A Step-by-Step Introduction to Error-Correcting Codes: From Hamming to Low-Density Parity-Check (LDPC)

Pedro Cosme da Costa Vieira (pedro.cosme.vieira@gmail.com)

PENTAGON, Portugal

April 27, 2026

Abstract: This work presents an introduction to error-correcting codes, starting from Hamming codes and progressing toward Low-Density Parity-Check (LDPC) codes as a natural generalization. Hamming codes are used to establish the fundamental concepts of parity-check matrices, syndrome computation, and error localization, providing a clear and accessible bridge to more advanced coding schemes. The LDPC framework is then explored through its sparse parity-check structure and iterative decoding algorithms. A simplified bit-flipping approach is presented to illustrate the core principles of LDPC decoding, including the evaluation of bit reliability based on the number of unsatisfied parity-check equations. For clarity and stability, a didactic version of the algorithm is considered, in which only the most likely erroneous bit is corrected at each iteration. The computational efficiency of LDPC decoding is highlighted, as it relies primarily on simple binary operations such as XOR, making it suitable for high-performance hardware implementations. The discussion is further extended to modern storage systems, such as solid-state drives (SSDs), where data is stored in multi-level cells with inherently analog voltage levels. In this context, the use of reliability information is introduced, showing how LDPC decoding can be enhanced by weighting the contribution of parity-check equations according to the confidence of the underlying measurements. Overall, this work provides a conceptual and practical overview of LDPC codes, emphasizing their connection to classical error-correcting methods and their relevance in contemporary high-density storage and communication systems.

Keywords: Error-correcting codes, Hamming codes, LDPC codes, Bit-flipping decoding, Soft-decision decoding.

The writing of this text received substantial assistance from ChatGPT 5.

1. Introduction

Storage, reading, and transmission of information are always affected by noise. In any physical system, signals can be distorted during storage or transmission, leading to errors in the received data.

When information is analogue, meaning it can take any value in a continuous range, error correction is fundamentally difficult. Small perturbations continuously shift the signal, and there is no clear boundary that allows an unambiguous decision about the original value. In contrast, modern digital systems encode

information in a discrete form, typically using binary symbols (0 and 1). This discretisation creates clear decision boundaries between valid states.

Once information is represented in binary form, it becomes possible to periodically sample and interpret the signal in a consistent way, preventing small analogue fluctuations from accumulating continuously over time. For example, if information is transmitted as voltage oscillations between -1V and +1V, it is possible to interpret the signal using a decision threshold. A simple rule may assign any voltage below 0V to the binary value 0, and any voltage above 0V to the binary value 1.

In practical systems, there is a constant trade-off between performance, cost, and reliability. Increasing data throughput, increasing storage density, and reducing the amount of redundancy all contribute to lowering the cost per bit. However, these improvements come at a price: the system becomes more sensitive to noise and physical imperfections. As a result, bit errors — commonly referred to as bit flips — become more likely. In this case, after thresholding the received signal, a logical 1 may be interpreted when the original transmitted value was 0, or vice versa. These errors accumulate unless additional mechanisms are introduced to detect and correct them.

To correct these logical errors, it is necessary to introduce redundant information into the original data. Conceptually, the simplest approach is to repeat each bit multiple times and decide its value by majority voting. For example, each bit can be transmitted three times, and the receiver assigns the final value based on the majority of the received copies. While this method is easy to understand and implement, it is highly inefficient. It introduces a large amount of redundancy, significantly increasing the amount of transmitted data without providing proportional gains in efficiency. For this reason, more advanced coding schemes are used in practice, which aim to achieve the same level of error protection while using far fewer redundant bits.

There are two main strategies for reducing the amount of redundant information while still enabling reliable error detection and correction.

The first approach consists of constructing parity-check equations over subsets of bits. By carefully designing these relationships, it becomes possible to detect and locate errors based on inconsistencies in the parity constraints, and in some cases even correct them. A classical example of this approach is the Hamming (1950) code, where parity bits are arranged in a structured way to enable single-bit error correction.

