\ c_7\ c_8]$$

that satisfy the four parity-check equations

$$\begin{aligned} c_1 \oplus c_5 \oplus c_7 \oplus c_8 &= 0, \\ c_3 \oplus c_5 \oplus c_6 &= 0, \\ c_2 \oplus c_6 \oplus c_7 &= 0, \\ c_4 \oplus c_6 \oplus c_8 &= 0. \end{aligned}$$

Checking the vector $\hat{\mathbf{c}} = [0\ 1\ 1\ 0\ 1\ 1\ 0\ 1]$, we see that

$$\begin{aligned} 0 \oplus 1 \oplus 0 \oplus 1 &= 0, \\ 1 \oplus 1 \oplus 1 &= 1, \\ 1 \oplus 1 \oplus 0 &= 0, \\ 0 \oplus 1 \oplus 1 &= 0, \end{aligned}$$

so $\hat{\mathbf{c}}$ is not valid codeword for this code. The codeword constraints can be re-written as

$$\begin{aligned} c_1 &= c_5 \oplus c_7 \oplus c_8, \\ c_2 &= c_6 \oplus c_7, \\ c_3 &= c_5 \oplus c_6, \\ c_4 &= c_6 \oplus c_8. \end{aligned}$$

The bits c_1, c_2, c_3, c_4 are the parity bits and c_5, c_6, c_7, c_8 are the message bits. The message bits are conventionally labeled by $\mathbf{u} = [u_1, u_2, \ldots u_k]$, where the vector $\mathbf{u}$ holds the k message bits. Under these constraints, we can produce the codeword from the message word. For the message $\mathbf{u} = [1\ 0\ 0\ 1]$, because $c_1 = 1 \oplus 0 \oplus 1 = 0, c_2 = 0 \oplus 0 = 0, c_3 = 1 \oplus 0 = 1$ and $c_4 = 0 \oplus 1 = 1$, the codeword is $c = [0\ 0\ 1\ 1\ 1\ 0\ 0\ 1]$. These

constraints can be written in matrix form as follows

$$[c_1\ c_2\ c_3\ c_4\ c_5\ c_6\ c_7\ c_8] = [u_1\ u_2\ u_3\ u_4] \underbrace{\begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}}_{G},$$

where the matrix G is called the *generation matrix* of the code. The (i, j)th entry of G is 1 if the ith message bit plays a role in determining the jth codeword bit. Thus, the codeword $\mathbf{c}$ corresponding to the binary message $\mathbf{u} = [u_1\ u_2\ u_3\ u_4]$ can be found using the matrix equation

$$\mathbf{c} = \mathbf{u}G.$$

For a binary code with k message bits and length n codewords, the generator matrix G is a $k \times n$ binary matrix. The ratio k/n is called the *rate* of the code.

A code with k message bits contains 2^k codewords. These codewords are a subset of the total possible 2^n binary vectors of length n. When the first or last k codeword bits contain the message bits, the code is called *systematic*. The generator matrix for these codes contains the k x k identity matrix I_k, as its first or last k columns.

Generally, if we have the parity-check matrix H, we can find the generator matrix G, by performing Gauss-Jordan elimination on H to derive

$$H = [I_{n-k}\ \ A],$$

where A is an $(n-k) \times k$ binary matrix and I_{n-k} is the identity matrix of order $n-k$. Then, the generator matrix is

$$G = [A^T\ \ I_k].$$

The row space of G is orthogonal to H. Thus, if G is the generator matrix for a code with parity-check matrix H, then

$$GH^T = 0.$$

Let us see an example. Suppose that we have the LDPC code with $rate = 1/2$ and $length = 10$, which is described by the parity-check matrix

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

We want to find the corresponding generator matrix. So we must perform Gauss-Jordan elimination to make one part of H, *square* and *invertible*.

The **first step** is to put H into $row - echelon$ form. In this form, the leading coefficient of a non-zero row is always strictly to the right of the leading coefficient of the row above it. Because of the need of having the same codeword set as the original, we can apply only *elementary row operations* in $GF(2)$, which are row interchanging or one row adding to another modulo 2. We interchange the first and the second row. Subsequently, for third row, we add the first and the third row and for the fourth row we add the second and the fourth row. Then, replacing the fifth row with the sum of all rows, we have the matrix H in row-echelon form

$$H_r = \begin{bmatrix} \mathbf{1} & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & \mathbf{1} & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & \mathbf{1} & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & \mathbf{1} & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 1 & 0 & 1 & 1 & 1 \end{bmatrix}.$$

The **second step** is to put parity-check matrix H into $reduced\ row - echelon$ form. In this form, every leading coefficient is 1 and is the only nonzero entry in its column. The first and the second columns are already correct. The entry in the third column above the diagonal is removed by replacing the second row, with the sum of the second and third row. To clear the fourth column, we add the forth row to the first, second and third row and to clear the fifth column, we add the fifth row to the first, second and

forth row. Now, the matrix is in $reduced\ row - echelon$ form.

$$H_{rr} = \begin{bmatrix} \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & \mathbf{1} & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & \mathbf{1} & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & \mathbf{1} & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 1 & 0 & 1 & 1 & 1 \end{bmatrix}.$$

We observe that the first part of the H_{rr} matrix is the identity matrix of order $n - k$. Because $GH^T = 0$, the generator matrix for the code is

$$G = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

The drawback of this approach is the complexity. The matrix G will most likely not be sparse and the matrix multiplication

$$\mathbf{c} = \mathbf{u}G$$

at the encoder, will have complexity in the order of n^2 operations. Because LDPC codes have large n, the multiplication will become very complex.

3.5 Efficient encoders based on approximate upper triangulations

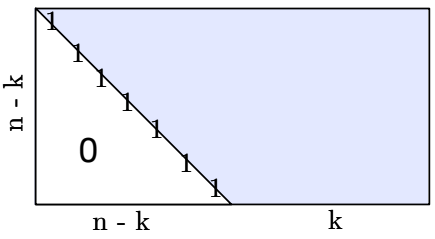

FIGURE 3.3: H in upper triangular form.

A better approach to encode is to avoid constructing G and use only parity-check matrix H. The **first step** is as before to put H into row-echelon form. Now the matrix is $H = (H_p \ H_s)$, where H_p is square in upper triangular form and has dimensions $(n-k) \times (n-k)$, as we see in Figure 3.3. The code consists of the set of n-tuples $\mathbf{x}$ such that $H\mathbf{x}^T = \mathbf{0}^T$, where $x = (p, s)$.

The **second step** is to use back substitution to find the parity-check bits. More precisely, for $l \in [n-k]$ calculate

$$p_l = -\sum_{j=l+1}^{n-k} H_{l,j}p_j - \sum_{j=1}^{k} H_{l,j+n-k}s_j.$$

According to Richardson and Urbanke in [11], there is a better way than this and we can perform encoding with linear complexity. **Firstly**, we put the parity-check matrix into *approximate upper triangular* form, using only row and column permutations, as in Figure 3.4.

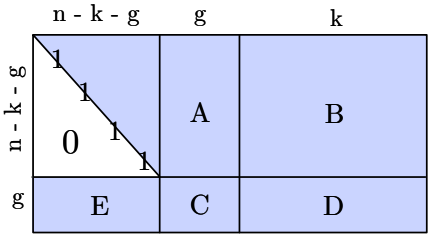

FIGURE 3.4: H in approximate upper triangular form.

Let us name T the first part of the matrix. This is upper triangular with ones along the diagonal and dimensions $(n-k-g) \times (n-k-g)$. The g rows of H, as we see in the figure, are called *gap* and we want the gap to be as small as possible to reduce the encoding complexity of the code. **Secondly**, we want to eliminate E. So, we multiply H_t from the left by $\begin{bmatrix} I & 0 \\ -ET^{-1} & I \end{bmatrix}$ to obtain

$$H' = \begin{bmatrix} I & 0 \\ -ET^{-1} & I \end{bmatrix} H_t = \begin{bmatrix} T & A & B \\ 0 & C - ET^{-1}A & D - ET^{-1}B \end{bmatrix}.$$

Finally, to perform encoding using H', the codeword $\mathbf{c} = [c_1 \ c_2 \dots c_N]$ is divided into three parts, so that $\mathbf{c} = [\mathbf{p_1} \ \mathbf{p_2} \ \mathbf{s}]$, where $\mathbf{s} = [s_1 \ s_2 \dots s_k]$ is the k-bit message, $\mathbf{p_1}$ holds the first $(n-k-g)$ parity bits and $\mathbf{p_2}$ the remaining g parity bits. The codeword $\mathbf{c} = [\mathbf{p_1} \ \mathbf{p_2} \ \mathbf{s}]$ must satisfy the parity-check equation $\mathbf{c}H^T = \mathbf{0}$. So, we have the system

of equations

$$T\mathbf{p_1}^T + A\mathbf{p_2}^T + B\mathbf{s}^T = \mathbf{0}^T$$
$$(C - ET^{-1}A)\mathbf{p_2}^T + (D - ET^{-1}B)\mathbf{s}^T = \mathbf{0}^T.$$

We define $\phi = C - ET^{-1}A$. If ϕ is invertible, then the solution of the above system gives $\mathbf{p_2}^T = -\phi^{-1}(D - ET^{-1}B)\mathbf{s}^T$ and $\mathbf{p_1}^T = -T^{-1}(A\mathbf{p_2}^T + B\mathbf{s}^T)$.

Let us see an example with the same parity-check matrix as the above example. We have the parity matrix

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

In the **first step**, we swap the first with the second row, the fourth with the third row and the second with the fifth column. Now, the parity check matrix gap is 2 and the matrix is

$$H_t = \left[\begin{array}{ccc|cc|ccccc} \mathbf{1} & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & \mathbf{1} & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & \mathbf{1} & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ \hline 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \end{array}\right].$$

In the **second** step, we want to set E to zero. We have

$$\begin{bmatrix} I & 0 \\ -ET^{-1} & I \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix},$$

and we perform the multiplication with H_t from the left, to give

$$H' = \left[\begin{array}{ccc|cc|ccccc} \mathbf{1} & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & \mathbf{1} & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & \mathbf{1} & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \end{array}\right].$$

We see that $\phi = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ is invertible. We want to send the message $\mathbf{s} = [1\ 0\ 1\ 1\ 0]$. Thus,

$$\mathbf{p_2}^T = -\phi^{-1}(D - ET^{-1}B)\mathbf{s}^T = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

$$\mathbf{p_1}^T = -T^{-1}(A\mathbf{p}_2^T + B\mathbf{s}^T) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \left(\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \right) = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

So, the codeword is $\mathbf{c} = [\mathbf{p_1}\ \mathbf{p_2}\ \mathbf{s}] = [1\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 0]$. This approach has almost linear encoding complexity.

Chapter 4

Construction of LDPC codes

4.1 Introduction

There are many different methods to construct LDPC codes. In this chapter, we shall preset Gallager [5], random [11, p. 78], and Progressive edge growth algorithm [12] construction.

4.2 Gallager codes

Gallager's original definition of a regular LDPC code was that it is a linear code whose $m \times n$ parity-check matrix H has $g \ll m$ ones in each column and $r \ll n$ ones in each row. The matrix H has the form

$$H = \begin{bmatrix} H_1 \\ H_2 \\ \vdots \\ H_g \end{bmatrix},$$

where the submatrices $H_1, H_2, \ldots, H_g$ have the following structure. For any integers μ and r greater than 1, each submatrix H is $\mu \times \mu r$ with row weight μ and column weight 1. The submatrix H_1 has the following specific form: for $i = 0, 1, \ldots, \mu - 1$, the ith row

contains all of its r 1s in columns ir to $(i+1)r-1$. The other submatrices are obtained by column permutations of H_1. This method has easy construction and it is shown that it has excellent distance properties, when $g \geq 3$. Contrary to these advantages, the absence of length-4 cycles in H is not guaranteed.

The following matrix is the first example given by Gallager

$$H = \left[\begin{array}{cccccccccccccccccccc}
1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1\\
\hline
1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&0&0&0&0\\
0&1&0&0&0&1&0&0&0&1&0&0&0&0&0&0&1&0&0&0\\
0&0&1&0&0&0&1&0&0&0&0&0&0&1&0&0&0&1&0&0\\
0&0&0&1&0&0&0&0&0&0&1&0&0&0&1&0&0&0&1&0\\
0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1\\
\hline
1&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0\\
0&1&0&0&0&0&1&0&0&0&1&0&0&0&0&1&0&0&0&0\\
0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&1&0\\
0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&1&0&0&0\\
0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&0&0&1
\end{array}\right].$$

It is a 15×20 matrix with $g = 3$, $r = 4$ and $\mu = 5$.

Example 4.2.1. In this example, we consider the performance of sum-product decoding over Gallager $(3, 6)$ and $(5, 10)$ regular codes on a BI-AWGN channel. The construction length of the matrices is 10^4 and 10^5, respectively.

We can see in Figure 4.1 that the Gallager $(3, 6)$ codes perform better than the Gallager $(5, 10)$ codes.

4.3 Random construction

Given a degree distribution pair (Λ, P), define an *ensemble* of bipartite graphs LDPC(Λ, P) in the following way. Each graph in LDPC(Λ, P) has $\Lambda(1)$ variable nodes and $P(1)$ check

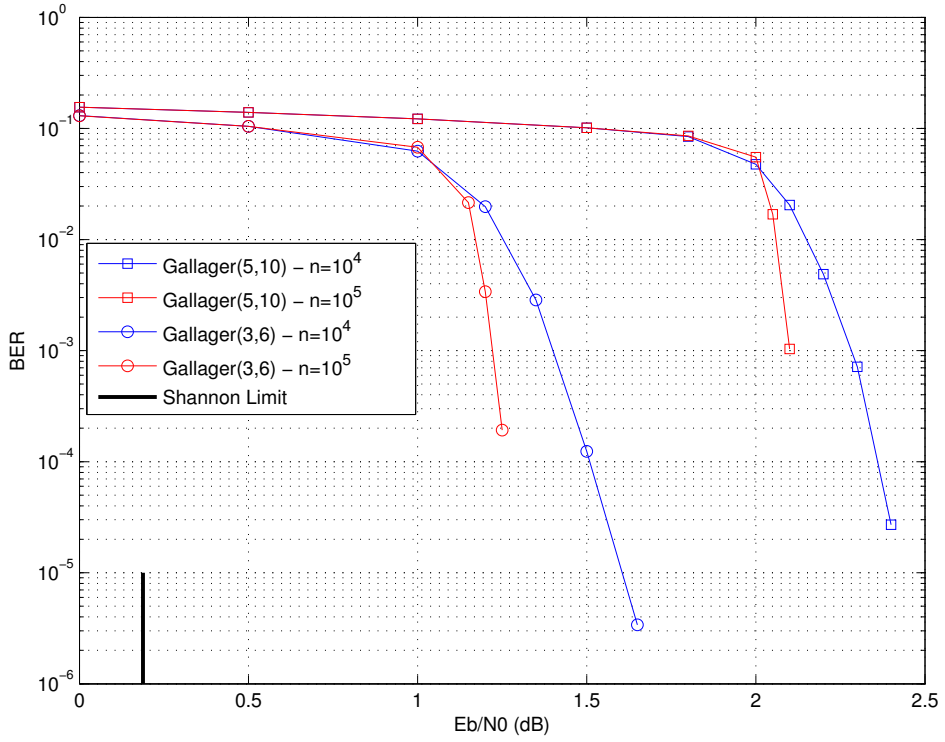

FIGURE 4.1: Decoding of Gallager (3,6) and (5,10) codes in the BI-AWGN channel.

nodes: Λ_i variable nodes and P_i check nodes have degree i. A node of degree i has i *sockets* from which the i edges emanate, so that in total there are $\Lambda'(1) = P'(1)$ sockets on each side. Label the sockets on each side with the elements of set $[\Lambda'(1)] = \{1, \ldots, \Lambda'(1)\}$ in some arbitrary but fixed way. Let σ be a permutation on $[\Lambda'(1)]$. Associate to σ a bipartite graph by connecting the i-th socket on the variable side to the $\sigma(i)$-th socket on the check side. Letting σ run over the set of permutations on $[\Lambda'(1)]$ generates a set of bipartite graphs. Finally, we define a probability distribution over the set of graphs by placing the uniform probability distribution on the set of permutations. This is the ensemble of bipartite graphs LDPC(Λ, P).

