BP + OSD Algorithm for Quantum Error Correction

Lecture Note based on arXiv:2005.07016

Roffe, White, Burton, and Campbell (2020)

Contents

1 Introduction

1.1 Overview

This lecture note introduces the **BP+OSD decoder** for quantum error correction:

- **BP** = Belief Propagation (a classical decoding algorithm)
- **OSD** = Ordered Statistics Decoding (a post-processing technique)

Together, BP+OSD provides a general-purpose decoder for **quantum LDPC codes** (Low-Density Parity Check codes).

1.2 Learning Objectives

By the end of this note, you will understand:

1. How classical error correction codes work
2. The Belief Propagation algorithm for decoding
3. Why BP fails for quantum codes (the degeneracy problem)
4. How OSD fixes the degeneracy problem
5. The complete BP+OSD decoding algorithm

2 Classical Error Correction

2.1 Linear Codes

All arithmetic in this note is performed in **binary** (modulo 2):

- $0 + 0 = 0, \quad 1 + 0 = 0 + 1 = 1, \quad 1 + 1 = 0$
- This is also written as XOR: $a \oplus b = (a + b) \bmod 2$
- Vectors and matrices use element-wise mod-2 arithmetic

2.2 Hamming Weight and Distance

> **Definition.** The **Hamming weight** of a binary vector $\boldsymbol{v}$ is the number of 1s it contains $|\boldsymbol{v}| = \sum_i v_i$. The **Hamming distance** between two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ is the number of positions where they differ:$d(\boldsymbol{u}, \boldsymbol{v}) = |\boldsymbol{u} + \boldsymbol{v}|$.

For example, for $\boldsymbol{v} = (1, 0, 1, 1, 0)$: Hamming weight $|\boldsymbol{v}| = 3$ and for $\boldsymbol{u} = (1, 1, 0, 1, 0)$ and $\boldsymbol{v} = (1, 0, 1, 1, 0)$: $\boldsymbol{u} + \boldsymbol{v} = (0, 1, 1, 0, 0)$ and $d(\boldsymbol{u}, \boldsymbol{v}) = 2$.

> **Definition.** An $[\boldsymbol{n}, \boldsymbol{k}, \boldsymbol{d}]$ **linear code** $\mathcal{C}$ is a set of binary vectors (called **codewords**) where n is the **block length** (number of bits in each codeword), k is the **dimension** (number of information bits encoded), and d is the **minimum distance** (minimum Hamming weight among non-zero codewords). The **rate** of the code is $R = k/n$.

A linear code can be defined by an $m \times n$ **parity check matrix** H. H_{ij} denotes the entry in row i, column j of matrix H. m is the number of rows in H (number of parity checks), n is the number of columns in H (number of bits), and $\text{rank}(H)$ is the number of linearly independent rows. By the rank-nullity theorem: $k = n - \text{rank}(H)$.

For example, The $[\mathbf{3, 1, 3}]$ **repetition code** encodes 1 bit into 3 bits by triplication.

Parity check matrix:

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

Verification: $H \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ ✓ and $H \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ ✓

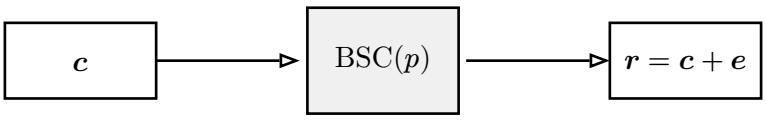
So the codewords are: $\mathcal{C}_H = \{(0, 0, 0), (1, 1, 1)\}$

Parameters: $n = 3$ bits, $k = 3 - 2 = 1$ info bit, $d = 3$ (weight of $(1, 1, 1)$)

2.3 Error Model and Syndrome

Definition. In the **binary symmetric channel** (BSC) with error probability p:

- Each bit is independently flipped with probability p
- Original codeword: $\boldsymbol{c}$
- Error pattern: $\boldsymbol{e}$ (a binary vector, $e_i = 1$ means bit i flipped)
- Received word: $\boldsymbol{r} = \boldsymbol{c} + \boldsymbol{e}$



$\boldsymbol{e}$: random error

Figure 1: Binary symmetric channel model

Definition. The **syndrome** of a received word $\boldsymbol{r}$ is:

$$\boldsymbol{s} = H \cdot \boldsymbol{r} \tag{2}$$

Since $H \cdot \boldsymbol{c} = \boldsymbol{0}$ for any codeword, we have:

$$\boldsymbol{s} = H \cdot \boldsymbol{r} = H \cdot (\boldsymbol{c} + \boldsymbol{e}) = H \cdot \boldsymbol{c} + H \cdot \boldsymbol{e} = \boldsymbol{0} + H \cdot \boldsymbol{e} = H \cdot \boldsymbol{e} \tag{3}$$

Key Point. The syndrome depends **only on the error**, not on which codeword was sent! This is what makes syndrome-based decoding possible.

2.4 The Decoding Problem

Given: Parity check matrix H and syndrome $\boldsymbol{s} = H \cdot \boldsymbol{e}$

Find: The most likely error $\boldsymbol{e}^*$ that could have produced $\boldsymbol{s}$

Definition. Maximum likelihood decoding finds:

$$\boldsymbol{e}^* = \arg\min_{\boldsymbol{e}: H \cdot \boldsymbol{e} = \boldsymbol{s}} |\boldsymbol{e}| \tag{4}$$

That is, the minimum Hamming weight error consistent with the syndrome.

2.5 Probabilistic Graphical Models

Before introducing the Belief Propagation algorithm, we need to understand how probabilistic inference problems can be represented as graphs.

> **Definition.** A **probabilistic graphical model** (PGM) is a graph-based representation of a probability distribution. Nodes represent random variables, and edges encode conditional independence relationships. PGMs enable efficient inference algorithms by exploiting the structure of the distribution.

There are two main families of PGMs:

- **Directed graphical models** (Bayesian networks): edges have direction, representing causal relationships
- **Undirected graphical models** (Markov networks): edges are undirected, representing symmetric dependencies

For error correction, we use undirected models because parity constraints are symmetric — no variable "causes" another.

2.6 Undirected Probabilistic Graphical Models

> **Definition.** An **undirected probabilistic graphical model** (also called a **Markov network** or **Markov random field**) represents a joint probability distribution as:
>
> $$P(\boldsymbol{x}) = \frac{1}{Z} \prod_{c \in \mathcal{C}} \psi_c(\boldsymbol{x}_c) \tag{5}$$
>
> where:
>
> - $\boldsymbol{x} = (x_1, ..., x_n)$ are random variables
> - $\mathcal{C}$ is a set of **cliques** (fully connected subgraphs)
> - $\psi_c(\boldsymbol{x}_c)$ is a **potential function** over variables in clique c
> - $Z = \sum_{\boldsymbol{x}} \prod_c \psi_c(\boldsymbol{x}_c)$ is the **partition function** (normalization constant)

> **Key Point.** The UAI format mentioned in the getting started guide represents exactly this structure: variables (detectors), cliques (error mechanisms), and potential functions (error probabilities).

For binary error correction with syndrome $\boldsymbol{s}$, we want to compute:

$$P(\boldsymbol{e} \mid \boldsymbol{s}) \propto \prod_c \psi_c(\boldsymbol{e}_c) \tag{6}$$

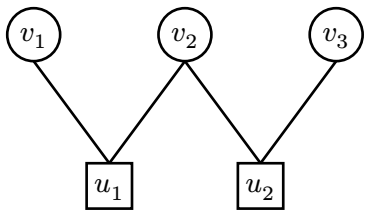
where each potential ψ_c encodes a parity constraint.

2.7 Factor Graphs

To understand the Belief Propagation algorithm, we need the concept of **factor graphs**.

> **Definition.** A **factor graph** is a bipartite graph $G = (V, U, E)$ representing the parity check matrix H. The **data nodes** are set $V = \{v_1, v_2, ..., v_n\}$ such that each node v_j corresponds to each column of H. A **parity nodes** are set $U = \{u_1, u_2, ..., u_m\}$ such that

each node u_i corresponds to each row of H. An **edges** $E = \{(v_j, u_i) : H_{ij} = 1\}$ connects v_j to u_i exists if $H_{ij} = 1$.



Factor graph for $H = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$

Figure 2: Factor graph for the $[3, 1, 3]$ repetition code with node conventions

The **neighborhoods** of nodes are defined as: $V(u_i) = \{v_j : H_{ij} = 1\}$.

2.8 Comparing Graph Representations

Three related graph representations appear in coding theory and probabilistic inference:

Definition. Comparison of graph representations:

1. **Undirected Probabilistic Graphical Model (Markov Network)**:
   - Nodes = random variables
   - Edges = direct dependencies between variables
   - Cliques = groups of mutually dependent variables
   - Represents: $P(\boldsymbol{x}) = \frac{1}{Z} \prod_c \psi_c(\boldsymbol{x}_c)$
2. **Factor Graph**:
   - Two types of nodes: variable nodes AND factor nodes
   - Bipartite structure: edges only between variables and factors
   - Explicitly represents factorization of the distribution
   - Represents: $P(\boldsymbol{x}) = \frac{1}{Z} \prod_i f_i\left(\boldsymbol{x}_{N(i)}\right)$
3. **Tanner Graph**:
   - A special case of factor graph for error correction codes
   - Variable nodes = bits (columns of H)
   - Factor nodes = parity checks (rows of H)
   - Represents: parity check matrix H structure

Key Point. Key relationships:

- Factor graphs are a **bipartite refinement** of Markov networks that make the factorization explicit
- Tanner graphs are factor graphs **specialized for linear codes** where factors represent parity constraints
- All three represent the same probability distribution, but factor graphs enable more efficient message-passing algorithms
- The UAI format represents Markov networks (cliques and potentials), while BP operates on the factor graph representation

Property	Markov Network	Factor Graph	Tanner Graph
Node types	Variables only	Variables + Factors	Bits + Checks
Graph structure	General	Bipartite	Bipartite
Edges represent	Dependencies	Factor membership	Parity constraints
Used for	General inference	Message passing	Code decoding
BP efficiency	Less efficient	Efficient	Efficient

Table 1: Comparison of graph representations

Why use factor graphs for BP? The bipartite structure of factor graphs makes message passing natural:

- Messages flow between variables and factors
- Each factor collects evidence from its variables
- Each variable aggregates information from its factors
- No need to handle complex clique structures

For error correction, the Tanner graph (factor graph) representation is ideal because:

- Parity checks are naturally factors (XOR constraints)
- Bits are naturally variables (error indicators)
- The sparse structure (H has few 1s) gives efficient $O(n)$ BP iterations

Definition. An **(l, q)-LDPC code** is a linear code whose parity check matrix H satisfies:

- Each column has at most l ones (each bit is in at most l checks)
- Each row has at most q ones (each check involves at most q bits)

The matrix H is called **sparse** because l and q are small constants independent of n.

Key Point. LDPC codes are important because their sparse structure enables efficient decoding via Belief Propagation with complexity $O(n)$ per iteration.

3 Belief Propagation Decoder

3.1 Introduction and Motivation

The rediscovery of Low-Density Parity-Check (LDPC) codes in the late 1990s marked a paradigm shift in coding theory, transitioning from algebraic decoding algorithms to probabilistic iterative decoding that approaches the Shannon limit [1]. Central to this revolution is the **Belief Propagation** (BP) algorithm [2], a message-passing protocol that operates on the graphical representation of codes.

Key Point. BP in Modern Communications: BP decoding powers critical communication standards:

- Wi-Fi (IEEE 802.11n/ac/ax)
- Satellite communication (DVB-S2)
- 5G New Radio

While its practical efficacy is undisputed, the mathematical rigor underlying its convergence behavior involves multiple theoretical frameworks: Density Evolution for asymptotic analysis, Bethe Free Energy for variational optimization, and trapping set theory for failure mechanisms.

The convergence of BP is understood through different lenses depending on the regime. In the asymptotic limit of infinite block length, convergence is probabilistic and governed by Density Evolution [3]. In finite-length regimes, convergence is variational, linked to minimization of the Bethe Free Energy [4]. However, combinatorial substructures known as trapping sets can arrest decoding, creating error floors [5].

3.2 The Message Passing Mechanism

Belief Propagation (BP), also called the **sum-product algorithm**, is an iterative message-passing algorithm on the factor graph.

Definition. The goal of BP is to compute, for each bit j, the **marginal probability**:

$$P_1(e_j) = P(e_j = 1 \mid \boldsymbol{s}), \tag{7}$$

given $\boldsymbol{s} = H \cdot \boldsymbol{e}$ is the syndrome of the error $\boldsymbol{e}$. This is called a **soft decision** – it tells us how likely each bit is to be flipped.

We use the following notation throughout:

- p as the channel error probability (probability each bit flips)
- $m_{v_j \to u_i}$ as the message from data node v_j to parity node u_i
- $m_{u_i \to v_j}$ as the message from parity node u_i to data node v_j
- Messages represent beliefs about whether $e_j = 1$

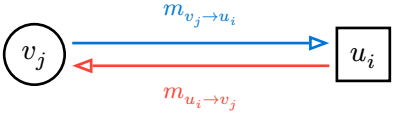


Figure 3: Messages passed between data and parity nodes

The BP-algorithm requires the quantification of how certain we are about the bit being flipped or not. This is done using **log-likelihood ratios** (LLR).

Definition. Instead of probabilities, BP uses **log-likelihood ratios** (LLR) for numerical stability:

$$\mathrm{LLR}(e_j) = \log \frac{P(e_j = 0)}{P(e_j = 1)} \tag{8}$$

- $\mathrm{LLR} > 0$ means $e_j = 0$ is more likely (bit probably correct)
- $\mathrm{LLR} < 0$ means $e_j = 1$ is more likely (bit probably flipped)
- $|\mathrm{LLR}|$ indicates confidence level

For the channel with error probability p, the **channel LLR** is:

$$p_l = \log \frac{1-p}{p} \tag{9}$$

Since $p < 0.5$ in practice, we have $p_l > 0$.

3.3 BP Algorithm: Step-by-Step

Step 1: Initialization

Set the channel LLR:

$$p_l = \log \frac{1-p}{p} \tag{10}$$

Initialize all messages from data nodes to parity nodes with the channel prior:

$$m_{v_j \to u_i} := p_l \quad \text{for all edges } (v_j, u_i) \tag{11}$$

Why? Before any message passing, the only information we have about each bit is from the **channel itself**. Since each bit flips independently with probability p, the initial belief is simply the channel's prior: "this bit is probably correct" (because $p < 0.5$, so $p_l > 0$).

Code: The implementation adds small epsilon values for numerical stability:

```
# Initialize channel LLRs
channel_llr = torch.log((1 - channel_probs + 1e-10) / (channel_probs + 1e-10))

# Initialize qubit-to-check messages with channel prior
msg_q2c = channel_llr[qubit_edges].unsqueeze(0).expand(batch_size, -1).clone()
msg_c2q = torch.zeros(batch_size, num_edges, device=device)
```

Step 2: Parity-to-Data Messages

Each parity node u_i sends a message to each connected data node v_j:

- **Min-sum form:**

$$m_{u_i \to v_j} = (-1)^{s_i} \cdot \alpha \cdot \prod_{v'_j \in V(u_i) \setminus v_j} \operatorname{sign}\left(m_{v'_j \to u_i}\right) \cdot \min_{v'_j \in V(u_i) \setminus v_j} |m_{v'_j \to u_i}| \tag{12}$$

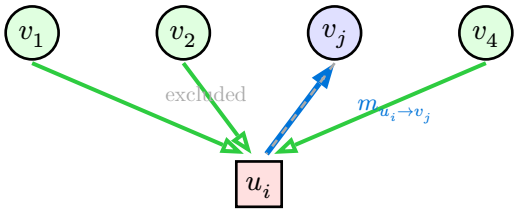
- **Sum-Product form:**

$$m_{u_i \to v_j} = (-1)^{s_i} \cdot 2 \tanh^{-1}\left(\prod_{v'_j \in V(u_i) \setminus v_j} \tanh\left(\frac{m_{v'_j \to u_i}}{2}\right)\right) \tag{13}$$

Where:

- s_i = the i-th syndrome bit (given as input, either 0 or 1)
- $V(u_i) \setminus v_j$ = all neighbors of u_i except v_j (defined in Section 3.5)
- $\operatorname{sign}(x) = +1$ if $x \geq 0$, else -1
- $\alpha = 1 - 2^{-t}$ is a **damping factor** at iteration t (helps convergence)

Why? A parity check enforces that XOR of all connected bits equals the syndrome bit s_i. The check node tells v_j: "Based on what I know about the **other** bits, here's how likely **you** are to be flipped." If $s_i = 0$, the parity check says "even number of flipped bits." If the other bits all look correct (positive LLR), then v_j should also be correct. If one other bit looks flipped (negative LLR), then v_j should be correct to maintain even parity. The formula computes this XOR-like logic in LLR form.