The second approach is based on algebraic coding over finite fields. In this case, the original data is interpreted as coefficients of a polynomial, and additional redundant information is generated by evaluating this polynomial at extra points. These redundant evaluations allow the reconstruction of the original polynomial even when some of the transmitted symbols are corrupted or lost. This is the principle behind Reed–Solomon (1960) codes.

These strategies have been previously introduced in Vieira (2026).

Low-Density Parity-Check (LDPC) codes, Gallager (1962), can be seen as a natural evolution of earlier linear block codes based on parity-check constraints, such as Hamming codes. Although LDPC codes were originally introduced in the 1960s, they only became practically relevant from the 1990s onwards, when advances in computational power made iterative decoding feasible.

Like Hamming codes, LDPC codes are defined through parity-check equations. However, instead of operating on small, tightly constrained blocks that yield exact algebraic corrections, LDPC codes operate on large blocks and rely on probabilistic iterative decoding.

In a simple Hamming code, for example, a block of 64 bits with 8 parity bits can typically guarantee the correction of a single-bit error through exact syndrome decoding. In contrast, LDPC codes may operate on blocks of thousands of bits (*e.g.*, 4096 bits with several hundred and even thousand parity bits). Error correction is not deterministic in the same sense as Hamming codes; instead, the decoder iteratively exchanges probabilistic information across a sparse parity-check graph. This allows the correction of a large number of errors with high probability, with performance improving as the noise level decreases.

Because of its efficiency and near-optimal performance close to the Shannon (1948) limit, LDPC has become one of the dominant error-correction schemes in modern digital systems. It is widely used in wireless communication standards such as Wi-Fi and 5G, as well as in data storage technologies, including modern hard disk drives (HDDs) and solid-state drives (SSDs).

Its success comes from the combination of strong error-correction capability, scalability to very large block sizes, and efficient hardware implementations, making it suitable for both high-speed communication and high-density storage systems.

2. From Hamming Codes to LDPC: Structured Parity and Scalable Error Correction

Hamming codes represent one of the earliest systematic approaches to error detection and correction in digital systems. Their central idea is to introduce redundancy in a structured way, allowing the receiver to not only detect that an error has occurred, but also to identify its exact position within a block.

In a classical Hamming construction, parity bits are inserted at positions that are powers of two (1, 2, 4, 8, …). These positions are reserved for parity, while the remaining positions carry the original data bits. The value of each parity bit is computed based on a specific subset of positions in the block, defined by the binary representation of the index of each bit.

To illustrate the principle, consider a small example with 8 data bits:

Data = 11111111.

We construct a 12-bit Hamming codeword (Hamming(12,8)) by inserting parity bits at positions 1, 2, 4 and 8, leading to a 12-bit codeword structured as 001011101111.

Positions 1, 2, 4 and 8 → parity bits
Positions 3, 5, 6, 7, 9, 10, 11 and 12 → data bits

```
def code_word(word):
    "Insert Hamming parity bits (initialized to 0)"
    n = len(word)
    # Determine number of parity bits needed
    r = 0
    while (2 ** r) < (n + r + 1):
        r += 1
    # Create result list with space for parity bits
```

```
    res = []
    j = 0  # index for original message
    for i in range(1, n + r + 1):
        if (i & (i - 1)) == 0:
            res.append(0)            # Position is power of 2 -> parity bit
        else:
            res.append(word[j])
            j += 1
    return res

word =[1,1,1,1,1,1,1,1]
print("codeword:",code_word(word))
```

```
codeword: [0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1]
```

Each parity bit is responsible for checking a subset of positions. These subsets are defined using the binary representation of the position index: a bit is included in a parity group if the corresponding binary digit is equal to 1.

Parity bit 1 → bit1 ⊕ bit 3 ⊕ bit 5 ⊕ bit 7 ⊕ bit 9 ⊕ bit 11 , = 1
Parity bit 2 → bit 2 ⊕ bit 3 ⊕ bit 6 ⊕ bit 7 ⊕ bit 10 ⊕ bit 11 , = 1
Parity bit 3 → bit 4 ⊕ bit 5 ⊕ bit 6 ⊕ bit 7 ⊕ bit 12 , = 0
Parity bit 4 → bit 8 ⊕ bit 9 ⊕ bit 10 ⊕ bit 11 ⊕ bit 12 , = 0

This structure ensures that any single-bit error produces a unique combination of parity violations, known as a syndrome, which directly identifies the position of the erroneous bit. As any single-bit error produces a unique combination of parity violations, it is possible to directly identify the position of the erroneous bit.

