Low Density Parity Check Codes: Encodings and Density Evolution

Brian Kurkoski
kurkoski@ice.uec.ac.jp

Review: The Communication Problem

$$\begin{tikzcd} {} \arrow[r, "\mathbf{u}"] & \boxed{\text{Encoder}} \arrow[r, "\mathbf{x}"] & \boxed{\text{Channel}} \arrow[r, "\mathbf{y}"] & \boxed{\text{Decoder}} \arrow[r, "\hat{\mathbf{x}}"] & {} \end{tikzcd}$$

The problem is to communicate reliably over a noisy channel. An **error correcting code** C is specified by a parity-check matrix H. The code C is the set of all length-N sequences $\mathbf{x}$ such that:

$$\mathbf{x}\, H^t = 0$$

Only binary codes are considered, and the dimension of the code is K.

If the information sequence, of length K, is $\mathbf{u}$. A matrix G for which:

$$\mathbf{u}\, G = \mathbf{x}$$

is called the generator.

Low-density parity check codes are a class of error correcting code which:

- Are powerful error correcting codes,
- Have a low-complexity decoding algorithm.

Outline: Low Density Parity Check Codes

Encoding LDPC codes

- Canonical form check matrix, generator
- Richardson-Urbanke encoding method
- Quasicyclic codes and shift register encoder

Message passing in the logarithm domain

- Derivation of log-domain messages
- Messages and their densities

Analysis of decoding algorithms

- Irregular LDPC codes
- Density evolution

Encoding

The generator matrix, *G,* for an error-correcting block code *C,* is a *K*-by-*N* matrix, *G* for the code *C:*

$$G = \begin{bmatrix} — & g_1 & — \\ — & g_2 & — \\ & \vdots & \\ — & g_K & — \end{bmatrix}$$

Each row of the matrix, g_i , is a 1-by-N vector.

A special form of *G* is its **systematic form**. Let I_K be the *K*-by-*K* identity matrix and let *P* be a *K*-by-(*N*–*K*) **parity submatrix**: The **canonical form** for G is:

$$G = \left[I_K \vdots P \right]$$

Let the **information vector** be $\mathbf{u} = (u_1\ u_2\ \ldots\ u_K)$. Vector **u** is has *K* elements, $u_i \in \{0,1\}$, where 0's and 1's are assumed equally likely.

Then, **x** is a codeword of *C*:

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

Encoding

$$G = \begin{bmatrix} — & g_1 & — \\ — & g_2 & — \\ & \vdots & \\ — & g_K & — \end{bmatrix}$$

- If **u** has a 1 in position *k* and 0's in all other positions, then:

$$\mathbf{x} = \mathbf{u} \cdot G = (0 \cdots 0 \underset{\text{Position } k}{1} 0 \cdots 0) \cdot G = g_k$$

- That is, g_k is a codeword.
- So, all the rows of G are codewords and we can write:

$$G \cdot H^t = \underline{0},$$

where $\underline{0}$ is a K-by-$(N-K)$ matrix of zeros.

Construction of *G*, Given *H*

Given a parity check matrix *H*, we want to construct the generator *G*.

Let *P* be the (*N*-*K*)-by-*K* parity submatrix. The **canonical form** for the parity-check matrix is the form:

$$H = \left[P^t \vdots I_{N-K} \right]$$

Then you can show that:

$$\underbrace{\left[I_k \vdots -P \right]}_{G} \cdot \underbrace{\left[\begin{array}{c} P \\ \cdots \\ I_k \end{array} \right]}_{H^t} = \underline{0}$$

That is, $G = [\ I_k \mid -P^t\ ]$ is a generator for the code *C*.

For binary codes, −1 = 1 and 0 = -0, so $P = -P$ and:

$$G = \left[I_K \vdots P \right]$$

Example Encoding: Hamming Code

For example, a possible parity-check matrix for the (7,4) Hamming Code is below. This matrix is already in the form $H = [P \mid I_k\,]$. Then, the generator $G = [\,I_k \mid P\,]$.

$$H = \left[\begin{array}{cccc|ccc} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right] \qquad G = \left[\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{array}\right]$$

Encoding. If the information vector is $\mathbf{u} = (1\ 0\ 0\ 1)$, then the corresponding codeword $\mathbf{x}$ is:

$$\begin{aligned} \mathbf{x} &= \mathbf{u}\,G \\ &= (1\ 0\ 0\ 1)\left[\begin{array}{ccccccc} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{array}\right] \\ &= (1\ 0\ 0\ 1\ 0\ 0\ 1) \end{aligned}$$

LDPC Code Example: Random code

Consider the following parity-check matrix.

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

We use Gaussian elimination to put the matrix in canonical form.

Performing row and column operations does not change the code.

LDPC Code Example: Canonical Form

Thus, the canonical form is:

$$\widetilde{H} = \left[\begin{array}{ccccc|ccccccccccccccc}
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}\right]$$

Note that randomly generated matrices may not be full rank. Thus, the effective rate of the code is slightly higher than the design rate. The dependent rows of the parity check matrix can be eliminated.

LDPC Code Example: Systematic Generator

So the generator matrix in systematic form is $G = [\ I \mid P\ ]$:

$$G = \left[\begin{array}{ccccc|ccccccccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0
\end{array}\right]$$

Note that P is non-sparse. This is true in general, and we assume the density of ones is 0.5. With the information vector $\mathbf{u}$, the systematic codeword is $\mathbf{x} = (\mathbf{u}, \mathbf{p})$, where:

$$\mathbf{p} = \mathbf{u}P.$$

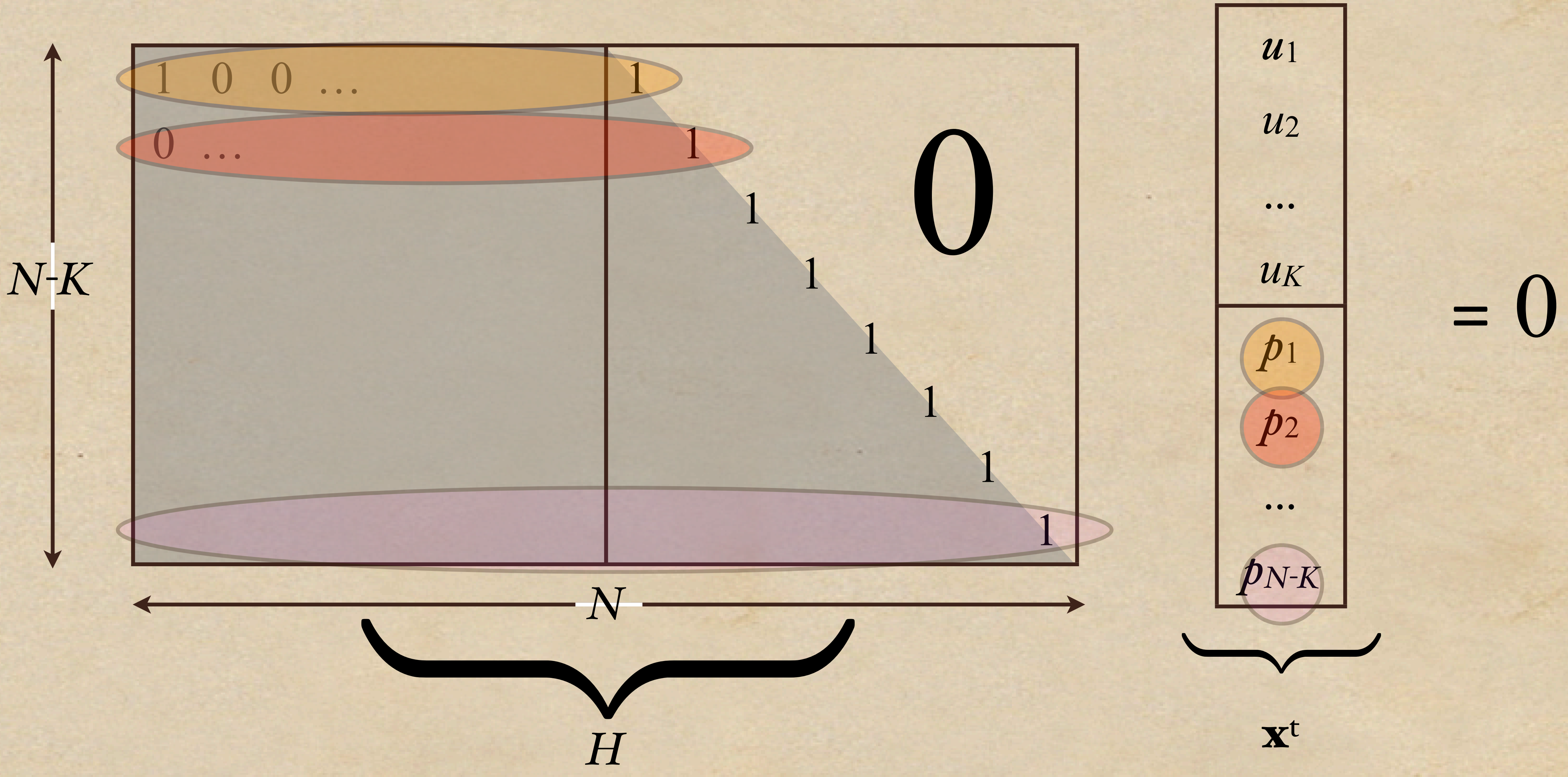
In general, the number of one's in P is $0.5(1\text{-}R)RN^2$. So $O(N^2)$ operations are required.

$\Rightarrow$ It would be much better to find an encoding $O(N)$ algorithm.