Check node collects info from v_1, v_2, v_4
to compute message to v_j (excluding v_j's own message)

Figure 4: Parity-to-data message: u_i uses info from all neighbors **except** v_j to tell v_j what it should be

Code: The implementation computes signs and magnitudes separately, using a sorting trick to efficiently find the minimum excluding each edge:

```
def _check_to_qubit_minsum(self, msg_q2c, syndromes):
    for c in range(num_checks):
        edges = self.check_to_edges[c]
        incoming = msg_q2c[:, edges]  # (batch, degree)

        # Separate signs and magnitudes
        signs = torch.sign(incoming)  # (batch, degree)
        mags = torch.abs(incoming)    # (batch, degree)

        # Product of all signs
        total_sign = torch.prod(signs, dim=1, keepdim=True)
```

```
        # Apply syndrome: flip sign if syndrome is 1
        syndrome_sign = 1 - 2 * syndromes[:, c:c+1]
        total_sign = total_sign * syndrome_sign

        # For each edge, divide out its sign to get product of others
        outgoing_signs = total_sign / (signs + 1e-10)

        # Min magnitude excluding each edge (second minimum trick)
        sorted_mags, _ = torch.sort(mags, dim=1)
        min_mag = sorted_mags[:, 0:1]
        second_min = sorted_mags[:, 1:2] if sorted_mags.shape[1] > 1 else min_mag

        # If edge has the min, use second_min; else use min
        is_min = (mags == min_mag)
        outgoing_mags = torch.where(is_min, second_min, min_mag)

        # Apply scaling factor
        scaling = 0.625
        msg_c2q[:, edges] = scaling * outgoing_signs * outgoing_mags
```

Key Point. Second Minimum Trick: To compute $\min_{v'_j \neq v_j} |m_{v'_j}|$ efficiently, we sort all magnitudes once. For each edge: if it holds the minimum, use the second minimum; otherwise use the first minimum. This avoids recomputing $O(d)$ minimums for each of d edges.

Sum-Product Code: The sum-product variant uses the tanh identity for exact computation:

```
def _check_to_qubit_sumproduct(self, msg_q2c, syndromes):
    for c in range(num_checks):
        edges = self.check_to_edges[c]
        incoming = msg_q2c[:, edges]  # (batch, degree)

        # Compute tanh(LLR/2) - clamp for numerical stability
        half_llr = torch.clamp(incoming / 2, min=-20, max=20)
        tanh_vals = torch.tanh(half_llr)  # (batch, degree)

        # Product of all tanh values
        total_prod = torch.prod(tanh_vals, dim=1, keepdim=True)

        # Apply syndrome: flip sign if syndrome is 1
        syndrome_sign = 1 - 2 * syndromes[:, c:c+1]
        total_prod = total_prod * syndrome_sign

        # For each edge, divide out its tanh contribution
        outgoing_prod = total_prod / (tanh_vals + 1e-10)

        # Clamp to valid range for atanh (-1, 1)
        outgoing_prod = torch.clamp(outgoing_prod, min=-1+1e-7, max=1-1e-7)

        # Convert back: 2 * atanh(prod)
        msg_c2q[:, edges] = 2 * torch.atanh(outgoing_prod)
```

Key Point. Min-Sum vs Sum-Product: Min-sum is an **approximation** of sum-product, not an equivalent algorithm. The relationship comes from the identity:

$$2\tanh^{-1}\left(\prod_i \tanh(x_i/2)\right) \approx \text{sign}\left(\prod_i x_i\right) \cdot \min_i |x_i| \tag{14}$$

This approximation holds because $\tanh(x/2) \approx \text{sign}(x)$ for large $|x|$, and the product of tanh values is dominated by the smallest magnitude input. The scaling factor $\alpha = 0.625$ in min-sum compensates for the systematic overestimation of confidence that results from this approximation [6].

Algorithm	**Formula**	**Trade-off**
Sum-Product	$2\tanh^{-1}(\prod \tanh(x/2))$	Exact but slower (tanh/atanh)
Min-Sum	$\alpha \cdot \text{sign}(\prod x) \cdot \min\|x\|$	Approximate but faster

Table 2: Comparison of check-to-qubit message algorithms

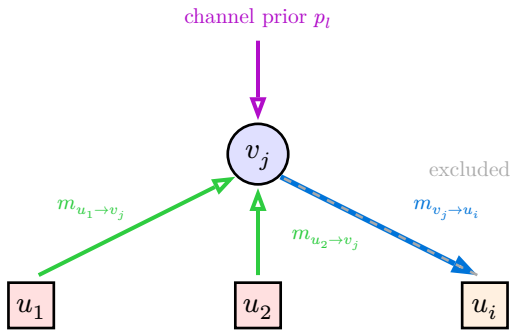
Step 3: Data-to-Parity Messages

Each data node v_j sends a message to each connected parity node u_i:

$$m_{v_j \to u_i} = p_l + \sum_{u_i' \in U(v_j) \setminus u_i} m_{u_i' \to v_j} \tag{15}$$

Where $U(v_j) \setminus u_i$ = all parity neighbors of v_j except u_i.

Why? A data node collects evidence from multiple parity checks. Each check provides independent information about whether this bit is flipped. The data node sums up all this evidence (in LLR, multiplication of probabilities becomes addition).



Data node sums: channel prior + messages from u_1, u_2 to send to u_i (excluding u_i's own message)

Figure 5: Data-to-parity message: v_j combines channel prior with info from other checks

Intuition: Why exclude u_i? To avoid **echo effects**. If v_j included the message it previously received from u_i, that information would bounce back, creating a feedback loop. On a **tree-structured graph**, this exclusion ensures each piece of evidence is counted exactly once, making BP exact. On graphs with cycles, this is an approximation.

Code: The implementation efficiently computes this by computing the total sum and subtracting each edge's contribution:

```
def _qubit_to_check(self, msg_c2q, channel_llr):
    for q in range(num_qubits):
        edges = self.qubit_to_edges[q]
        incoming = msg_c2q[:, edges]  # (batch, degree)

        # Sum of all incoming messages plus channel prior
        total_sum = incoming.sum(dim=1, keepdim=True) + channel_llr[q]

        # For each edge, subtract its own contribution
        msg_q2c[:, edges] = total_sum - incoming
```

Key Point. Why subtract? Instead of computing $n-1$ sums for each of n edges (expensive), we compute one total sum and subtract each term. This reduces complexity from $O(d^2)$ to $O(d)$ per qubit.

Step 4: Compute Soft Decisions

For each bit j, compute the total belief (sum of all evidence):

$$P_1(e_j) = p_l + \sum_{u_i \in U(v_j)} m_{u_i \to v_j} \tag{16}$$

Why? Unlike Step 3, here we include **all** incoming messages (no exclusion). This is the final belief about bit j, combining the channel prior with evidence from **every** connected parity check. The result is the log-posterior probability ratio.

Code: The implementation sums all incoming messages (no exclusion) and converts to probability:

```
def _compute_marginals(self, msg_c2q, channel_llr):
    for q in range(num_qubits):
        edges = self.qubit_to_edges[q]

        # Total LLR = channel + sum of ALL incoming
        total_llr = channel_llr[q] + msg_c2q[:, edges].sum(dim=1)

        # Convert LLR to probability: P(1) = sigmoid(-LLR)
        marginals[:, q] = torch.sigmoid(-total_llr)
```

Key Point. Sigmoid conversion: The relationship between LLR and probability is:

$$\text{LLR} = \log \frac{P(0)}{P(1)} \Rightarrow P(1) = \frac{1}{1+e^{\text{LLR}}} = \sigma(-\text{LLR}) \tag{17}$$

PyTorch's `torch.sigmoid(-total_llr)` computes this efficiently.

Step 5: Make Hard Decisions

Convert soft decisions to a binary estimate:

$$e_j^{\text{BP}} = \begin{cases} 1 \text{ if } P_1(e_j) < 0 & \text{(more likely flipped)} \\ 0 \text{ otherwise} & \text{(more likely correct)} \end{cases} \tag{18}$$

This gives us the BP estimate $\boldsymbol{e}^{\text{BP}} = (e_1^{\text{BP}}, e_2^{\text{BP}}, ..., e_n^{\text{BP}})$.

Why? The sign of LLR directly tells us the most likely value: $P_1 > 0$ means $P(e_j = 0) > P(e_j = 1)$, so the bit is probably correct. $P_1 < 0$ means the bit is probably flipped.

Step 6: Check Convergence

Verify if the estimate satisfies the syndrome equation:

$$H \cdot \boldsymbol{e}^{\text{BP}} = \boldsymbol{s} \quad ? \tag{19}$$

- **If yes:** BP has **converged**. Return $\boldsymbol{e}^{\text{BP}}$ and soft decisions P_1.
- **If no:** Go back to Step 2 and repeat.
- **If max iterations reached:** BP has **failed to converge**.

Why iterate? On graphs with cycles, a single pass doesn't propagate information globally. Each iteration allows beliefs to travel further through the graph. Eventually, if the error is correctable, the hard decisions will satisfy all parity checks.

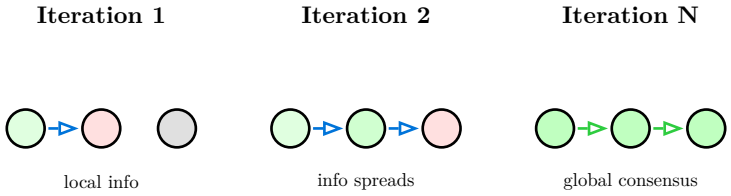


Figure 6: Information propagates further with each iteration until convergence

3.3.1 Damping for Convergence

Theory: Damping prevents oscillation by mixing old and new messages:

$$m^{(t+1)} = \gamma \cdot m^{(t)} + (1 - \gamma) \cdot m_{\text{new}} \tag{20}$$

where $\gamma \in [0, 1)$ is the damping factor.

Code: Applied after computing new check-to-qubit messages:

```
# Main iteration loop
for _ in range(max_iter):
    # Compute new check-to-qubit messages
    msg_c2q_new = self._check_to_qubit_minsum(msg_q2c, syndromes)

    # Damping: blend old and new messages
    msg_c2q = damping * msg_c2q + (1 - damping) * msg_c2q_new

    # Update qubit-to-check messages
    msg_q2c = self._qubit_to_check(msg_c2q, channel_llr)
```

Damping Value	Behavior	Use Case
$\gamma = 0$	No damping (full update)	Simple graphs, fast convergence
$\gamma = 0.2$	Light damping	Typical default
$\gamma = 0.5$	Strong damping	Graphs with short cycles

Table 3: Effect of damping factor on BP convergence

3.3.2 Summary: Complete BP Iteration

Putting it all together, one BP iteration consists of:

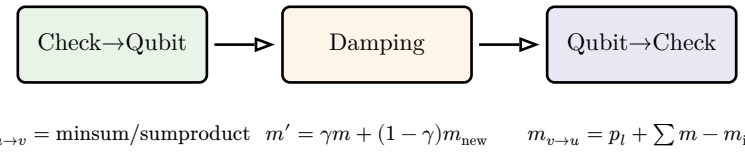


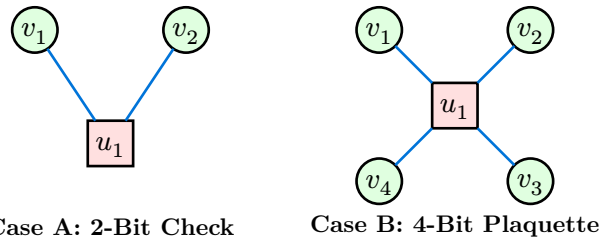
Figure 7: One BP iteration: message updates with damping

3.4 BP Dynamics: Classical vs. Quantum Constraints

To understand why BP fails on quantum codes, we compare its behavior on a minimal classical graph (2 bits) versus a minimal quantum stabilizer (4 bits).

3.4.1 Problem Setup: 2-Bit vs. 4-Bit Parity

Consider two scenarios with channel error probability $p = 0.1$ and observed syndrome $s = 1$ (odd parity): (i). A check node u_1 connected to 2 variables v_1, v_2 (e.g., a simple parity check). (ii). A check node u_1 connected to 4 variables $v_1, ..., v_4$ (e.g., a surface code plaquette).



Case A: 2-Bit Check **Case B: 4-Bit Plaquette**

Figure 8: Comparison of message passing geometry

Both cases start with the same channel LLR:

$$\text{LLR}_{\text{channel}} = \ln\left(\frac{1-p}{p}\right) = \ln(9) \approx 2.197. \tag{21}$$

The check node sends a message to v_1 based on evidence from **all other** neighbors. Formula: $m_{u\to v} = (-1)^s \cdot 2\tanh^{-1}\left(\prod_{v'\in N(u)\setminus v} \tanh(m_{v'\to u}/2)\right)$

Case A (2-Bit): Strong Correction v_1 has only 1 neighbor (v_2). The product contains a single term $\tanh(1.1) \approx 0.8$.

$$m_{u_1\to v_1} = -2\tanh^{-1}(0.8) \approx -2.197 \tag{22}$$

The check says: "My other neighbor is definitely correct, so **you** must be wrong." The message magnitude **matches** the channel prior but flips the sign.

Case B (4-Bit): Signal Dilution v_1 has 3 neighbors (v_2, v_3, v_4). The product accumulates uncertainty: $0.8^3 \approx 0.512$.

$$m_{u_1\to v_1} = -2\tanh^{-1}(0.512) \approx -1.13 \tag{23}$$

The check says: "I see an error, but it could be any of you 4. I suspect you, but weakly." The penalty (-1.13) is **weaker** than the channel prior (+2.197).

3.4.2 Step 3: Variable Update and Failure

We now compute the updated belief for v_1:

Metric	Case A (2-Bit)	Case B (4-Bit)
Prior LLR	$+2.197$	$+2.197$
Check Msg	-2.197	-1.13
Net LLR	**0**	$\mathbf{+1.067}$
Prob($e_1 = 1$)	50%	$\approx 26\%$
Hard Decision	$e_1 = 0$ or 1	$e_1 = 0$

Why Quantum BP Fails:

1. **Case A (Classical):** The LLR drops to 0. BP correctly identifies "maximum uncertainty." It knows it cannot distinguish between e_1 and e_2.
2. **Case B (Quantum):** The LLR remains positive (+1.067). BP remains "confident" that the bit is correct.
   - The hard decision outputs $\boldsymbol{e} = (0, 0, 0, 0)$.
   - **Parity Check:** $0 + 0 + 0 + 0 = 0 \neq 1$.
   - **Result:** BP fails to find a valid solution because the "blame" is diluted across too many qubits.

In a full surface code, this effect compounds. Multiple conflicting checks reduce the LLR toward 0, leading to a state of "Split Belief" where the decoder is paralyzed by symmetry.

Key Point. Summary of the Failure Mode:
- **Ambiguity Dilution:** In high-weight stabilizers (like 4-bit plaquettes), the check node message is too weak to overturn the channel prior.
- **Invalid Hard Decisions:** Unlike the 2-bit case where uncertainty is explicit ($L = 0$), the 4-bit case results in false confidence ($L > 0$) that violates the parity constraint.

3.4.3 The Degeneracy Problem

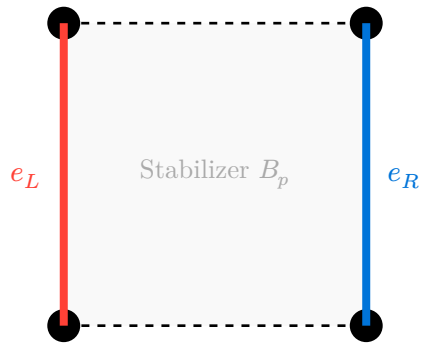
The Degeneracy Problem In quantum codes, **multiple distinct error patterns** can be physically equivalent (differing only by a stabilizer). BP, which is designed to find a **unique** correct error, fails to distinguish between these equivalent patterns, leading to decoding failure.