2.1 Matrix Representation of Hamming Codes

Hamming codes can be reformulated using linear algebra over the finite field GF(2), where addition corresponds to the XOR operation and multiplication corresponds to the AND operation.

In this representation, a Hamming code is defined through a parity-check matrix H, which encodes all parity constraints of the system. A valid codeword x must satisfy:

$$H \cdot x = 0$$

where all operations are performed in GF(2).

Each column of the matrix H corresponds to a bit position in the codeword, while each row represents a parity-check equation. A value of 1 in the matrix indicates that a given bit participates in a specific parity constraint. For example, in the Hamming (8, 4) code example, the H matrix is:

H = [[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
[0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0],
[0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1],
[0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]]

```
def generate_H_Hamming(n):
    "Generate Hamming parity-check matrix H (r x n), n: codeword length"
```

```
    # Determine number of parity bits
    r = 0
    while (2 ** r) < (n + 1):
        r += 1
    H = []
    # Each row corresponds to one parity bit
    for i in range(r):
        row = []
        for j in range(1, n + 1):
            # Take i-th bit of j (LSB = row 0)
            bit = (j >> i) & 1
            row.append(bit)
        H.append(row)
    return H

H = generate_H_Hamming(12)
print("H_Hamming:",H)
```

```
H_Hamming: [[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0], [0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0],
[0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1], [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]]
```

```
def sum_gf2(a,b): # XOR
    return a^b

def mult_gf2(a,b): # AND
    return a&b

def mult_matrix_gf2(H, X):
    """
    Multiply matrix H by vector X in GF(2)
    H: list of lists (rows)
    X: list (vector)
    """
    result = []
    for row in H:
        s = 0
        for a, b in zip(row, X):
            s = sum_gf2(s, mult_gf2(a, b))
        result.append(s)
    return result
```

Now, it is easy to compute the values of the parity-bits:

```
word=[1,1,1,1,1,1,1,1]
codeword = code_word(word)
parity_bits = mult_matrix_gf2(H, codeword)
print(' parity_bits:', parity_bits)
```

```
parity_bits: [1, 1, 0, 0]
```

```
def insert_parity_bits(codeword, syndrome):
    "Insert syndrome bits into parity positions (powers of 2)"
    res = codeword[:]
    j = 0  # index for syndrome
    for i in range(1, len(res) + 1):
        if (i & (i - 1)) == 0:
            res[i - 1] = syndrome[j]
            j += 1
    return res
```

```
#Verify
Codeword = insert_parity_bits(codeword, parity_bits)
print("codeword =", codeword)
print("syndrome =", mult_matrix_gf2(H, codeword))

codeword = [1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1]
syndrome = [0, 0, 0, 0]
```

We can now verify and correct a maximum of one error.

```
def correct_codeword(codeword, H):
    "Detect and correct a single-bit error in a Hamming codeword"
    # Compute syndrome
    syndrome = mult_matrix_gf2(H, codeword)
    # If syndrome is all zeros -> no error
    if sum(syndrome) == 0:
        return codeword, 0  # no correction needed
    # Convert syndrome (LSB first) to position
    error_pos = 0
    for i, bit in enumerate(syndrome):
        error_pos += bit * (2 ** i)
    # Flip the erroneous bit (1-based index)
    corrected = codeword[:] # Copy codeword
    idx = error_pos - 1
    corrected[idx] = 1 if corrected[idx] == 0 else 0
    return corrected, error_pos

#Aplication
word_64=[1,1,0,0,1,1,1,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,1,1,1,1,0,0,0,0,1,1
,1,1,1,1,0,0,0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,0,0,0]
codeword = code_word(word_64)
H = generate_H_Hamming(len(codeword))
syndrome = mult_matrix_gf2(H, codeword)
codeword=insert_parity_bits(codeword, syndrome)
codeword_error = codeword[:]
codeword_error[37] = 1 - codeword_error[37] # flip one bit
x, y = correct_codeword(codeword_error, H)
print(f"Codeword is{'' if x==codeword else 'not'} correct, error position: {y}")

Codeword is correct, error position: 38
```