Encoding by Back-Substitution

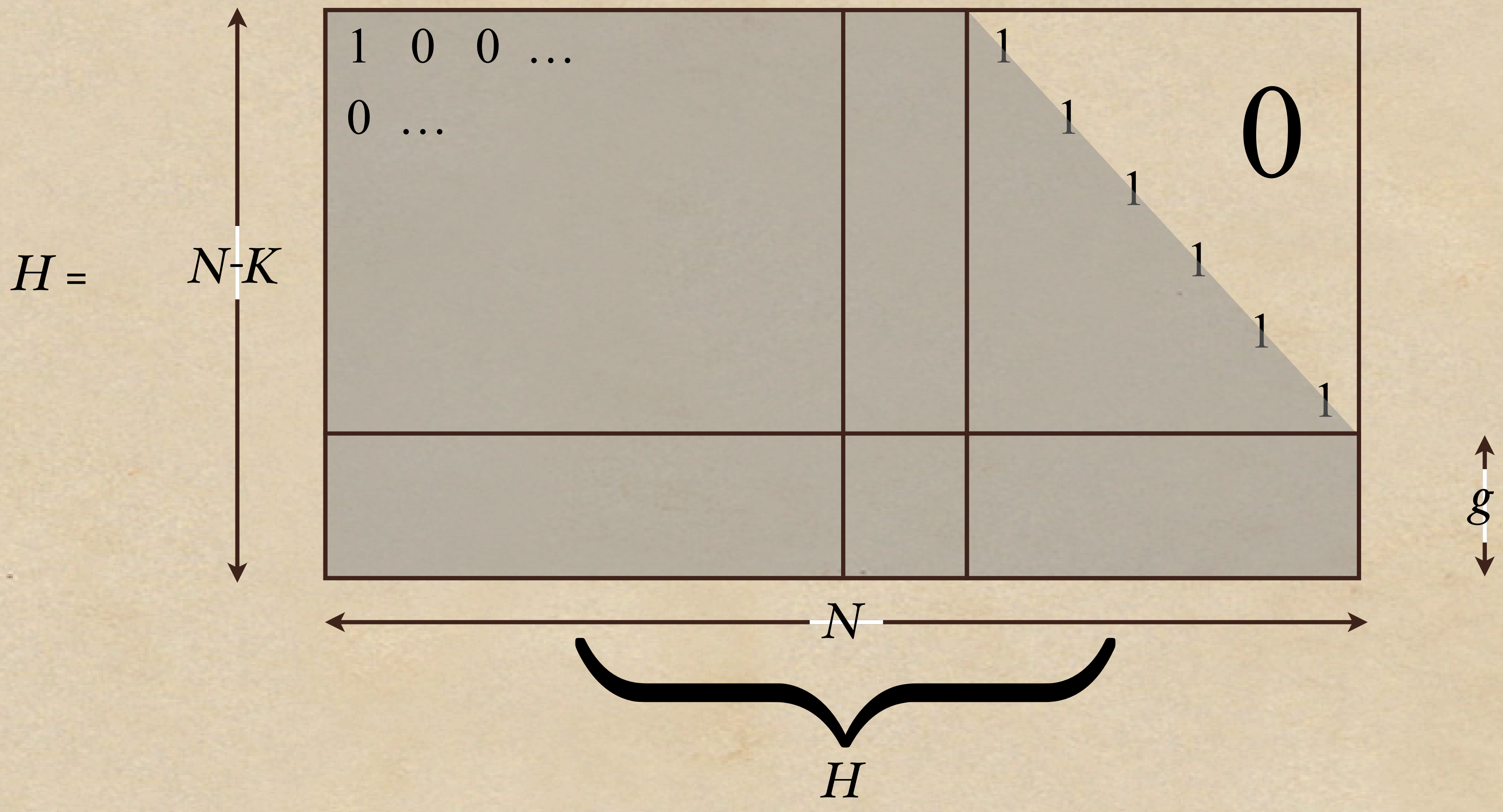
As before $\mathbf{x} = (\mathbf{u}, \mathbf{p})$. The bits $\mathbf{p} = (p_1\, p_2\, \ldots\, p_{N\text{-}K})$ are unknowns to be determined.

If we can write H in the following form:



Efficient Encoding [Richardson and Urbanke]

Suppose that H can be written in the following way:



The term g is called the "gap" and for efficient encoding, it should be as small as possible. H is partitioned as follows:

$$H = \begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix}$$

Efficient Encoding [Richardson and Urbanke]

Suppose that H can be written in the following way:

$$H = \left[\begin{array}{ccc} A & B & T \\ C & D & E \end{array} \right]$$

where T is a lower-triangular submatrix. If we multiply H on the left by:

$$\left[\begin{array}{cc} I & 0 \\ -ET^{-1} & I \end{array} \right]$$

we obtain:

$$\widetilde{H} = \left[\begin{array}{ccc} A & B & T \\ \Gamma & \phi & 0 \end{array} \right]$$

where $\Gamma = -ET^{-1}A + C$ and $\phi = -ET^{-1}B + D$.

Efficient Encoding [Richardson and Urbanke]

Let $\mathbf{x} = (\mathbf{u}, \mathbf{p_1}, \mathbf{p_2})$. The information $\mathbf{u}$ is known, we are trying to find $\mathbf{p_1}, \mathbf{p_2}$. The parity check $\widetilde{H}x^t$ becomes:

$$\begin{bmatrix} A & B & T \\ \Gamma & \phi & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{u}^t \\ \mathbf{p}_1^t \\ \mathbf{p}_2^t \end{bmatrix} = 0$$

First, we solve:

$$\begin{aligned} \Gamma\mathbf{u}^t + \phi\mathbf{p}_1^t &= 0 \\ \mathbf{p}_1^t &= -\phi^{-1}\Gamma\mathbf{u}^t \longleftarrow \end{aligned}$$

If $-\phi^{-1}\Gamma$ is precomputed, complexity is $O(g \cdot k)$. However, $O(g^2 + n)$ complexity is possible

Then, since we know $\mathbf{p}_1$ we can solve:

$$\begin{aligned} A\mathbf{u}^t + B\mathbf{p}_1^t + T\mathbf{p}_2^t &= 0 \\ \mathbf{p}_2^t &= T^{-1}(A\mathbf{u}^t + B\mathbf{p}_1^t) \longleftarrow \end{aligned}$$

$O(n)$ complexity

Quasi-Cyclic Codes

The regular LDPC codes considered this far were random constructions.

Quasi-cyclic codes are structured codes:

Cyclic Code. A code for which the a cyclic shift of a codeword is also a codeword:

$\mathbf{x}_1 = (x_1, x_2, \ldots, x_N)$ is a codeword $\rightarrow$ $\mathbf{x}_2 = (x_N, x_1, \ldots, x_{N-1})$ is a codeword.

Quasi-Cyclic Code. A code for which a cyclic shift of a codeword by l is also a codeword:

$\mathbf{x}_1 = (x_1, x_2, \ldots, x_N)$ is a codeword $\rightarrow$ $\mathbf{x}_2 = (x_{N-l+1}, x_{N+l+1}, \ldots, x_{N-l})$ is a codeword.

We are interested in quasi-cyclic codes because they have can be encoded very simply, using only shift registers.

Also, certain quasi-cycle codes have a low-density parity check matrix, and so they are suitable for decoding by the message-passing algorithm.

Circulant

Quasi-cyclic codes are constructed using circulants.

A v-by-v matrix A is **circulant** if each row is a right shift of the row above. For example:

$$A = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The algebra of binary circulant matrices is isomorphic to the algebra of polynomials modulo x^{n-1} over GF(2).

Note that for the above example, the matrix A is low density.

Construction of Quasi-Cyclic Codes

Consider the parity-check matrix of a code generated from l circulant matrices of size v-by-v:

$$H = [A_1 \; A_2 \; \cdots A_l]$$

If the circulant matrices are low density, then H will also be low density. If A_l is invertible, then the generator matrix can be written in systematic form:

$$G = \left[\begin{array}{c|c} & (A_l^{-1}A_1)^t \\ I_{v(l-1)} & \vdots \\ & (A_l^{-1}A_{l-1})^t \end{array} \right]$$

Good quasi-cyclic codes can be designed with careful selection of A_i. One such code performed within 1.1 dB of the AWGN Shannon limit for rate $R = 0.88$ [Chen, Xu, Djurdjevic, Lin, *Trans. Comm.*, July 2004].

Example of Efficient Encoding

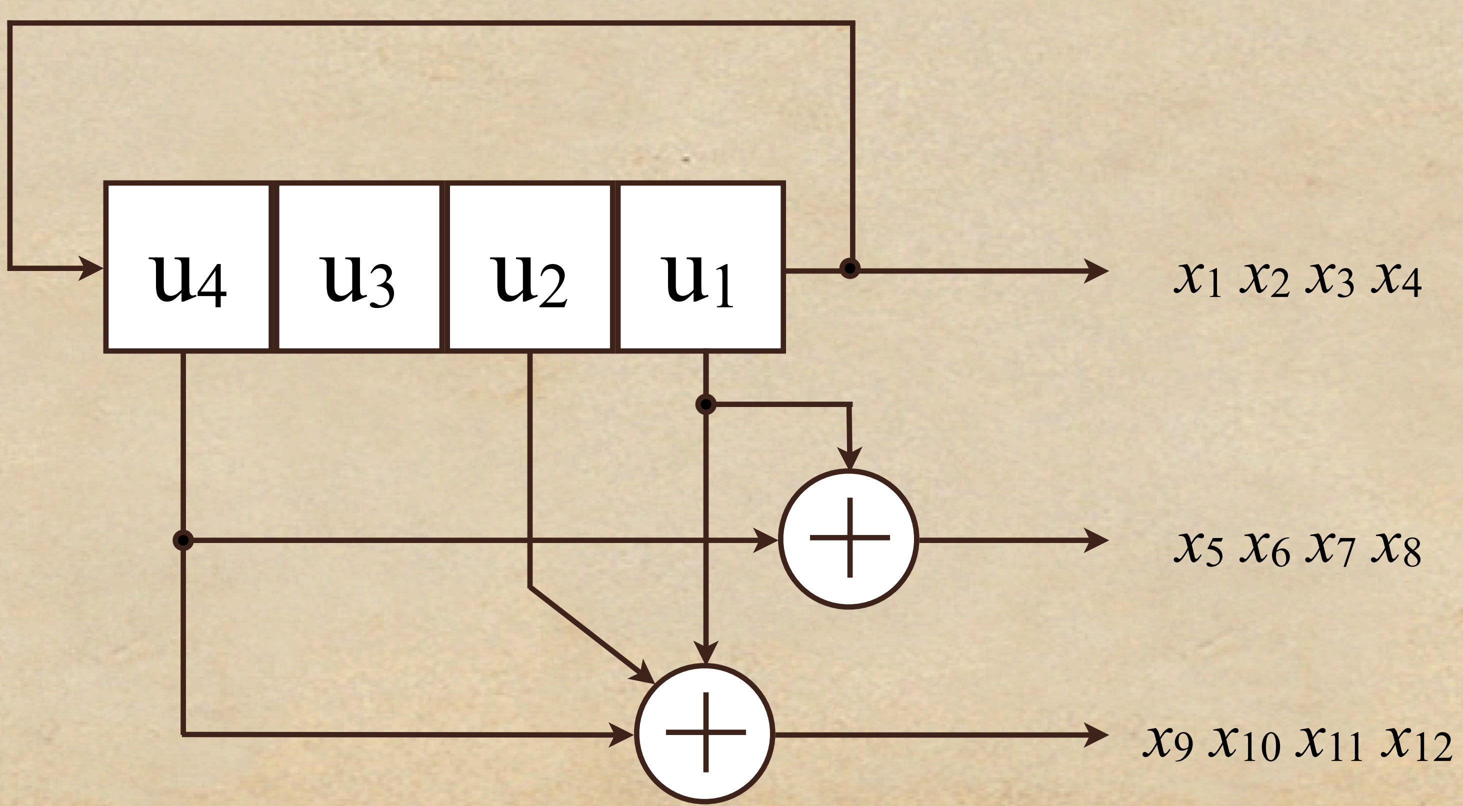
Consider the following quasi-cyclic code:

$$
\begin{aligned}
H &= \left[\begin{array}{c|c|c} A_1 & A_2 & A_3 \\ \hline A_4 & A_5 & A_6 \end{array}\right] \\
H &= \left[\begin{array}{cccc|cccc|cccc}
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\hline
1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}\right] \\
G &= \left[\begin{array}{cccc|cccc|cccc}
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1
\end{array}\right]
\end{aligned}
$$

Example of Encoding Using a Shift Register

1. Preload the information bits **u** into the shift register
2. Clock the shift register 4 times.

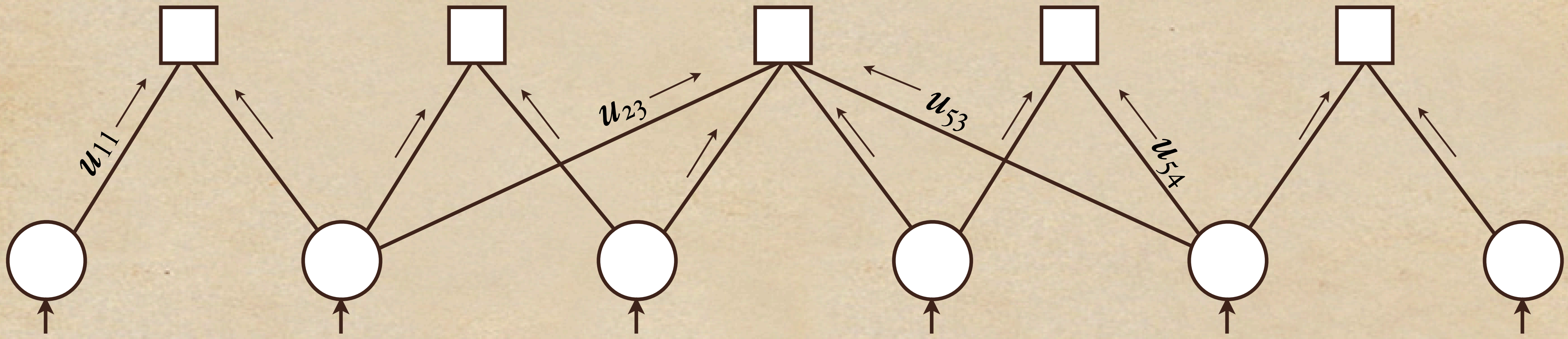
$$
\begin{aligned}
\mathbf{x} &= \mathbf{u}G \\
\mathbf{x} &= (u_1\ u_2\ u_3\ u_4)\left[\begin{array}{cccc|cccc|cccc}
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1
\end{array}\right]
\end{aligned}
$$



Review: Message Passing Decoding of LDPC Codes

The parity-matrix of an LDPC code can be represented by a Tanner graph:

- Check nodes are represented by squares
- Bit nodes are represented by circles
- There is an edge between a square and a circle if there is a 1 in the parity-check matrix.



Messages are passed iteratively between bit nodes and check nodes.

The messages passed a **probabilities**.

Log-Domain Decoding

Before, we considered message passing where messages were probabilities:

check-to-bit message: $v_i = P(x_i = 0)$

bit-to-check message: $u_i = P(x_i = 0)$

Now, we consider messages in the log domain:

check-to-bit message: $$\boxed{\begin{aligned} V_i &\overset{\text{def}}{=} \log \frac{P(x_i = 1)}{P(x_i = 0)} = \log \frac{1 - v_i}{v_i} \\ U_i &\overset{\text{def}}{=} \log \frac{P(x_i = 1)}{P(x_i = 0)} = \log \frac{1 - u_i}{u_i} \end{aligned}}$$

bit-to-check message:

We call V_i and U_i **L-Values**. Also, the channel *a posteriori* probabilities are represented using the **log-likelihood ratio (LLR)**, Y_i:

$$Y_i = \log \frac{P(x_i = 1|y_i)}{P(x_i = 0|y_i)}$$

If $Y_i < 0$, then choose $x_i = 0$. If $Y_i \geq 0$, then choose $x_i = 1$.

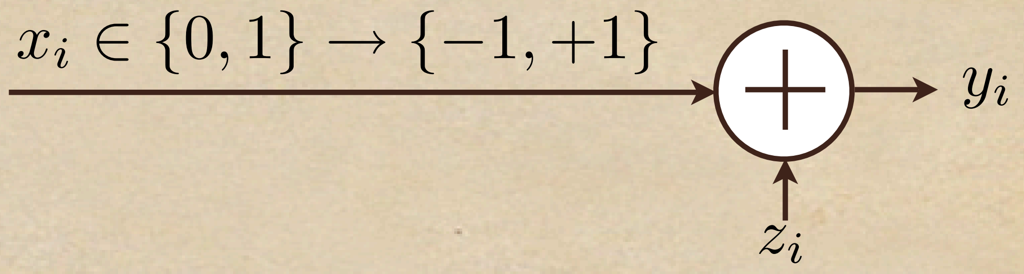
⇒ In practice, we will find that using logarithm domain messages is easier than using probability messages.

AWGN Channel *A Posteriori* Probability: P(x_i | y_i)

The message-passing algorithm requires we compute the channel P(x_i | y_i). Assume the z_i are independent.

In probability domain:

$$\begin{aligned} P(x=0|y) &= ke^{(y+1)^2/2\sigma^2} \\ P(x=1|y) &= ke^{(y-1)^2/2\sigma^2} \\ k &= \frac{1}{2\sqrt{2\pi}\sigma P(y)} \end{aligned}$$



$$z_i \sim \mathcal{N}(0,\sigma^2)$$

$$f_Z(z) = \frac{1}{\sqrt{2\pi}\sigma} e^{z^2/2\sigma^2}$$

In the log domain:

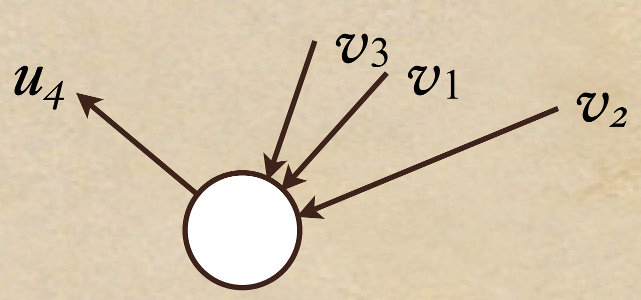
$$\begin{aligned} Y_i &= \log \frac{P(x_i=1|y_i)}{P(x_i=0|y_i)} \\ Y_i &= \log \frac{ke^{(y_i+1)^2/2\sigma^2}}{ke^{(y_i-1)^2/2\sigma^2}} \end{aligned}$$

$$\boxed{Y_i = \frac{2}{\sigma^2} y_i} \longleftarrow \text{Very Simple}$$

Log Domain Decoding: Bit Node Function

Before, we showed that if the bit node input and output message are:

$v_1 = P(x=0)$, $v_2 = P(x=0)$, $v_3 = P(x=0)$, $u_4 = P(x=0)$



Then showed that the output message is:

$$u_4 = \frac{v_1 v_2 v_3}{v_1 v_2 v_3 + (1-v_1)(1-v_2)(1-v_3)}$$

In the log domain, the L-values are:

$$V_i = \log \frac{1-v_i}{v_i}, \quad U_i = \log \frac{1-u_i}{u_i}$$

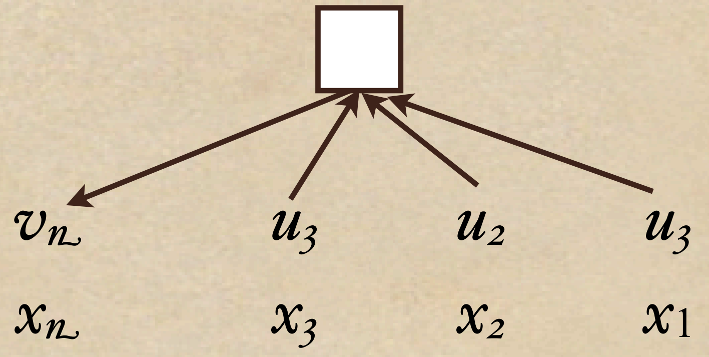
Then we can show that the bit node in the log domain is:

$$\begin{aligned}
U_4 &= \log \frac{1-u_4}{u_4} \\
U_4 &= \log \frac{(1-v_1)(1-v_2)(1-v_3)}{v_1 v_2 v_3} \\
U_4 &= \log \frac{1-v_1}{v_1} + \log \frac{1-v_2}{v_2} + \log \frac{1-v_3}{v_3} \\
U_4 &= V_1 + V_2 + V_3
\end{aligned}$$

Log Domain-Decoding: Check Node Function

At the check node, we know $x_1 + x_2 + \ldots + x_n = 0$.