Definition. Two errors $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ are **degenerate** if they produce the same syndrome $\boldsymbol{s}$ and their difference forms a stabilizer:

$$H \cdot \boldsymbol{e}_1 = H \cdot \boldsymbol{e}_2 = \boldsymbol{s} \quad \text{and} \quad \boldsymbol{e}_1 + \boldsymbol{e}_2 \in \text{Stabilizers} \tag{24}$$

Unlike classical codes where distinct low-weight errors are rare, quantum stabilizer codes are **constructed** from local degeneracies (the stabilizers themselves).

Fro example, consider the Toric code where qubits reside on the edges of a lattice. A stabilizer B_p acts on the 4 qubits surrounding a plaquette. The weight-2 error occurs on the **left** edges is degenerate with the weight-2 error on the **right** two edges.



e_L and e_R are equivalent (differ by B_p)

Both have weight 2. Both satisfy local checks.

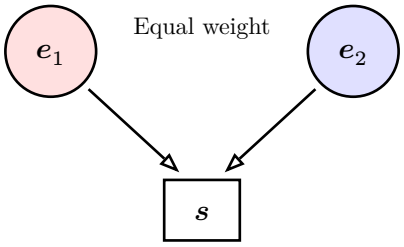
Figure 9: Symmetric degeneracy in a Toric code plaquette

When BP runs on this symmetric structure, it encounters a fatal ambiguity.

1. **Perfect Symmetry:** The graph structure for e_L is identical to the graph structure for e_R.
2. **Message Stalling:** BP receives equal evidence for the left-side error and the right-side error.
3. **Marginal Probability 0.5:** The algorithm converges to a state where every qubit in the loop has $P(e_i) \approx 0.5$.

> **Definition. The Hard Decision Failure:** When $P(e_i) \approx 0.5$, the log-likelihood ratio is ≈ 0.
>
> - If we threshold **below** 0.5: The decoder outputs $\boldsymbol{e} = \mathbf{0}$ (no error). This fails to satisfy the syndrome.
> - If we threshold **above** 0.5: The decoder outputs $\boldsymbol{e} = e_L + e_R$. This is the stabilizer itself (a closed loop), which effectively applies a logical identity but fails to correct the actual open string error.

This is why BP alone typically exhibits **zero threshold** on the Toric code: as the code size increases, the number of these symmetric loops increases, and BP gets confused by all of them simultaneously.



Same syndrome!

Figure 10: Two errors with the same syndrome cause BP to fail

When BP encounters degenerate errors of equal weight:

1. BP assigns high probability to **both** solutions $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$
2. The beliefs "split" between the two solutions
3. BP outputs $\boldsymbol{e}^{\text{BP}} \approx \boldsymbol{e}_1 + \boldsymbol{e}_2$
4. Check: $H \cdot \boldsymbol{e}^{\text{BP}} = H \cdot (\boldsymbol{e}_1 + \boldsymbol{e}_2) = \boldsymbol{s} + \boldsymbol{s} = \mathbf{0} \neq \boldsymbol{s}$

5. **BP fails to converge!**

Key Point. For the Toric code, degeneracy is so prevalent that **BP alone shows no threshold** — increasing code distance makes performance worse, not better!

4 Ordered Statistics Decoding (OSD)

4.1 The Key Insight

The parity check matrix H (size $m \times n$ with $n > m$) has more columns than rows and cannot be directly inverted. However, we can select a subset of $r = \text{rank}(H)$ linearly independent columns to form an invertible $m \times r$ submatrix.

Definition. For an $m \times n$ matrix H with $\text{rank}(H) = r$:

- **Basis set** $[S]$: indices of r linearly independent columns
- **Remainder set** $[T]$: indices of the remaining $k' = n - r$ columns
- $H_{[S]}$: the $m \times r$ submatrix of columns in $[S]$ (this is invertible!)
- $H_{[T]}$: the $m \times k'$ submatrix of columns in $[T]$

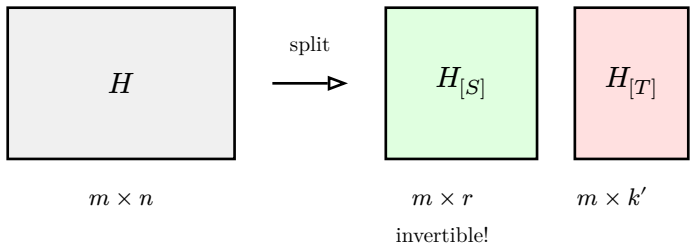


Figure 11: Splitting H into basis and remainder parts

OSD then resolves split beliefs and **forcing a unique solution**. It first choose the basis selection $[S]$ through BP soft decisions guide. and then calculate the matrix inversion on $H_{[S]}$ eliminates ambiguity.

4.2 OSD-0: The Basic Algorithm

Ordered Statistics Decoding (OSD) was introduced by Fossorier and Lin as a soft-decision decoding algorithm for linear block codes that approaches maximum-likelihood performance [7]. The algorithm was later extended with computationally efficient variants [8]. For quantum LDPC codes, the BP+OSD combination was shown to be remarkably effective by Panteleev and Kalachev, and further developed by Roffe et al. [9].

Definition. OSD-0 finds a solution by:

1. Choosing a "good" basis $[S]$ using BP soft decisions P_1
2. Solving for the basis bits via matrix inversion
3. Setting all remainder bits to zero

Algorithm 2: OSD-0

Input:

- Parity matrix H (size $m \times n$, rank r)
- Syndrome $\boldsymbol{s}$
- BP soft decisions $P_1(e_1), ..., P_1(e_n)$ (from Algorithm 1)

Steps:

1. **Rank bits by probability:** Sort bit indices by P_1 values: most-likely-flipped first. Result: ordered list $[O_{\text{BP}}] = (j_1, j_2, ..., j_n)$
2. **Reorder columns:** $H_{[O_{\text{BP}}]}$ = matrix H with columns reordered by $[O_{\text{BP}}]$
3. **Select basis:** Scan left-to-right, select first r linearly independent columns. Basis indices: $[S]$. Remainder indices: $[T]$ (size $k' = n - r$)
4. **Solve on basis:** $\boldsymbol{e}_{[S]} = H_{[S]}^{-1} \cdot \boldsymbol{s}$
5. **Set remainder to zero:** $\boldsymbol{e}_{[T]} = \mathbf{0}$ (zero vector of length k')
6. **Remap to original ordering:** Combine $\left(\boldsymbol{e}_{[S]}, \boldsymbol{e}_{[T]}\right)$ and undo the permutation

Output: $\boldsymbol{e}^{\text{OSD-0}}$ satisfying $H \cdot \boldsymbol{e}^{\text{OSD-0}} = \boldsymbol{s}$

Figure 12: OSD-0 algorithm

The GPU-accelerated implementation in `batch_osd.py` realizes OSD-0 through Gaussian elimination rather than explicit matrix inversion. Below is the mapping between the theoretical definition and the code logic.

4.2.1 Step 1: Choosing a "Good" Basis (Sorting)

The definition requires selecting basis columns $[S]$ based on BP soft decisions. For quantum error correction, the implementation sorts by **probability descending** rather than reliability $|p - 0.5|$.

```
# Sort by probability descending (highest probability first)
# High-probability errors become pivots; low-probability errors become free variables
sorted_indices = np.argsort(probs)[::-1]
```

Key Point. Why probability-based sorting for quantum codes?

Traditional OSD uses reliability $|p - 0.5|$, but this fails for quantum codes where:

- Most qubits have $p \approx 0$ (no error) $\rightarrow$ reliability ≈ 0.5
- Identified errors have $p \approx 1 \rightarrow$ reliability ≈ 0.5

Both have similar reliability despite being very different! Sorting by probability directly places likely errors in $[S]$ (pivots) and unlikely errors in $[T]$ (free variables set to 0).

4.2.2 Step 2: Solving for Basis Bits (RREF)

Instead of computing $H_{[S]}^{-1}$ explicitly, the code computes the **Reduced Row Echelon Form** (RREF) of the augmented matrix $[H_{\text{sorted}} \mid \boldsymbol{s}]$. This is numerically stable and solves the linear system in one pass. For example, consider a parity check matrix H with 3 checks and 6 variables, and syndrome $\boldsymbol{s} = (1, 0, 1)^T$. Assume BP has already sorted columns by probability (column 0 = highest probability error). Then the initial augmented matrix $[H_{\text{sorted}} \mid \boldsymbol{s}]$ writes:

$$\left(\begin{array}{cccccc|c} 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 \end{array}\right) \tag{25}$$

Iteration 1: Process column 0

- Find pivot: Row 0 has a 1 in column 0 ✓
- Eliminate: Row 1 also has 1 in column 0, so XOR row 1 with row 0

$$\left(\begin{array}{cccccc|c} \mathbf{1} & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 \end{array}\right) \quad \text{pivot\_cols} = [0] \tag{26}$$

Iteration 2: Process column 1

- Find pivot: Row 1 has a 1 in column 1 ✓
- Eliminate: Row 2 also has 1 in column 1, so XOR row 2 with row 1

$$\left(\begin{array}{cccccc|c} \mathbf{1} & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & \mathbf{1} & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right) \quad \text{pivot\_cols} = [0, 1] \tag{27}$$

Iteration 3: Process column 2

- Find pivot: No rows have a 1 in column 2 below the current pivot row ×
- Skip this column (it becomes a free variable)

Iteration 4: Process column 3

- Find pivot: Row 2 has a 1 in column 3 ✓
- Eliminate: Rows 0 and 1 have 1s in column 3, XOR both with row 2

$$\left(\begin{array}{cccccc|c} \mathbf{1} & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & \mathbf{1} & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \end{array}\right) \quad \text{pivot\_cols} = [0, 1, 3] \tag{28}$$

Final Result: Pivot columns $[S] = \{0, 1, 3\}$ (basis variables) and free columns $[T] = \{2, 4, 5\}$ (remainder variables).

In the code, the H matrix is stored as a 2D 'int8' array to fully utilized the GPU's integer arithmetic capabilities. The fuctiion 'compute_rref' implements Gaussian elimination over GF(2):

```
def _get_rref_cached(self, sorted_indices: np.ndarray, syndrome: np.ndarray):
    # Reorder columns by sorted indices
    H_sorted = self.H[:, sorted_indices]
    # Build augmented matrix [H_sorted | s]
    augmented = np.hstack([H_sorted, syndrome.reshape(-1, 1)]).astype(np.int8)
    # Compute RREF in-place
    pivot_cols = self._compute_rref(augmented)
    return augmented, pivot_cols

def _compute_rref(self, M: np.ndarray) -> List[int]:
    m, n = M.shape
    pivot_row = 0
    pivot_cols = []

    for col in range(n - 1):  # Don't pivot on syndrome column
        if pivot_row >= m:
```

```
            break
        # Find a row with 1 in this column
        candidates = np.where(M[pivot_row:, col] == 1)[0]
        if len(candidates) == 0:
            continue  # No pivot in this column

        # Swap to bring pivot to current row
        swap_r = candidates[0] + pivot_row
        if swap_r != pivot_row:
            M[[pivot_row, swap_r]] = M[[swap_r, pivot_row]]

        pivot_cols.append(col)

        # Eliminate all other 1s in this column (XOR in GF(2))
        rows_to_xor = np.where(M[:, col] == 1)[0]
        rows_to_xor = rows_to_xor[rows_to_xor != pivot_row]
        if len(rows_to_xor) > 0:
            M[rows_to_xor, :] ^= M[pivot_row, :]

        pivot_row += 1

    return pivot_cols
```

- `pivot_cols`: The column indices where pivots were found. These form the basis set $[S]$.
- After RREF, the basis submatrix has an identity-like structure, and the syndrome column contains the solution values.

4.2.3 Step 3: Reading the OSD-0 solution:

Continue from the previous example, the result can be read from the pivot collumns and syndrome column $\boldsymbol{s}$, as illustated by the following steps:

- Set free variables to zero: $e_2 = e_4 = e_5 = 0$
- Read pivot values from the transformed syndrome column:
  - Row 0: pivot at column 0, syndrome value $= 1 \rightarrow e_0 = 1$
  - Row 1: pivot at column 1, syndrome value $= 1 \rightarrow e_1 = 1$
  - Row 2: pivot at column 3, syndrome value $= 0 \rightarrow e_3 = 0$
- Solution in sorted order then must be $\boldsymbol{e}_{\text{sorted}} = (1, 1, 0, 0, 0, 0)$, and can be verified through:

$$H_{\text{sorted}} \cdot \boldsymbol{e}_{\text{sorted}} = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ \mathbf{0} \\ 0 \\ \mathbf{0} \\ \mathbf{0} \end{pmatrix} = \begin{pmatrix} 1 \oplus 0 \\ 1 \oplus 1 \\ 0 \oplus 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \boldsymbol{s} \quad \checkmark \tag{29}$$

In the code, the solution is extracted by initializing the solution vector to zeros and updating **only** the pivot positions using the transformed syndrome.

```
# OSD-0 Solution: Initialize all bits to 0 (remainder bits stay 0)
solution_base = np.zeros(self.num_errors, dtype=np.int8)

# Build pivot-to-row mapping and extract solution
pivot_row_map = {}
for r in range(augmented.shape[0]):
    row_pivots = np.where(augmented[r, :self.num_errors] == 1)[0]
    if len(row_pivots) > 0:
```

```
        col = row_pivots[0]
        if col in pivot_cols:
            pivot_row_map[col] = r
            # Pivot bit = transformed syndrome value
            solution_base[col] = augmented[r, -1]
```

- `solution_base = np.zeros(...)`: Ensures $\boldsymbol{e}_{[T]} = \mathbf{0}$ (OSD-0 constraint).
- `augmented[r, -1]`: The transformed syndrome value. Since the basis submatrix is now identity-like, this directly gives $\boldsymbol{e}_{[S]}$.

4.2.4 Step 4: Inverse Mapping (Unsort)

Finally, the solution is mapped back to the original column ordering:

```
# Remap from sorted order back to original order
estimated_errors = np.zeros(self.num_errors, dtype=int)
estimated_errors[sorted_indices] = final_solution_sorted
return estimated_errors
```

Theoretical Step	**Code Realization (`batch_osd.py`)**
1. Sort by soft decisions	`np.argsort(probs)[::-1]` (probability descending)
2. Select basis $[S]$	`pivot_cols` from `_compute_rref`
3. Matrix inversion	RREF transforms basis to identity structure
4. Solve $\boldsymbol{e}_{[S]}$	`solution_base[col] = augmented[r, -1]`
5. Set $\boldsymbol{e}_{[T]} = \mathbf{0}$	`np.zeros(...)` initialization
6. Unsort	`estimated_errors[sorted_indices] = solution`

Table 4: Mapping OSD-0 theory to `batch_osd.py` implementation

4.3 Higher-Order OSD (OSD-λ)

OSD-0 assumes the remainder error bits are zero ($\boldsymbol{e}_{[T]} = \mathbf{0}$). While this provides a valid solution, it forces all "correction" work onto the basis bits $[S]$, which may result in a high-weight (improbable) error pattern.