3. From Dense to Sparse Matrices: The Origin of LDPC

In contrast to Hamming codes, Low-Density Parity-Check (LDPC) codes operate on much larger blocks and introduce a significantly higher level of redundancy. For example, an LDPC code may use 512 parity bits to protect a 4096-bit (4 Kib) data block. This corresponds to a redundancy ratio of 1/9, which is how LDPC redundancy is usually expressed. In this example, the physical sector contains 4608 bits in total, of which only 4096 bits are visible to and usable by the system, the remainder being dedicated to error correction.

Unlike Hamming codes, LDPC codes do not rely on fixed positional rules for parity bits. Instead, the structure of the code is fully defined by the parity-check matrix H, which determines how data and parity bits are related. While, in principle, bits may be placed at the beginning, at the end, or interleaved, both the encoder and the decoder must share a consistent understanding of this structure through H. In practice, simple layouts — such as grouping parity bits at the end — are often adopted for implementation convenience.

The essential structure of LDPC codes is defined by the parity-check matrix H, where each row represents a parity-check equation and each column corresponds to a bit in the codeword. A value of 1 in the matrix indicates that a given bit participates in a given parity constraint. To ensure useful redundancy, each bit must participate in multiple parity-check equations, and each equation must involve multiple bits. For example, in a system with n = 4096 + 512 = 4608-bit and m = 512 parity-check equations, one may design a regular LDPC structure where each bit participates in dv = 25 equations. This implies that each parity-check equation involves approximately 4608*25/512 = 225 bits chosen from the 4608 bits, a ration 1/0 of approximately 5%.

$$dc = (bits+m) \cdot dv / m,$$

where dc is the number of bits in each parity-check equations, dv is the number of parity-check equations where each bit appears, n = (message_bits + m) is the total number of bits (message + redundancy) and m is the number of bits of the redundancy.

With a redundancy of 1/9 in a 4608-bit codeword, it is not meaningful to define a strict upper bound on the number of correctable errors, such as “maximum of 512/8 errors”. Unlike block codes with bounded-distance decoding, LDPC codes rely on iterative, probabilistic decoding. As a result, their error-correction capability is characterized statistically rather than deterministically. In practice, the decoder can successfully correct a relatively large number of errors when their density is below a certain threshold, but the probability of decoding failure increases as the number of errors grows, which is typically expressed in terms of the bit error rate (BER). In practice, there are systems with unequal error protection (UEP) that use lower BER targets (*i.e.*, higher redundancy) in regions storing critical information, and higher BER targets (*lower redundancy*) in less critical areas.

Matrix H can be constructed in a pseudo-random manner, subject to structural constraints defined by parameters such as n, m, dv, and dc. In practical systems such as HDDs, SSDs or G5 protocol, LDPC matrices are not generated randomly during operation. Instead, a predefined set of parity-check matrices is implemented in the controller firmware. These matrices are fixed and shared by both encoding and decoding processes, although different matrices or configurations may be selected dynamically depending on operating conditions such as error rates or device aging.

3.1 - Encoder - Computing parity bits.

Computing parity bits is based on the parity-check matrix H, which contains one row per parity constraint. In this example, H has 512 rows (one for each parity-bit) and 4608 columns (one for each bit of the codeword) over the finite field GF(2). If the redundant bits are placed at the end of the codeword, the matrix can be partitioned as H = [Hd|Hp], where Hd corresponds to the data columns (512 × 4608) and Hp corresponds to the parity columns (512 × 512). For simplicity, we will assume that Hp is the identity matrix.

```
def split_H(H, k):
    "Split H into data and parity parts, k is the number of data bits"
    H_d = [row[:k] for row in H]
    H_p = [row[k:] for row in H]
    return H_d, H_p
```