Before, we showed that at the check node:



$$v_n = \frac{1}{2}\Big((2u_1 - 1)\cdots(2u_{n-1} - 1) + 1\Big)$$

Using the definitions as before:

$$\begin{aligned} U_i &= \log\frac{1-u_i}{u_i} \quad \Rightarrow \quad u_i = \frac{1}{e^{U_i}+1} \\ 2u_i - 1 &= \frac{e^{U_i/2} - e^{-U_i/2}}{e^{U_i/2} + e^{-U_i/2}} = \tanh(\frac{U_i}{2}), \\ \text{and also } 2v_n - 1 &= \tanh(\frac{V_n}{2}) \quad \Rightarrow \quad V_n = 2\tanh^{-1}\Big(2v_n - 1\Big) \end{aligned}$$

and so,

$$v_n = \frac{1}{2}\left(\tanh\frac{U_1}{2}\cdots\tanh\frac{U_{n-1}}{2} + 1\right)$$

and then,

$$\boxed{V_n = 2\tanh^{-1}\prod_i \tanh\frac{U_i}{2}}$$

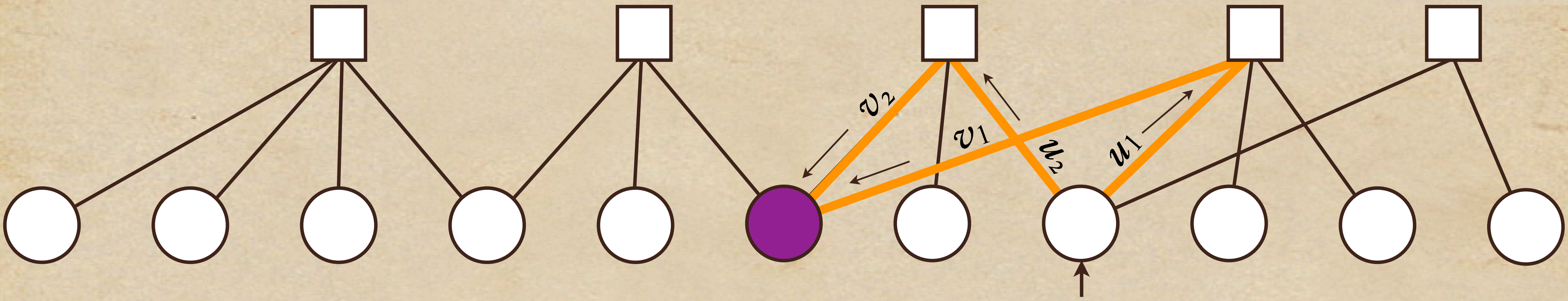
Loops

Loops in the graph are a problem because information comes back through the loop

Loops violate the independence assumption:

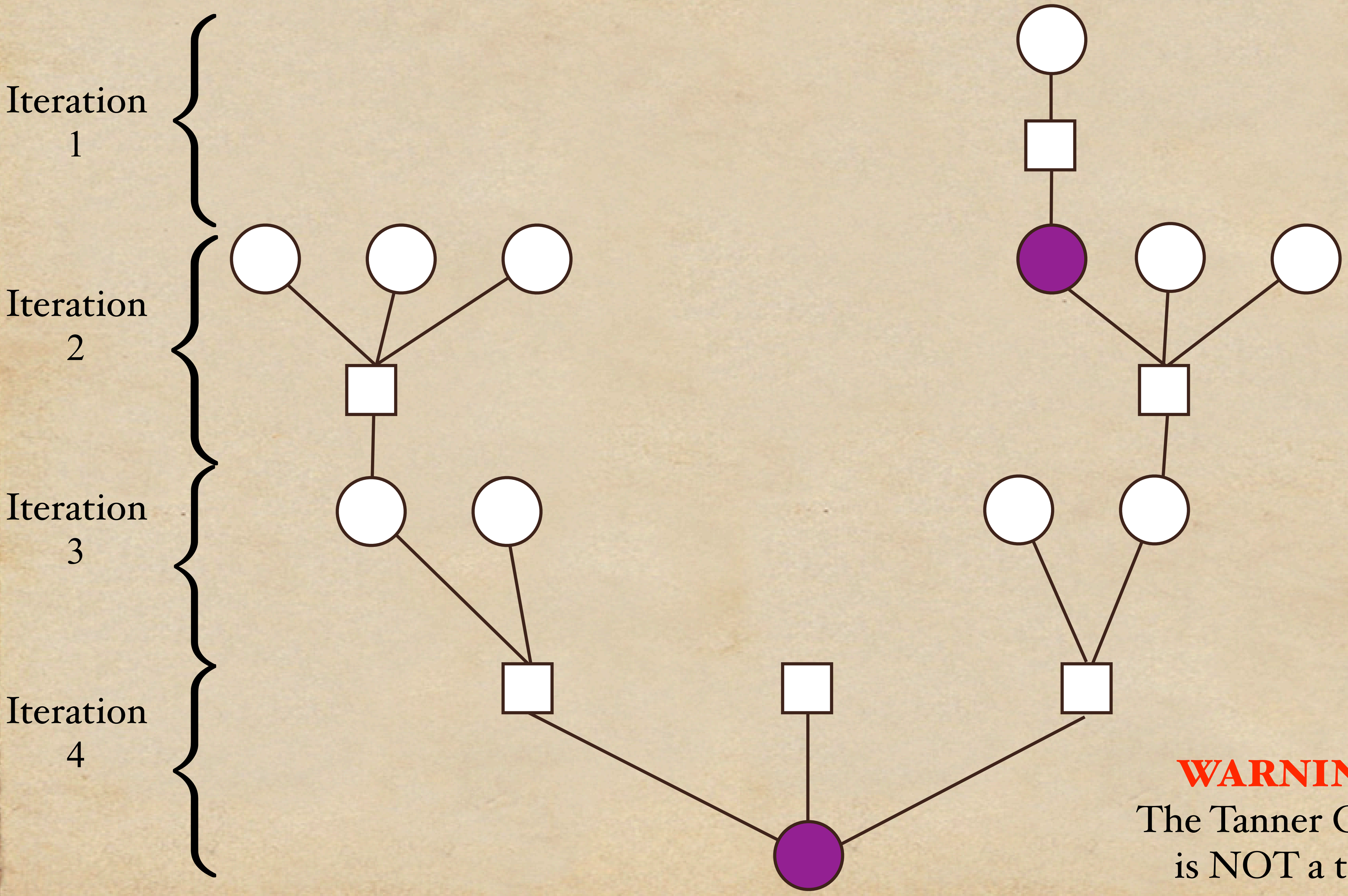
- v_1 was assumed independent of v_2
- Since v_1 depends on u_2, v_2 depends on u_2.
- Clearly, u_1 and u_2 are not independent.

Independence does not hold.



There are of course, codes whose graphs contain no loops. These are not powerful codes, however.

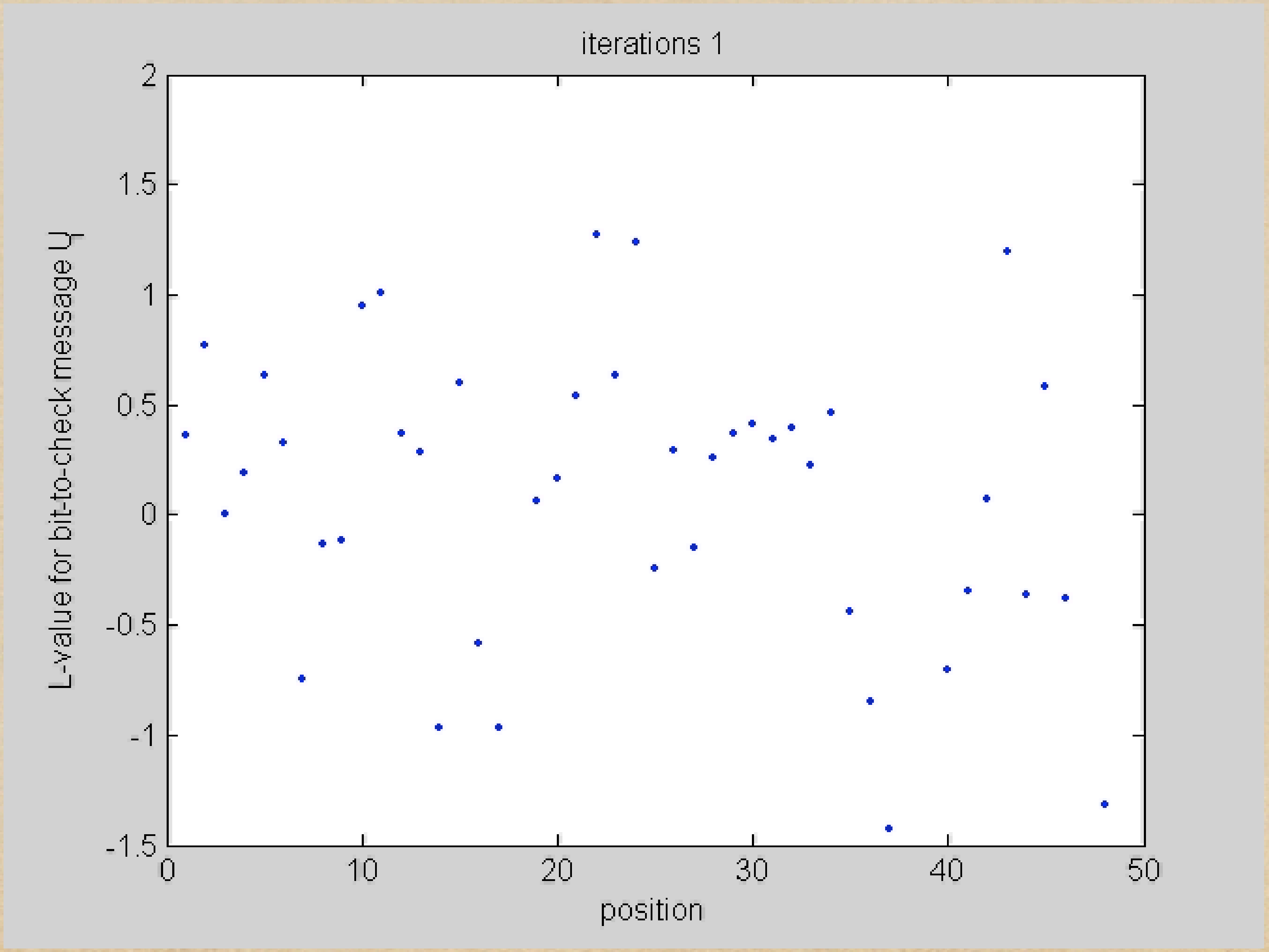
Tree View of Message Passing



WARNING
The Tanner Graph
is NOT a tree

Sample of 48 Bit-to-Check Messages

Consider the actual value of some bit-to-check messages, while decoding a length 16 LDPC code.