Example 4.3.1. In this example, we have the comparison between the Gallager and the regular-random codes of 10^4 and 10^5 length for the BI-AWGN.

We can see in Figure 4.2 that the random codes have slightly better performance.

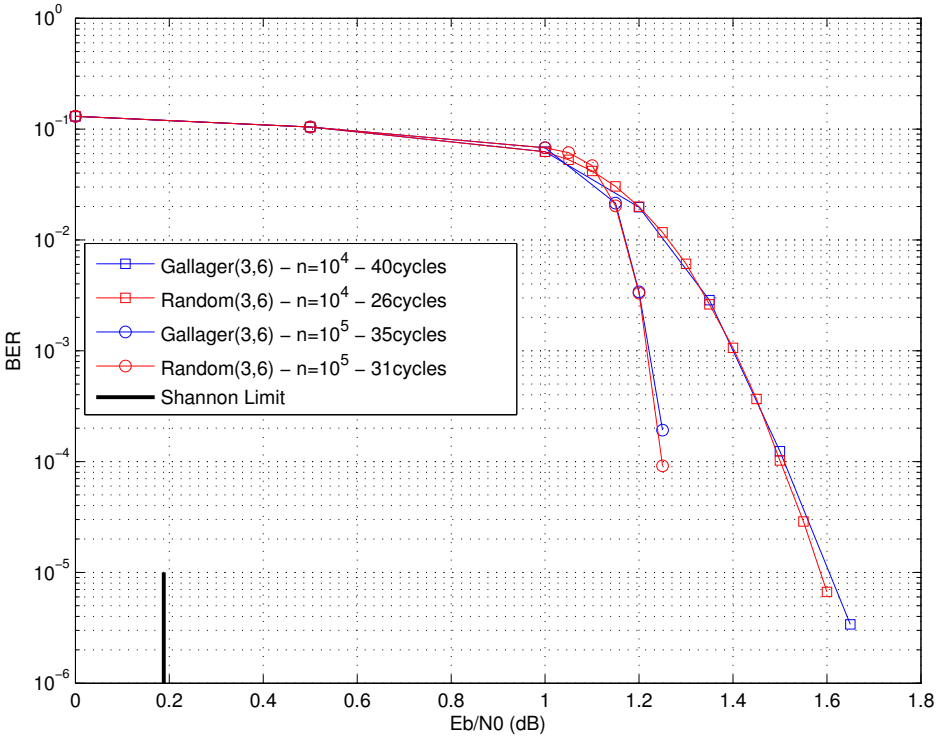

FIGURE 4.2: Decoding of random regular and Gallager ldpc codes in the BI-AWGN channel.

4.4 Progressive edge growth algorithm

Progressive edge growth is an algorithm for constructing Tanner graphs with large girth by progressively establishing edges or connections, between variable and check nodes in an edge-by-edge manner. Given the number of check nodes m, the number of variable nodes n and the variable node degree sequence, we place edges to keep the girth as large as possible.

A Tanner graph is denoted as (V, E) with V the set of nodes, i.e. $V = V_c \cup V_s$, where $V_c = \{c_0, c_1, \ldots, c_{m-1}\}$ is the set of check nodes and $V_s = \{s_0, s_1, \ldots, s_{n-1}\}$ the set of variable nodes. E is the set of edges such that $E = V_c \times V_s$ with edge $(c_i, s_j) \in E$ if and only if $h_{i,j} \neq 0$, $h_{i,j} \in H$, $0 \leq i \leq m-1, 0 \leq j \leq n-1$. Denote the variable node degree distribution by $D_s = d_{s_0}, d_{s_1}, \ldots, d_{s_{n-1}}$, in which d_{s_j} is the degree of symbol node $s_j, 0 \leq j \leq n-1$ in the non decreasing order, i.e. $d_{s_0} \leq d_{s_1} \cdots \leq d_{s_{n-1}}$, and the check node degree distribution by $D_c = d_{c_0}, d_{c_1}, \ldots, d_{c_{m-1}}$, in which d_{c_j} is the degree of check node $c_j, 0 \leq j \leq m-1$ and $d_{c_0} \leq d_{c_1} \cdots \leq d_{c_{m-1}}$. Let also the set of edges

E be partitioned in terms of V_s as $E = E_{s_0} \cup E_{s_1} \cup \cdots \cup E_{s_{n-1}}$, with E_{s_j} containing all edges incident on check node s_j. Finally, denote the kth edge incident on s_j by $E^k_{s_j}, 0 \leq k \leq d_{s_j} - 1$.

For a given symbol node s_j, define its neighbor within depth l, $N^l_{s_j}$, as the set consisting of all check nodes reached by a tree spreading from symbol node s_j within depth l, such as every route from a check node to another check node counts as 1. Its complementary set, $\bar{N}^l_{s_j}$, is defined as $V_c \setminus N^l_{s_j}$, or equivalently $\bar{N}^l_{s_j} \cup N^l_{s_j} = V_c$.

Algorithm 1 Progressive Edge-Growth

for $j = 0$ **to** $n - 1$ **do**
begin
  for $k = 0$ **to** $d_{s_j} - 1$ **do**
  begin
    if $k = 0$ **then**
      $E^0_{s_j} \longleftarrow \text{edge}(c_i, s_j)$, where $E^0_{s_j}$ is the first edge incident to s_j, and c_i is a check node having the lowest check degree under the current graph setting $E_{s_0} \cup E_{s_1} \cup \cdots \cup E_{s_{j-1}}$.
    else
      expanding a tree from symbol node s_j up to depth l under the current graph setting such that $\bar{N}^l_{s_j} \neq \emptyset$ but $\bar{N}^{l+1}_{s_j} = \emptyset$, or the cardinality of $\bar{N}^l_{s_j}$ stops increasing but is less than m, then $E^k_{s_j} \longleftarrow \text{edge}(c_i, s_j)$, where $E^k_{s_j}$ is the k-th edge incident to s_j and c_j is one check node picked from the set $\bar{N}^l_{s_j}$ having the lowest check-node degree.
    end if
  end for
end for

Because the low-degree variable nodes are the most susceptible to error (they receive the least amount of neighborly help), edge placement begins with the lowest-degree of them and progresses to variable nodes of increasing degree. The algorithm does not move to the next variable node until all of the edges of the current variable node have been attached. The first edge attached to a VN is connected to a lowest-degree check node under the current state of the graph. Subsequent attachments of edges to the VN are done in such a way that the (local) girth for that VN is maximum. Thus, if the

current state of the graph is such that one or more check nodes cannot be reached from the current variable node by traversing the edges connected so far, then the edge should be connected to an unreachable check node so that no cycle is created. Otherwise, if all check nodes are reachable from the current variable node along some number of edges, the new edge should be connected to a check node of lowest degree that results in the largest girth seen by the current variable node. This lowest-degree check node strategy will yield a fairly uniform check node degree distribution.

Example 4.4.1. Let us present an example of symbol node degree $D_s = \{1, 1, 2, 2, 2, 3\}$ irregular Tanner graph. This code has 6 variable nodes and 4 check nodes. We start by assigning degrees to the variable nodes of the empty graph (I) and we proceed according to the progressive edge growth algorithm.

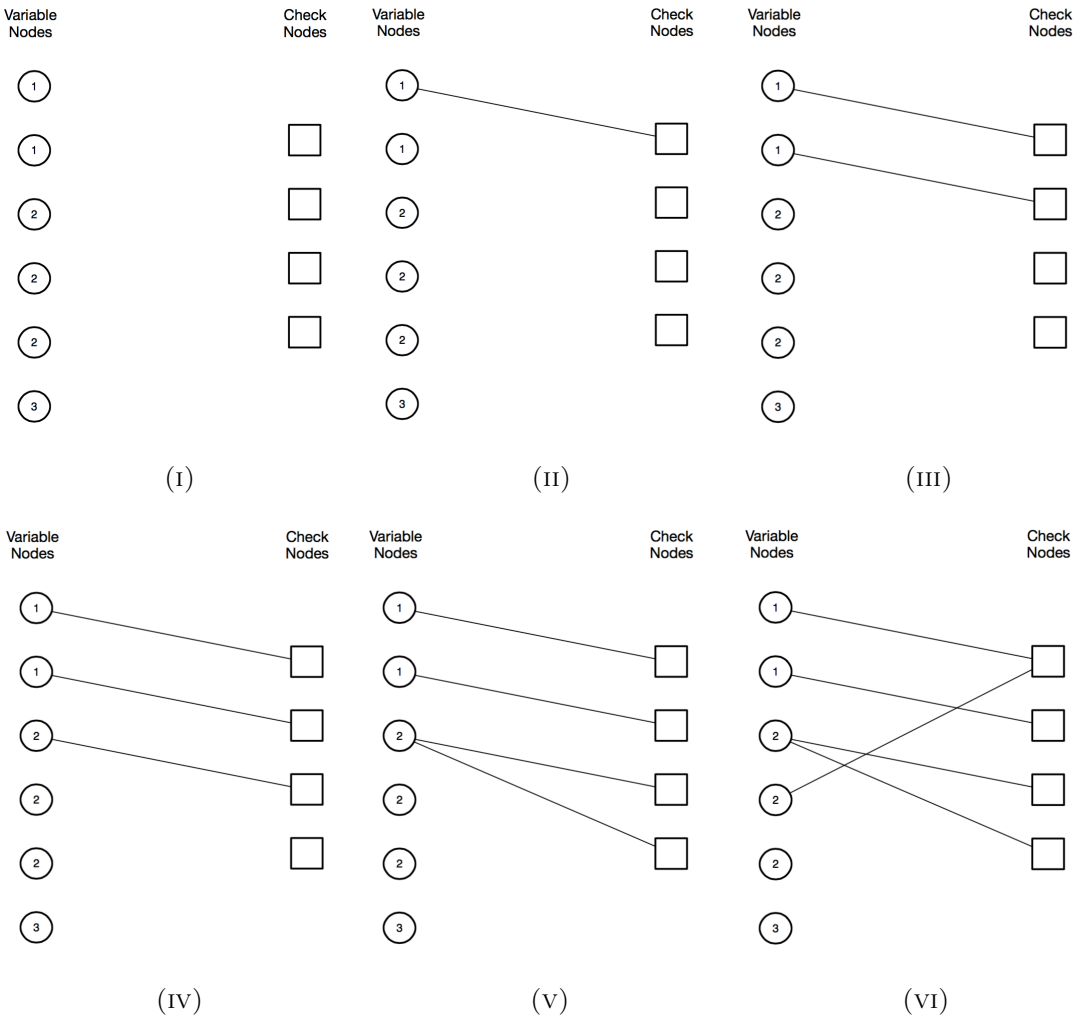

FIGURE 4.3: The beginning steps of progressive edge growth algorithm.

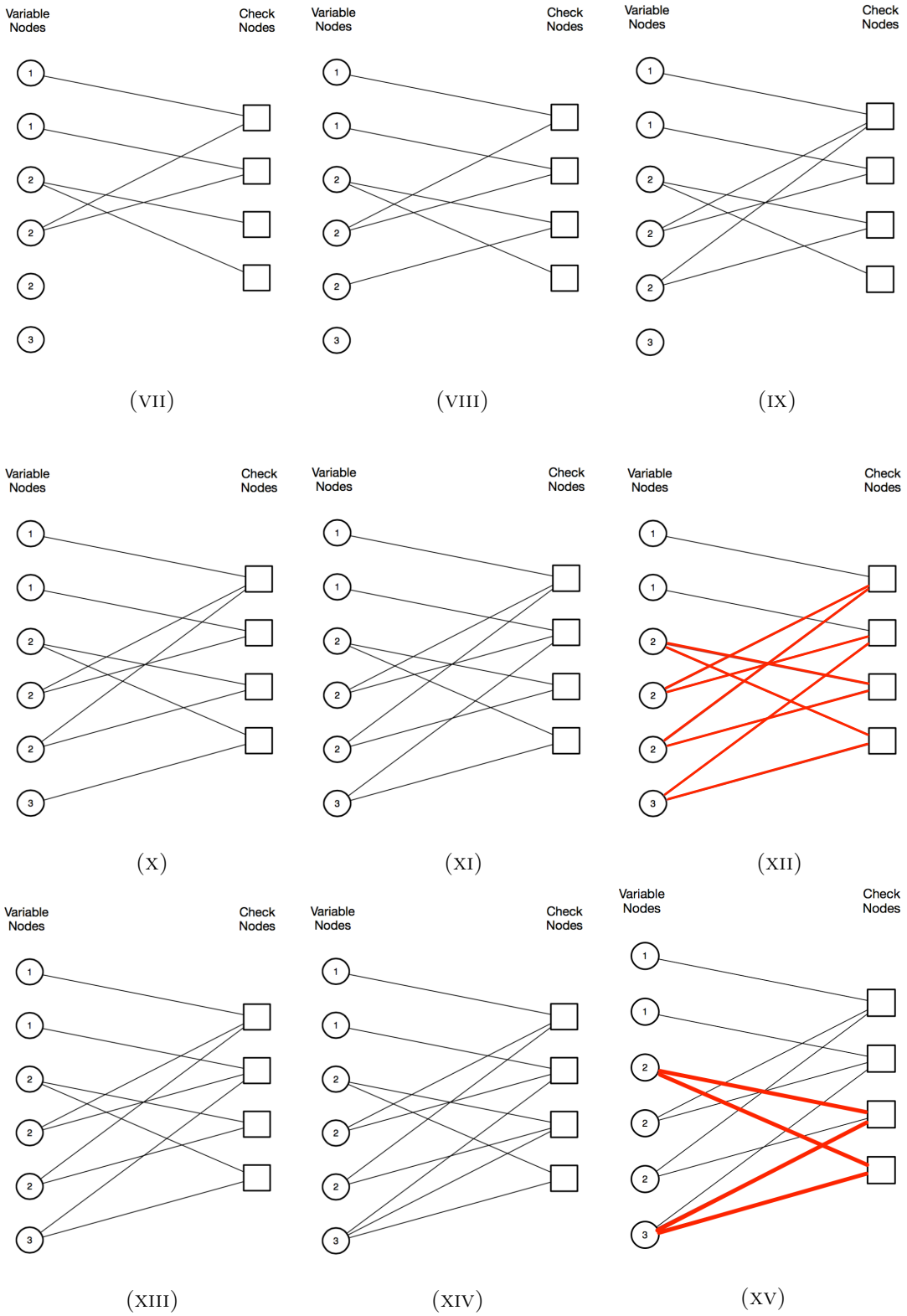

FIGURE 4.3: The last steps of progressive edge growth algorithm with the creation of cycles with lengths 4 and 8.

