DOCUMENTATION FOR PRODUCT CODE ENCODER/DECODER

1. Overview

This script implements encoding and decoding for a Product Code using simulated transmission over an AWGN BPSK channel. Parity bits are computed using per-pair XORs along both rows and columns, and the decoder uses a min-sum based message-passing approach to iteratively decode the received bits.

Let, $L(d) = log(\frac{P(d=+1}{P(d=-1)})$ is the apriori LLR for input bit d. The output LLR

$$L(\hat{d}) = L_c(x) + L(d) + L_e(\hat{d})$$

where $L_c(x)$ is the LLR of channel measurement at receiver, calculated using MAP- Maximum a posteriori rule, $L_e(\hat{d})$ is the extrinsic LLR value.

$$L_c(x_k) = \log_e \left[\frac{p(x_k \mid d_k = +1)}{p(x_k \mid d_k = -1)} \right]$$

which simplifies to

$$L_c(x_k) = \frac{2}{\sigma^2} x_k$$

. For the product code, the iterative decoding algorithm proceeds as follows:

(1) Set the a priori LLR, $L(d)$
(2) Decode horizontally, and using $L(\hat{d}) = L_c(x) + L(d) + L_e(\hat{d})$ obtain the horizontal extrinsic LLR as shown below:
$$L_{eh}(\hat{d}) = L(\hat{d}) - L_c(x) - L(d)$$
(3) Set $L(d) = L_{eh}(\hat{d})$ for the vertical decoding of step 4.
(4) Similarly, decode vertically and obtain the vertical extrinsic LLR as shown below:
$$L_{ev}(\hat{d}) = L(\hat{d}) - L_c(x) - L(d)$$
(5) Set $L(d) = L_{ev}(\hat{d})$ and repeat steps 2 through 5.
(6) After enough iterations (that is, repetitions of steps 2 through 5) to yield a reliable decision, go to step 7.
(7) The soft output is
$$L(\hat{d}) = L_c(x) + L_{eh}(\hat{d}) + L_{ev}(\hat{d})$$

2. Dependencies

```
import numpy as np
from itertools import combinations
```

3. Function Descriptions

3.1. simulate_awgn_bpsk_channel.

Purpose: Simulates BPSK transmission over an AWGN channel and returns the Log-Likelihood Ratio (LLR) values.
Inputs: The bits/ bit matrix which is to be passed through the channel and signal to noise ratio in decibel. **Returns:** `llr` Log likelihood ratio matrix of the input matrix after noise addition.

```
def simulate_awgn_bpsk_channel(bits, snr_db):
    snr_linear = 10**(snr_db / 10)
    sigma = np.sqrt(1 / snr_linear)
    bpsk = 2 * bits - 1
    noise = np.random.normal(0, sigma, bits.shape)
    received = bpsk + noise
    llr = 2 * received / sigma**2
    return llr
```

3.2. encoder.

Purpose: Encodes the $d \times d$ input data matrix, generates per-pair parity values along rows and columns, and simulates their transmission over AWGN.
For calculating the parity dictionary, we XOR the respective data_matrix values.

$$d_i \oplus d_j = p_{ij}$$

```
def encoder(data_matrix, snr_db):
     data_matrix = np.array(data_matrix)
    d= data_matrix.shape[0]
    assert data_matrix.shape==(d,d)
    Lc_matrix =simulate_awgn_bpsk_channel(data_matrix, snr_db)
    parity_h={}
    parity_v={}
    for i in range(d):
        for j1, j2 in combinations(range(d),2): parity_h[(i,j1,j2)]=
            simulate_awgn_bpsk_channel(np.array(data_matrix[i,j1]^data_matrix[i,j2
            ]),snr_db)
    for j in range(d):
        for i1,i2 in combinations(range(d),2): parity_v[(j,i1,i2)]=
            simulate_awgn_bpsk_channel(np.array(data_matrix[i1,j]^data_matrix[i2,j
            ]), snr_db)
    return Lc_matrix, parity_h, parity_v
```

Returns:

- `Lc_matrix` – LLR for each data bit.
- `parity_h` – Dictionary of row-wise XOR parities with keys `(i,j1,j2)` for row `i` and pair`(j1,j2)`
- `parity_v` – Dictionary of column-wise XOR parities with keys `(i1,i2,j)` with column `j` and pair `(i1,i2)`

3.3. min_sum_xor.

Purpose: Implements a min-sum approximation for XOR constraints in decoding.

```
def min_sum_xor(l1, l2):
    return -np.sign(l1) * np.sign(l2) * np.minimum(np.abs(l1), np.abs(l2))
```

3.4. decoder.

Purpose: Decodes the data matrix using iterative message passing. Each iteration consists of row-wise and column-wise extrinsic LLR updates using min-sum decoding.
We use the information obtained of d_i from the d_js using the XORed parity bits.

$$d_i = d_j \oplus p_{ij} \qquad i, j = 1, 2$$

```
def decoder(Lc_matrix, prior_matrix, parity_h, parity_v, max_iters):
    ...
```

For example: If the data matrix were 2x2, the Extrinsic LLR of d_1 will be sum of these two:

$$L_{eh}(\hat{d}_1) = \Big[L_c(x_2) + L(\hat{d}_2)\Big] \boxplus L_c(x_{12})$$

$$L_{ev}(\hat{d}_1) = \Big[L_c(x_3) + L(\hat{d}_3)\Big] \boxplus L_c(x_{13})$$

For Horizontal Extrinsic value updates,

```
    L_horizontal = L.copy()
        for (i, j1, j2), parity in parity_h.items():
            ext_j1 = min_sum_xor(L[i, j2] + Lc_matrix[i, j2], parity)
            ext_j2 = min_sum_xor(L[i, j1] + Lc_matrix[i, j1], parity)
            L_horizontal[i, j1] += ext_j1
            L_horizontal[i, j2] += ext_j2
```

Returns:

- `L` – Final LLR matrix after decoding.
- `decoded_bits` – Final decision bits from LLRs.

4. Example Execution

```
Lc_matrix, parity_h, parity_v = encoder([[1,0],[0,1]], 10*np.log10(0.5))
print(Lc_matrix)
L, decoded_bits = decoder(Lc_matrix, [[0,0],[0,0]], parity_h, parity_v, 2)
print(L)
print(decoded_bits)
```

```
[[-1.86797365 -2.34015361]
 [-2.56058726  0.60310293]]
[[ 6 -5]
 [-4  8]]
[[1 0]
 [0 1]]
```

5. Notes

- This implementation is designed for square matrices ($d \times d$).
- XOR parity is taken over all possible pairs in each row and column.
- LLRs are updated iteratively using a hard min-sum rule.
- The decoder supports use of non-zero a priori input LLRs.