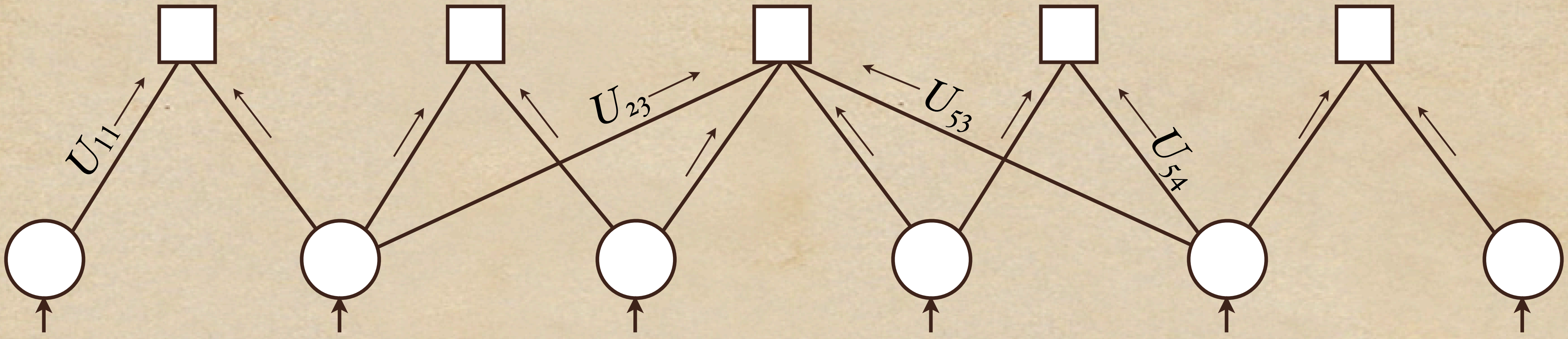


Message Densities

During decoding, the check to bit messages are updated on each iteration.

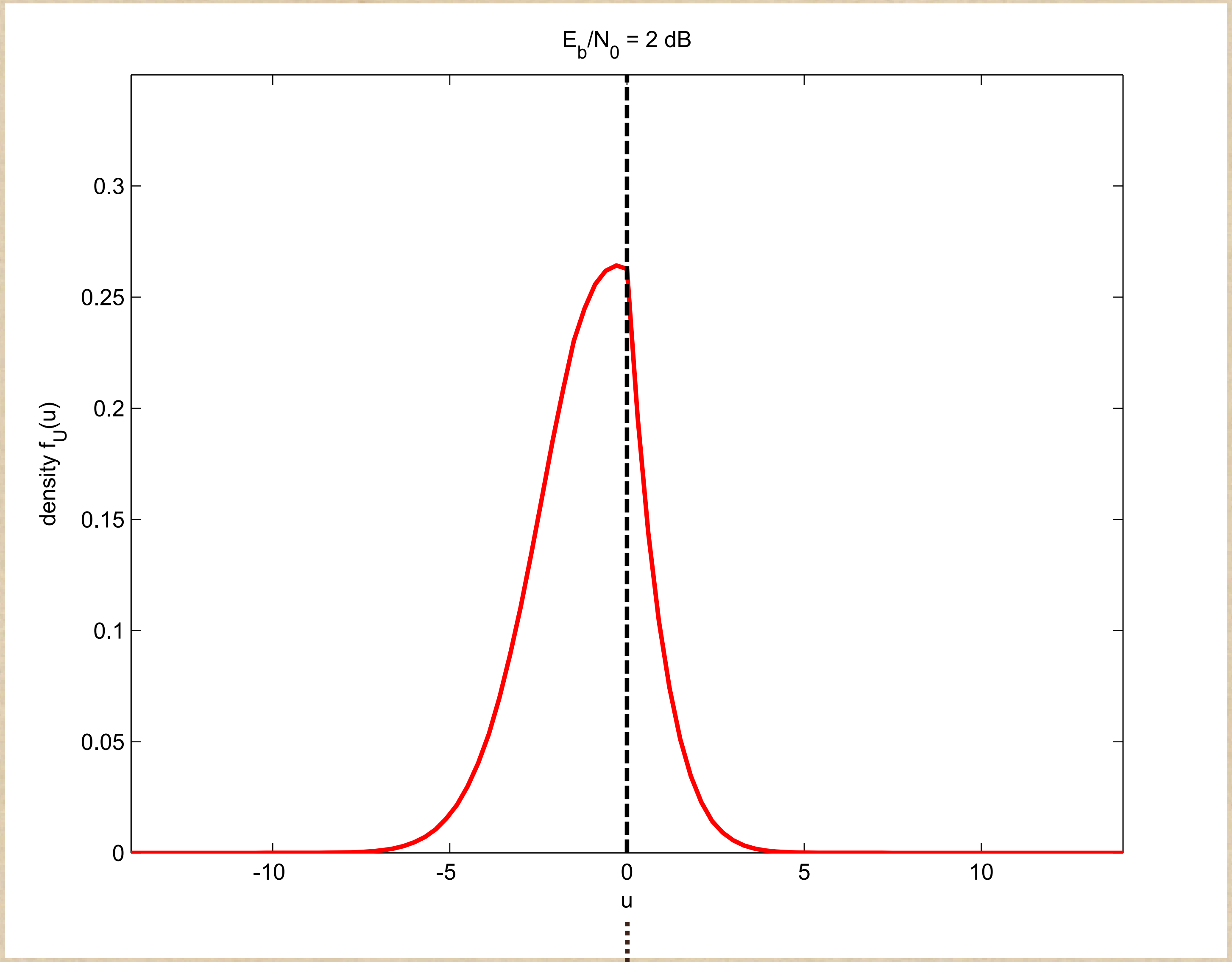
Under the assumption that the messages U_{ij} are independent, then a good question is what the **probability density function** of $U = U_{ij}$:

$$f_U(u) \; ?$$



Message Density $f_U(u)$ on Iteration 1

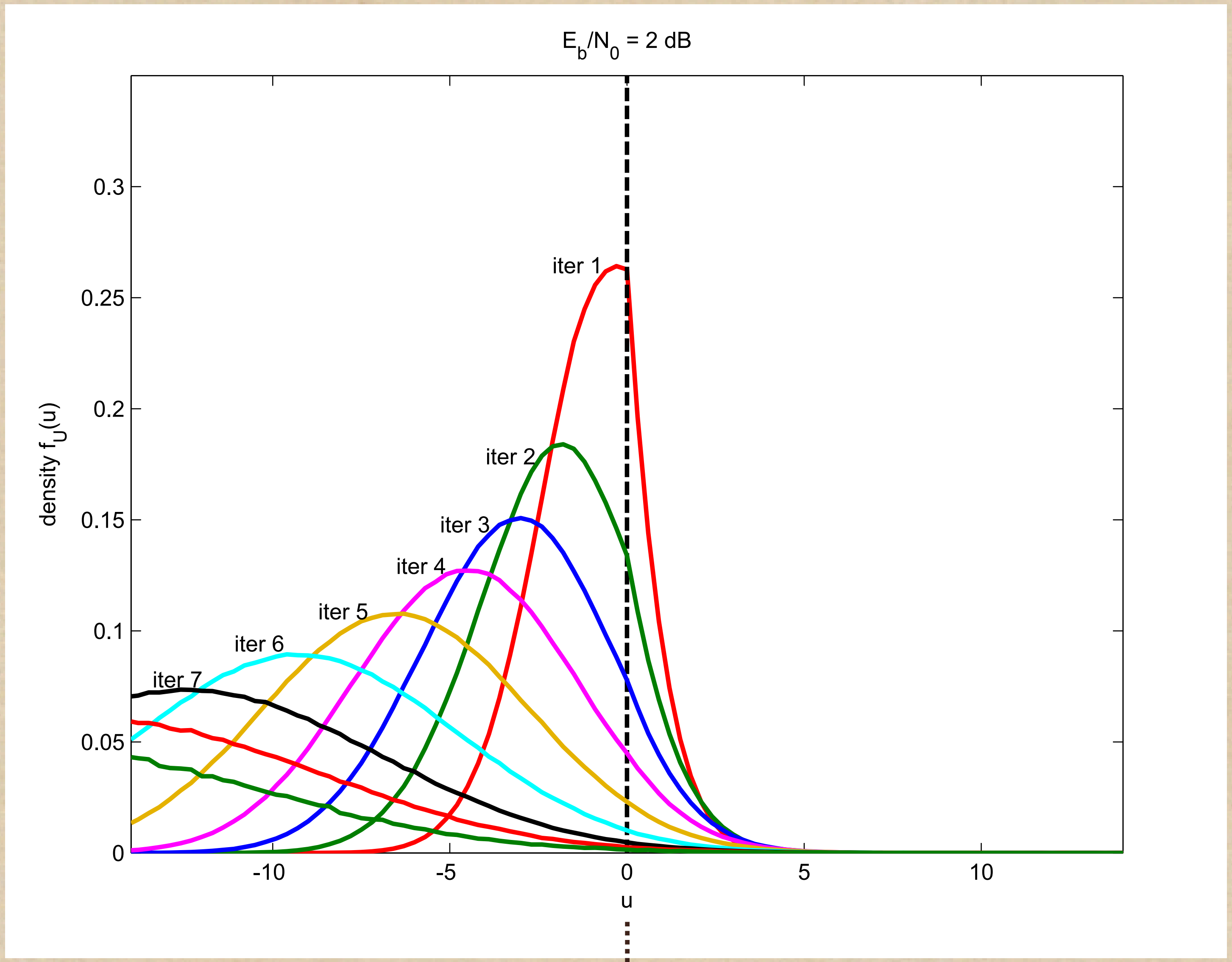
Assume that the all-zeros codeword was transmitted.