We see the progress of constructing a PEG parity-check matrix. The bold red edges

show that a cycle has been created. In the subfigure (XII), the algorithm created an 8-length cycle and in the subfigure (XV) the algorithm created a 4-length cycle. The created code has girth 4.

Example 4.4.2. Here is a decoding example with message passing algorithm for random-regular and peg-regular codes. The length of the codes is 500 and 10^4 bits, respectively. We transmit all-zero codeword through a BI-AWGN channel.

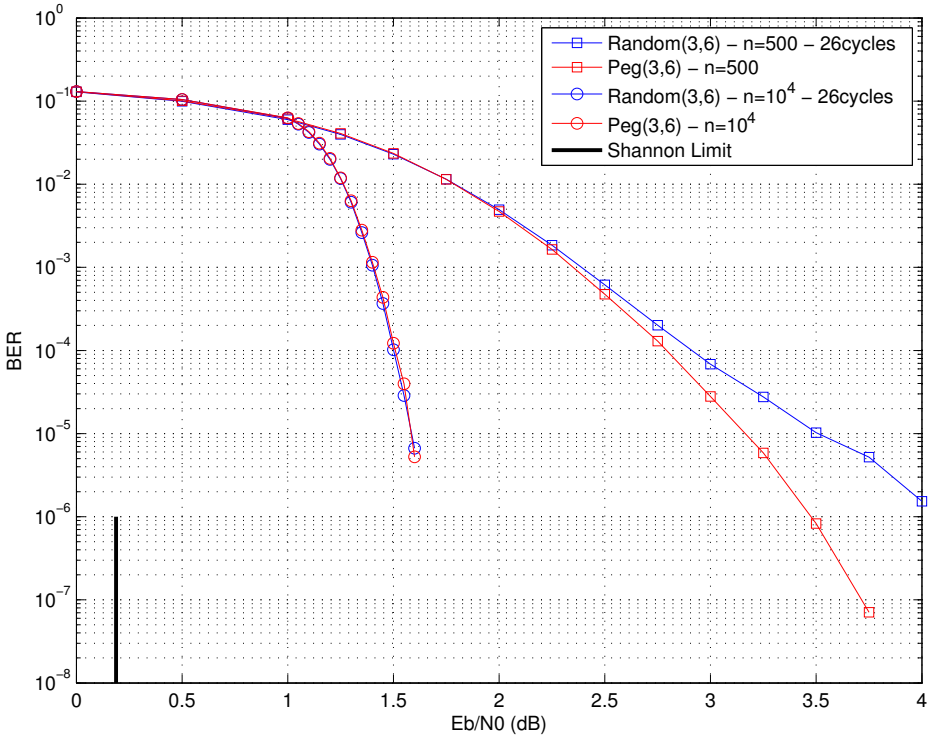

FIGURE 4.4: Decoding of random (3,6) and peg (3,6) codes in the BI-AWGN channel.

We see that random codes created with some 4-length cycles, something that decreases their performance against PEG codes which have no cycles of the same length. Of course, the length plays important role and we see that the codes with larger length are superior to these with smaller length.

Example 4.4.3. Now we construct PEG-irregular code with ensemble

$$L(x) = 0.477081x^2 + 0.280572x^3 + 0.0349963x^4 + 0.0963301x^5 + 0.0090884x^7 + 0.00137443x^{14} + 0.10055777x^{15}.$$

We can easily find the correspondent $\lambda(x)$

$$\lambda(x) = \frac{L'(x)}{L'(1)} = 0.23802x + 0.20997x^2 + 0.349199x^3 + 0.12015x^4 + 0.01587x^6 + 0.0048x^{13} + 0.376268x^{14}.$$

We define the $rate = 1/2$ and the length of the code $n = 5 \times 10^4$ and the algorithms creates it with $girth = 8$. Additionally, we construct the same code with target $girth = 4$. To compare their performance, we create a random-irregular code from the same ensemble, which has 582 cycles of 4-length. We keep this code and we eliminate the 4-length cycles. Here are the histograms of the codes degree distributions

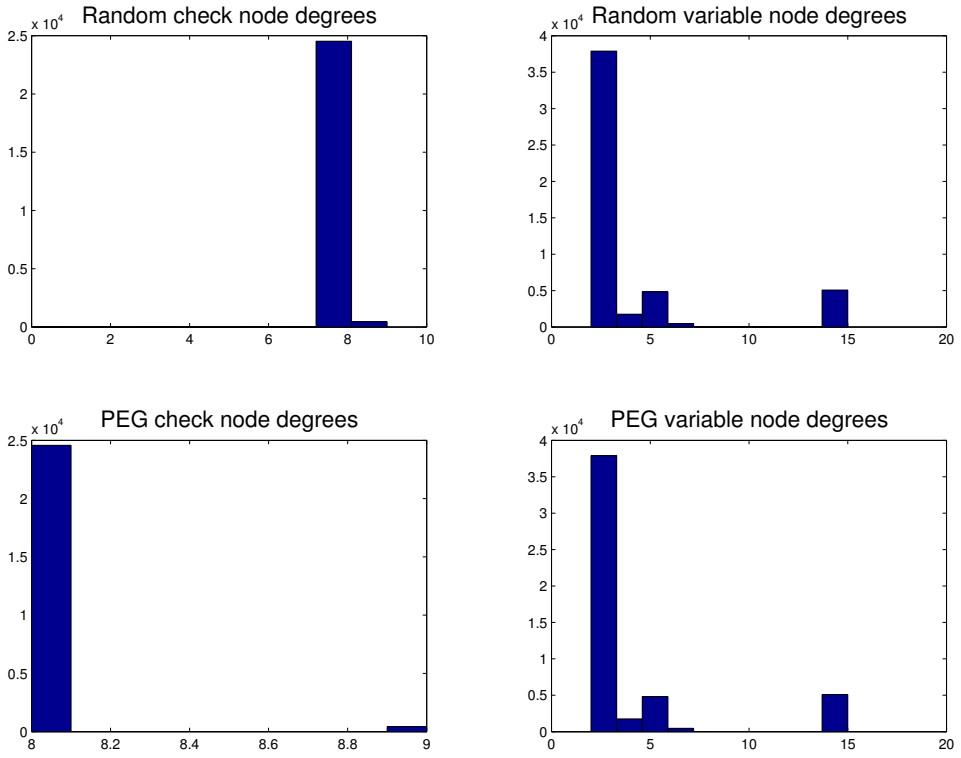

FIGURE 4.5: Degree distribution histograms of a random and a peg code same ensemble.

So, now we are sure that they are from the same ensemble. We transmit the all-zero codeword through the BI-AWGN channel and we decode them with the message-passing algorithm.

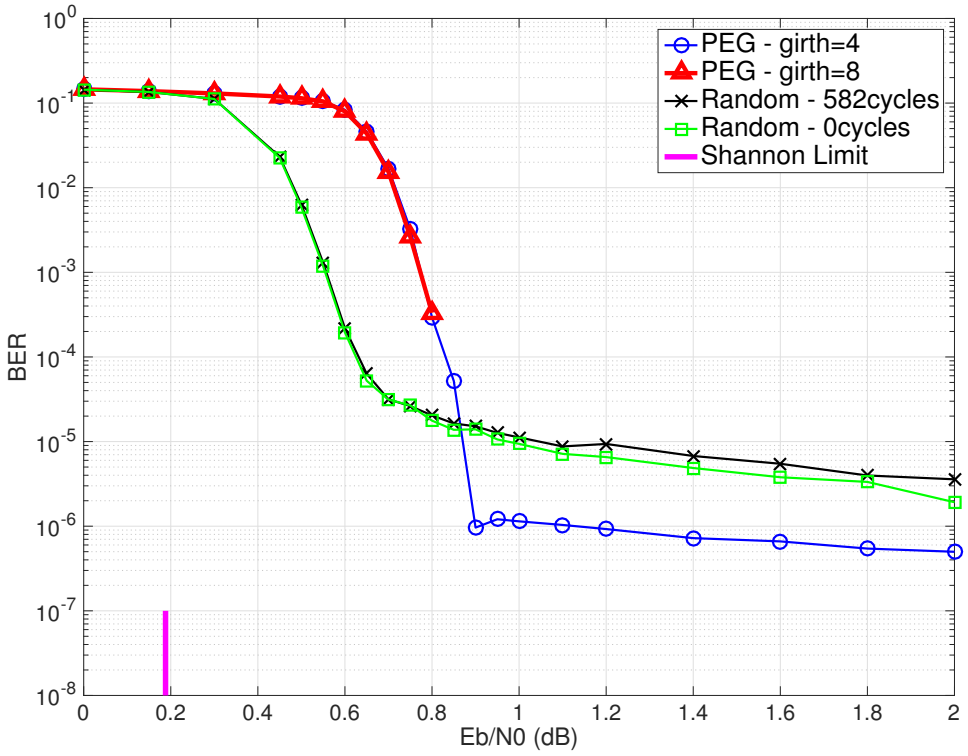

FIGURE 4.6: Decoding of peg and random irregular codes in the BI-AWGN channel.

We see that the waterfall region of the random codes are closer to the Shannon limit and the elimination of 4-length cycles has improved their error floor region. For the PEG codes, we see that they perform better in the error floor region and the PEG with girth equal to 8 has no error floor at all.

Chapter 5

Analysis and Design of LDPC Codes

5.1 Introduction

In this chapter, we examine the iterative decoding performance of LDPC code ensembles in the low-SNR region. We show that the iterative decoding of long LDPC codes displays a threshold effect such that communication is reliable below this threshold and unreliable above it. The threshold is a function of the code ensemble properties and the tools introduced in this chapter allow the designer to predict the decoding threshold and its gap from Shannon's limit. The ensemble properties for LDPC codes are the degree distributions which are the design targets for the code design. Our focus is on the binary AWGN channel.

The analysis below is based on the "local tree assumption," which means that the girth of the graph is large enough so that the subgraph forms a tree (there are no repeated nodes in the subgraph). So, we can analyze the decoding algorithm straightforwardly because incoming messages to every node are independent. Moreover, Richardson and Urbanke have shown in their paper [13] that, for almost all randomly constructed codes and for almost all inputs, the decoder performance will be close to the decoder performance under the local tree assumption with high probability, if the block length of the code is long enough.

We assume that the all-zeros codeword $\mathbf{c} = [0\,0\,\ldots\,0]$ is sent. Under the BPSK mapping, this means that the all-ones word $\mathbf{x} = [+1 + 1 \cdots + 1]$ is transmitted over the channel. An error will be made at the decoder after the maximum number of iterations if any of the signs of the variable nodes LLRs, L_j^{total}, are negative. Let $p_v^{(l)}$ denote the pdf of a message m_v to be passed from a variable node v to some check node during the lth iteration. So, no decision error will occur if

$$\lim_{l\to\infty} \int_{-\infty}^{0} p_v^{(l)}(\tau)\mathrm{d}\tau = 0.$$

Note that $p_v^{(l)}(\tau)$ depends on the channel parameter σ. Then, the decoding threshold σ^* is given by

$$\sigma^* = \sup\left\{\sigma : \lim_{l\to\infty} \int_{-\infty}^{0} p_v^{(l)}(\tau)\mathrm{d}\tau = 0\right\}.$$

In Figure 5.1, we see the evolution of the BER of a regular (3,6) LDPC code with 10^4-length, as a function of the number of iterations l for various values of σ.

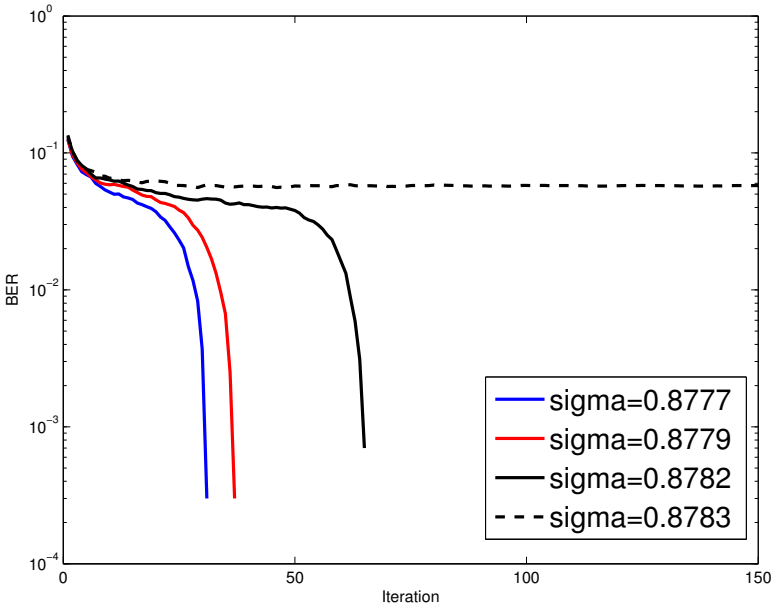

FIGURE 5.1: Evolution of BER of a regular (3,6) LDPC code as a function of the number of iterations l for various values of σ.

We can see that for $\sigma = 0.8777, 0.8779, 0.8782$ the error probability converges to zero, whereas for $\sigma = 0.8783$ the error probability converges to a non-zero value. So, $\sigma^* \approx 0.878221$.

We recall the log-likelihood ratios (LLRs) from previous chapter and we use:

$$v = \log \frac{p_{Y|X}(y|x=1)}{p_{Y|X}(y|x=-1)},$$

as the output message of a variable node, where x is the bit value of the node and y denotes all the information available to the node up to the present iteration obtained from the edges other than the one carrying u. Likewise, we define the output message of a check node as u. Under the message passing algorithm, we recall that v is equal to the sum of all incoming LLRs:

$$v = \sum_{i=0}^{d_v-1} u_i, \tag{5.1}$$

where $u_i, i = 1, \ldots, d_v - 1$, are the incoming LLRs from the neighbors of the variable node except the check node that gets the message v, and u_0 is the observed LLR of the output bit associated with the variable node. The density of the sum of d_v independent random variables $u_i, i = 0, \ldots, d_v - 1$, can be calculated by convolution of densities of the u_i's using Fourier transform in the frequency domain.