> **Definition. Higher-order OSD** improves this by testing non-zero configurations for the remainder bits $\boldsymbol{e}_{[T]}$. For any chosen hypothesis $\boldsymbol{e}_{[T]}$, the corresponding basis bits $\boldsymbol{e}_{[S]}$ are uniquely determined to satisfy the syndrome:
>
> $$\boldsymbol{e}_{[S]} = H_{[S]}^{-1} \cdot \left(\boldsymbol{s} + H_{[T]} \cdot \boldsymbol{e}_{[T]}\right) \tag{30}$$

It is straightforward to show that the constructed error $\boldsymbol{e} = \left(\boldsymbol{e}_{[S]}, \boldsymbol{e}_{[T]}\right)$ always satisfies the parity check equation $H \cdot \boldsymbol{e} = \boldsymbol{s}$ and the OSD-0 is a special case of OSD-λ when $\lambda = 0$.

$$\begin{aligned} H \cdot \boldsymbol{e} &= \begin{pmatrix} H_{[S]} & H_{[T]} \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{e}_{[S]} \\ \boldsymbol{e}_{[T]} \end{pmatrix} = H_{[S]} \cdot \boldsymbol{e}_{[S]} + H_{[T]} \cdot \boldsymbol{e}_{[T]} \\ &= H_{[S]} \cdot \left[H_{[S]}^{-1} \cdot \left(\boldsymbol{s} + H_{[T]} \cdot \boldsymbol{e}_{[T]}\right)\right] + H_{[T]} \cdot \boldsymbol{e}_{[T]} \\ &= I \cdot \left(\boldsymbol{s} + H_{[T]} \cdot \boldsymbol{e}_{[T]}\right) + H_{[T]} \cdot \boldsymbol{e}_{[T]} \\ &= \boldsymbol{s} + H_{[T]} \cdot \boldsymbol{e}_{[T]} + H_{[T]} \cdot \boldsymbol{e}_{[T]} = \boldsymbol{s} + \mathbf{0} = \boldsymbol{s} \end{aligned} \tag{31}$$

Then the problem change to find the minimum soft-weight solution for the remainder bits $\boldsymbol{e}_{[T]}$. A naive way to do this is to implement an exhaustive search (OSD-E) testing on all $2^{k'}$ patterns. This guarantees finding the minimum weight solution. Unfortunately, the remainder set $[T]$ has size $k' = n - r$, which is exponentially large in the code parameters $n - r$. To make this feasible, we restrict the search to the search depth λ, i.e., the **most suspicious** λ bits in $[T]$ (those with highest error probability among free variables) and accelerate this serch process by using the GPU. The `batch_osd.py` implementation accelerates OSD-E by evaluating all 2^{λ} candidates in parallel on GPU. Here is how it works:

Step 1: Identify Search Columns

Select the λ free variables with highest error probability (most suspicious):

```
# Get free columns (not pivots)
all_cols = set(range(self.num_errors))
free_cols = sorted(list(all_cols - set(pivot_cols)))

# Sort free columns by probability (highest first = most suspicious)
free_cols_with_prob = [(col, probs[sorted_indices[col]]) for col in free_cols]
free_cols_with_prob.sort(key=lambda x: -x[1])

# Select top osd_order free variables for search
search_cols = [col for col, _ in free_cols_with_prob[:osd_order]]
```

Step 2: Generate All 2^{λ} Candidates

```
# Exhaustive: Generate all 2^k combinations using bit manipulation
num_candidates = 1 << len(search_cols)  # 2^k
candidates_np = np.array([
    [(i >> j) & 1 for j in range(len(search_cols))]
    for i in range(num_candidates)
], dtype=np.int8)
```

Step 3: Parallel Evaluation on GPU

All candidates are evaluated simultaneously using batched matrix operations:

```
def _evaluate_candidates_gpu(self, candidates, augmented, search_cols, probs_sorted,
pivot_cols):
    num_candidates = candidates.shape[0]

    # Transfer to GPU
    M_subset = torch.from_numpy(augmented[:, search_cols]).float().to(self.device)
    syndrome_col = torch.from_numpy(augmented[:, -1]).float().to(self.device)

    # Compute modified syndromes for ALL candidates in parallel
    # target_syndrome = (s + M @ e_T) % 2
    target_syndromes = (syndrome_col.unsqueeze(0) + candidates.float() @ M_subset.T)
% 2

    # Initialize solution matrix (num_candidates × n)
    cand_solutions = torch.zeros(num_candidates, self.num_errors, device=self.device)

    # Set free variable values from candidates
    cand_solutions[:, search_cols] = candidates.float()

    # Solve for pivot variables using the RREF structure
    for r in range(augmented.shape[0]):
```

```
            row_pivots = torch.where(augmented_torch[r, :] == 1)[0]
            if len(row_pivots) > 0:
                pivot_c = row_pivots[0].item()
                if pivot_c in pivot_cols:
                    # Pivot value = modified syndrome for this row
                    cand_solutions[:, pivot_c] = target_syndromes[:, r]

        # Compute soft-weighted costs and return best solution
        costs = self._compute_soft_weight_gpu(cand_solutions, probs_sorted)
        best_idx = torch.argmin(costs)
        return cand_solutions[best_idx]
```

Step 4: Soft-Weight Cost Function

In our code, the OSD uses the **soft-weighted cost** based on log-probabilities [9]:

> **Definition. Soft-Weighted Cost (Log-Probability Weight).** For an error pattern $\boldsymbol{e} = (e_1, ..., e_n)$ with bit-wise error probabilities $p_i = P(e_i = 1)$, the soft-weighted cost is:
>
> $$W_{\text{soft}(\boldsymbol{e})} = \sum_{i:e_i=1} (-\log p_i) = -\sum_{i=1}^{n} e_i \cdot \log p_i \tag{32}$$
>
> Lower cost indicates a more probable error pattern.

There actually are several other cost functions that can be used for OSD, such as the Hamming weight, Euclidean distance, and LLR-based weight, as listed in the following table:

Cost Function	**Formula**	**Properties**
Hamming Weight [10]	$W_{H(\boldsymbol{e})} = \sum_i e_i$	Counts flipped bits; ignores probabilities
Soft Weight (Log-Prob) [9]	$W_{\text{soft}(\boldsymbol{e})} = -\sum_i e_i \log p_i$	Weights by $-\log p_i$; approximates ML
Euclidean Distance [11]	$d_E^2 = \sum_i (r_i - c_i)^2$	For AWGN channels with continuous signals
LLR-Based Weight [12]	$W_{\text{LLR}(\boldsymbol{e})} = \sum_i e_i \lvert L_i \rvert$	Uses log-likelihood ratios $L_i = \log\left(\frac{p_i}{1-p_i}\right)$

Table 5: Comparison of cost functions for selecting the best error pattern

Then why soft weight is preferred for BP+OSD? This is because the soft weight approximates the Maximum-Likelihood Decoding (ML) objective [13]. The ML decoder selects the error pattern $\boldsymbol{e}^*$ that maximizes the posterior probability

$$\boldsymbol{e}^* = \arg\max_{\boldsymbol{e}} P(\boldsymbol{e} \mid \text{syndrome}). \tag{33}$$

Taking the logarithm, which is a monotonic transformation, and suppose the error e_i are independent, each with probability $P(e_i = 1) = p_i$, this becomes:

$$\boldsymbol{e}^* = \arg\max_{\boldsymbol{e}} \sum_i [e_i \log p_i + (1 - e_i) \log(1 - p_i)] \tag{34}$$

For sparse errors where most $e_i = 0$, minimizing $W_{\text{soft}(\boldsymbol{e})} = -\sum_i e_i \log p_i$ closely approximates the ML objective.

Key Point. Question: Why not use the Hamming weight or Euclidean distance as the cost function?

In the code, the soft-weight cost function is implemented as follows:

```
def _compute_soft_weight_gpu(self, solutions, probs):
    # Clip to avoid log(0)
    probs_clipped = torch.clamp(probs, 1e-10, 1 - 1e-10)
    # Log-probability weights: -log(p) penalizes flipping low-probability bits
    log_weights = -torch.log(probs_clipped)
    # Total cost = sum of weights for flipped bits
    costs = (solutions * log_weights).sum(dim=1)
    return costs
```

4.4 Combination Sweep Strategy (OSD-CS)

To allow for a larger search depth (e.g., $\lambda \approx 50 - 100$) without exponential cost, we use the **combination sweep** strategy, first proposed for reducing error floors in classical LDPC codes and adapted for quantum codes by Roffe et al. [9].

Definition. OSD-CS assumes the true error pattern on the remainder bits is **sparse**. Instead of checking **all** 2^λ patterns on λ most suspicious bits, it only checks those with low Hamming weight (w = 0,1,2). Exausted on those strings only take $C_\lambda^0 + C_\lambda^1 + C_\lambda^2 = 1 + \lambda + \frac{\lambda(\lambda-1)}{2}$ candidates. With $\lambda = 60$: approximately $1 + k + 1770$ configurations (vs 2^k for exhaustive search!)

Algorithm Steps:

1. **Sort:** Select the λ most suspicious positions in $[T]$ (highest probability among free variables).
2. **Sweep:** Generate candidate vectors $\boldsymbol{e}_{[T]}$ with:
   - **Weight 0:** The zero vector (equivalent to OSD-0).
   - **Weight 1:** All single-bit flips among the λ bits.
   - **Weight 2:** All pairs of bit flips among the λ bits.
3. **Select:** Calculate $\boldsymbol{e}_{[S]}$ for each candidate, compute the soft-weighted cost, and pick the best one.

Method	Complexity	Use Case
OSD-0	$O(1)$	Fastest, baseline performance
OSD-E (Exhaustive)	$O(2^\lambda)$	Optimal for small λ (≤ 15)
OSD-CS (Comb. Sweep)	$O(\lambda^2)$	Near-optimal for large λ (≈ 60)

Table 6: Comparison of OSD Search Strategies

Why OSD-CS works for Quantum Codes? In the low-error regime relevant for QEC, it is statistically very unlikely that the optimal solution requires flipping 3+ bits in the specific subset of "uncertain" remainder bits. Checking only weights 0, 1, and 2 captures the vast majority of likely error configurations while reducing complexity from exponential to polynomial (quadratic) [9].

Definition. Combination sweep is a greedy search testing configurations by likelihood:

1. **Sort remainder bits:** Order bits in $[T]$ by error probability (most likely first)
2. **Test weight-0:** The zero vector (OSD-0 baseline)
3. **Test weight-1:** Set each single bit in $\boldsymbol{e}_{[T]}$ to 1 (all k possibilities)
4. **Test weight-2:** Set each pair among the first λ bits to 1

Keep the minimum soft-weight solution found.

4.4.1 OSD-CS Implementation in `batch_osd.py`

The GPU-accelerated implementation realizes OSD-CS by explicitly generating sparse error patterns (weights 0, 1, and 2) instead of iterating through all binary combinations.

Step 1: Generating Sparse Candidates

The `_generate_osd_cs_candidates` method generates candidate vectors $\boldsymbol{e}_{[T]}$ with structured loops:

```
def _generate_osd_cs_candidates(self, k: int, osd_order: int) -> np.ndarray:
    """Generate OSD-CS (Combination Sweep) candidate strings."""
    candidates = []

    # Weight 0: Zero vector (OSD-0 baseline)
    candidates.append(np.zeros(k, dtype=np.int8))

    # Weight 1: Single-bit flips (k candidates)
    for i in range(k):
        candidate = np.zeros(k, dtype=np.int8)
        candidate[i] = 1
        candidates.append(candidate)

    # Weight 2: Two-bit flips (limited to osd_order)
    for i in range(min(osd_order, k)):
        for j in range(i + 1, min(osd_order, k)):
            candidate = np.zeros(k, dtype=np.int8)
            candidate[i] = 1
            candidate[j] = 1
            candidates.append(candidate)

    return np.array(candidates, dtype=np.int8)
```

- `np.zeros(k)`: The "baseline" hypothesis (OSD-0 solution).
- `range(k)` loop: Adds k candidates, each with a single bit flip at index `i`.
- Nested `range(limit)` loop: Adds $\binom{\lambda}{2}$ candidates, representing pairs of flips at indices (i, j).

Step 2: Integration into the Solve Loop

The `solve` method switches between OSD-E and OSD-CS based on the `osd_method` parameter:

```
# Generate candidates based on method
if osd_method == 'combination_sweep':
    # OSD-CS: O(λ²) sparse candidates
    candidates_np = self._generate_osd_cs_candidates(len(search_cols), osd_order)
else:
    # Exhaustive: O(2^k) all combinations
    num_candidates = 1 << len(search_cols)
    candidates_np = np.array([[(i >> j) & 1 for j in range(len(search_cols))]
```

```
                                for i in range(num_candidates)], dtype=np.int8)

# Transfer to GPU and evaluate all candidates in parallel
candidates = torch.from_numpy(candidates_np).to(self.device)
best_solution_sorted = self._evaluate_candidates_gpu(
    candidates, augmented, search_cols, probs_sorted, pivot_cols
)
```

Both methods use the same GPU evaluation function—only the candidate generation differs.

Complexity Comparison

Method	**Candidates**	**Example ($\lambda = 15$)**
OSD-E	2^λ	32768
OSD-CS	$1 + k + \binom{\lambda}{2}$	$\approx 1 + k + 105$

Table 7: Number of candidates for OSD-E vs OSD-CS

Weight Class	**Code Realization**
Weight 0	`candidates.append(np.zeros(k))`
Weight 1	`for i in range(k): candidate[i] = 1`
Weight 2	`for i in range(limit): for j in range(i+1, limit): ...`

Table 8: Mapping OSD-CS theory to `batch_osd.py` implementation

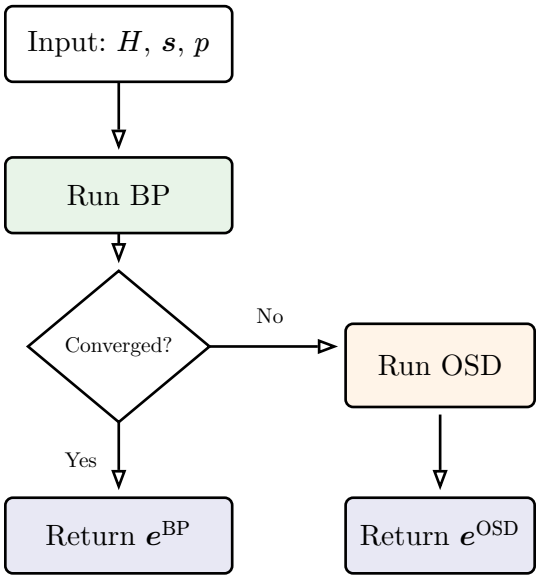


Figure 13: BP+OSD decoder flowchart

Key Point.

- If BP succeeds (converges): use BP result — fast!
- If BP fails: use OSD to resolve degeneracy — always gives valid answer

5 Quantum Error Correction Basics

5.1 Qubits and Quantum States

Definition. A **qubit** is a quantum two-level system. Its state is written using **ket notation**:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{35}$$

where:

- $|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}$ and $|1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$ are the **computational basis states**
- α, β are complex numbers with $|\alpha|^2 + |\beta|^2 = 1$
- The ket symbol $|\cdot\rangle$ is standard notation for quantum states

Common quantum states include:

- $|0\rangle, |1\rangle$ = computational basis
- $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ = superposition (plus state)
- $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ = superposition (minus state)

5.2 Pauli Operators

Definition. The **Pauli operators** are the fundamental single-qubit error operations:

Symbol	Matrix	Binary repr.	Effect on states
$\mathbb{1}$ (Identity)	$\begin{pmatrix}1&0\\0&1\end{pmatrix}$	$(0,0)$	No change
X (bit flip)	$\begin{pmatrix}0&1\\1&0\end{pmatrix}$	$(1,0)$	$\|0\rangle \leftrightarrow \|1\rangle$
Z (phase flip)	$\begin{pmatrix}1&0\\0&-1\end{pmatrix}$	$(0,1)$	$\|+\rangle \leftrightarrow \|-\rangle$
$Y = iXZ$	$\begin{pmatrix}0&-i\\i&0\end{pmatrix}$	$(1,1)$	Both flips

Table 9: Pauli operators

Key Point. Quantum errors are modeled as random Pauli operators:

- **X errors** = bit flips (like classical errors)
- **Z errors** = phase flips (uniquely quantum, no classical analogue)
- **Y errors** = both (can be written as $Y = iXZ$)

5.3 Binary Representation of Pauli Errors

Definition. An n-qubit Pauli error E can be written in **binary representation**:

$$E \mapsto \boldsymbol{e}_Q = (\boldsymbol{x}, \boldsymbol{z}) \tag{36}$$

where:

- $\boldsymbol{x} = (x_1, ..., x_n)$ indicates X components ($x_j = 1$ means X error on qubit j)
- $\boldsymbol{z} = (z_1, ..., z_n)$ indicates Z components ($z_j = 1$ means Z error on qubit j)

For example, the error $E = X_1 Z_3$ on 3 qubits (X on qubit 1, Z on qubit 3) has binary representation:

$$\boldsymbol{e}_Q = (\boldsymbol{x}, \boldsymbol{z}) = ((1, 0, 0), (0, 0, 1)) \tag{37}$$

5.4 CSS Codes

Definition. A **CSS code** (Calderbank-Shor-Steane code) is a quantum error-correcting code with a structure that allows X and Z errors to be corrected independently.

A CSS code is defined by two classical parity check matrices H_X and H_Z satisfying:

$$H_X \cdot H_Z^T = \mathbf{0} \quad \text{(orthogonality constraint)} \tag{38}$$

The combined quantum parity check matrix is:

$$H_{\text{CSS}} = \begin{pmatrix} H_Z & \mathbf{0} \\ \mathbf{0} & H_X \end{pmatrix} \tag{39}$$

Key Point. The orthogonality constraint $H_X \cdot H_Z^T = \mathbf{0}$ ensures that the quantum stabilizers **commute** (a necessary condition for valid quantum codes).