$\longrightarrow U > 0 \rightarrow$ error

Message Density $f_U(u)$ on Iteration 1 to 9

Assume that the all-zeros codeword was transmitted.



Successful decoding: U → -∞

Density Evolution: Irregular LDPC Codes

- Gallager's regular codes have a fixed row and column weight:

  Column weight *j*

  Row weight *k*

- Irregular LDPC Codes (Luby, *et al.* 2001)

  Allow both the column weight and row weight to vary, specified by a **degree distribution**:

  Variable degree distribution: $\lambda(x) = \sum_{i=2} \lambda_i x^{i-1}$

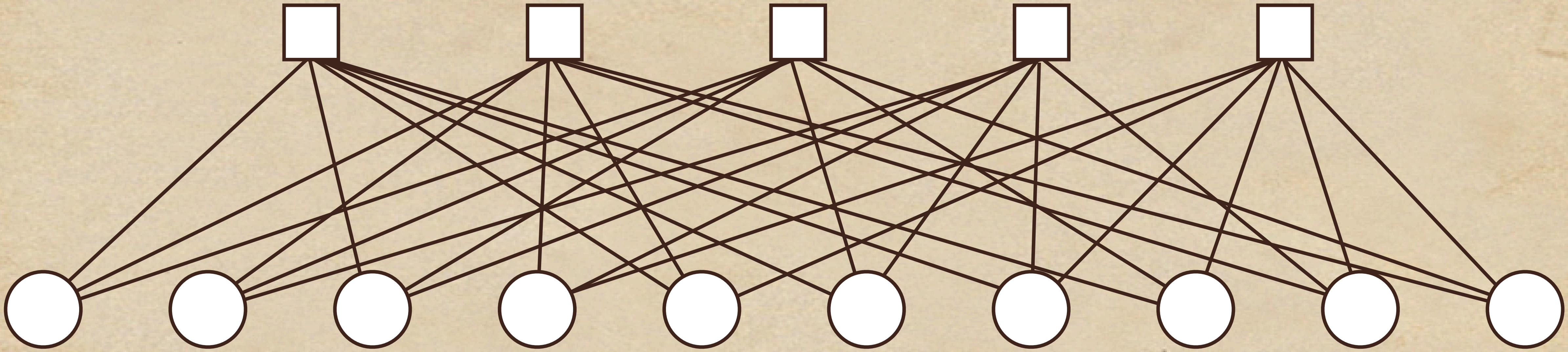
  Check degree distribution: $\rho(x) = \sum_{i=2} \rho_i x^{i-1}$

  where λ_i is the fraction of edges connected to variable nodes with degree *i*, and ρ_i is the fraction of edges connected to check nodes with degree *i*. Further : $\sum_i \lambda_i = 1.$ and $\sum_i \rho_i = 1.$

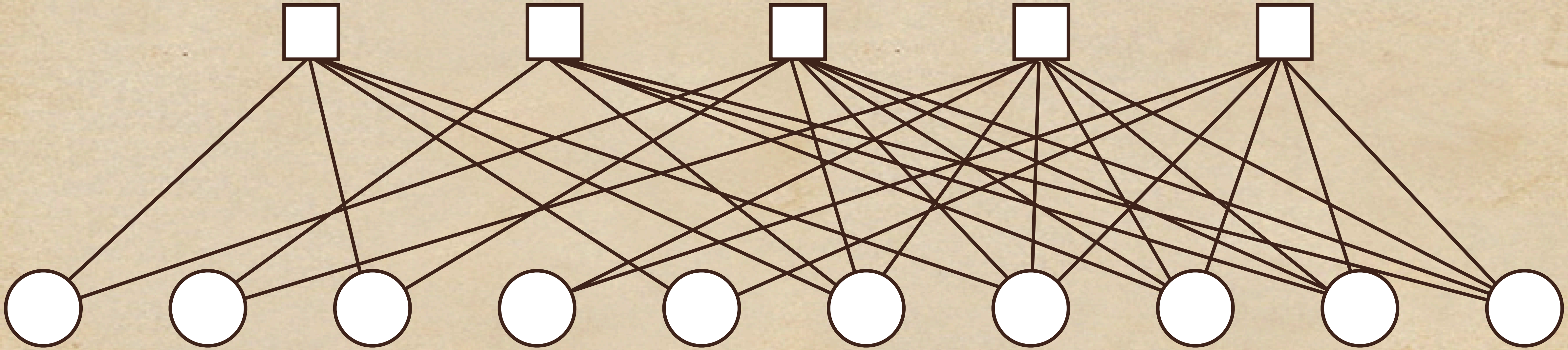
  The **rate** of irregular LDPC Codes is given by:

$$R = 1 - \frac{\int_0^1 \rho(x)dx}{\int_0^1 \lambda(x)dx}$$

Example of Degree Distributions



Regular LDPC Code: $\lambda(x) = 1 \cdot x^2, \quad \rho(x) = 1 \cdot x^5$

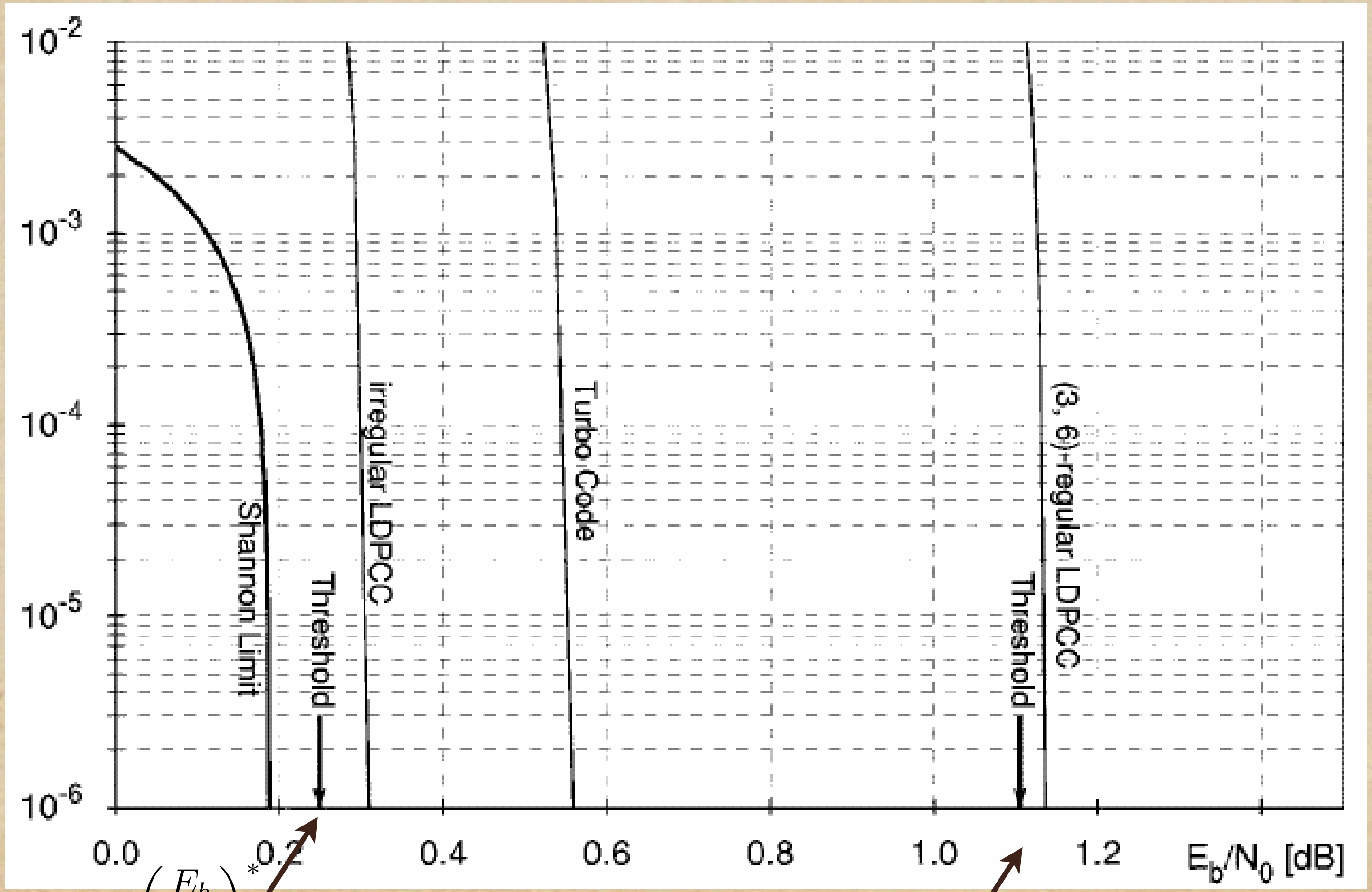


Irregular LDPC Code: $\lambda(x) = \frac{1}{3}x + \frac{2}{3}x^3, \quad \rho(x) = \frac{10}{30}x^4 + \frac{6}{30}x^5 + \frac{14}{30}x^6$

LDPC Code Noise Threshold

A channel has a noise parameter σ . For increasing values of σ, the channel is increasing unreliable. For example, on the AWGN channel, σ^2 is the channel noise variance.

Noise Threshold. The noise threshold σ^* is the channel parameter such that: if $\sigma < \sigma^*$, the probability of bit error for an arbitrarily long LDPC code goes to zero for a large number of iterations. On the other hand, if $\sigma > \sigma^*$, the probability of bit error is strictly non-zero.