Since the codeword c=[d|p] must satisfy $H.c^T=0$, it must also satisfy:

$$Hd \cdot d^T \oplus Hp \cdot p^T = 0 \quad \Leftrightarrow \quad Hd \cdot d^T = Hp \cdot p^T \quad \Leftrightarrow \quad p = Hp^{-1} \cdot (Hd \cdot d^T)$$

$\Rightarrow p = Hd \cdot d^T$, assuming $Hp = I$

The vector p defines the parity bits as a function of the data bits. This operation is computationally efficient and well suited for hardware implementation, as it relies on simple binary operations over GF(2), primarily XOR operations, which can be implemented with low complexity and high parallelism.

```
def compute_parity(H, data):
    "Compute parity bits assuming systematic form [data | parity], H = [A | I]"
    n = len(data)
    r = len(H)
    # Spliting H
    A = [row[:n] for row in H]
    parity = []
    for i in range(r):
        s = 0
        for j in range(n):
            s ^= int(A[i][j]) & int(data[j])
        parity.append(int(s))
    return parity

def code_word(word,parity):
    "include parity bits at the end"
    codeword = word[:]
    codeword.extend(parity)
    return codeword

#verify
H=[[1,0,1,1,1,0,1,1,1,0,0,0],
   [1,0,0,1,0,1,1,0,0,1,0,0],
   [0,1,0,0,0,0,1,1,0,0,1,0],
   [1,1,1,1,1,1,0,1,0,0,0,1]]
word= [1,0,1,1,0,0,1,0]
parity=compute_parity(H, word)
codeword = code_word(word,parity)
print("parity LDPC:",parity, "codeword:",codeword)
print("syndrome:",mult_matrix_gf2(H,codeword))
```

```
parity LDPC: [0, 1, 1, 1] codeword: [1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1]
syndrome: [0, 0, 0, 0]
```

3.2 - Decoder – Correcting errors.

Let cw_r be the received codeword (possibly corrupted by errors). The decoding process starts by computing the syndrome $S = H \cdot cw_r^T$. While a zero syndrome indicates that the received codeword, cw_r, satisfies all parity constraints, in practical communication or storage systems with 4608-bit blocks, the presence of noise makes it highly likely that errors have occurred. Therefore, LDPC decoding is typically performed as an iterative process, which progressively refines the estimate of the transmitted codeword.

In general, decoding requires solving a system of linear equations defined by $H \cdot cw_r^T = 0$, where the number of variables (the total number of bits, n) exceeds the number of equations (the number of redundant bits). This results in an underdetermined system with multiple valid solutions. However, these solutions are not equally likely: the correct codeword is expected to differ from cw_r in only a small number of positions, corresponding to the most probable error locations.

LDPC decoding is typically performed as an iterative process, which progressively refines the estimate of the solution of system of linear equations, *i.e.*, the transmitted codeword by identifying and correcting the bits that are most likely to be in error. This algorithm is commonly referred to as belief propagation (BP). The iterative decoding process begins by initializing the estimated codeword, cw_e, with the received codeword, cw_r:

$$cw_e = cw_r$$

The syndrome $S = H \cdot cw_e^T$ is then computed to identify which parity-check equations are violated.

Each non-zero component of S corresponds to an unsatisfied parity-check equation.

For each bit j, the number of unsatisfied parity-check equations in which it participates is counted and compared to the total number of checks connected to that bit. The resulting ratio provides a measure of how likely the bit is to be erroneous. For example, if a bit is involved in five parity checks and three of them are unsatisfied, the ratio is 3/5.