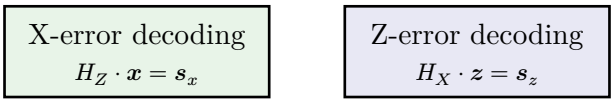
5.5 Syndrome Measurement in CSS Codes

For a CSS code with error $E \mapsto \boldsymbol{e}_Q = (\boldsymbol{x}, \boldsymbol{z})$:

Definition. The **quantum syndrome** is:

$$\boldsymbol{s}_Q = (\boldsymbol{s}_x, \boldsymbol{s}_z) = (H_Z \cdot \boldsymbol{x}, H_X \cdot \boldsymbol{z}) \tag{40}$$

- $\boldsymbol{s}_x = H_Z \cdot \boldsymbol{x}$ detects X (bit-flip) errors
- $\boldsymbol{s}_z = H_X \cdot \boldsymbol{z}$ detects Z (phase-flip) errors



Two independent classical problems!

Figure 14: CSS codes allow independent X and Z decoding

Key Point. CSS codes allow **independent decoding**:

- Decode X errors using matrix H_Z and syndrome $\boldsymbol{s}_x$
- Decode Z errors using matrix H_X and syndrome $\boldsymbol{s}_z$

Each is a classical syndrome decoding problem — so BP can be applied!

5.6 Quantum Code Parameters

Quantum codes use double-bracket notation $[[n, k, d]]$:

- n = number of physical qubits
- k = number of logical qubits encoded
- d = code distance (minimum weight of undetectable errors)

Compare to classical $[n, k, d]$ notation (single brackets).

Definition. A **quantum LDPC (QLDPC) code** is a CSS code where H_{CSS} is sparse.

An $(\boldsymbol{l_Q}, \boldsymbol{q_Q})$**-QLDPC code** has:

- Each column of H_{CSS} has at most l_Q ones
- Each row of H_{CSS} has at most q_Q ones

5.7 The Hypergraph Product Construction

Definition. The **hypergraph product** constructs a quantum CSS code from a classical code.

Given classical code with $m \times n$ parity check matrix H:

$$H_X = (H \otimes \mathbb{1}_n \;\; \mathbb{1}_m \otimes H^T) \tag{41}$$

$$H_Z = (\mathbb{1}_n \otimes H \;\; H^T \otimes \mathbb{1}_m) \tag{42}$$

Where:
- $\otimes$ = **Kronecker product** (tensor product of matrices)
- $\mathbb{1}_n = n \times n$ identity matrix
- H^T = transpose of H

A well-known example is the **Toric Code**, which is the hypergraph product of the ring code (cyclic repetition code). From a classical $[n, 1, n]$ ring code, we obtain a quantum $[[2n^2, 2, n]]$ Toric code. Its properties include:
- $(4, 4)$-QLDPC: each stabilizer involves at most 4 qubits
- High threshold (10.3% with optimal decoder)
- Rate $R = \frac{2}{2n^2} \to 0$ as $n \to \infty$

5.8 Manifest of BP+OSD threshold analysis

In this section, we implement the BP+OSD decoder on the rotated surface code datasets. The end-to-end workflow consists of three stages: (1) generating detector error models from noisy circuits, (2) building the parity check matrix with hyperedge merging, and (3) estimating logical error rates using soft XOR probability chains.

5.8.1 Step 1: Generating Rotated Surface Code DEM Files

The first step is to generate a **Detector Error Model (DEM)** from a noisy quantum circuit using **Stim**. The DEM captures the probabilistic relationship between physical errors and syndrome patterns.

> **Definition.** A **Detector Error Model (DEM)** is a list of **error mechanisms**, each specifying a probability p of occurrence, a set of **detectors** (syndrome bits) that flip when the error occurs, and optionally, **logical observables** that flip when the error occurs.

We use Stim's built-in circuit generator to create rotated surface code memory experiments with circuit-level depolarizing noise:

```
import stim

circuit = stim.Circuit.generated(
    "surface_code:rotated_memory_z",
    distance=d,         # Code distance
    rounds=r,           # Number of syndrome measurement rounds
    after_clifford_depolarization=p,      # Noise after gates
    before_round_data_depolarization=p,   # Noise on idle qubits
    before_measure_flip_probability=p,    # Measurement errors
    after_reset_flip_probability=p,       # Reset errors
)

# Extract DEM from circuit
dem = circuit.detector_error_model(decompose_errors=True)
```

The DEM output uses a compact text format. Key elements include:

Syntax	Meaning
`error(0.01) D0 D1`	Error with $p = 0.01$ that triggers detectors D_0 and D_1
`error(0.01) D0 D1 ^ D2`	**Correlated error**: triggers $\{D_0, D_1\}$ AND $\{D_2\}$ simultaneously
`error(0.01) D0 L0`	Error that triggers D_0 and flips logical observable L_0
`detector D0`	Declares detector D_0 (syndrome bit)
`logical_observable L0`	Declares logical observable L_0

Table 10: DEM syntax elements. The `^` separator indicates correlated fault mechanisms.

> **Key Point.** The `^` **separator** is critical for correct decoding. In `error(p) D0 D1 ^ D2`, the fault triggers **both** patterns $\{D_0, D_1\}$ and $\{D_2\}$ simultaneously with probability p.

These must be treated as separate columns in the parity check matrix H, each with the same probability p.

5.8.2 Step 2: Building the Parity Check Matrix H

Converting the DEM to a parity check matrix H for BP decoding requires two critical processing stages.

5.8.2.1 Stage 1: Separator Splitting

DEM errors with `^` separators represent correlated faults that trigger multiple detector patterns simultaneously. These must be split into **separate columns** in H:

Key Point. Example: Consider `error(0.01) D0 D1 ^ D2 L0`. This splits into two components:

- Component 1: detectors $= \{D_0, D_1\}$, observables $= \{\}$, probability $= 0.01$
- Component 2: detectors $= \{D_2\}$, observables $= \{L_0\}$, probability $= 0.01$

Each component becomes a **separate column** in the H matrix with the same probability.

The splitting algorithm (from `_split_error_by_separator`):

```
def _split_error_by_separator(targets):
    components = []
    current_detectors, current_observables = [], []

    for t in targets:
        if t.is_separator():  # ^ found
            components.append({
                "detectors": current_detectors,
                "observables": current_observables
            })
            current_detectors, current_observables = [], []
        elif t.is_relative_detector_id():
            current_detectors.append(t.val)
        elif t.is_logical_observable_id():
            current_observables.append(t.val)

    # Don't forget the last component
    components.append({"detectors": current_detectors,
                       "observables": current_observables})
    return components
```

5.8.2.2 Stage 2: Hyperedge Merging

After splitting, errors with **identical detector patterns** are merged into single **hyperedges**. This is essential because:

1. Errors with identical syndromes are **indistinguishable** to the decoder
2. Detectors are XOR-based: two errors triggering the same detector cancel out
3. Merging reduces the factor graph size and improves threshold performance

Definition. Hyperedge Merging: When two error mechanisms have identical detector patterns, their probabilities are combined using the **XOR formula**:

$$p_{\text{combined}} = p_1 + p_2 - 2p_1p_2 \tag{43}$$

This formula computes $P(\text{odd number of errors fire}) = P(A \oplus B)$.

Proof. For independent errors A and B:

$$\begin{aligned} P(A \oplus B) &= P(A) \cdot (1 - P(B)) + P(B) \cdot (1 - P(A)) \\ &= P(A) + P(B) - 2P(A)P(B) \end{aligned} \tag{44}$$

This is exactly the probability that an **odd** number of the two errors occurs, which determines the net syndrome flip (since two flips cancel). □

For observable flip tracking, we compute the **conditional probability** $P(\text{obs flip} \mid \text{hyperedge fires})$:

```
# When merging error with probability prob into existing hyperedge:
if has_obs_flip:
    # New error flips observable: XOR with existing flip probability
    obs_prob_new = obs_prob_old * (1 - prob) + prob * (1 - obs_prob_old)
else:
    # New error doesn't flip observable
    obs_prob_new = obs_prob_old * (1 - prob)

# Store conditional probability: P(obs flip | hyperedge fires)
obs_flip[j] = obs_prob / p_combined
```

Mode	H Columns (d=3)	Description
No split, no merge	~ 286	Raw DEM errors as columns
Split only	~ 556	After ^ separator splitting
Split + merge (optimal)	~ 400	After hyperedge merging

Table 11: Effect of separator splitting and hyperedge merging on H matrix size for $d = 3$ rotated surface code. The split+merge approach provides the optimal balance.

The final output is a tuple $(H, \text{priors}, \text{obs\_flip})$ where:

- H: Parity check matrix of shape $(\text{num\_detectors}, \text{num\_hyperedges})$
- priors: Prior error probabilities per hyperedge
- obs_flip: Observable flip probabilities $P(\text{obs flip} \mid \text{hyperedge fires})$

5.8.3 Step 3: Estimating Logical Error Rate

With the parity check matrix H constructed, we can now decode syndrome samples and estimate the logical error rate.

5.8.3.1 Decoding Pipeline

The BP+OSD decoding pipeline consists of three stages:

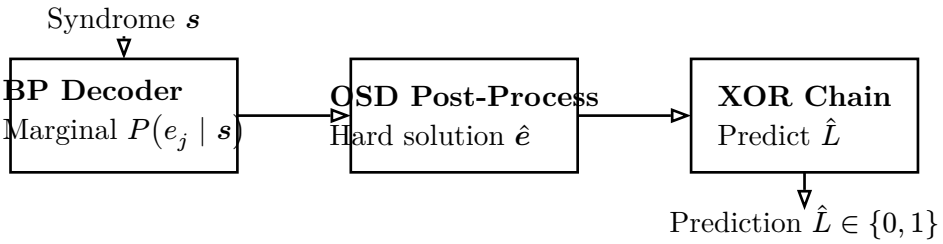


Figure 15: BP+OSD decoding pipeline: BP computes soft marginals, OSD finds a hard solution, XOR chain predicts observable.

1. **BP Decoding**: Given syndrome $\boldsymbol{s}$, run belief propagation on the factor graph to compute marginal probabilities $P(e_j = 1 \mid \boldsymbol{s})$ for each hyperedge j.
2. **OSD Post-Processing**: Use Ordered Statistics Decoding to find a hard solution $\hat{\boldsymbol{e}}$ satisfying $H\hat{\boldsymbol{e}} = \boldsymbol{s}$, ordered by BP marginals.
3. **XOR Probability Chain**: Compute the predicted observable value using soft probabilities.

5.8.3.2 XOR Probability Chain for Observable Prediction

The key insight is that observable prediction must account for the **soft** flip probabilities stored in `obs_flip`. When hyperedges are merged, `obs_flip[j]` contains $P(\text{obs flip} \mid \text{hyperedge } j \text{ fires})$, not a binary indicator.

Theorem (XOR Probability Chain). Given a solution $\hat{\boldsymbol{e}}$ and observable flip probabilities obs_flip, the probability of an odd number of observable flips is computed iteratively:

$$P_{\text{flip}} = P_{\text{flip}} \cdot (1 - \text{obs\_flip}[j]) + \text{obs\_flip}[j] \cdot \left(1 - P_{\text{flip}}\right) \tag{45}$$

for each j where $\hat{e}_j = 1$. The predicted observable is $\hat{L} = \mathbb{1}\left[P_{\text{flip}} > 0.5\right]$.

The implementation:

```
def compute_observable_predictions_batch(solutions, obs_flip):
    batch_size = solutions.shape[0]
    predictions = np.zeros(batch_size, dtype=int)

    for b in range(batch_size):
        p_flip = 0.0
        for i in np.where(solutions[b] == 1)[0]:
            # XOR probability: P(A XOR B) = P(A)(1-P(B)) + P(B)(1-P(A))
            p_flip = p_flip * (1 - obs_flip[i]) + obs_flip[i] * (1 - p_flip)
        predictions[b] = int(p_flip > 0.5)

    return predictions
```

Key Point. If `merge_hyperedges=False`, then `obs_flip` contains binary values $\{0, 1\}$, and the XOR chain reduces to simple parity: $\hat{L} = \sum_j \hat{e}_j \cdot \text{obs\_flip}[j] \bmod 2$.

5.8.3.3 Logical Error Rate Estimation

The logical error rate (LER) is estimated by comparing predictions to ground truth:

$$\text{LER} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}\left[\hat{L}^{(i)} \neq L^{(i)}\right] \tag{46}$$

where N is the number of syndrome samples, $\hat{L}^{(i)}$ is the predicted observable for sample i, and $L^{(i)}$ is the ground truth.

5.8.3.4 Threshold Analysis

The **threshold** p_{th} is the physical error rate below which increasing code distance reduces the logical error rate. For rotated surface codes with circuit-level depolarizing noise, the threshold is approximately **0.7%** (Bravyi et al., Nature 2024).

Distance	$p = 0.005$	$p = 0.007$	$p = 0.009$
$d = 3$	~ 0.03	~ 0.06	~ 0.10
$d = 5$	~ 0.01	~ 0.04	~ 0.09
$d = 7$	~ 0.005	~ 0.03	~ 0.08

Table 12: Example logical error rates for BP+OSD decoder. Below threshold ($p < 0.007$), larger distances achieve lower LER. Above threshold, the trend reverses.

At the threshold, curves for different distances **cross**: below threshold, larger d gives lower LER; above threshold, larger d gives **higher** LER due to more opportunities for errors to accumulate.

5.9 BP Convergence and Performance Guarantees

Theorem (BP Convergence on Trees). [2], [14] If the factor graph $G = (V, U, E)$ is a **tree** (contains no cycles), then BP converges to the **exact** marginal probabilities $P(e_j = 1 \mid \boldsymbol{s})$ in at most d iterations, where d is the diameter of the tree (maximum distance between any two nodes).

Proof. We prove exactness by induction on the tree structure, using the factorization property of graphical models.

Factorization on Trees: For a tree-structured factor graph, the joint probability distribution factors as:

$$P(\boldsymbol{x}) = \frac{1}{Z} \prod_{a \in \mathcal{F}} \psi_{a(\boldsymbol{x}_{\mathcal{N}(a)})} \tag{47}$$

where $\mathcal{F}$ is the set of factors, $\mathcal{N}(a)$ are neighbors of factor a, and Z is the partition function.

Key Property: On a tree, removing any node v separates the graph into disjoint connected components (subtrees). By the global Markov property, variables in different subtrees are conditionally independent given v.

Base Case (Leaf Nodes): Consider a leaf variable node v with single neighbor (factor) a. The message $\mu_{v \to a}(x_v)$ depends only on the local evidence $P(y_v \mid x_v)$. Since there are no other dependencies, this message is exact at iteration 1.

Inductive Step: Assume messages from all nodes at distance $> k$ from root are exact. Consider node u at distance k with neighbors $\mathcal{N}(u) = \{a_1, ..., a_m\}$.

For message $\mu_{u \to a_i}(x_u)$, the BP update is:

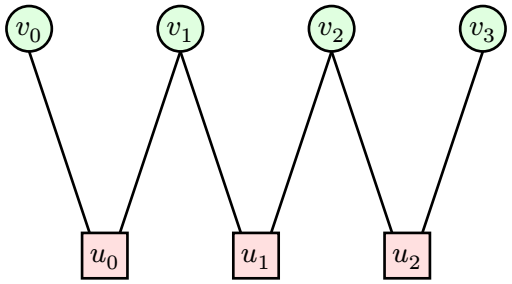
$$\mu_{u \to a_i}(x_u) \propto P(y_u \mid x_u) \prod_{a_j \in \mathcal{N}(u) \setminus \{a_i\}} \mu_{a_j \to u}(x_u) \tag{48}$$

By the separation property, removing u creates m independent subtrees rooted at $\{a_1, ..., a_m\}$. By the inductive hypothesis, messages from these subtrees are exact marginals of their respective subtrees. Since subtrees are conditionally independent given u, the product of messages equals the joint probability of all subtree configurations, making $\mu_{u \to a_i}(x_u)$ exact.

Termination: After d iterations (where d is the tree diameter), messages have propagated from all leaves to all nodes. Each node's belief $b_{v(x_v)} \propto P(y_v \mid x_v) \prod_{a \in \mathcal{N}(v)} \mu_{a \to v}(x_v)$ equals the exact marginal $P(x_v \mid \boldsymbol{y})$ by the factorization property.