$\left(\frac{E_b}{N_0}\right)^*$

$\left(\frac{E_b}{N_0}\right)^*$

$$\left(\frac{E_b}{N_0}\right)^* = \frac{1}{2R(\sigma^*)^2}$$

From Richardson and Urbanke, *Transactions on Information Theory,* Feb. 2001

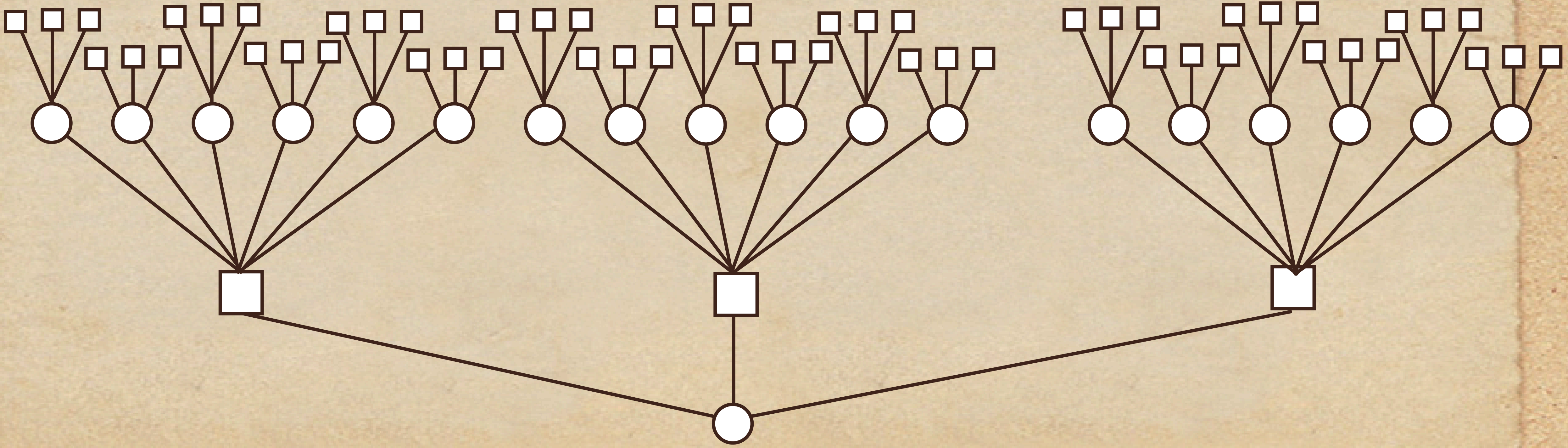
Density Evolution

Density Evolution is a numerical technique that finds the threshold for a given LDPC code degree distribution pair, $\lambda(x), \rho(x)$.

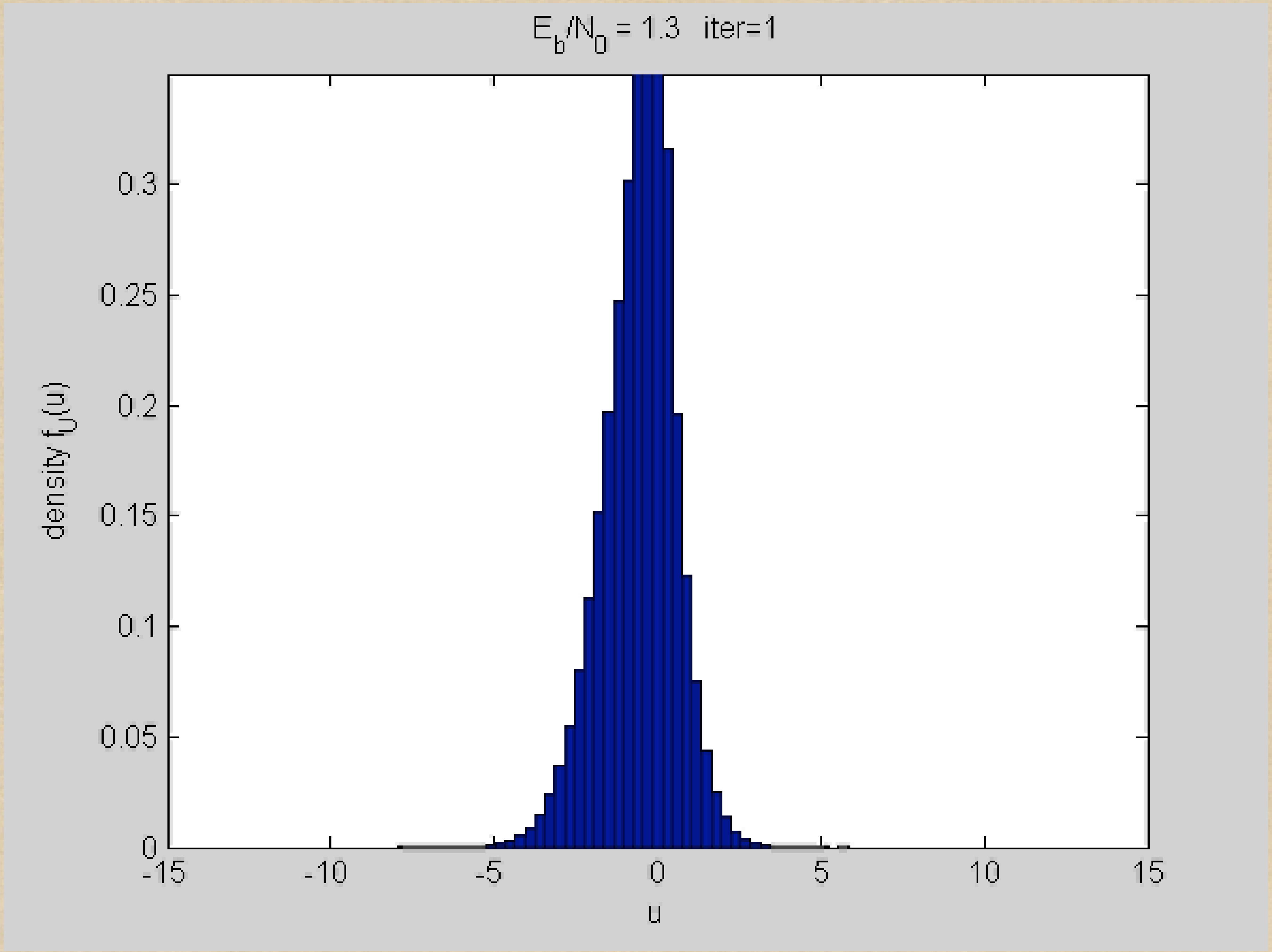
Density evolution quantizes the message densities $f_U(u)$ and $f_V(v)$, and computes densities $f_U(u) \rightarrow f_V(v) \rightarrow f_U(u) \rightarrow f_V(v) \rightarrow$ ... as they evolve from iteration to iteration.

Denisty evolution assumes independence of the messages, and so it can be though of analyzing the decoding algorithm's behavior on a tree-like graph.

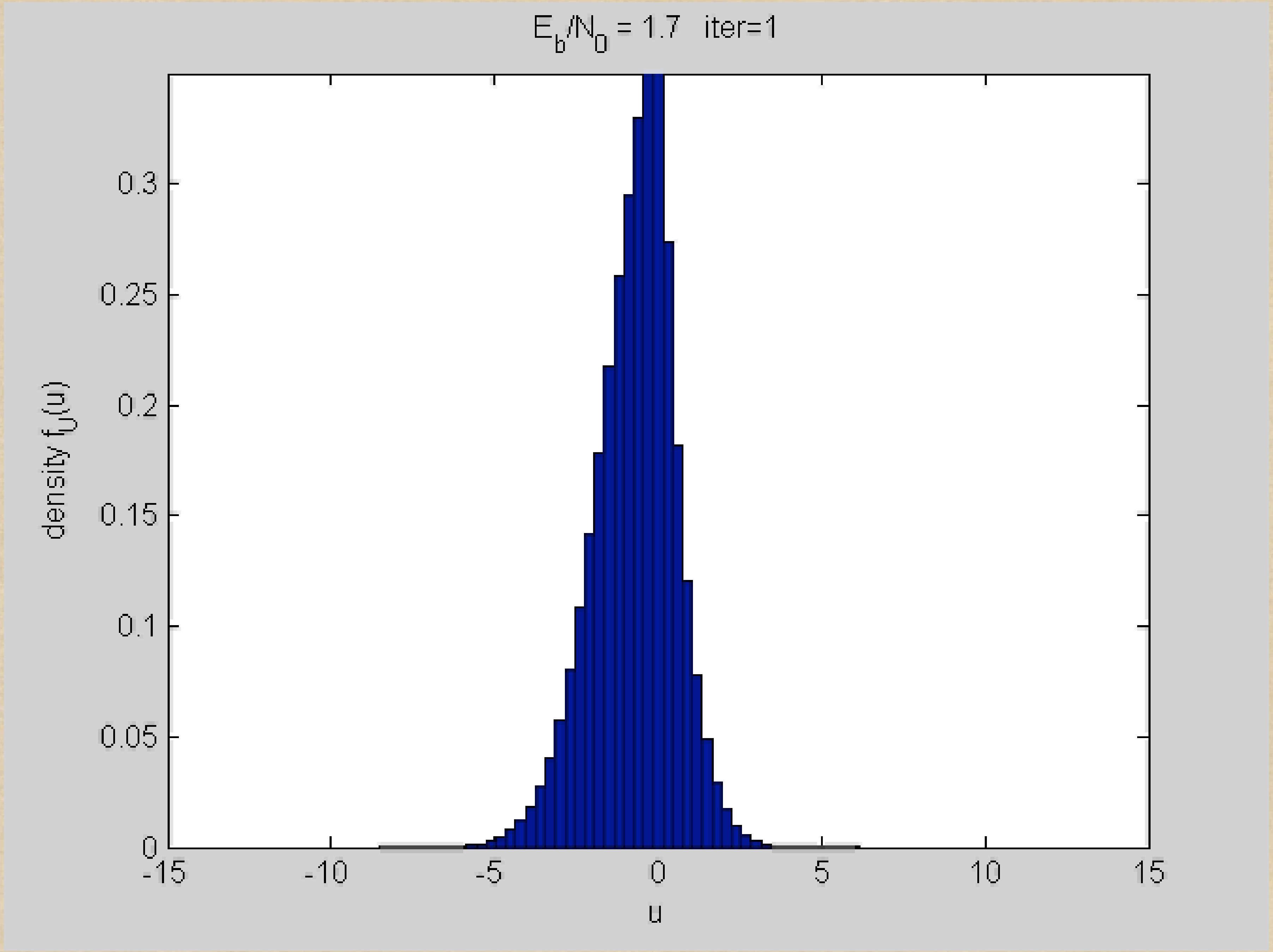
Long LDPC codes can be constructed to have a large girth, so they are effectively free of loops. Thus, density evolution is a good approximation of long LDPC codes.