In practical implementations, multiple candidate bits may be flipped in each iteration, which is particularly important in high-density storage systems such as SSDs, where a block of 4608 bits may contain on the order of tens of errors. Attempting to correct only a single error per iteration would lead to slow convergence in such scenarios. However, for didactic purposes, and because it tends to provide more stable convergence behaviour, we will consider a simplified approach in which only the most likely erroneous bit is corrected at each iteration.

```
def bit_flipping_iteration(H, cw_e, syndrome):
    n = len(cw_e) #codeword number of bits
    r = len(H)    #parity number of bits / number of equations
    new_cw = cw_e[:]
    mx, k = 0, 0 # maximum probability
    for j in range(n): # for each bit in the codeword
        score = 0
        degree = 0
        for i in range(r): # see in all equations
            if H[i][j]: # ==1
                degree += 1
                score += syndrome[i] # may de used a soft-valued metric
        if score/degree > mx: # identifies the maximum probability
            mx=score/degree
            k = j
    new_cw[k] = 1 - new_cw[k] # flip the bit most probable of been a flip-error
    return new_cw

#decoder verification
codeword_e=codeword[:]
codeword_e[6]= 1 - codeword_e[6] #error
print(codeword,"cw correct")
print(codeword_e, "cw_e with one error")
for i in range(1,50): # maximum number of iterations
    S = mult_matrix_gf2(H, codeword_e)
    if sum(S) > 0:
        codeword_e = bit_flipping_iteration(H, codeword_e, S)
    else: break
print(codeword_e, "number of iterations:", i - 1, "syndrome:", S)
```

```
[1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1] cw correct
```

```
[1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1] cw_e with one error
[1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1] number of iterations: 1 syndrome: [0, 0, 0, 0]
```

In the 8+4 example, only one iterations were required to reach the correct solution. In cases with a higher number of errors, such as 50 errors in a 4096+512 bit configuration, more iterations will be needed. However, the process remains computationally efficient, as it relies primarily on simple XOR operations, which are extremely fast to execute in hardware.

For a throughput of 3 GB/s (≈24 Gbit/s), an LDPC processor operating at 800 MHz would ideally require about 30 bits per cycle. However, due to the iterative nature of LDPC decoding, the effective parallelism increases to hundreds or thousands of bits per cycle, and is typically implemented in hardware. In a mature 90 nm technology node, a processing core with this level of parallelism translates to an area on the order of 0.1 mm², with a corresponding silicon cost extremely low — on the order of a few cents or less per unit.

3.3 - Correcting analogue information

A key aspect of modern error correction, is the analogue nature of the information.

In simplified terms, the reading of a QLC NAND cell in an SSD can be modelled as a voltage ranging between 0 V and 4.0 V. Each of the 16 possible levels occupies an interval with some variation, for example ±0.15 V due to noise and physical effects. Values below 0.25 V may be interpreted as level 0, while values above 3.75 V are interpreted as level 15.

When converting this voltage into a 4-bit value, the reliability of the bits is not uniform. In general, the most significant bits are more robust, while the least significant bits are more sensitive to small variations in the measured voltage. This can be intuitively modelled by assigning higher confidence to more significant bits and lower confidence to less significant ones.

In an extreme simplified view, the read operation can be interpreted as producing a value between 0 and 15:

Read Value = Read_Voltage/0.25,

where most of the decision is reliable, MSB, and only a small uncertainty remains around the least significant level, LSB. In this sense, one may think of the measured value as a base integer to which a small uncertainty of −1, 0, or +1 may be added, reflecting the proximity to adjacent levels. The most reliable part of the value corresponds to the more significant bits, while the uncertainty is mainly associated with the least significant bit.

True Values	7	3	14	12	8	4	3	9	7
Read Volts	1,80	0,85	3,60	2,98	2,10	1,03	0,84	2,31	1,79
Read Values	6,70	2,90	13,90	11,42	7,90	3,62	2,86	8,74	6,66
MSB	6	2	13	11	7	3	2	8	6
Codeword	0,70	0,90	0,90	0,42	0,90	0,62	0,86	0,74	0,66
Prediction	1	1	1	0	1	1	1	1	1
Distance	0,30↓	0,10↓	0,10↓	-0,42↑	0,10↓	0,38↓	0,14↓	0,26↓	0,34↓

Table 1 - Example of 4 x 9 bits read from a QLC-NAND

Under this conceptual interpretation, the LDPC decoding focus on one bit, where the decision is close to a threshold and therefore more uncertain. In such cases, the decoder effectively enforces a stronger correction constraint, analogous to requiring a majority-like consistency condition (for example, a 1/2 or higher agreement among parity-check observations) but the correction effort is concentrated on a subset of the bits, roughly one bit per cell.