Therefore, BP computes exact marginals on trees in d iterations. □

Example: Consider the $[7, 4, 3]$ Hamming code with tree-structured factor graph:



Tree structure: diameter $d = 4$, BP converges in 4 iterations

Figure 16: Tree-structured code where BP gives exact solution

For this tree with syndrome $\boldsymbol{s} = (1, 0, 0)$ and $p = 0.1$:

- BP converges in $d = 4$ iterations
- Output: $\boldsymbol{e}^{\text{BP}} = (1, 0, 0, 0, 0, 0, 0)$ (single bit flip at position 0)
- This is the **exact** maximum likelihood solution

Theorem (BP Performance on Graphs with Cycles). [15], [16] For an (l, q)-LDPC code with factor graph of **girth** g (minimum cycle length), BP provides the following guarantees:

1. **Local optimality:** If the true error $\boldsymbol{e}^*$ has Hamming weight $|\boldsymbol{e}^*| < g/2$, then BP converges to $\boldsymbol{e}^*$ with high probability (for sufficiently small p).
2. **Approximation bound:** For codes with girth $g \geq 6$ and maximum degree $\Delta = \max(l, q)$, if BP converges, the output $\boldsymbol{e}^{\text{BP}}$ satisfies:

$$|\boldsymbol{e}^{\text{BP}}| \leq (1 + \varepsilon(g, \Delta)) \cdot |\boldsymbol{e}^*| \tag{49}$$

   where $\varepsilon(g, \Delta) \to 0$ as $g \to \infty$ for fixed Δ.
3. **Iteration complexity:** BP requires $O(g)$ iterations to propagate information across the shortest cycle.

Proof. **Part 1 (Local optimality):** Consider an error $\boldsymbol{e}^*$ with $|\boldsymbol{e}^*| < g/2$. In the factor graph, the neighborhood of radius $r = \lfloor g/2 \rfloor - 1$ around any error bit is a tree (no cycles within distance r). Within this tree neighborhood:

- BP computes exact marginals (by Theorem 1)
- The error bits are separated by distance $\geq g/2$
- No interference between error regions

Therefore, BP correctly identifies each error bit independently, giving $\boldsymbol{e}^{\text{BP}} = \boldsymbol{e}^*$.

Part 2 (Approximation bound): For $|\boldsymbol{e}^*| \geq g/2$, cycles create dependencies. The approximation error comes from:

- **Double-counting:** Evidence circulates through cycles
- **Correlation:** Nearby error bits are not independent

For girth g, the correlation decays exponentially with distance. The number of length-g cycles through a node is bounded by Δ^g. Using the correlation decay lemma for loopy belief propagation, the relative error in log-likelihood ratios is:

$$\varepsilon(g, \Delta) \leq C \cdot \Delta^{2-g/2} \tag{50}$$

for some constant C. This translates to the weight approximation bound.

Part 3 (Iteration complexity): Information propagates one edge per iteration. To detect a cycle of length g, messages must travel distance g, requiring $O(g)$ iterations. □

Key Point. Practical implications:

- Codes with large girth g (e.g., $g \geq 8$) allow BP to correct more errors
- Random LDPC codes typically have $g = O(\log n)$, giving good BP performance
- Structured codes (e.g., Toric code with $g = 4$) have small girth, leading to BP failures
- The degeneracy problem in quantum codes compounds the cycle problem, making OSD necessary

5.9.1 Density Evolution Framework

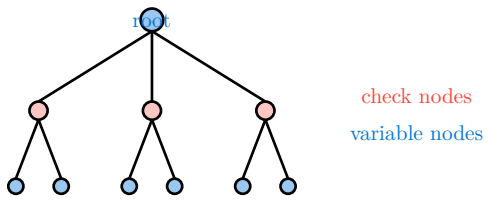
We now develop the rigorous theoretical foundations that explain **when** and **why** BP converges in different regimes, drawing from asymptotic analysis via density evolution [3], variational optimization through statistical physics [4], and combinatorial failure modes [5].

For infinite-length random LDPC codes, convergence is analyzed through the **density evolution** method [15], which tracks the probability distributions of messages rather than individual message values.

Definition. Cycle-Free Horizon: For a random LDPC code with block length $n \to \infty$, the **computation tree** of depth l rooted at any edge is the subgraph containing all nodes reachable within l hops. The cycle-free horizon property states:

$$\lim_{n\to\infty} \mathbb{P}(\text{cycle in depth-}l \text{ tree}) = 0 \tag{51}$$

This means that for any fixed number of iterations l, the local neighborhood appears tree-like with probability approaching 1 as $n \to \infty$.



Computation tree of depth $l = 2$: no cycles

Figure 17: Locally tree-like structure in large random graphs

Key Point. The cycle-free horizon is the mathematical justification for applying tree-based convergence proofs to loopy graphs in the asymptotic limit. It explains why BP performs well on long random LDPC codes despite the presence of cycles.

Definition. Concentration Theorem: Let Z be a performance metric (e.g., bit error rate) of BP after l iterations on a code randomly drawn from ensemble $\mathcal{C}(n, \lambda, \rho)$, where $\lambda(x)$ and $\rho(x)$ are the variable and check node degree distributions. For any $\varepsilon > 0$:

$$\mathbb{P}(|Z - \mathbb{E}[Z]| > \varepsilon) \leq e^{-\beta n \varepsilon^2} \tag{52}$$

where $\beta > 0$ depends on the ensemble parameters.

Interpretation: As $n \to \infty$, almost all codes in the ensemble perform identically to the ensemble average. Individual code performance concentrates around the mean with exponentially small deviation probability.

Key Point. Concentration visualization: The performance metric Z concentrates exponentially around its ensemble average $\mathbb{E}[Z]$. For large block length n, the probability of deviation greater than ε decays as $e^{-\beta n \varepsilon^2}$, meaning almost all codes perform identically to the average.

The proof of the Concentration Theorem uses martingale theory:

Proof. **Proof sketch via Doob's Martingale:**

1. **Martingale Construction:** View code selection as revealing edges sequentially. Define $Z_i = \mathbb{E}[Z \mid \text{first } \ i \ \text{ edges revealed}]$. This forms a Doob martingale: $\mathbb{E}[Z_{i+1} \mid Z_0, ..., Z_i] = Z_i$.
2. **Bounded Differences:** In a sparse graph with maximum degree Δ, changing a single edge affects at most $O(\Delta^l)$ messages after l iterations. Since Δ is constant and l is fixed, the change in Z is bounded: $|Z_i - Z_{i-1}| \leq c/n$ for some constant c.
3. **Azuma-Hoeffding Inequality:** For a martingale with bounded differences $|Z_i - Z_{i-1}| \leq c_i$:

$$\mathbb{P}(|Z_m - Z_0| > \varepsilon) \leq 2 \exp\left(-\frac{\varepsilon^2}{2\sum_{i=1}^{m} c_i^2}\right) \tag{53}$$

4. **Application:** With $m = O(n)$ edges and $c_i = O(1/n)$, we have $\sum c_i^2 = O(1/n)$, giving:

$$\mathbb{P}(|Z - \mathbb{E}[Z]| > \varepsilon) \leq 2 \exp\left(-\frac{\varepsilon^2 n}{2C}\right) = e^{-\beta n \varepsilon^2} \tag{54}$$

where $\beta = 1/(2C)$. □

Theorem (Threshold Theorem). For a code ensemble with degree distributions $\lambda(x), \rho(x)$ and a symmetric channel with noise parameter σ (e.g., standard deviation for AWGN), there exists a unique **threshold** σ^* such that:

1. If $\sigma < \sigma^*$ (low noise): As $l \to \infty$, the probability of decoding error $P_e^{(l)} \to 0$
2. If $\sigma > \sigma^*$ (high noise): $P_e^{(l)}$ remains bounded away from zero

The threshold is determined by the fixed points of the density evolution recursion:

$$P_{l+1} = \Phi(P_l, \sigma) \tag{55}$$

where Φ is the density update operator combining variable and check node operations.

Key Point. Threshold phenomenon: There exists a sharp transition at σ^*. Below this threshold (low noise), BP converges to zero error as iterations increase. Above threshold (high noise), errors persist. This sharp phase transition is characteristic of random LDPC ensembles.

Key Point. Why the threshold exists: The density evolution operator Φ has two competing fixed points:

- **All-correct fixed point:** Messages concentrate at $\pm\infty$ (high confidence)
- **Error fixed point:** Messages remain near zero (low confidence)

Below threshold, the all-correct fixed point is stable and attracts all trajectories. Above threshold, the error fixed point becomes stable, trapping the decoder.

5.9.2 Variational Perspective: Bethe Free Energy

The density evolution framework applies to infinite-length codes. For finite loopy graphs, we need a different lens: **statistical physics** [4]. This reveals that BP is actually performing **variational optimization** of an energy function.

Definition. Bethe Free Energy: For a factor graph with variables $\boldsymbol{x} = (x_1, ..., x_n)$ and factors ψ_a, let $b_{i(x_i)}$ be the **belief** (pseudo-marginal) at variable i and $b_{a(\boldsymbol{x}_a)}$ be the belief at factor a. The Bethe Free Energy is:

$$F_{\text{Bethe}(b)} = \sum_a \sum_{\boldsymbol{x}_a} b_{a(\boldsymbol{x}_a)} E_{a(\boldsymbol{x}_a)} - H_{\text{Bethe}(b)} \tag{56}$$

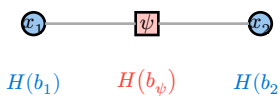
where the **Bethe entropy** approximates the true entropy using local entropies:

$$H_{\text{Bethe}} = \sum_a H(b_a) + \sum_i (1 - d_i) H(b_i) \tag{57}$$

Here d_i is the degree of variable i, and $H(b) = -\sum_x b(x) \log b(x)$ is the Shannon entropy.

Constraints: Beliefs must be normalized and **marginally consistent**:

$$\sum_{\boldsymbol{x}_a \setminus x_i} b_{a(\boldsymbol{x}_a)} = b_{i(x_i)} \quad \text{for all } i \in a \tag{58}$$



Bethe entropy: $H_{\text{Bethe}} = H(b_\psi) + (1-2)H(b_1) + (1-2)H(b_2)$

($d_1 = d_2 = 2$ for this graph)

Figure 18: Bethe entropy decomposes global entropy into local terms

Key Point. Intuition: The Bethe approximation treats each factor independently, summing local entropies. The $(1 - d_i)$ correction prevents double-counting: a variable connected to d_i factors appears in d_i factor entropies, so we subtract $(d_i - 1)$ copies of its individual entropy.

Theorem (Yedidia-Freeman-Weiss). [4] A set of beliefs $\setminus \{b_i, b_a \setminus\}$ is a **fixed point** of the Sum-Product BP algorithm if and only if it is a **stationary point** (critical point) of the Bethe Free Energy $F_{\text{Bethe}(b)}$ subject to normalization and marginalization constraints.

Equivalently: BP performs coordinate descent on the Bethe Free Energy. Each message update corresponds to minimizing F_{Bethe} with respect to one edge's belief.

Proof. **Proof sketch via Lagrangian:**

1. **Constrained optimization:** Form the Lagrangian:

$$\mathcal{L} = F_{\text{Bethe}(b)} + \sum_{i,a} \sum_{x_i} \lambda_{ia}(x_i) \left(b_{i(x_i)} - \sum_{\boldsymbol{x}_a \setminus x_i} b_{a(\boldsymbol{x}_a)} \right) + \text{normalization terms} \tag{59}$$

2. **Stationarity conditions:** Taking $\partial \mathcal{L} / \partial b_a = 0$ and $\partial \mathcal{L} / \partial b_i = 0$:

$$b_{a(\boldsymbol{x}_a)} \propto \psi_{a(\boldsymbol{x}_a)} \prod_{i \in a} \exp(\lambda_{ia}(x_i)) \tag{60}$$

$$b_{i(x_i)} \propto \prod_{a \in i} \exp(\lambda_{ai}(x_i)) \tag{61}$$

3. **Message identification:** Define messages $\mu_{i \to a}(x_i) = \exp(\lambda_{ia}(x_i))$. Substituting and enforcing marginalization constraints yields exactly the BP update equations:

$$\mu_{i \to a}(x_i) \propto P(y_i \mid x_i) \prod_{a' \in i \setminus a} \mu_{a' \to i}(x_i) \tag{62}$$

$$\mu_{a \to i}(x_i) \propto \sum_{\boldsymbol{x}_a \setminus x_i} \psi_{a(\boldsymbol{x}_a)} \prod_{i' \in a \setminus i} \mu_{i' \to a}(x_{i'}) \tag{63}$$

□

Key Point. Energy landscape interpretation: BP performs gradient descent on the Bethe Free Energy landscape. On trees, there's a single global minimum (correct solution). On loopy graphs, local minima can trap the decoder, corresponding to incorrect fixed points. The contour lines represent energy levels, with BP trajectories flowing toward minima.

Key Point. Implications for convergence:

- **On trees:** Bethe approximation is exact ($F_{\text{Bethe}} = F_{\text{Gibbs}}$), so BP finds the global minimum
- **On loopy graphs:** F_{Bethe} is an approximation. BP finds a local minimum, which may not be the true posterior

- **Stable fixed points** correspond to local minima of F_{Bethe}
- **Unstable fixed points** (saddle points) cause oscillations

This explains why BP can converge to incorrect solutions: it gets trapped in local minima created by graph cycles.

5.9.3 Sufficient Conditions for Convergence

While density evolution guarantees asymptotic convergence and Bethe theory explains fixed points, neither provides **guarantees** for specific finite loopy graphs. We now present rigorous sufficient conditions [17].

Definition. Dobrushin's Influence Matrix: For a graphical model, the influence C_{ij} measures the maximum change in the marginal distribution of variable i caused by fixing variable j:

$$C_{ij} = \sup_{x_j, x_{j'}} \| P(x_i \mid x_j) - P(x_i \mid x_{j'}) \|_{\text{TV}} \tag{64}$$

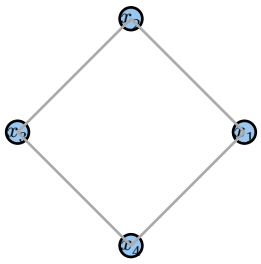
where $\| \cdot \|_{\text{TV}}$ is the total variation distance.

The **Dobrushin interdependence matrix** $\boldsymbol{C}$ has entries C_{ij} for $i \neq j$ and $C_{ii} = 0$.

Theorem (Dobrushin's Uniqueness Condition). If the Dobrushin matrix satisfies:

$$\|\boldsymbol{C}\|_\infty = \max_i \sum_{j \neq i} C_{ij} < 1 \tag{65}$$

then:
1. The Gibbs measure has a unique fixed point
2. BP converges exponentially fast to this fixed point from any initialization
3. The convergence rate is $\lambda = \|\boldsymbol{C}\|_\infty$



Example: 4-cycle with weak coupling
$\|\boldsymbol{C}\|_\infty = \max_i \sum_j C_{ij} = 2 \cdot 0.3 = 0.6 < 1$ ✓

Figure 19: Dobrushin condition: information dissipates through the graph

Key Point. Limitation for LDPC codes: Error correction codes are designed to **propagate** information over long distances. Parity checks impose hard constraints (infinite coupling strength). Therefore, useful LDPC codes typically **violate** Dobrushin's condition.

While sufficient, Dobrushin's condition is far from necessary. It applies mainly to high-noise regimes where correlations are weak.

Theorem (Contraction Mapping Convergence). View BP as a mapping $\boldsymbol{T} : \mathcal{M} \to \mathcal{M}$ on the space of messages. If $\boldsymbol{T}$ is a **contraction** under some metric d:

$$d(\boldsymbol{T}(\boldsymbol{m}), \boldsymbol{T}(\boldsymbol{m}')) \leq \lambda \cdot d(\boldsymbol{m}, \boldsymbol{m}') \tag{66}$$

with Lipschitz constant $\lambda < 1$, then:

1. BP has a unique fixed point $\boldsymbol{m}^*$
2. BP converges geometrically: $d\big(\boldsymbol{m}^{(t)}, \boldsymbol{m}^*\big) \leq \lambda^t d\big(\boldsymbol{m}^{(0)}, \boldsymbol{m}^*\big)$

Proof. **Proof:** Direct application of the Banach Fixed Point Theorem. The contraction property ensures:

- **Uniqueness:** If $\boldsymbol{m}^*$ and $\boldsymbol{m}'^*$ are both fixed points, then:

$$d(\boldsymbol{m}^*, \boldsymbol{m}'^*) = d(\boldsymbol{T}(\boldsymbol{m}^*), \boldsymbol{T}(\boldsymbol{m}'^*)) \leq \lambda \cdot d(\boldsymbol{m}^*, \boldsymbol{m}'^*) \tag{67}$$

Since $\lambda < 1$, this implies $d(\boldsymbol{m}^*, \boldsymbol{m}'^*) = 0$, so $\boldsymbol{m}^* = \boldsymbol{m}'^*$.