Moreover, we recall from the message passing decoding that the processing rule for the check nodes is:

$$u = 2 \tanh^{-1} \left(\prod_{j=1}^{d_c-1} \tanh \left(\frac{v_j}{2} \right) \right),$$

or equivalently

$$\tanh \frac{u}{2} = \prod_{j=1}^{d_c-1} \tanh \frac{v_j}{2}, \tag{5.2}$$

where $v_j, j = 1, \ldots, d_c - 1$, are the incoming LLRs from the $d_c - 1$ neighbors of a check node, and u is the message sent to the remaining neighbor.

In [14], the authors propose modeling the messages of the BP algorithm as Gaussian random variables based on the observations that message distributions are close to the Gaussian distribution. Since a Gaussian is completely specified by its mean and variance, we need to keep only these quantities during the iterations. Additionally, because of the

symmetry condition proposed in [15], a symmetric Gaussian has mean m and variance $\sigma^2 = 2m$, which means that we need to keep only the mean.

5.2 Gaussian approximation for regular LDPC codes

We know that the LLR message u_0 from the channel is Gaussian with mean $2/\sigma_n^2$ and variance $4/\sigma_n^2$. We denote the means of u and v by m_u and m_v respectively. Because of the fact that the code ensemble is regular, all message means are equal during each iteration. So, (5.1) becomes

$$m_v^{(l)} = m_{u_0} + (d_v - 1)m_u^{(l-1)}, \tag{5.3}$$

where l denotes the l iteration and m_{u_0} the mean of the channel message.

Now, we have to calculate the updated mean $m_u^{(l)}$. We are taking means on both sides of (5.2):

$$E\Big[\tanh\frac{u^{(l)}}{2}\Big] = E\Big[\prod_{j=1}^{d_c-1}\tanh\frac{v_j^{(l)}}{2}\Big] \tag{5.4}$$

$$= \Big(E\Big[\tanh\frac{v^{(l)}}{2}\Big]\Big)^{d_c-1} \tag{5.5}$$

where we have omitted the index j because the v_j's are i.i.d. The variables $u^{(l)}$ and $v^{(l)}$ are Gaussian $\mathcal{N}(m_u^{(l)}, 2m_u^{(l)})$ and $\mathcal{N}(m_v^{(l)}, 2m_v^{(l)})$, respectively. So,

$$E\Big[\tanh\frac{u}{2}\Big] = \frac{1}{\sqrt{4\pi m_u}}\int_{\mathbb{R}}\tanh\frac{u}{2}e^{-\frac{(u-m_u)^2}{4m_u}}\mathrm{d}u.$$

We will now define the auxiliary function $\phi(x)$

$$\phi(x) = \begin{cases} 1 - \frac{1}{\sqrt{4\pi x}}\int_{\mathbb{R}}\tanh\frac{u}{2}\exp^{-\frac{(u-x)^2}{4x}}\mathrm{d}u, & \text{if } x > 0 \\ 1, & \text{if } x = 0. \end{cases}.$$

Note that $\phi(x)$ need only be defined for $x \geq 0$ because we assume that only +1s are transmitted and hence all message means are positive. It can be shown that $\phi(x)$ is continuous and decreasing for $x \geq 0$, with $\lim_{x\to 0}\phi(x) = 1$ and $\lim_{x\to\infty}\phi(x) = 0$.

Without much sacrifice in accuracy, the authors of [14] found a good approximation of $\phi(x)$

$$\phi(x) = \begin{cases} e^{-0.4527x^{0}.86+0.0218}, & x < 10, \\ \sqrt{\frac{\pi}{x}} e^{-\frac{x}{4}(1-\frac{20}{7x})}, & x \geq 10. \end{cases}$$

We recall from (5.5) that

$$E\Big[\tanh\frac{u^{(l)}}{2}\Big] = \Big(E\Big[\tanh\frac{v^{(l)}}{2}\Big]\Big)^{d_c-1},$$

and we get

$$1-\phi(m_u^{(l)}) = (1-\phi(m_v^{(l)}))^{d_c-1}.$$

Now, from (5.3), we have

$$1-\phi(m_u^{(l)}) = (1-\phi(m_{u_0}+(d_v-1)m_u^{(l-1)}))^{d_c-1}.$$

Finally, the update rule for $m_u^{(l)}$ becomes

$$m_u^{(l)} = \phi^{-1}\Big(1-[1-\phi(m_{u_0}+(d_v-1)m_u^{(l-1)})]^{d_c-1}\Big), \tag{5.6}$$

where $m_u^{(0)} = 0$ is the initial value for m_u.

Example 5.2.1. Consider the (3,6)-regular code on a binary AWGN channel with 10^4-length and $\sigma = 0.8782 < \sigma^*$. We see the evolution of densities for the messages emitted from variable and check nodes as well as the Gaussian approximation densities.

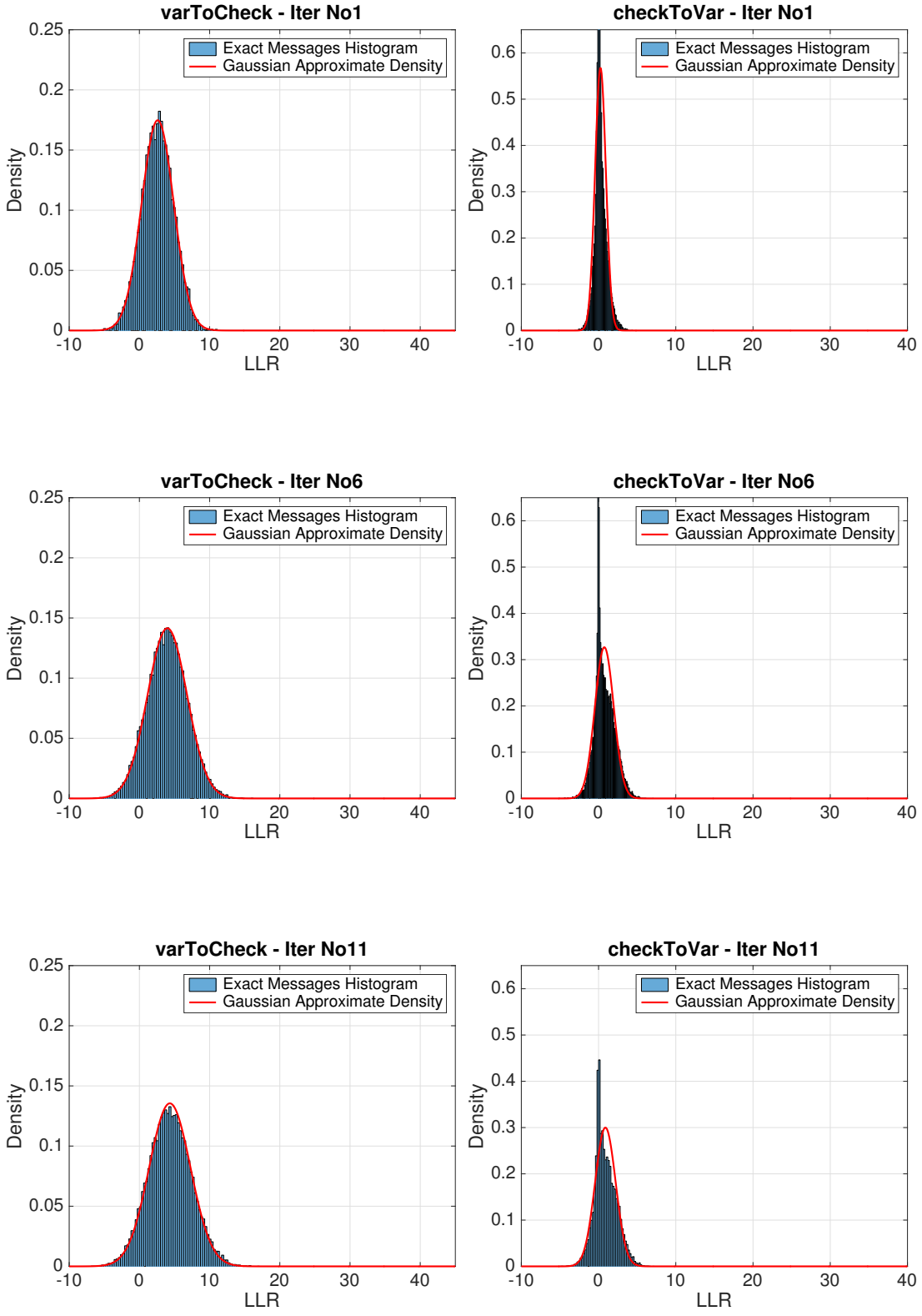

FIGURE 5.4: Exact and Gaussian approximation message densities for a regular (3,6) LDPC code at 1st, 6th, and 11th iteration of message-passing decoding with $\sigma < \sigma^*$.

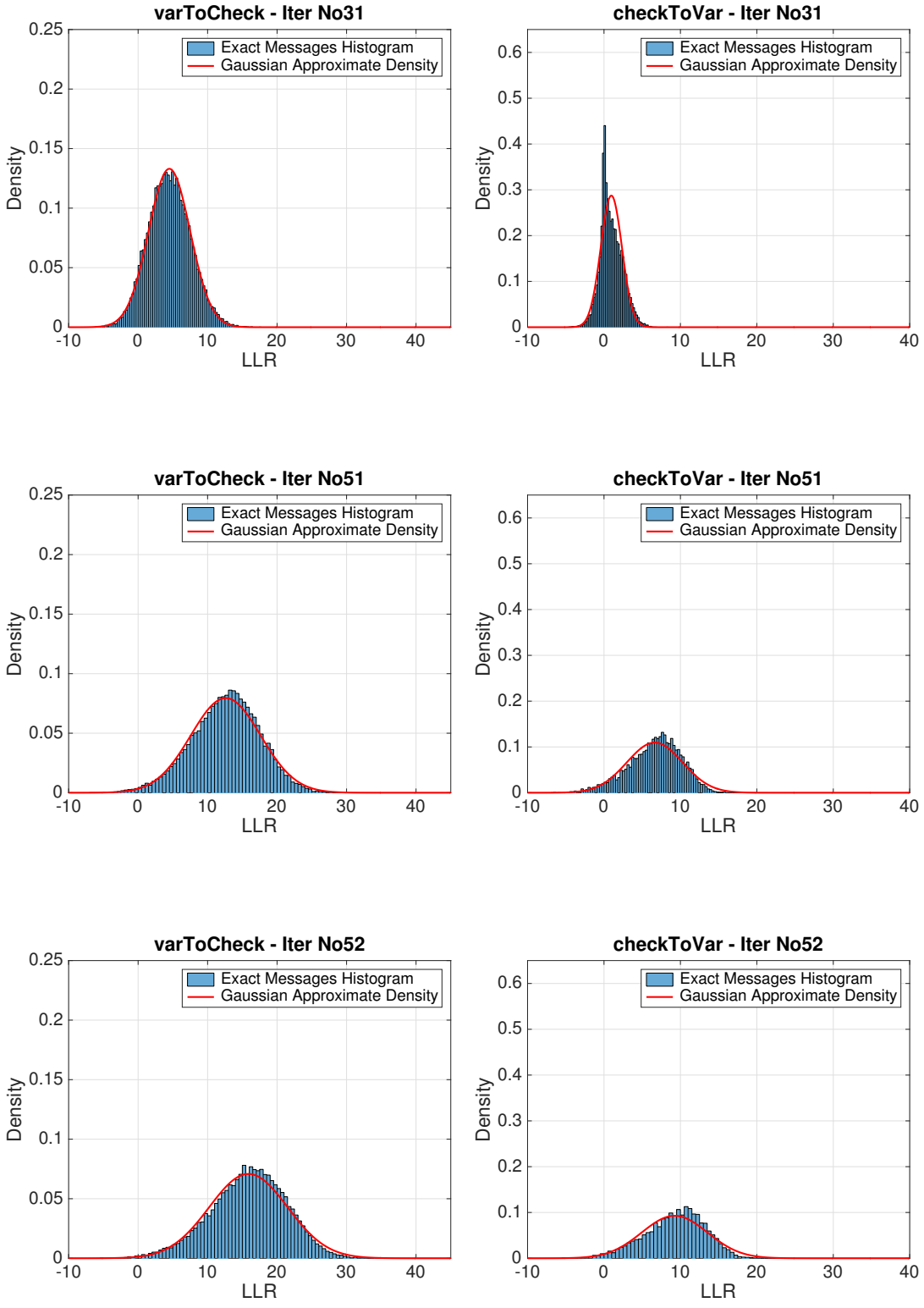

FIGURE 5.7: Exact and Gaussian approximation message densities for a regular (3,6) LDPC code at 31st, 51st, and 52nd iteration of message-passing decoding with $\sigma < \sigma^*$.

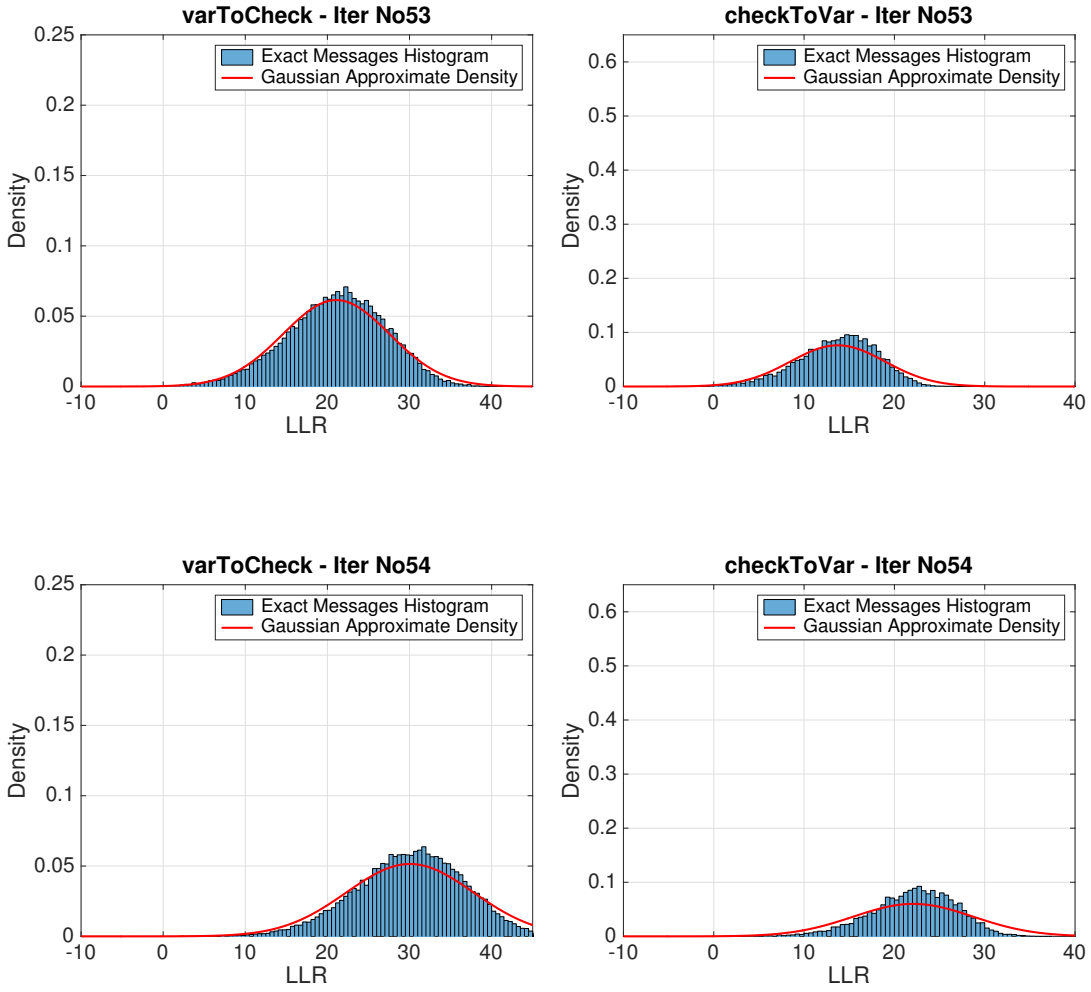

FIGURE 5.9: Exact and Gaussian approximation message densities for a regular (3,6) LDPC code at 53rd and 54th iteration of message-passing decoding with $\sigma < \sigma^*$.