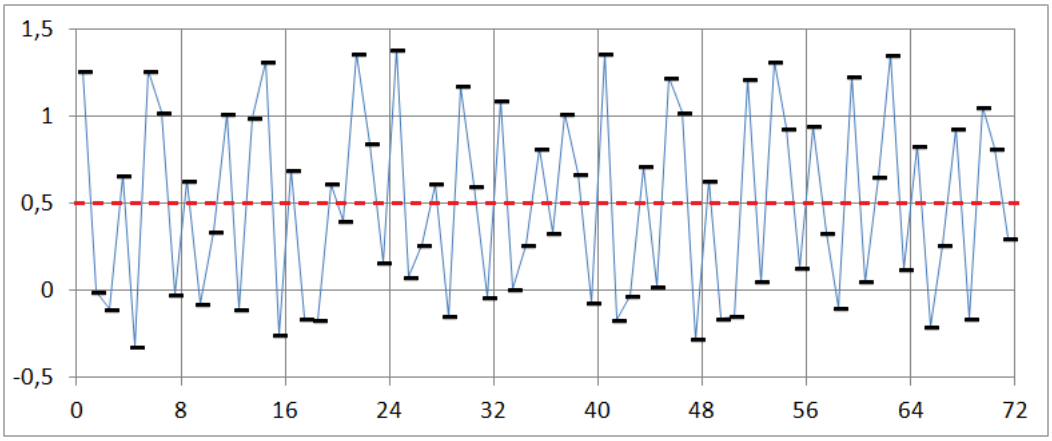


Figure 1 - Example of least significant unity from 256+32 bits read from a QLC NAND

4. Conclusion

This work started with Hamming codes as an entry point to error correction, using their simplicity to illustrate the fundamental idea of parity-check equations and syndrome-based decoding. From this perspective, LDPC codes can be naturally understood as a generalization: instead of a small number of dense parity relations, we move to a large set of sparse constraints, enabling powerful error-correction capabilities while maintaining computational efficiency.

The bit-flipping algorithm was presented as an intuitive decoding strategy, where each bit is evaluated based on the number of unsatisfied parity-check equations in which it participates. Even in its simplest form—flipping only the most likely erroneous bit per iteration—it demonstrates convergence and provides insight into how iterative decoding operates. Although practical implementations often update multiple bits simultaneously to accelerate convergence, the simplified approach highlights the underlying mechanism in a clear and stable way.

The examples discussed show that only a small number of iterations may be required in low-error scenarios, while more demanding cases naturally require additional iterations. Nevertheless, the computational cost remains low, as the decoding process relies predominantly on simple XOR operations, making it well suited for hardware implementation and high-throughput applications such as SSDs.

A key aspect of modern storage systems is the analogue nature of the stored information. Multi-level memory cells encode several bits using different voltage levels, which are inherently noisy and cannot be read as exact binary values. LDPC codes are particularly effective in this context because they can exploit reliability information. By extending the decoding process to include weighted contributions from parity-check equations, it is possible to move beyond purely binary decisions and improve both convergence speed and error-correction performance.

In summary, LDPC codes combine structural simplicity, scalability, and the ability to incorporate soft information, making them a cornerstone of modern error-correction systems. Starting from the conceptual foundation provided by Hamming codes, it becomes clear how increasingly sophisticated decoding strategies emerge from the same basic principles, ultimately enabling reliable communication and storage in highly noisy environments.

5. References

Gallager, R. G. (1962), “Low-density parity-check codes”, *IRE Transactions on Information Theory*, 8(1), 21–28.

Hamming, R. W. (1950), “Error detecting and error correcting codes”, *Bell System Technical Journal*, 29(2), 147–160.

Shannon, C. E. (1948), “A Mathematical Theory of Communication”, *Bell System Technical Journal*, 27(3), 379–423 and 27(4), 623–656.

Vieira, P. C. C. (2026), “A Step-by-Step Introduction to Error-Correcting Codes: From Hamming to Reed-Solomon”, *ResearchGate*, April, DOI: 10.13140/RG.2.2.24819.49441