- **Convergence:** For any initialization $\boldsymbol{m}^{(0)}$:

$$d\big(\boldsymbol{m}^{(t+1)}, \boldsymbol{m}^*\big) = d\big(\boldsymbol{T}\big(\boldsymbol{m}^{(t)}\big), \boldsymbol{T}(\boldsymbol{m}^*)\big) \leq \lambda \cdot d\big(\boldsymbol{m}^{(t)}, \boldsymbol{m}^*\big) \tag{68}$$

Iterating gives $d\big(\boldsymbol{m}^{(t)}, \boldsymbol{m}^*\big) \leq \lambda^t d\big(\boldsymbol{m}^{(0)}, \boldsymbol{m}^*\big)$. □

Key Point. Spectral radius condition: For binary pairwise models, the contraction constant can be computed from the **spectral radius** of the interaction matrix:

$$\rho(\boldsymbol{A}) < 1, \quad \text{where } A_{ij} = \tanh|J_{ij}| \tag{69}$$

This is sharper than Dobrushin's condition (which corresponds to the L_∞ norm of $\boldsymbol{A}$).

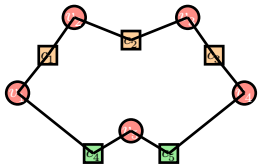
5.9.4 Failure Mechanisms: Trapping Sets

The previous sections explain when BP converges. We now characterize when and why it **fails** [5]. In the high-SNR regime, BP can get trapped in incorrect fixed points due to specific graph substructures.

Definition. (a,b) Absorbing Set: A subset $\mathcal{D} \subseteq V$ of a variable nodes is an (a, b) absorbing set if:

1. The induced subgraph contains exactly b **odd-degree** check nodes (unsatisfied checks)
2. Every variable node $v \in \mathcal{D}$ has **strictly more** even-degree neighbors than odd-degree neighbors in the induced subgraph

Interpretation: If the variables in $\mathcal{D}$ are in error, each receives more "confirming" messages (from satisfied checks) than "correcting" messages (from unsatisfied checks), causing the decoder to stabilize in the error state.



5 error variables (red)
3 odd-degree checks (orange)
Even-degree checks (green)

Canonical $(5, 3)$ absorbing set: each variable has ≥ 2 even neighbors

Figure 20: The $(5, 3)$ absorbing set: a stable error configuration

Key Point. Why BP gets trapped:

1. **Majority vote:** Each variable node performs a weighted majority vote of its check neighbors
2. **Satisfied checks dominate:** In an absorbing set, satisfied checks (even degree) outnumber unsatisfied checks (odd degree) for each variable
3. **Reinforcement loop:** Satisfied checks send messages that **confirm** the error state, while unsatisfied checks send weak correction signals
4. **Stable fixed point:** The configuration becomes a local minimum of the Bethe Free Energy

This is the primary cause of **error floors** in LDPC codes: at high SNR, rare noise patterns that activate absorbing sets dominate the error probability.

Theorem (Absorbing Sets and Error Floors). For an LDPC code with minimum absorbing set size $(a_{\min}, b_{\min})$, the error floor is dominated by:

$$P_{\text{error}} \approx \binom{n}{a_{\min}} \cdot p^{a_{\min}} \cdot (1-p)^{n-a_{\min}} \cdot P_{\text{trap}} \tag{70}$$

where P_{trap} is the probability that BP fails to correct the absorbing set configuration.

Implication: Error floor height is determined by the **size** and **multiplicity** of small absorbing sets. Code design focuses on eliminating small absorbing sets.

6 Minimum Weight Perfect Matching (MWPM) Decoder

6.1 Maximum Likelihood Decoding and MWPM

Maximum Likelihood Decoding (MLD) seeks the most probable error pattern $\boldsymbol{e}$ given syndrome $\boldsymbol{s}$ and error probabilities $p(\boldsymbol{e})$.

Definition. Maximum Likelihood Decoding Problem: Given parity check matrix $\boldsymbol{H} \in \mathbb{F}_2^{m\times n}$, syndrome $\boldsymbol{s} \in \mathbb{F}_2^m$, and error weights $w_i = \ln\left(\frac{1-p_i}{p_i}\right)$, find:

$$\min_{\boldsymbol{c}\in\mathbb{F}_2^n} \sum_{i\in[n]} w_i c_i \quad \text{subject to} \quad \boldsymbol{H}\boldsymbol{c} = \boldsymbol{s} \tag{71}$$

For certain code structures, MLD can be efficiently reduced to a graph matching problem.

Theorem (MLD to MWPM Reduction). If every column of $\boldsymbol{H}$ has at most 2 non-zero elements (each error triggers at most 2 detectors), then MLD can be deterministically reduced to Minimum Weight Perfect Matching with boundaries in polynomial time.

Definition. Detector Graph: Given $\boldsymbol{H} \in \mathbb{F}_2^{m\times n}$, construct graph $G = (V, E)$ where:

- Vertices: $V = [m] \cup \{0\}$ (detectors plus boundary vertex)
- Edges: For each column i of $\boldsymbol{H}$:
  - If column i has weight 2 (triggers detectors x_1, x_2): edge (x_1, x_2) with weight w_i
  - If column i has weight 1 (triggers detector x): edge $(x, 0)$ with weight w_i

Proof. **Reduction procedure:**

1. **Graph construction:** Build detector graph G from $\boldsymbol{H}$ as defined above. The boundary operator $\partial : \mathbb{F}_2^n \to \mathbb{F}_2^{m+1}$ maps edge vectors to vertex vectors, corresponding to the parity check matrix.
2. **Syndrome to boundary:** Given syndrome $\boldsymbol{s} \in \mathbb{F}_2^m$, identify the set $D \subseteq V$ of vertices with non-zero syndrome values. This becomes the boundary condition for the matching problem.
3. **Shortest path computation:** For all pairs (u, v) where $u, v \in D \cup \{0\}$, compute shortest paths using Dijkstra's algorithm. This requires $O(|D|^2)$ shortest path computations, constructing a complete weighted graph on $D \cup \{0\}$.
4. **MWPM with boundary:** Solve MWPM on the complete graph with boundary vertex $\{0\}$. The solution gives edges whose boundary equals D, which corresponds to the minimum weight error pattern satisfying $\boldsymbol{H}\boldsymbol{c} = \boldsymbol{s}$.

Since each step is polynomial time, MLD reduces to MWPM in polynomial time. □

6.2 The Matching Polytope

Definition. Weighted Perfect Matching: Given weighted graph $G = (V, E, W)$ where $W = \{w_e \in \mathbb{R} \mid e \in E\}$:

- A **matching** $M \subseteq E$ has no two edges sharing a vertex
- A **perfect matching** covers every vertex in V

- The **weight** of matching M is $\sum_{e \in M} w_e$

The integer programming formulation uses indicator variables $x_e \in \{0, 1\}$:

$$\min_{\boldsymbol{x}} \sum_{e \in E} w_e x_e \quad \text{subject to} \quad \sum_{e \in \delta(\{v\})} x_e = 1 \ \forall v \in V, \quad x_e \in \{0, 1\} \tag{72}$$

where $\delta(\{v\})$ denotes edges incident to vertex v.

Theorem (Matching Polytope Characterization). Define the odd set family $\mathcal{O}(G) = \{U \subseteq V : |U| \text{ is odd and } \geq 3\}$. Let:

$$\begin{aligned} P_1(G) &= \text{conv}\left\{ \boldsymbol{x} : x_e \in \{0, 1\}, \sum_{e \in \delta(\{v\})} x_e = 1 \ \forall v \in V \right\} \\ P_2(G) &= \left\{ \boldsymbol{x} : x_e \geq 0, \sum_{e \in \delta(\{v\})} x_e = 1 \ \forall v \in V, \sum_{e \in \delta(U)} x_e \geq 1 \ \forall U \in \mathcal{O}(G) \right\} \end{aligned} \tag{73}$$

If edge weights are rational, then $P_1(G) = P_2(G)$.

Implication: The integer program can be relaxed to a linear program by replacing $x_e \in \{0, 1\}$ with $x_e \geq 0$ and adding the **blossom constraints** $\sum_{e \in \delta(U)} x_e \geq 1$ for all odd sets U.

Key Point. Why blossom constraints matter: An odd set U cannot have a perfect matching using only internal edges (odd number of vertices). Therefore, at least one edge must connect to the outside: $\sum_{e \in \delta(U)} x_e \geq 1$. This constraint is necessary and sufficient for the convex hull to equal the integer hull.

6.3 Dual Formulation and Optimality Conditions

Definition. MWPM Dual Problem: The dual of the MWPM linear program is:

$$\max_{\boldsymbol{y}} \sum_{v \in V} y_v + \sum_{O \in \mathcal{O}(G)} y_O \tag{74}$$

subject to:

$$\begin{gathered} \lambda_e = w_e - \left(y_{v_1} + y_{v_2}\right) - \sum_{O : e \in \delta(O)} y_O \geq 0 \quad \forall e \in E \\ y_O \geq 0 \quad \forall O \in \mathcal{O}(G) \end{gathered} \tag{75}$$

where λ_e is the **slack** of edge e.

Theorem (KKT Complementary Slackness). Primal solution $\boldsymbol{x}$ and dual solution $(\boldsymbol{y}, \{y_O\})$ are optimal if and only if:

1. **Primal feasibility:** $\sum_{e \in \delta(\{v\})} x_e = 1$, $\sum_{e \in \delta(U)} x_e \geq 1$, $x_e \geq 0$
2. **Dual feasibility:** $\lambda_e \geq 0$, $y_O \geq 0$
3. **Complementary slackness:**
   - $\lambda_e x_e = 0$ (tight edges are in matching)

- $y_O\left(\sum_{e\in\delta(O)} x_e - 1\right) = 0$ (tight odd sets have positive dual)

6.4 The Blossom Algorithm

The Blossom algorithm, developed by Edmonds (1965), solves MWPM by maintaining primal and dual feasibility while growing alternating trees.

Definition. Alternating structures:

- **M-alternating walk:** Path $(v_0, v_1, ..., v_t)$ where edges alternate between M and $E \setminus M$
- **M-augmenting path:** M-alternating walk with both endpoints unmatched
- **M-blossom:** Odd-length cycle in an M-alternating walk where edges alternate in/out of M

Key Point. Algorithm overview:

1. **Initialization:** Start with empty matching $M = \emptyset$, dual variables $y_v = 0$
2. **Main loop:** While M is not perfect:
   - **Search:** Find M-alternating walks from unmatched vertices
   - **Augment:** If M-augmenting path found, flip edges along path (add unmatched edges, remove matched edges)
   - **Shrink:** If M-blossom found, contract it to a single vertex, update dual variables
   - **Grow:** If no path/blossom found, increase dual variables to make new edges tight, add to search tree
   - **Expand:** When blossom dual variable reaches zero, uncontract it
3. **Termination:** When all vertices are matched, return M

Theorem (Blossom Algorithm Correctness and Complexity). The Blossom algorithm:
1. Maintains primal feasibility, dual feasibility, and complementary slackness throughout
2. Terminates with an optimal MWPM
3. Runs in $O(|V|^3)$ time with careful implementation

Iteration bounds:

- Augmentations: at most $|V|/2$
- Contractions: at most $2|V|$
- Expansions: at most $2|V|$
- Edge additions: at most $3|V|$
- Total: $O(|V|^2)$ iterations, each taking $O(|V|)$ time

Key Point. Why MWPM for quantum codes:

- Surface codes and other topological codes have parity check matrices where each error triggers at most 2 stabilizers
- This structure allows efficient MLD via MWPM
- Blossom algorithm provides polynomial-time optimal decoding
- Practical implementations achieve near-optimal thresholds (10-11% for surface codes)

- Contrast with BP: MWPM finds global optimum but is slower; BP is faster but can get trapped in local minima

7 Results and Performance

7.1 Error Threshold

Definition. The **threshold** p_{th} is the maximum error rate below which the logical error rate decreases with increasing code distance.

- If $p < p_{\text{th}}$: Larger codes $\rightarrow$ exponentially better protection
- If $p > p_{\text{th}}$: Larger codes $\rightarrow$ worse protection (error correction fails)

7.2 Experimental Results

Code Family	BP Only	BP+OSD-0	BP+OSD-CS
Toric	N/A (fails)	$9.2 \pm 0.2\%$	$\mathbf{9.9 \pm 0.2}\%$
Semi-topological	N/A (fails)	$9.1 \pm 0.2\%$	$\mathbf{9.7 \pm 0.2}\%$
Random QLDPC	$6.5 \pm 0.1\%$	$6.7 \pm 0.1\%$	$\mathbf{7.1 \pm 0.1}\%$

Table 13: Observed thresholds from the paper

Key Results for Toric Code

- **BP alone:** Complete failure due to degeneracy (no threshold)
- **BP+OSD-CS:** 9.9% threshold (optimal decoder achieves 10.3%)
- **Improvement:** Combination sweep gains 0.7% over OSD-0
- **Low-error regime:** Exponential suppression of logical errors

7.3 Complexity

Component	Complexity	Notes
BP (per iteration)	$O(n)$	Linear in block length
OSD-0	$O(n^3)$	Dominated by matrix inversion
Combination sweep	$O(\lambda^2)$	$\lambda = 60 \rightarrow$ 1830 trials
Total	$O(n^3)$	Practical for moderate n

Table 14: Complexity analysis

8 Tropical Tensor Network

In this section, we introduce a complementary approach to decoding: **tropical tensor networks**. While BP+OSD performs approximate inference followed by algebraic post-processing, tropical tensor networks provide a framework for **exact** maximum a posteriori (MAP) inference by reformulating the problem in terms of tropical algebra.

The key insight is that finding the most probable error configuration corresponds to an optimization problem that can be solved exactly using tensor network contractions in the tropical semiring. This approach is particularly powerful for structured codes where the underlying factor graph has bounded treewidth.

8.1 Tropical Semiring

Definition. The **tropical semiring** (also called the **max-plus algebra**) is the algebraic structure $(\mathbb{R} \cup \{-\infty\}, \oplus, \otimes)$ where:

- **Tropical addition**: $a \oplus b = \max(a, b)$
- **Tropical multiplication**: $a \otimes b = a + b$ (ordinary addition)
- **Additive identity**: $-\infty$ (since $\max(a, -\infty) = a$)
- **Multiplicative identity**: 0 (since $a + 0 = a$)

Key Point. The tropical semiring satisfies all semiring axioms:

- Associativity: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
- Commutativity: $a \oplus b = b \oplus a$
- Distributivity: $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$

This algebraic structure allows us to replace standard summation with maximization while preserving the correctness of tensor contractions.

The tropical semiring was first systematically studied in the context of automata theory and formal languages [18]. Its connection to optimization problems makes it particularly useful for decoding applications.

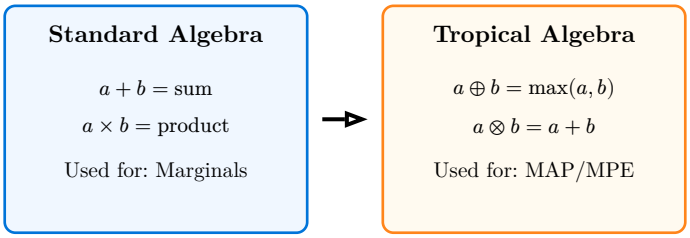


Figure 21: Standard algebra vs tropical algebra: switching the algebraic structure transforms marginalization into optimization

8.2 From Probabilistic Inference to Tropical Algebra

Recall that the MAP (Maximum A Posteriori) decoding problem seeks:

$$\boldsymbol{e}^* = \arg \max_{\boldsymbol{e}: H\boldsymbol{e}=\boldsymbol{s}} P(\boldsymbol{e}) \tag{76}$$

For independent bit-flip errors with probability p, the probability factors as:

$$P(\boldsymbol{e}) = \prod_{i=1}^{n} P(e_i) = \prod_{i=1}^{n} p^{e_i}(1-p)^{1-e_i} \tag{77}$$

Taking the logarithm transforms products into sums:

$$\log P(\boldsymbol{e}) = \sum_{i=1}^{n} \log P(e_i) = \sum_{i=1}^{n} [e_i \log p + (1 - e_i) \log(1 - p)] \tag{78}$$

Key Point. In the log-probability domain:

- **Products become sums**: $\log(P \cdot Q) = \log P + \log Q$
- **Maximization is preserved**: $\arg\max_x f(x) = \arg\max_x \log f(x)$

This means finding the MAP estimate for a function $\prod_f \varphi_f(\boldsymbol{e}_f)$ is equivalent to:

$$\boldsymbol{e}^* = \arg \max_{\boldsymbol{e}: H\boldsymbol{e}=\boldsymbol{s}} \sum_f \log \varphi_f(\boldsymbol{e}_f) \tag{79}$$

where each factor φ_f contributes additively in log-space.