We see that for $\sigma < \sigma^*$, the densities "move to the right", indicating that the error probability decreases as a function of the number of iterations. Furthermore, we see that the Gaussian approximation works well for the messages output from variable nodes.

Now, we see the densities of messages emitted from variable and check nodes for $\sigma = 0.8783 > \sigma^*$ for various iterations.

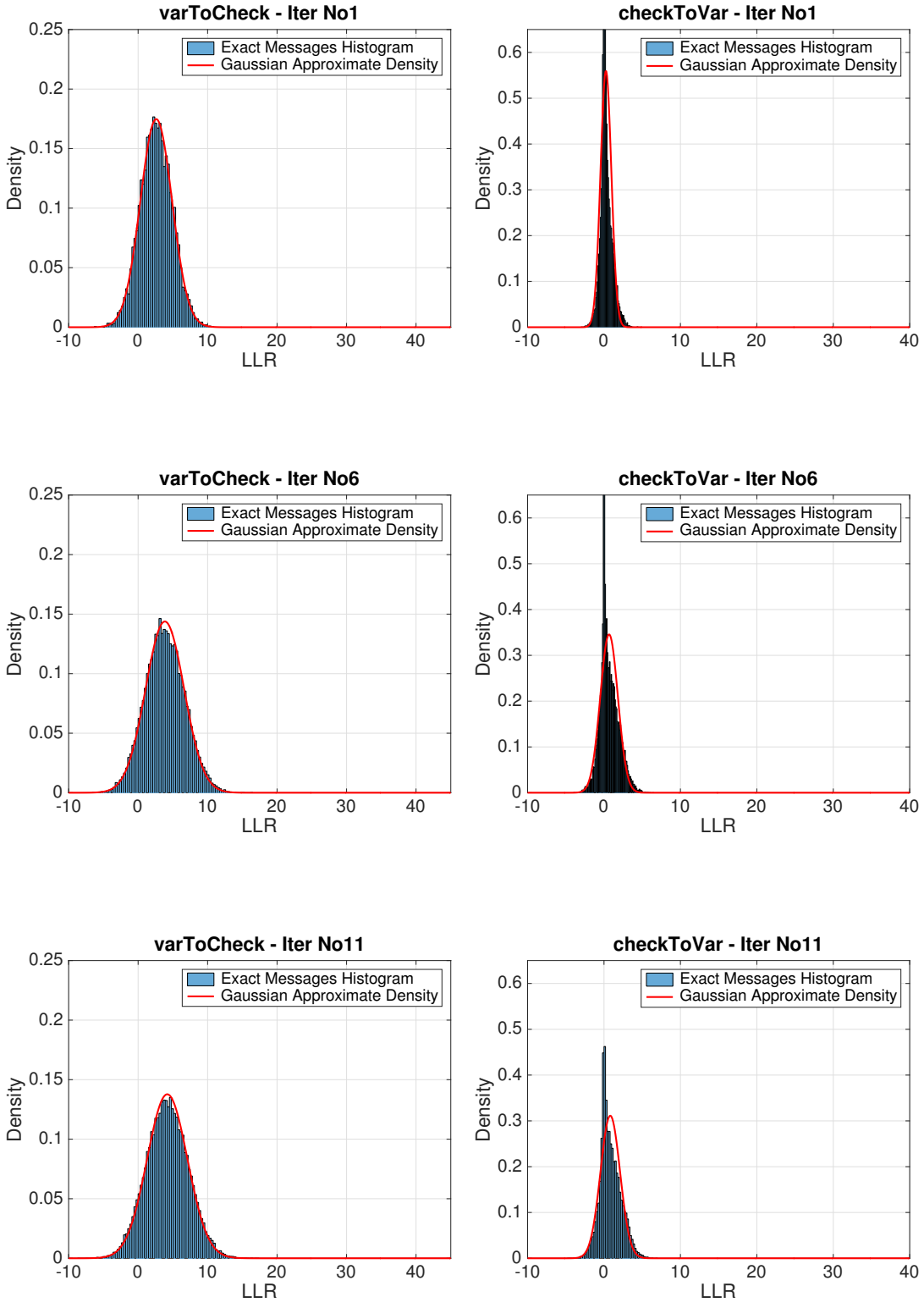

FIGURE 5.12: Exact and Gaussian approximation message densities for a regular (3,6) LDPC code at 1st, 6th and 11th iteration of message-passing decoding with $\sigma > \sigma^*$.

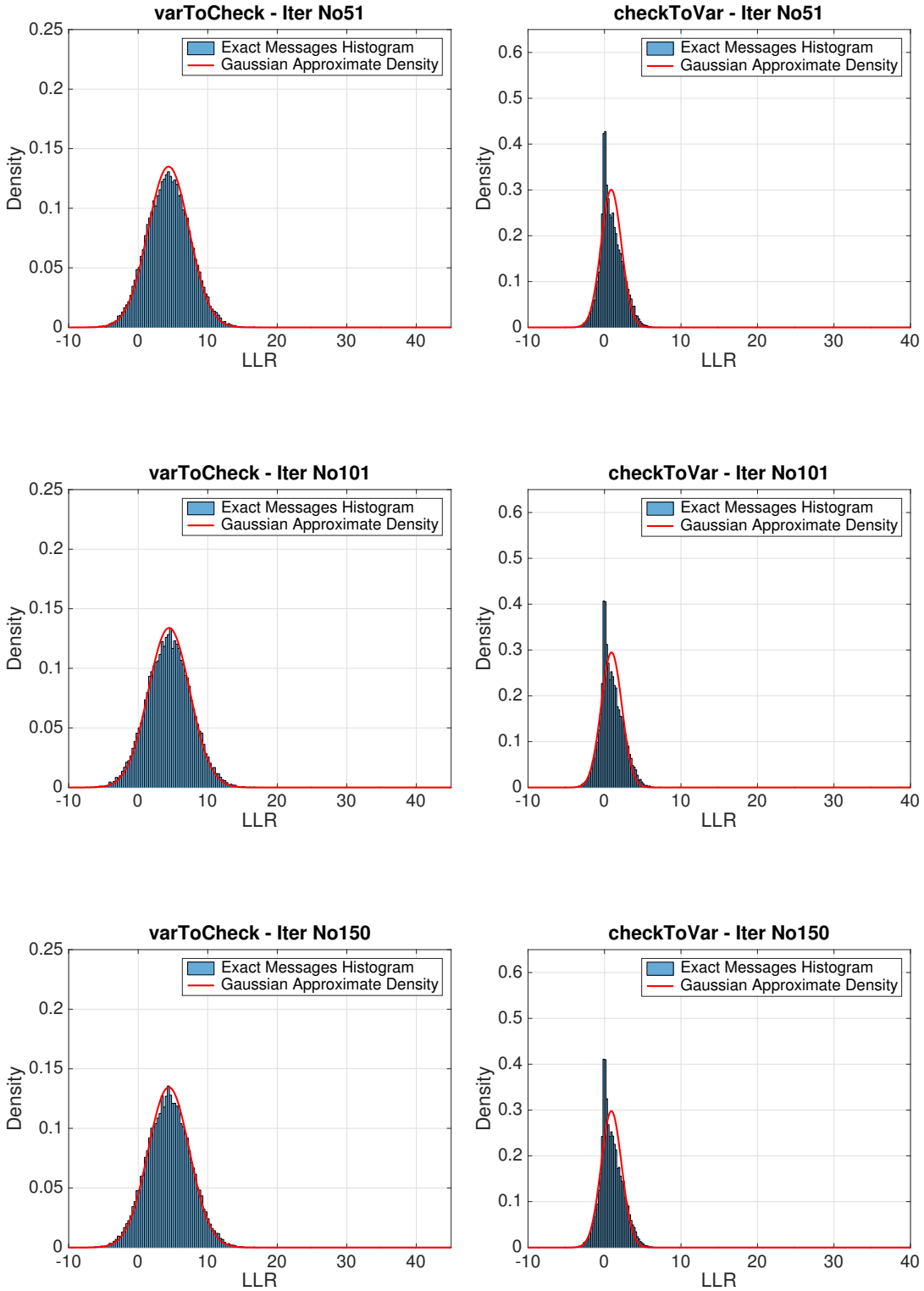

FIGURE 5.15: Exact and Gaussian approximation message densities for a regular (3,6) LDPC code at 51st, 101st and 150th iteration of message-passing decoding with $\sigma > \sigma^*$.

So, for $\sigma > \sigma^*$, the densities start "moving to the right," but stop at a fixed point after a certain number of iterations.

Additionally, we show the decoding performance of the regular $(3, 6)$ code with the threshold and Shannon limit in Figure 5.16 as far as the average iterations for the particular threshold in Figure 5.17.

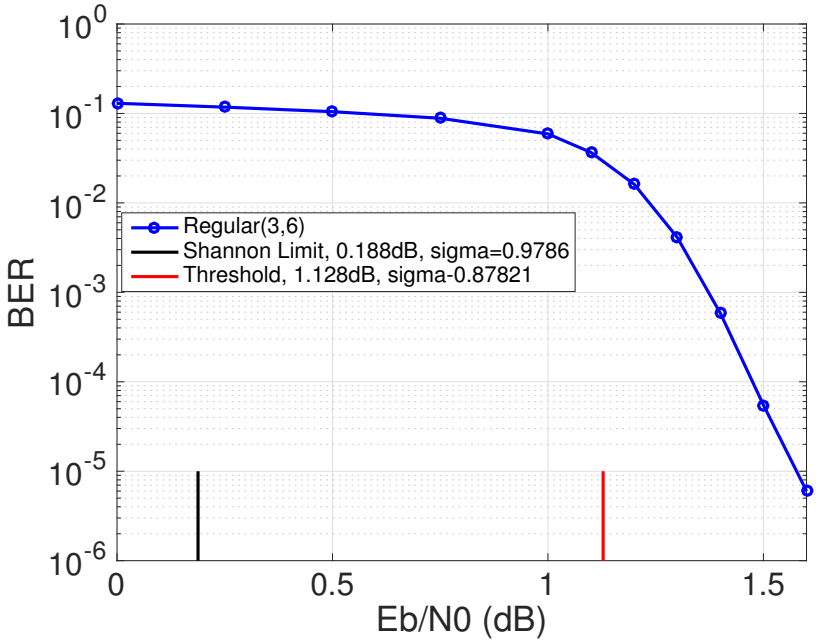

FIGURE 5.16: The decoding performance on a binary AWGN channel of length-10^4 rate-1/2 Regular$(3, 6)$ LDPC code.

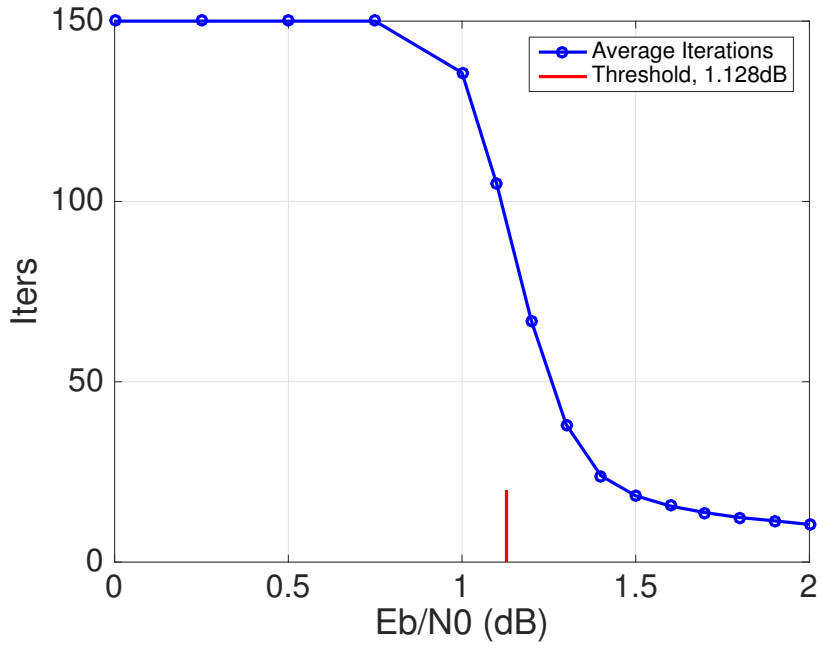

FIGURE 5.17: The average number of iterations of decoding the Regular$(3, 6)$ LDPC code.

5.3 Gaussian approximation for irregular LDPC codes

We consider an ensemble of random codes with degree distributions $\lambda(x)$ and $\rho(x)$. A node will get iid messages from its neighbors, where each of these messages is a random mixture of different Gaussian densities from neighbors with different degrees. Now, the mean $m_{v,i}^{(l)}$ of the output message of a degree-i variable node at lth iteration is given by

$$m_{v,i}^{(l)} = m_{u0} + (i-1)m_u^{(l-1)}, \tag{5.7}$$

where m_{u0} is the mean of u_0 and $m_u^{(l-1)}$ is the mean of u at the $(l-1)$ iteration. The variance of the output density is given by $2m_{v,i}^{(l)}$, as in the last section. Because a randomly chosen edge is connected to a degree-i variable node with probability λ_i, averaging over all degrees i yields that at the lth iteration, an incoming message v to a check node will have the following Gaussian mixture density $f_v^{(l)}$

$$f_v^{(l)}(\tau) = \sum_{i=1}^{d_v} \lambda_i \mathcal{N}(\tau; m_{v,i}^{(l)}, 2m_{v,i}^{(l)}). \tag{5.8}$$

In this expression, $\mathcal{N}(\tau; \mu, \sigma^2)$ represents the pdf for the Gaussian random variable with mean m and variance σ^2, at point τ.