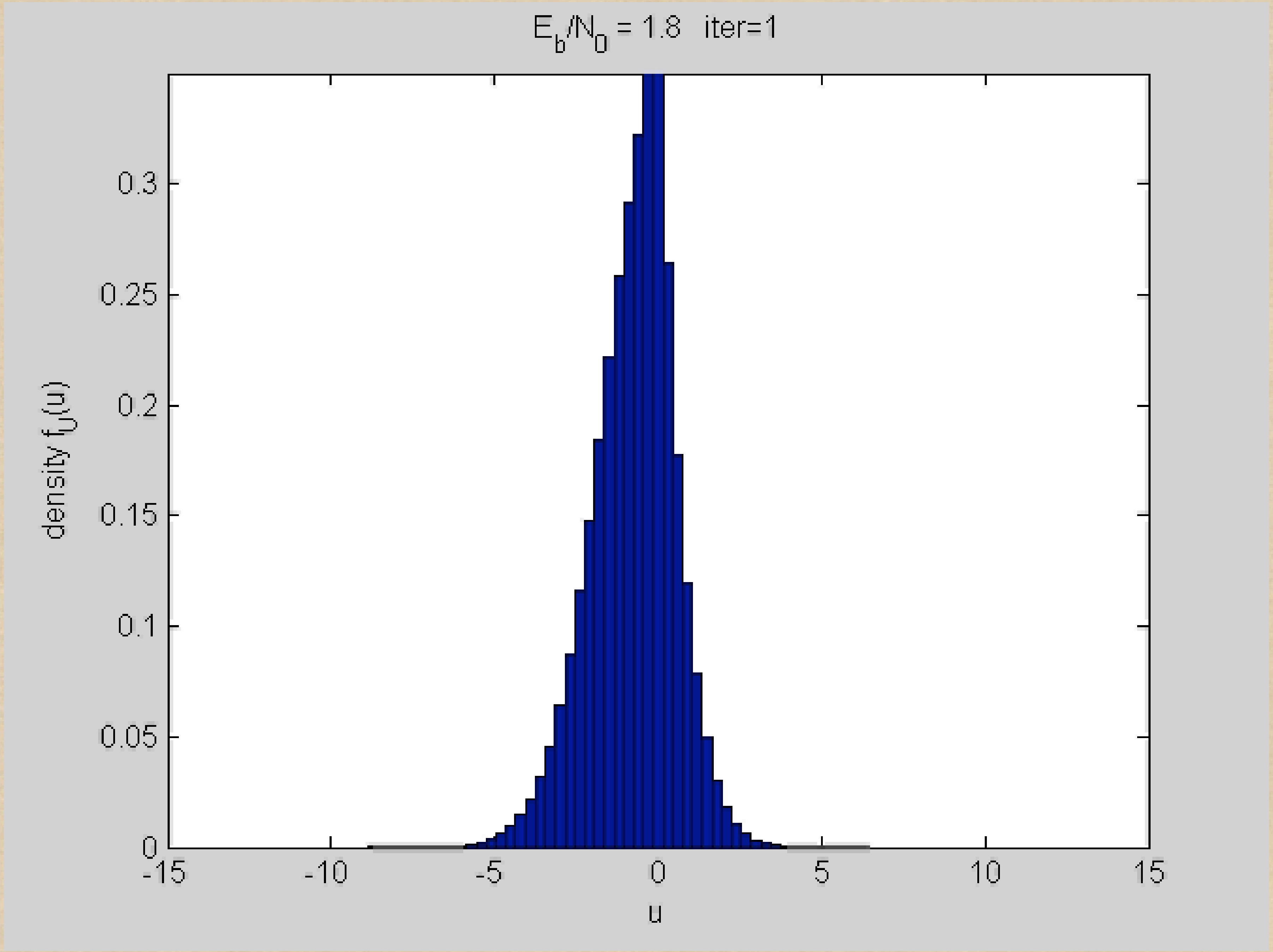
Density for Finite Length Code: $E_b/N_0 = 1.3$ dB



Density for Finite Length Code: $E_b/N_0 = 1.7$ dB



Density for Finite Length Code: $E_b/N_0 = 1.8$ dB



If this was an infinite length code: 1.7 dB < $(E_b/N_0)^*$ < 1.8 dB

Irregular LDPC Degree Optimization

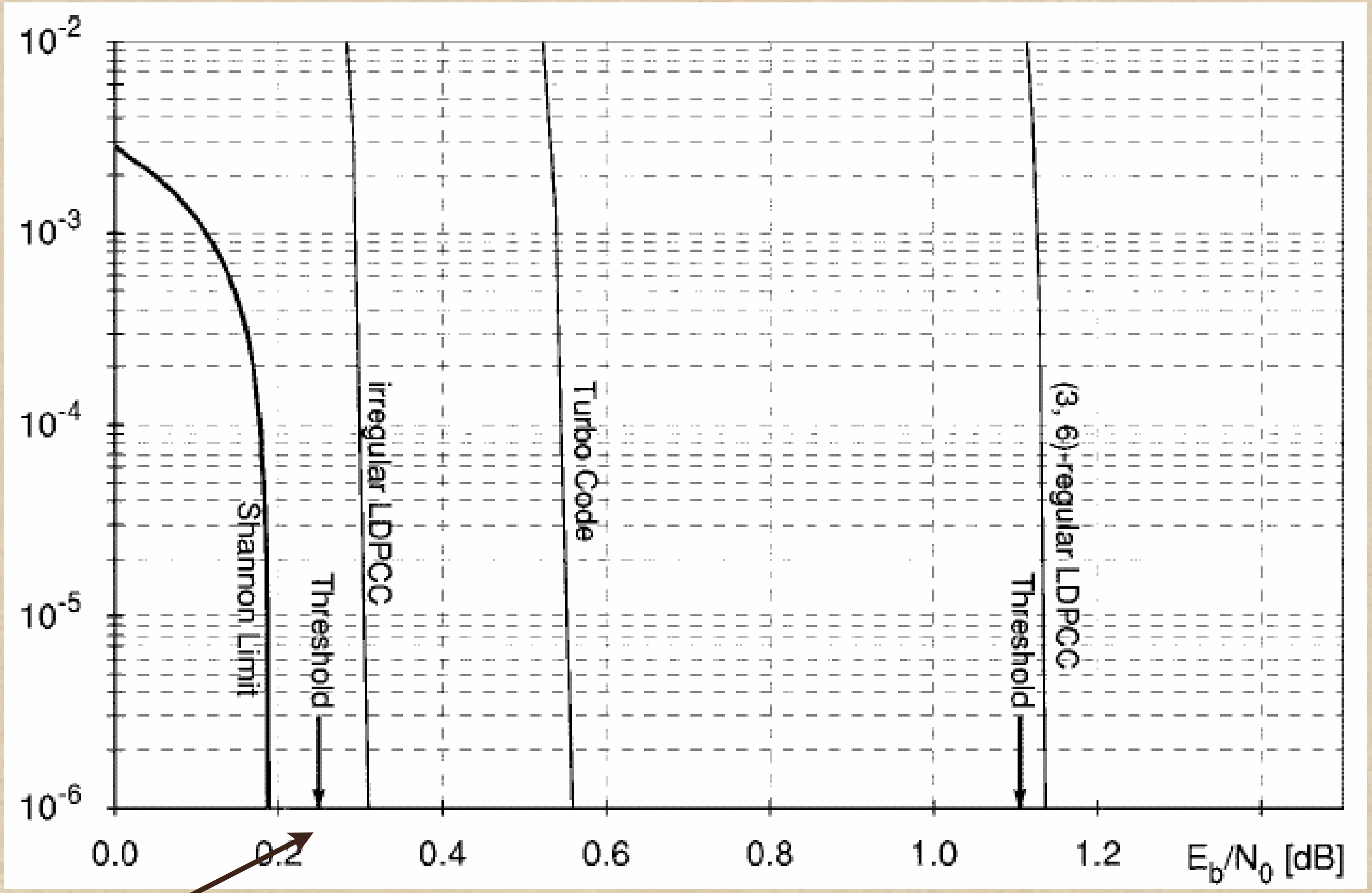
Let $\widetilde{\lambda}(x), \widetilde{\rho}(x)$ be the search algorithm's best degree distribution pair, with threshold $\widetilde{\sigma}^*$.

1. Choose an initial degree distribution pair $\lambda_0(x), \rho_0(x)$, compute the corresponding threshold σ_0^*. Set the best $\widetilde{\lambda}(x) \leftarrow \lambda_0(x), \widetilde{\rho}(x) \leftarrow \rho_0(x), \widetilde{\sigma}^* \leftarrow \sigma_0^*$.
2. By some optimization procedure, select a new degree distribution $\lambda(x), \rho(x)$. Compute the corresponding noise threshold σ^*.
3. If there is an improvement, that is $\sigma^* > \widetilde{\sigma}^*$, then update:

$$\begin{aligned} \widetilde{\lambda}(x) &\leftarrow \lambda(x) \\ \widetilde{\rho}(x) &\leftarrow \rho(x) \\ \widetilde{\sigma}^* &\leftarrow \sigma^* \end{aligned}$$

4. Repeat Steps 2, 3 until some stopping condition is reached.
5. The noise threshold is $\widetilde{\sigma}^*$, with degree distributions $\widetilde{\lambda}(x), \widetilde{\rho}(x)$.

Density Evolution Result: Irregular LDPC Code



$$\begin{aligned}\lambda(x) &= 0.18x^1 + 0.21x^2 + 0.0027x^3 + 0.00009x^7 + 0.15x^7 + .092x^8 \\ &\quad + .028x^9 + 0.012x^{14} + 0.072x^{29} + 0.26x^{49} \\ \rho(x) &= 0.34x^8 + 0.089x^9 + 0.57x^{10}\end{aligned}$$