The connection to tropical algebra becomes clear: if we replace standard tensor contractions (sum over products) with tropical contractions (max over sums), we transform marginal probability computation into MAP computation [2].

Operation	**Standard (Marginals)**	**Tropical (MAP)**
Combine factors	$\varphi_a \cdot \varphi_b$	$\log \varphi_a + \log \varphi_b$
Eliminate variable	$\sum_x$	$\max_x$
Result	Partition function Z	Max log-probability

Table 15: Correspondence between standard and tropical tensor operations

Example: Consider a simple Markov chain with three binary variables $x_1, x_2, x_3 \in \{0, 1\}$ and two factors:

$$P(x_1, x_2, x_3) = \varphi_1(x_1, x_2) \cdot \varphi_2(x_2, x_3) \tag{80}$$

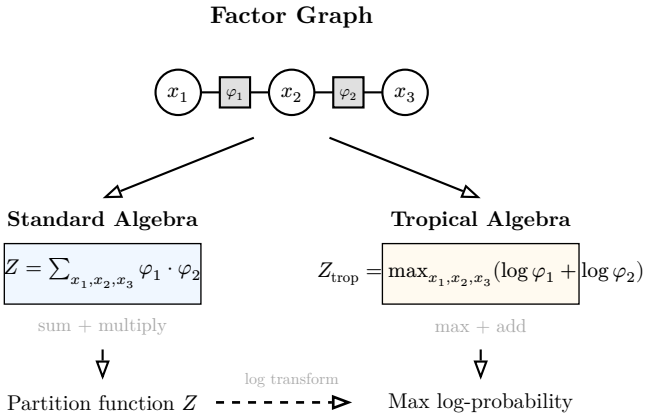


Figure 22: Standard vs tropical contraction of a Markov chain. The same factor graph structure supports both marginal computation (standard algebra) and MAP inference (tropical algebra).

The partition function in standard algebra sums over all configurations:

$$Z = \sum_{x_1, x_2, x_3} \varphi_1(x_1, x_2) \cdot \varphi_2(x_2, x_3) \tag{81}$$

The same structure in tropical algebra computes the maximum log-probability:

$$Z_{\text{trop}} = \max_{x_1,x_2,x_3} [\log \varphi_1(x_1, x_2) + \log \varphi_2(x_2, x_3)] \tag{82}$$

Key Point. Beyond a Change of Language [19]: Tropical tensor networks provide computational capabilities unavailable in traditional approaches:

1. **Automatic Differentiation for Configuration Recovery**: Backpropagating through tropical contraction yields gradient "masks" that directly identify optimal variable assignments $\boldsymbol{x}^*$—no separate search phase is needed.
2. **Degeneracy Counting via Mixed Algebras**: By tracking (Z_{trop}, n) where n counts multiplicities, one simultaneously finds the optimal value AND counts all solutions achieving it in a single contraction pass.
3. **GPU-Accelerated Tropical BLAS**: Tropical matrix multiplication maps to highly optimized GPU kernels, enabling exact ground states for 1024-spin Ising models and 512-qubit D-Wave graphs in under 100 seconds.

8.3 Tensor Network Representation

A tensor network represents the factorized probability dis tribution as a graph where nodes of tensors correspond to factors φ_f and the edges of correspond to functions that contract the variables.

Definition. Given a factor graph with factors $\{\varphi_f\}$ and variables $\{x_i\}$, the corresponding **tensor network** consists of:

- A tensor T_f for each factor, with indices corresponding to the variables in φ_f
- The **contraction** of the network computes: $\sum_{x_1,\ldots,x_n} \prod_f T_f(\boldsymbol{x}_f)$

In the tropical semiring, this becomes: $\max_{x_1,\ldots,x_n} \sum_f T_f(\boldsymbol{x}_f)$

The efficiency of tensor network contraction depends critically on the **contraction order**—the sequence in which variables are eliminated.

Key Point. The **treewidth** of the factor graph determines the computational complexity:

- A contraction order exists with complexity $O(n \cdot d^{w+1})$ where w is the treewidth
- For sparse graphs (like LDPC codes), treewidth can be small, enabling efficient exact inference
- Tools like `omeco` find near-optimal contraction orders using greedy heuristics

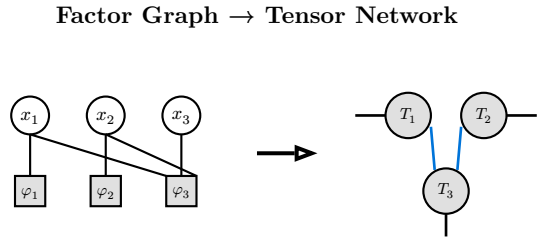


Figure 23: Factor graph representation as a tensor network. Edges between tensors represent indices to be contracted (summed/maximized over).

The contraction process proceeds by repeatedly selecting a variable to eliminate:

```
# Conceptual contraction loop (simplified)
for var in elimination_order:
    bucket = [tensor for tensor in tensors if var in tensor.indices]
    combined = tropical_contract(bucket, eliminate=var)
    tensors.update(combined)
```

8.4 Backpointer Tracking for MPE Recovery

A critical challenge with tensor network contraction is that it only computes the **value** of the optimal solution (the maximum log-probability), not the **assignment** that achieves it.

> **Definition.** A **backpointer** is a data structure that records, for each max operation during contraction:
> - The indices of eliminated variables
> - The arg max value for each output configuration
>
> Formally, when computing $\max_x T(y, x)$, we store: $\text{bp}(y) = \arg\max_x T(y, x)$

The recovery algorithm traverses the contraction tree in reverse:

Contraction Tree with Backpointers

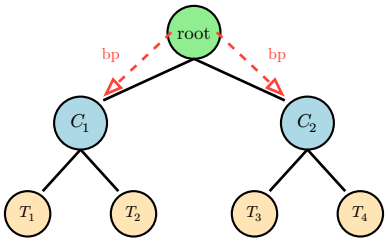


Figure 24: Contraction tree with backpointers. During contraction (bottom-up), backpointers record argmax indices. During recovery (top-down, dashed arrows), backpointers are traced to reconstruct the optimal assignment.

The implementation in the `tropical_in_new/` module demonstrates this pattern:

```
# From tropical_in_new/src/primitives.py
@dataclass
class Backpointer:
    """Stores argmax metadata for eliminated variables."""
    elim_vars: Tuple[int, ...]      # Which variables were eliminated
    elim_shape: Tuple[int, ...]     # Domain sizes
    out_vars: Tuple[int, ...]       # Remaining output variables
    argmax_flat: torch.Tensor       # Flattened argmax indices

def tropical_reduce_max(tensor, vars, elim_vars, track_argmax=True):
    """Tropical max-reduction with optional backpointer tracking."""
    # ... reshape tensor to separate kept and eliminated dimensions ...
    values, argmax_flat = torch.max(flat, dim=-1)
    if track_argmax:
        backpointer = Backpointer(elim_vars, elim_shape, out_vars, argmax_flat)
    return values, backpointer
```

The recovery algorithm traverses the tree from root to leaves:

```
# From tropical_in_new/src/mpe.py
def recover_mpe_assignment(root) -> Dict[int, int]:
    """Recover MPE assignment from a contraction tree with backpointers."""
```

```
    assignment: Dict[int, int] = {}

    def traverse(node, out_assignment):
        assignment.update(out_assignment)
        if isinstance(node, ReduceNode):
            # Use backpointer to recover eliminated variable values
            elim_assignment = argmax_trace(node.backpointer, out_assignment)
            child_assignment = {**out_assignment, **elim_assignment}
            traverse(node.child, child_assignment)
        elif isinstance(node, ContractNode):
            # Propagate to both children
            elim_assignment = argmax_trace(node.backpointer, out_assignment)
            combined = {**out_assignment, **elim_assignment}
            traverse(node.left, {v: combined[v] for v in node.left.vars})
            traverse(node.right, {v: combined[v] for v in node.right.vars})

    # Start from root with initial assignment from final tensor
    initial = unravel_argmax(root.values, root.vars)
    traverse(root, initial)
    return assignment
```

8.5 Application to Error Correction Decoding

For quantum error correction, the MAP decoding problem is:

$$\boldsymbol{e}^* = \arg\max_{\boldsymbol{e}:H\boldsymbol{e}=\boldsymbol{s}} P(\boldsymbol{e}) \tag{83}$$

The syndrome constraint $H\boldsymbol{e} = \boldsymbol{s}$ can be incorporated as hard constraints (factors that are $-\infty$ for invalid configurations and 0 otherwise) [20].

Aspect	**BP+OSD**	**Tropical TN**
Inference type	Approximate marginals	Exact MAP
Degeneracy handling	OSD post-processing	Naturally finds one optimal
Output	Soft decisions $\to$ hard	Direct hard assignment
Complexity	$O(n^3)$ for OSD	Exp. in treewidth
Parallelism	Iterative	Highly parallelizable

Table 16: Comparison of BP+OSD and tropical tensor network decoding approaches

Key Point. Advantages of tropical tensor networks for decoding:

- **Exactness**: Guaranteed to find the MAP solution (no local minima)
- **No iterations**: Single forward pass plus backtracking
- **Natural for structured codes**: Exploits graph structure via contraction ordering

Limitations:

- Complexity grows exponentially with treewidth
- For dense or high-treewidth codes, may be less efficient than BP+OSD
- Requires careful implementation of backpointer tracking

The tensor network approach is particularly well-suited to codes with local structure, such as topological codes where the treewidth grows slowly with system size [21].

8.6 Complexity Considerations

The computational complexity of tropical tensor network contraction is governed by the **treewidth** of the underlying factor graph.

Definition. The **treewidth** w of a graph is the minimum width of any tree decomposition, where width is one less than the size of the largest bag. Intuitively, it measures how "tree-like" the graph is.

Code Type	Treewidth	Contraction Complexity
1D repetition	$O(1)$	$O(n)$
2D toric	$O(\sqrt{n})$	$O\left(n \cdot 2^{\sqrt{n}}\right)$
LDPC (sparse)	$O(\log n)$ to $O(\sqrt{n})$	Varies
Dense codes	$O(n)$	$O(2^n)$ – intractable

Table 17: Treewidth and complexity for different code families

Key Point. For LDPC codes used in quantum error correction:

- The sparse parity check matrix leads to bounded-degree factor graphs
- Greedy contraction order heuristics (like those in `omeco`) often find good orderings
- The practical complexity is often much better than worst-case bounds suggest

The tropical tensor network approach provides a systematic way to exploit code structure for efficient exact decoding when the treewidth permits.

9 Summary

9.1 Key Takeaways

1. **Classical BP** computes marginal probabilities via message passing on factor graphs
2. **Quantum codes suffer from degeneracy**: multiple errors can produce the same syndrome, causing BP to output invalid solutions (split beliefs)
3. **OSD resolves degeneracy** by selecting a basis guided by BP soft decisions, then solving via matrix inversion to get a unique valid solution
4. **Combination sweep** efficiently improves OSD-0 by testing low-weight configurations of the remainder bits
5. **BP+OSD is general**: works for Toric codes, semi-topological codes, and random QLDPC codes, achieving near-optimal thresholds

9.2 The BP+OSD Recipe

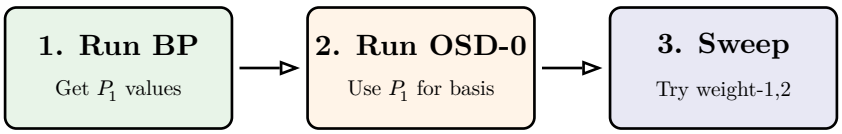


Figure 25: BP+OSD in three steps

10 References

Bibliography

[1] D. J. MacKay, *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

[2] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[3] T. Richardson and R. Urbanke, "The Capacity of Low-Density Parity-Check Codes Under Message-Passing Decoding," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 599–618, 2001.

[4] J. S. Yedidia, W. T. Freeman, and Y. Weiss, "Understanding Belief Propagation and Its Generalizations," *Exploring Artificial Intelligence in the New Millennium*. pp. 239–269, 2003.

[5] L. Dolecek, Z. Zhang, V. Anantharam, M. J. Wainwright, and B. Nikolić, "Analysis of Absorbing Sets and Fully Absorbing Sets of Array-Based LDPC Codes," *IEEE Transactions on Information Theory*, vol. 56, no. 1, pp. 181–201, 2010.

[6] J. Chen, A. Dholakia, E. Eleftheriou, M. P. Fossorier, and X.-Y. Hu, "Reduced-Complexity Decoding of LDPC Codes," *IEEE Transactions on Communications*, vol. 53, no. 8, pp. 1288–1299, 2005.

[7] M. P. Fossorier and S. Lin, "Soft-Decision Decoding of Linear Block Codes Based on Ordered Statistics," *IEEE Transactions on Information Theory*, vol. 41, no. 5, pp. 1379–1396, 1995.

[8] M. P. Fossorier and S. Lin, "Computationally Efficient Soft-Decision Decoding of Linear Block Codes Based on Ordered Statistics," *IEEE Transactions on Information Theory*, vol. 42, no. 3, pp. 738–750, 1996.

[9] J. Roffe, D. R. White, S. Burton, and E. Campbell, "Decoding Across the Quantum Low-Density Parity-Check Code Landscape," *Physical Review Research*, vol. 2, no. 4, p. 43423, 2020.

[10] R. W. Hamming, "Error Detecting and Error Correcting Codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[11] G. D. Forney, "Generalized Minimum Distance Decoding," *IEEE Transactions on Information Theory*, vol. 12, no. 2, pp. 125–131, 1966.

[12] J. Hagenauer, E. Offer, and L. Papke, "Iterative Decoding of Binary Block and Convolutional Codes," *IEEE Transactions on Information Theory*, vol. 42, no. 2, pp. 429–445, 1996.

[13] C. Yue, M. Shirvanimoghaddam, Y. Li, and B. Vucetic, "A Revisit to Ordered Statistics Decoding: Distance Distribution and Decoding Rules," *IEEE Transactions on Information Theory*, vol. 67, no. 7, pp. 4288–4337, 2021.

[14] A. Montanari, "Belief Propagation." [Online]. Available: https://web.stanford.edu/~montanar/RESEARCH/BOOK/partD.pdf

[15] T. Richardson and R. Urbanke, *Modern Coding Theory*. Cambridge University Press, 2008.

[16] S. Tatikonda and M. I. Jordan, "Loopy Belief Propagation and Gibbs Measures," in *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, 2002, pp. 493–500.

[17] A. T. Ihler, J. W. Fisher III, and A. S. Willsky, "Loopy Belief Propagation: Convergence and Effects of Message Errors," *Journal of Machine Learning Research*, vol. 6, pp. 905–936, 2005.

[18] J.-É. Pin, "Tropical Semirings," *Idempotency*. Cambridge University Press, pp. 50–69, 1998.

[19] J.-G. Liu, L. Wang, and P. Zhang, "Tropical Tensor Network for Ground States of Spin Glasses," *Physical Review Letters*, vol. 126, no. 9, p. 90506, 2021.

[20] T. Farrelly, R. J. Harris, N. A. McMahon, and T. M. Stace, "Parallel Decoding of Multiple Logical Qubits in Tensor-Network Codes," *arXiv preprint arXiv:2012.07317*, 2020.

[21] R. Orús, "Tensor Networks for Complex Quantum Systems," *Nature Reviews Physics*, vol. 1, no. 9, pp. 538–550, 2019.

End of Lecture Note