Using (5.8) and the definition of ϕ, we have

$$\begin{aligned} E\Big[\tanh\frac{v^{(l)}}{2}\Big] &= \int_{-\infty}^{+\infty} \tanh\frac{v^{(l)}}{2}\Big(\sum_{i=1}^{d_v}\lambda_i\mathcal{N}(\tau; m_{v,i}^{(l)}, 2m_{v,i}^{(l)})\Big)\mathrm{d}\tau \\ &= \sum_{i=1}^{d_v}\lambda_i\int_{-\infty}^{+\infty}\tanh\frac{v^{(l)}}{2}\mathcal{N}(\tau; m_{v,i}^{(l)}, 2m_{v,i}^{(l)})\mathrm{d}\tau \\ &= \sum_{i=1}^{d_v}\lambda_i\left(1-\phi(m_{v,i}^{(l)})\right) \\ &= 1-\sum_{i=1}^{d_v}\lambda_i\phi(m_{v,i}^{(l)}). \end{aligned} \tag{5.9}$$

Furthermore, for the message $u^{(l)}$ that leaves a check node of degree j, we have

$$\begin{aligned} E\Big[\tanh\frac{u_j^{(l)}}{2}\Big] &= \Big(E\Big[\tanh\frac{v^{(l)}}{2}\Big]\Big)^{j-1} \\ \Leftrightarrow \Big(1-\phi(m_{u_j}^{(l)})\Big) &= \Big(1-\sum_{i=1}^{d_v}\lambda_i\phi(m_{v,i}^{(l)})\Big)^{j-1} \\ \Leftrightarrow m_{u_j}^{(l)} &= \phi^{-1}\Big(1-\Big(1-\sum_{i=1}^{d_v}\lambda_i\phi(m_{v,i}^{(l)})\Big)^{j-1}\Big). \end{aligned}$$

Finally, if we average over all check-node degrees j, we have

$$\begin{aligned} m_u^{(l)} &= \sum_{j=1}^{d_c}\rho_j m_{u_j}^{(l)} = \sum_{j=1}^{d_c}\rho_j\phi^{-1}\Big(1-\Big(1-\sum_{i=1}^{d_v}\lambda_i\phi(m_{v,i}^{(l)})\Big)^{j-1}\Big) \\ &= \sum_{j=1}^{d_c}\rho_j\phi^{-1}\Big(1-\Big(1-\sum_{i=1}^{d_v}\lambda_i\phi(m_{u_0}+(i-1)m_u^{(l-1)})\Big)^{j-1}\Big), \end{aligned} \tag{5.10}$$

where $m_{u_0}=2/\sigma_n^2$ is the mean of the message from the channel, σ_n^2 is the noise variance of the AWGN channel, and the initial value of $m_u^{(0)}$ is set to 0.

Equation (5.10) is the Gaussian-approximation recursion. Its notation can be simplified as follows. Let $s\in[0,\infty)$, $t\in[0,\infty)$, and define

$$\begin{aligned} f_j(s,t) &= \phi^{-1}\Big(1-\Big[1-\sum_{i=1}^{d_v}\lambda_i\phi(s+(i-1)t)\Big]^{j-1}\Big), \\ f(s,t) &= \sum_{j=1}^{d_c}\rho_j f_j(s,t). \end{aligned} \tag{5.11}$$

Hence, the simplified recursion is

$$t_l = f(s,t_{l-1}), \tag{5.12}$$

where $s=m_{u_0}$, $t_l=m_u^{(l)}$, and $t_0=0$. The threshold is

$$s^* = \inf\Big\{s : s\in[0,\infty) \text{ and } \lim_{l\to\infty} t_l(s)=\infty\Big\}.$$

We can find an alternative expression to (5.11), which will be more useful. From (5.7), we have

$$\begin{aligned} m_{v,i}^{(l)} &= m_{u_0} + (i-1)m_u^{(l-1)} \\ \overset{(5.10)}{\Leftrightarrow}\ m_{v,i}^{(l)} &= m_{u_0} + (i-1)\sum_{j=1}^{d_c}\rho_j\phi^{-1}\Big(1-\Big(1-\sum_{i=1}^{d_v}\lambda_i\phi(m_{v,i}^{(l-1)})\Big)^{j-1}\Big) \\ \Leftrightarrow\ \phi(m_{v,i}^{(l)}) &= \phi\Big(m_{u_0} + (i-1)\sum_{j=1}^{d_c}\rho_j\phi^{-1}\Big(1-\Big(1-\sum_{i=1}^{d_v}\lambda_i\phi(m_{v,i}^{(l-1)})\Big)^{j-1}\Big)\Big). \end{aligned}$$

Averaging over all variable-node degrees i, gives

$$\sum_{i=1}^{d_v}\lambda_i\phi(m_{v,i}^{(l)}) = \sum_{i=1}^{d_v}\lambda_i\phi\Big(m_{u_0} + (i-1)\sum_{j=1}^{d_c}\rho_j\phi^{-1}\Big(1-\Big(1-\sum_{i=1}^{d_v}\lambda_i\phi(m_{v,i}^{(l-1)})\Big)^{j-1}\Big)\Big).$$

We set $r_l := \sum_{i=1}^{d_v}\lambda_i\phi(m_{v,i}^{(l)})$ and $s := m_{u_0}$. Then, for $0 < s < \infty$, $0 \le r \le 1$, we define $h_i(s,r)$ and $h(s,r)$ as

$$h_i(s,r) = \Big(s + (i-1)\sum_{j=1}^{d_c}\rho_j\phi^{-1}\Big(1-(1-r)\Big)^{j-1}\Big) \tag{5.13}$$

$$h(s,r) = \sum_{i=1}^{d_v}\lambda_i h_i(s,r). \tag{5.14}$$

Then, (5.12) becomes equivalent to

$$r_l = h(s, r_{l-1}). \tag{5.15}$$

The initial value of r_0 is $\phi(s)$. It is easy to see that $t_l \to \infty$ iff $r_l \to 0$.

For **code design** using Gaussian approximation, we simplify the problem and assume that the check node degree distribution is given. Now, we only need to look for a good variable node degree distribution. The problem now consists of maximizing the rate of the code. We know that

$$r = 1 - \frac{\int_0^1\rho(x)dx}{\int_0^1\lambda(x)dx} = 1 - \frac{\sum_i\rho_i/i}{\sum_i\lambda_i/i}.$$

So, for fixed $\rho(x)$, we need to maximize $\sum_i \lambda_i/i$. So we can design an LDPC code by solving the following optimization problem

$$\begin{aligned} &\max \sum_{i=2}^{d_v} \lambda_i/i \\ &\text{subject to} \quad \sum_i \lambda_i = 1 \\ &\qquad\qquad r > h(s,r)\ \forall r \in (0, \phi(s)). \end{aligned} \tag{5.16}$$

Example 5.3.1. Consider the evolution of message densities for the binary AWGN channel and the rate-1/2 degree distribution pair

$$\lambda(x) = 0.212332x + 0.197596x^2 + 0.0142733x^4 + 0.0744898x^5 + 0.0379457x^6$$
$$+0.0693008x^7 + 0.086264x^8 + 0.00788586x^{10} + 0.0168657x^{11} + 0.283047x^{30},$$

$$\rho(x) = x^8.$$

We set $\sigma = 0.939 < \sigma^*$, so we expect that the messages "move to the right".

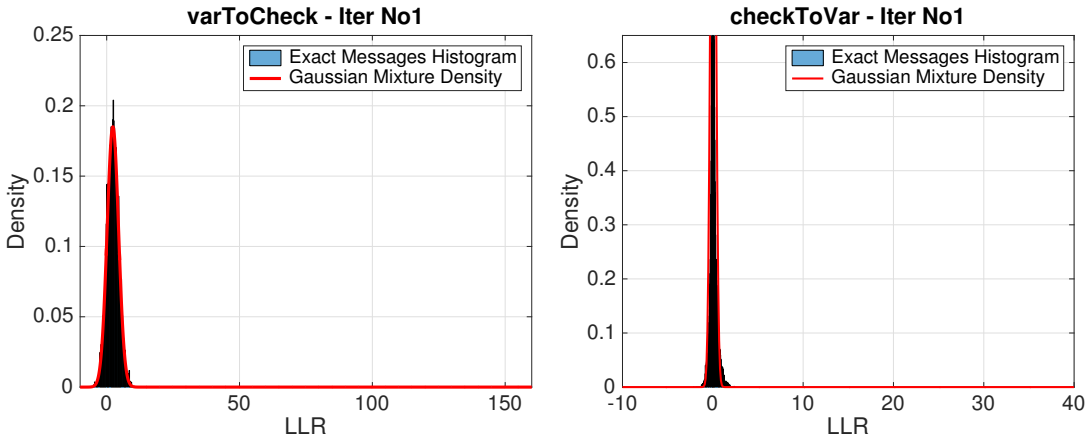

FIGURE 5.18: Exact and Gaussian-mixture message densities for an irregular LDPC code at 1st iteration of message-passing decoding with $\sigma < \sigma^*$.

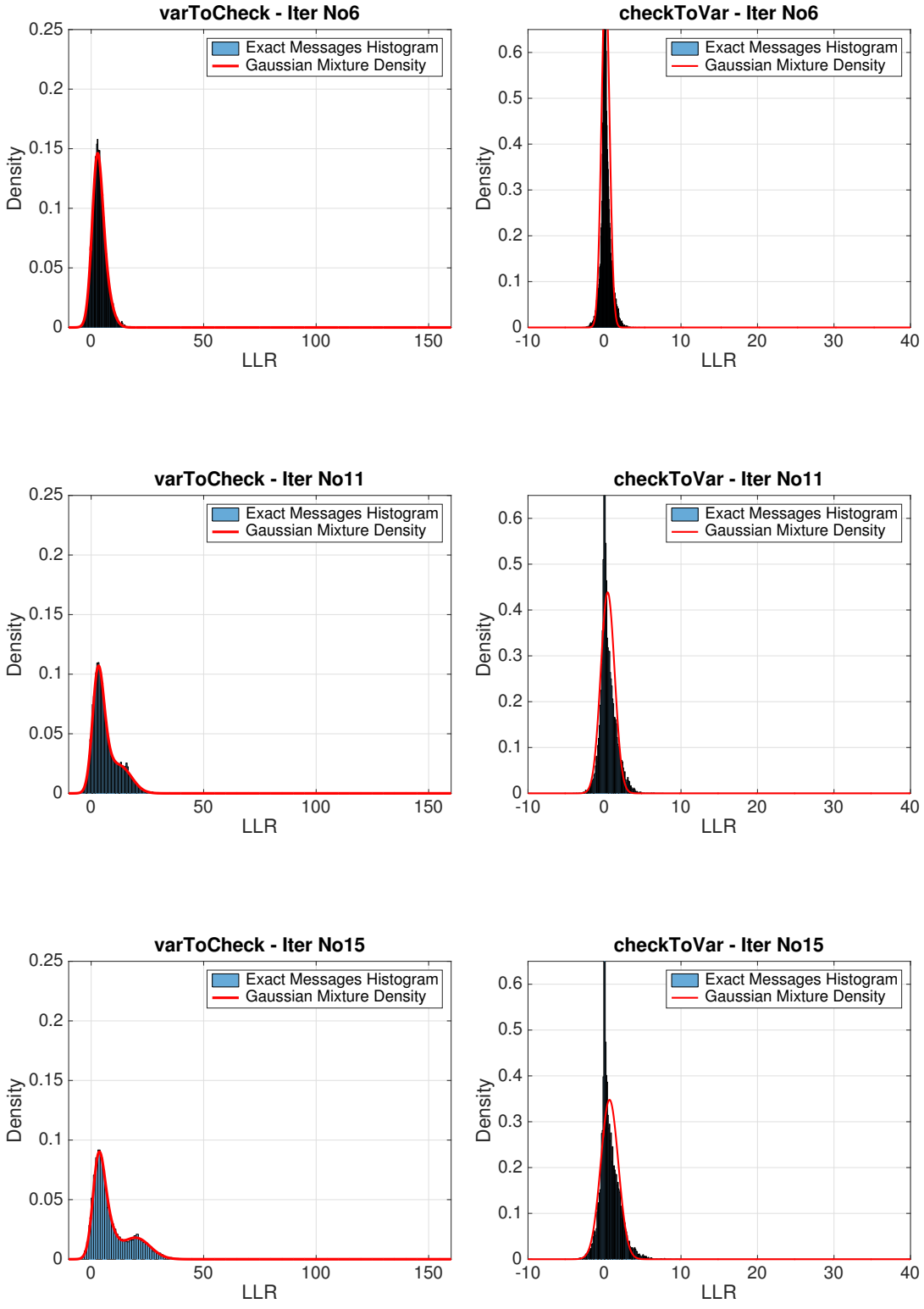

FIGURE 5.21: Exact and Gaussian-mixture message densities for an irregular LDPC code at 6th, 11th, and 15th iteration of message-passing decoding with $\sigma < \sigma^*$.

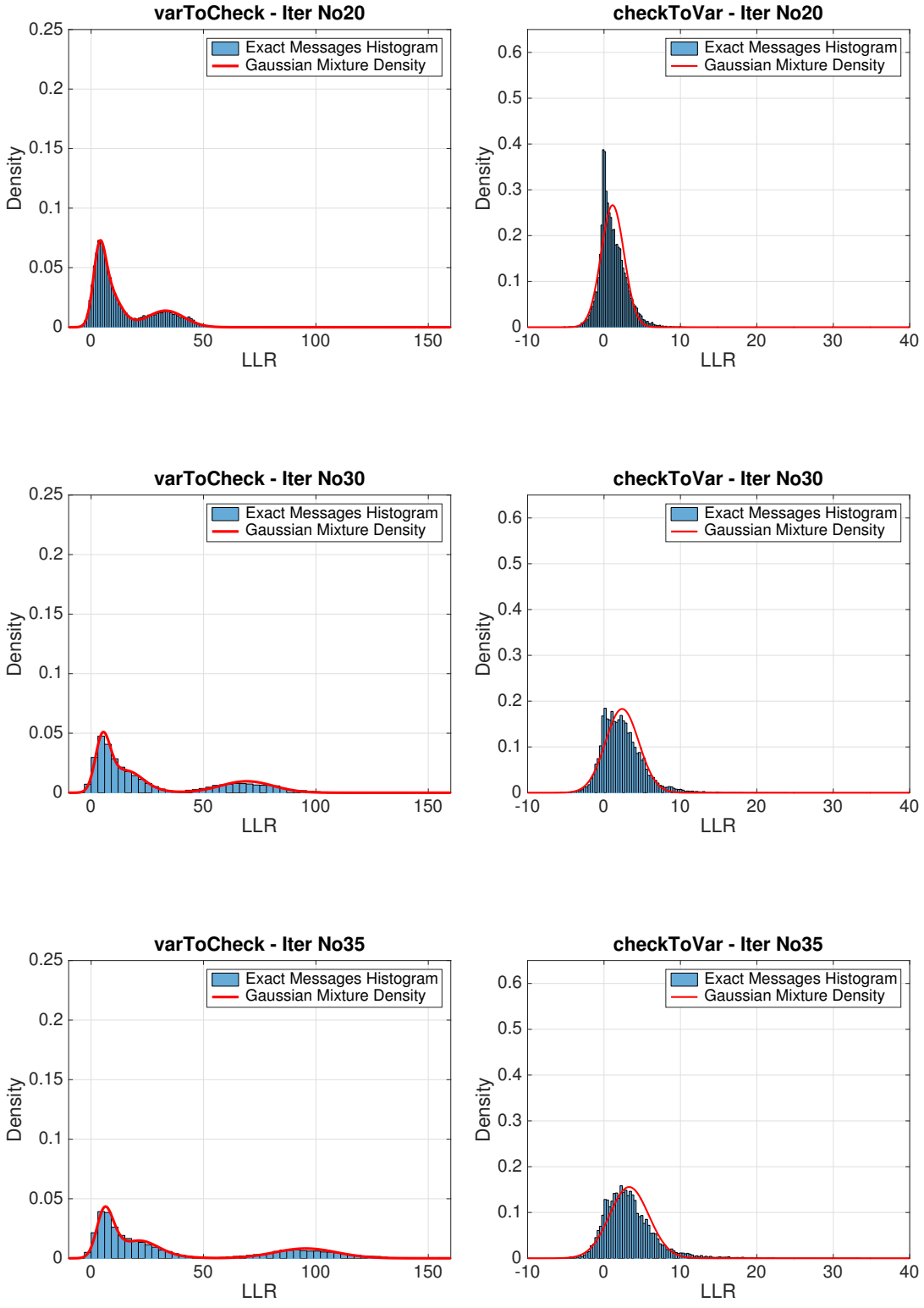

FIGURE 5.24: Exact and Gaussian-mixture message densities for an irregular LDPC code at 20th, 30th and 35th iteration of message-passing decoding with $\sigma < \sigma^*$.

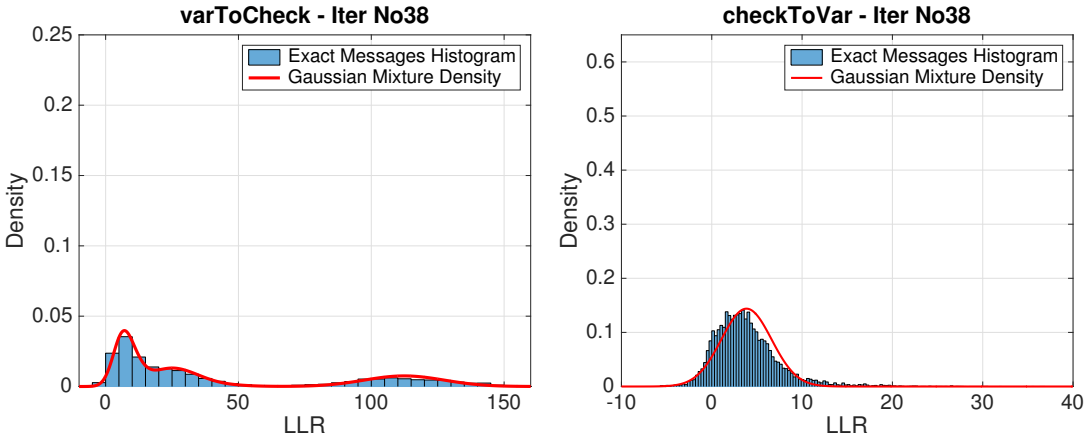

FIGURE 5.25: Exact and Gaussian-mixture message densities for an irregular LDPC code at 38th iteration of message-passing decoding with $\sigma < \sigma^*$.

As the iteration number is increased, the area under the curve for negative LLRs decreases and so the probability of error decreases. Moreover, we see that Gaussian mixture densities and the exact densities are very close for the message emitted from variable nodes. For the output messages of the check nodes, some Gaussian mixtures are very different from the exact densities. However, because of the fact that the two densities in the other direction are very close suggests that the approximation is working well.

Chapter 6

Accumulator-Based LDPC Codes

6.1 Introduction

The accumulator-based codes that were invented first are called repeat-accumulate (RA) codes [16]. Despite their simple structure, they were shown to provide good performance and, more importantly, they belong to the category of efficiently encodable LDPC codes. RA codes are LDPC codes that can be decoded as serial turbo codes, but are more commonly treated as LDPC codes.

6.2 Repeat-accumulate codes

As shown in Figure 6.1, an RA code consists of a serial concatenation, through an interleaver, of a single rate-$1/q$ repetition code with an accumulator having transfer function $1/(1+D)$. The accumulator simply outputs the modulo-2 sum of the current input bit and the previous output bit. More precisely, it provides a running sum of all past inputs. To ensure a large minimum Hamming distance, the interleaver should be designed so that consecutive 1s at its input are widely separated at its output. To see this, observe that two 1s separated by $(s-1)$ 0s at the accumulator input, will yield a run of s 1s at the accumulator output. So, we want s to be large, in order to achieve large codeword Hamming weight and, consequently, large minimum distance of the code.

RA codes that transmit both the message bits and parity bits are called systematic RA codes. Contrary to this, the RA codes that transmit only the parity bits are called

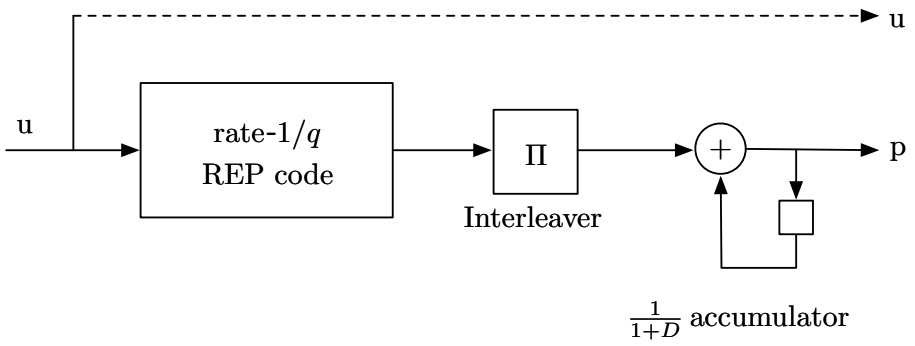

FIGURE 6.1: A repeat-accumulate code block diagram.

non-systematic. In the first case, the information word $\mathbf{u}$, is combined with $\mathbf{p}$ to yield the codeword $\mathbf{c} = [\mathbf{u}\ \mathbf{p}]$, so that the code rate is $1/(1+q)$. For non-systematic RA codes, the accumulator output $\mathbf{p}$ is the codeword and the code rate is $1/q$. The main limitations of the RA codes are the code rate, which cannot be higher than $1/2$, and performance at small or medium lengths.

6.3 Irregular repeat-accumulate codes

The irregular repeat-accumulate (IRA) codes [17] generalize the RA codes in that the repetition rate may differ for each of the k information bits and that linear combinations of the repeated bits are sent through the accumulator. Further, IRA codes are typically systematic. IRA codes provide two advantages over RA codes. Firstly, they allow flexibility in the choice of the repetition rate for each information bit so that high-rate codes can be designed. Secondly, their irregularity allows operation closer to the capacity limit.

The Tanner graph for IRA codes is presented in Figure 6.2. The variable repetition rate is accounted for in the graph by letting the variable node degrees $d_{b,i}$ vary. The accumulator is depicted by the lowest part of the graph. Figure 6.2 also includes the representation for RA codes. For an RA code, each information bit node is connected to exactly q check nodes ($d_{b,i} = q$) and each check node is connected to exactly one information bit node ($d_{c,j} = 1$).

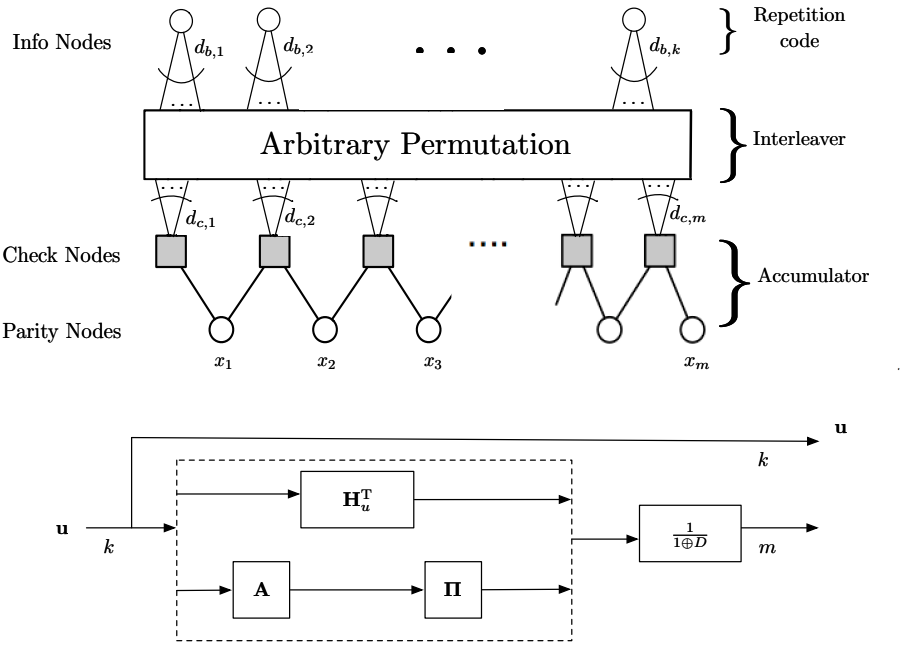

FIGURE 6.2: A Tanner graph (up) and encoder (down) for irregular repeat-accumulate codes.

To determine the code rate for an IRA code, define $\bar{q}$ to be the average repetition rate of the information bits,

$$\bar{q} = \frac{1}{k} \sum_{i=1}^{k} d_{b,i},$$

and $\bar{d_c}$ as the average of the degrees $\{d_{c,j}\}$,

$$\bar{d_c} = \frac{1}{m} \sum_{j=1}^{m} d_{c,j}.$$

Then, the code rate for systematic IRA codes is

$$R = \frac{1}{1 + \bar{q}/\bar{d_c}}.$$

For non-systematic IRA codes, $R = \bar{d_c}/\bar{q}$.

The parity-check matrix for systematic RA and IRA codes has the form

$$\mathbf{H} = [\mathbf{H}_u \ \mathbf{H}_p], \tag{6.1}$$

where $\mathbf{H}_p$ is an $m \times m$ dual-diagonal matrix,

$$\mathbf{H}_p = \begin{bmatrix} 1 & & & & \\ 1 & 1 & & & \\ & \ddots & \ddots & & \\ & & 1 & 1 & \\ & & & 1 & 1 \end{bmatrix}. \tag{6.2}$$

For RA codes, $\mathbf{H}_u$ is a regular matrix having column weight q and row weight 1. For IRA codes, $\mathbf{H}_u$ has column weights $\{d_{b,i}\}$ and row weights $\{d_{c,j}\}$. The encoder of Figure 6.2 is obtained by noting that the generator matrix corresponding to $\mathbf{H} = [\mathbf{H}_u \ \mathbf{H}_p]$ is

$$\mathbf{G} = [\mathbf{I} \ \ \mathbf{H}_u^{\mathrm{T}} \mathbf{H}_p^{-\mathrm{T}}],$$

and writing $\mathbf{H}_u$ as $\mathbf{\Pi}^{\mathrm{T}} \mathbf{A}^{\mathrm{T}}$, where $\mathbf{\Pi}$ is a permutation matrix. Note also that

$$\mathbf{H}_p^{-\mathrm{T}} = \begin{bmatrix} 1 & 1 & \cdots & & 1 \\ & 1 & 1 & \cdots & 1 \\ & & \ddots & & \vdots \\ & & & 1 & 1 \\ & & & & 1 \end{bmatrix}, \tag{6.3}$$

performs the same computation as the $1/(1 \oplus D)$ accumulator.

Given the code rate, length, and degree distributions, an IRA code is defined entirely by the matrix $\mathbf{H}_u$ (equivalently, by $\mathbf{A}$ and $\mathbf{\Pi}$).

6.4 Repeat-accumulate codes on the AWGN channel

In an iterative sum-product message-passing decoding algorithm, we recall that all messages are assumed to be log-likelihood ratios. The outgoing message from a variable node to a check node is

$$v = \sum_{i=0}^{d_b - 1} u_i, \tag{6.4}$$

where $u_i, i = 1, \ldots, d_b - 1$, are the incoming LLRs from the neighbors of the variable node except the check node that gets the message v, and u_0 is the observed LLR of the output bit associated with the variable node.

Moreover, the outgoing message from a check node to a variable node is

$$u = 2 \tanh^{-1} \left(\prod_{j=1}^{d_c+1} \tanh \left(\frac{v_j}{2} \right) \right)$$

or equivalently

$$\tanh \frac{u}{2} = \prod_{j=1}^{d_c+1} \tanh \frac{v_j}{2}, \tag{6.5}$$

where $v_j, j = 1, \ldots, d_c - 1$, are the incoming LLRs from $d_c - 1$ information node neighbors of a check node, and the additional 2 are the incoming LLRs from the parity node neighbors.

We recall that λ_i is the fraction of edges between the information and the check nodes that are adjacent to an information node of degree i. Now let ρ_i be the fraction of such edges that are adjacent to a check node of degree $i + 2$ (i.e. one which is adjacent to $i + 1$ information nodes). Then, the degree distributions from an edge perspective are

$$\lambda(x) = \sum_{i=1}^{d_b} \lambda_i x^{i-1}, \qquad \rho(x) = \sum_{i=1}^{d_c} \rho_i x^{i-1},$$

where d_b and d_c denote the maximum information and check node degree respectively. The rate of the IRA code for a (λ, ρ) degree distribution is given by

$$\text{Rate} = \left(1 + \frac{\sum_j \rho_j / j}{\sum_j \lambda_j / j} \right)^{-1}. \tag{6.6}$$

For the AWGN channel, we have only two possible inputs, 0 and 1. If the X is the input, the output is $Y = (-1)^X + Z$, with $Z \sim \mathcal{N}(0, \sigma^2)$. For a given noise variance σ^2, our objective is to find a degree distribution $\lambda(x)$ such that the ensemble message error probability approaches zero, while the rate is as large as possible. In the AWGN, we deal with probability densities which force us to work on approximate design methods.

6.4.1 Gaussian approximation

We recall that a Gaussian distribution $f(x)$ is called consistent if $f(x) = f(-x)\exp^x, \forall x \leq 0$. The consistency condition implies that the mean and variance satisfy $\sigma^2 = 2m$. For the sum-product algorithm, consistency is preserved at message updates of both the variable and check nodes. Thus, if we assume Gaussian messages and require consistency, we only need to keep track of the means. So, $\mathcal{N}(m, 2m)$ represents the pdf for a consistent Gaussian random variable. Moreover, in our analysis we consider concentrated $\rho(x) = x^{a-1}$.

The expected value of $\tanh\frac{x}{2}$ for a consistent Gaussian distributed random variable x with mean m is

$$E\Big[\tanh\frac{x}{2}\Big] = \frac{1}{\sqrt{4\pi m}}\int_{\mathbb{R}}\tanh\frac{x}{2}e^{-\frac{(x-m)^2}{4m}}\mathrm{d}x \triangleq \phi(m). \tag{6.7}$$

It is easy to see that $\phi(u)$ is a monotonic and increasing function of u. We denote its inverse function by $\phi^{-1}(u)$.

Now, let $m_{u_I}^{(l)}$ and $m_{u_P}^{(l)}$ be the means of the messages from check nodes to information nodes and parity nodes respectively, at the lth iteration. We want to obtain expressions for $m_{u_I}^{(l+1)}$ and $m_{u_P}^{(l+1)}$ in terms of $m_{u_I}^{(l)}$ and $m_{u_P}^{(l)}$. A message from a degree-i information node to a check node at lth iteration is Gaussian with

$$m_{v_I,i}^{(l)} = m_{u_0} + (i-1)m_{u_I}^{(l-1)}, \tag{6.8}$$

where m_{u_0} is the mean of channel message. So, if v_I denotes the message on a randomly selected information node to a check node, the Gaussian mixture density for the information node messages is

$$f_{v_I}^{(l)}(\tau) = \sum_{i=1}^{d_b}\lambda_i\mathcal{N}(\tau; m_{v_I,i}^{(l)}, 2m_{v_I,i}^{(l)}). \tag{6.9}$$

This pdf is important to us, because we will need to take the expected value of a function of the form $\tanh(v/2)$ with respect to the pdf of v. Specifically, we have

$$\begin{aligned} E\Big[\tanh\frac{v_I^{(l)}}{2}\Big] &= \int_{-\infty}^{+\infty} \tanh\frac{v_I^{(l)}}{2}\Big(\sum_{i=1}^{d_b}\lambda_i\mathcal{N}(\tau; m_{v_I,i}^{(l)}, 2m_{v_I,i}^{(l)})\Big)\mathrm{d}\tau \\ &= \sum_{i=1}^{d_b}\lambda_i\int_{-\infty}^{+\infty}\tanh\frac{v_I^{(l)}}{2}\mathcal{N}(\tau; m_{v_I,i}^{(l)}, 2m_{v_I,i}^{(l)})\mathrm{d}\tau \\ &= \sum_{i=1}^{d_b}\lambda_i\phi(m_{v_I,i}^{(l)}) \\ &= \sum_{i=1}^{d_b}\lambda_i\phi(m_{u_0}+(i-1)m_{u_I}^{(l-1)}). \end{aligned} \tag{6.10}$$

Similarly, if v_P denotes the message on a randomly selected edge from a parity node to a check node, we have

$$E\Big[\tanh\frac{v_P^{(l)}}{2}\Big] = \phi(m_{u_P}^{(l-1)} + m_{u_0}). \tag{6.11}$$

From (6.5), we have

$$E\Big[\tanh\frac{u^{(l)}}{2}\Big] = \prod_{j=1}^{d_c+1} E\Big[\tanh\frac{v_j^{(l)}}{2}\Big]. \tag{6.12}$$

Denote a message from a check node to an information node and a parity node by u_I and u_P, respectively. Now, from (6.10),(6.11) and (6.12), we have the following equalities

$$\begin{aligned} E\Big[\tanh\frac{u_I^{(l)}}{2}\Big] &= E\Big[\tanh\frac{v_I^{(l)}}{2}\Big]^{a-1}E\Big[\tanh\frac{v_P^{(l)}}{2}\Big]^2 \\ &= \Big(\sum_{i=1}^{d_b}\lambda_i\phi(m_{u_0}+(i-1)m_{u_I}^{(l-1)})\Big)^{a-1}\Big(\phi(m_{u_P}^{(l-1)}+m_{u_0})\Big)^2, \end{aligned} \tag{6.13}$$

$$\begin{aligned} E\Big[\tanh\frac{u_P^{(l)}}{2}\Big] &= E\Big[\tanh\frac{v_I^{(l)}}{2}\Big]^{a}E\Big[\tanh\frac{v_P^{(l)}}{2}\Big] \\ &= \Big(\sum_{i=1}^{d_b}\lambda_i\phi(m_{u_0}+(i-1)m_{u_I}^{(l-1)})\Big)^{a}\Big(\phi(m_{u_P}^{(l-1)}+m_{u_0})\Big), \end{aligned} \tag{6.14}$$

where a is the check node degree of our concentrated $\rho(x) = x^{a-1}$. Using the definition of $\phi(m)$ in (6.7), we have the following recursion for $m_{u_I}^{(l)}$ and $m_{u_P}^{(l)}$

$$\phi(m_{u_I}^{(l+1)}) = \Big(\sum_{i=1}^{d_b} \lambda_i \phi(m_{u_0} + (i-1)m_{u_I}^{(l)})\Big)^{a-1} \Big(\phi(m_{u_P}^{(l)} + m_{u_0})\Big)^2, \tag{6.15}$$

$$\phi(m_{u_P}^{(l+1)}) = \Big(\sum_{i=1}^{d_b} \lambda_i \phi(m_{u_0} + (i-1)m_{u_I}^{(l)})\Big)^{a} \Big(\phi(m_{u_P}^{(l)} + m_{u_0})\Big). \tag{6.16}$$

In order to have small bit error probability, the means $m_{u_I}^{(l)}$ and $m_{u_P}^{(l)}$ should approach infinity as l approaches infinity.

6.4.2 Design of IRA codes

We now assume that iterative decoding has reached a fixed point of (6.15) and (6.16), so $m_{u_I}^{(l+1)} = m_{u_I}^{(l)} = m_{u_I}$ and $m_{u_P}^{(l+1)} = m_{u_P}^{(l)} = m_{u_P}$. Denote $\sum_{i=1}^{d_b} \lambda_i \phi(m_{u_0} + (i-1)m_{u_I})$ by x. From (6.10) we can see that $0 < x < 1$ and $\lim_{m_{u_I} \to \infty} x = 1$. From (6.15), it is easy to show that m_{u_I} is a function of x, denoted by f, so $m_{u_I} = f(x)$. Then, dividing (6.15) by the square of (6.16) gives

$$\phi(m_{u_I}^{(l+1)}) = \phi^2(m_{u_P}^{(l+1)})/x^{a+1} = \phi^2(f(x))/x^{a+1}. \tag{6.17}$$

Now, replacing $m_{u_I}^{(l+1)}$ with $\phi^{-1}(\phi^2(f(x))/x^{a+1})$ into the definition of x, we obtain the following equation for the fixed point x

$$x = \sum_{i=1}^{d_b} \lambda_i \phi\Big(m_{u_0} + (i-1)\phi^{-1}\big(\frac{\phi^2(f(x))}{x^{a+1}}\big)\Big). \tag{6.18}$$

If this equation does not have a solution in the interval $[0, 1]$, then the decoding bit error probability converges to zero. Therefore, if we have

$$F(x) \triangleq \sum_{i=1}^{d_b} \lambda_i \phi\Big(m_{u_0} + (i-1)\phi^{(-1)}\big(\frac{\phi^2(f(x))}{x^{a+1}}\big)\Big) > x, \tag{6.19}$$

for any $x \in [x_0, 1)$, where x_0 is the value of x at the first iteration, then the iterative decoding is successful.

a	8	8
λ_2		0.0577128
λ_3	0.252744	0.117057
λ_7		0.2189922
λ_8		0.0333844
λ_{11}	0.081476	
λ_{12}	0.327162	
λ_{18}		0.2147221
λ_{20}		0.0752259
λ_{46}	0.184589	
λ_{48}	0.154029	
λ_{55}		0.0808676
λ_{58}		0.202038
rate	0.50227	0.497946

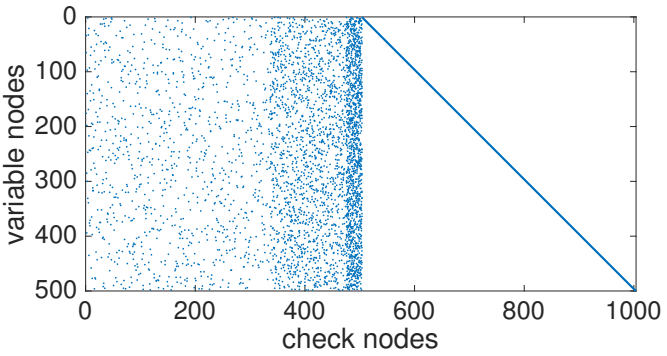

FIGURE 6.3: The optimization table of IRA codes and the sparsity of the $\lambda_2 = 0$ and length= 1000 IRA code.

The rate of the code is given by

$$\frac{\sum_i \lambda_i / i}{1/a + \sum_i \lambda_i / i}. \tag{6.20}$$

So, to maximize the rate, we should maximize $\sum_i \lambda_i / i$. Thus, we can design an IRA code by solving the following linear programming problem

$$\begin{aligned} &\max \sum_{i=1}^{d_b} \lambda_i / i \\ &\text{subject to } F(x) > x, \forall x \in [x_0, 1]. \end{aligned} \tag{6.21}$$

Using this linear programming methodology, the authors of [17], designed IRA codes for rate= 0.5, which are shown in the Table of Figure 6.3.

We decode the IRA code from the table with $\lambda_2 = 0$ for length $n = 10^4, 10^5, 10^6$ and we observe the results in Figure 6.4.

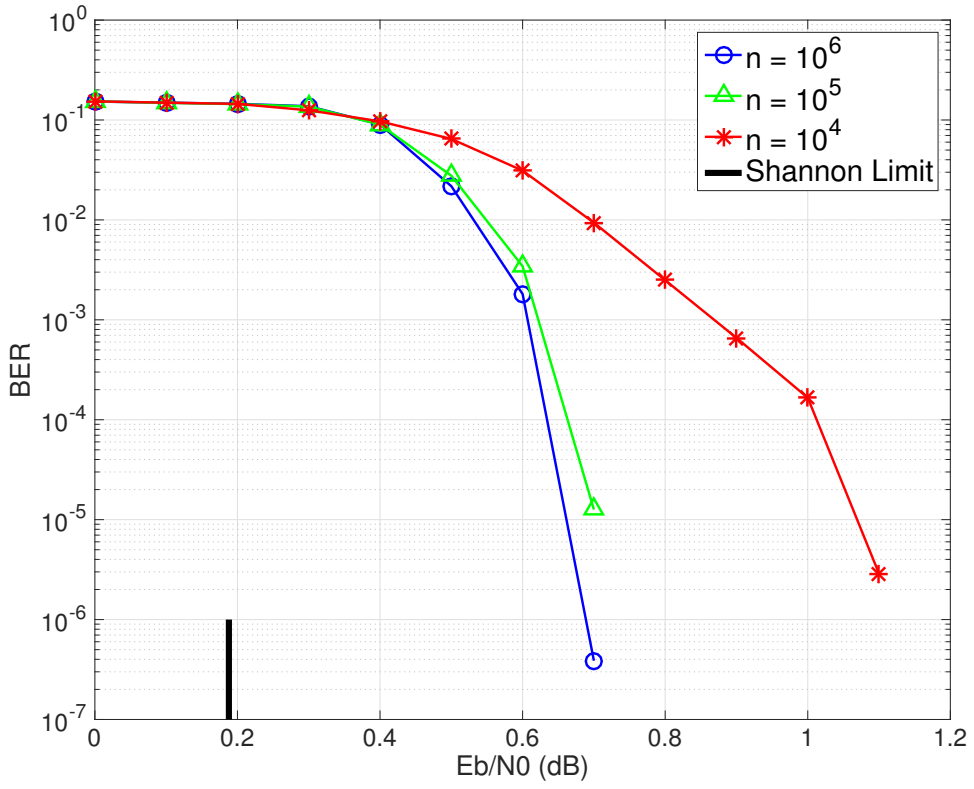

FIGURE 6.4: The error rate for a binary AWGN channel using sum-product algorithm for different lengths of rate-1/2 optimized irregular repeat-accumulate codes.

Chapter 7

Conclusion

In this thesis, we summarized fundamental concepts related with LDPC and Repeat-Accumulate codes.

These codes can be studied using concepts from graph theory. We considered in detail the encoding procedure as well as the decoding procedure with examples and performance figures. We constructed codes with several methods and saw the differences of their performance. Furthermore, we analyzed them using Gaussian approximation, something that led us to new design methods.

Because of the limited degrees of freedom, repeat-accumulate are inferior to LDPC codes for decoding but they are easier to encode.

Bibliography

[1] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, vol. 27: pp. 379–423, 623–656, July, October, 1948.

[2] William E. Ryan and Shu Lin. *Channel Codes, Classical and Modern.* Cambridge University Press, 2008.

[3] Daniel J. Costello Shu Lin. *Error Control Coding.* Pearson Prentice Hall, 2004.

[4] Sarah J. Johnson. *Iterative Error Correction.* Cambridge University Press, 2010.

[5] Robert G Gallager. *Low Density Parity Check Codes.* M.I.T. Press, 1963.

[6] R. Michael Tanner. A recursive approach to low complexity codes. *IEEE Trans. Inform. Theory*, vol. IT-27: pp. 533–547, Sep. 1981.

[7] D. MacKey and R. Neal. Good codes based on very sparse matrices. *Cryptography and Coding, 5th IMA Conf.*, Oct. 1995.

[8] D. MacKey. Good error correcting codes based on very sparse matrices. *IEEE Trans. Inform. Theory*, vol. 45, no. 3: pp. 399–431, March 1999.

[9] Thomas J. Richardson Sae-Young Chung, G. David Forney and Rudiger Urbanke. On the design of low-density parity-check codes within 0.0045 db of the shannon limit. *IEEE Communication Letters*, vol. 5, no. 2: pp. 58–60, Feb. 2001.

[10] M. Amin Shokrollahi Thomas J. Richardson and Rudiger L. Urbanke. Design of capacity-approaching irregular low-density parity-check codes. *IEEE Trans. Inform. Theory*, vol. 47, no. 2: pp. 619–637, Feb. 2001.

[11] T. J. Richardson and R. Urbanke. *Modern Coding Theory.* Cambridge University Press, 2009.

[12] Evangelos Eleftheriou Xiao-Yu Hu and Dieter-Michael Arnold. Regular and irregular prograssive edge-growth tanner graphs. *IEEE Trans. Inform. Theory*, vol. 51, no. 1: pp. 386–398, Jan. 2005.

[13] T.J. Richardson and R.Urbanke. The capacity of low-density parity-check codes under message- passing decoding. *IEEE Trans. Inform. Theory*, vol. 47: pp. 599–618, Feb. 2001.

[14] T. J. Richardson S. Y. Chung and R. L. Urbanke. Analysis of sum-product decoding of low-density parity-check codes using a gaussian approximation. *IEEE Trans. Inform. Theory*, vol. 47, no. 2: pp. 657–670, Feb. 2001.

[15] A. Shokrollahi T. J. Richardson and R. Urbanke. Design of capacity- approaching low-density parity-check codes. *IEEE Trans. Inform. Theory*, vol. 47: pp. 619–637, Feb. 2001.

[16] H. Jin D. Divsalar and R. McEliece. Coding theorems for turbo-like codes. *Proc. 36th Annual Allerton Conf. on Communication, Control, and Computing*, pages pp. 201–210, Sep. 1998.

[17] Aamod Khandekar Hui Jin and Robert McEliece. Irregular repeat–accumulate codes. *2nd Int. Conference on Turbo Codes*, Sep. 2000.