Quantum Error Correction

INTRODUCTION TO QUANTUM AI

Mohammad Hasan
Harshwardhan
Aditya Mishra

Information Reconciliation for QKD using LDPC Codes

The Goal of QKD: Quantum Key Distribution (like BB84) allows Alice and Bob to produce a shared, secret "sifted key."

The Real-World Problem: In a practical QKD system, the channel is noisy. This leads to a **Quantum Bit Error Rate (QBER)**, meaning Alice's and Bob's keys are **highly correlated, but not identical.**

Our Project's Focus: These errors *must* be corrected before the key can be used. This is the **Information Reconciliation** step.

Our Approach: We implement a powerful classical algorithm, **Low-Density Parity-Check (LDPC) coding**, to find and remove these errors.

How LDPC Reconciliation Works

The Core Components

1. The "Rulebook" (The {H} Matrix): The entire code is defined by a public, shared **Parity Check Matrix**, {H} (created by the gallager_H function).

2. Alice's Role (Sending the "Hint"):

- Alice has key vAlice.
- She calculates the **syndrome (sAlice)**(using the syndrome function).
- She sends **sAlice** publicly to Bob.

3. Bob's Role (Finding the Error):

- Bob has his noisy key, **rBob**.
- He uses **sAlice** *and his key **rBob*** to run the **Decoder** (bit_flipping_decoder).
- The decoder's job is to find the most likely error vector, **eEst.**

4. The Correction:

- Bob corrects his key: **vCorrected = rBob + eEst**.
  If successful, **vCorrected** *is now identical to **vAlice**.*

Mapping the Process to Our Python Code

Key Functions and Logic

gallager_H (The {H} Matrix Generator):

- Builds the sparse {H} matrix, ensuring a regular structure (dv connections per column, dc connections per row).

syndrome (Alice's Action):

- Calculates the matrix-vector product to generate the syndrome **s**.

bit_flipping_decoder (Bob's Engine):

- Uses an **iterative majority logic** approach.
- In each iteration: it finds **unsatisfied checks** and flips the key bits that are connected to the majority of those unsatisfied checks,
- gradually driving the error estimate **eEST** to the correct value.

reconcile_keys_ldpc (The Full Workflow):

- Orchestrates the entire process: syndrome computation, decoding, and final key correction.

Demonstration: Simulating and Correcting Errors

How We Tested (The demo_reconciliation function)

Generate vAlice (Random 1024-bit key).

Simulate Noise: Apply simulated channel error using a **noise probability** (e.g., 2%), generating the noisy key **rBob.**

Run Reconciliation and collect statistics.

Example Results

Metric	Value	Interpretation
Key Length (n)	1024 bits	
Errors **Before**	23	The number of bits Bob needs to correct (QBER = 2.2461%).
Errors **After**	0	**SUCCESS!** The keys are now identical.
Iterations	8	The speed of the simple Bit-Flipping decoder.

```
Building LDPC H (m=512, n=1024, dv=4, dc=8) ...
Simulated channel noise prob=0.0200 -> errors before = 23 (QBER ~ 2.2461%)
Reconciliation stats: {'n': 1024, 'success': True, 'iterations': 8, 'errors_before': 23, 'errors_after': 0}
Reconciliation SUCCESS: Bob recovered Alice's key exactly.
errors after = 0, iterations = 8
Syndrome match after reconciliation: True
```

Code

```python
"""
LDPC-based information reconciliation (syndrome decoding via bit-flipping)
Integrates with BB84 sifted keys: Alice sends syndrome s = H @ alice_key (mod 2),
Bob uses his received key and s to estimate error vector e and correct his key.

Fixed: Ensure parameters satisfy Gallager construction; demo uses compatible dv/dc.
"""

import numpy as np
import random
from typing import Tuple

# -------------------------
# LDPC matrix construction (Gallager style)
# -------------------------
def gallager_H(n: int, m: int, dv: int, dc: int, seed: int = None) -> np.ndarray:
    """
    Build a regular (dv, dc) LDPC parity check matrix H of size m x n
    using a simple Gallager-like construction.

    Requirement (for this simple builder): n * dv == m * dc and m must be divisible by dv.
    If those conditions are violated, the function raises a ValueError with a hint.
    """
    if seed is not None:
```

```
    if seed is not None:
        random.seed(seed)
        np.random.seed(seed)

    if n * dv != m * dc:
        raise ValueError("n * dv must equal m * dc for regular LDPC (choose dv/dc so n*dv == m*dc)")

    # number of submatrices (layers)
    p = dv
    rows_per_layer = m // p
    if rows_per_layer * p != m:
        raise ValueError(
            f"m ({m}) must be divisible by dv ({dv}) for this simple construction. "
            "Choose dv/dc such that m % dv == 0."
        )

    H = np.zeros((m, n), dtype=int)

    # Create p permutation-blocks and set ones_per_row_layer positions per row
    # We place ones in a structured way so columns will approximately get dv ones.
    ones_per_row_layer = max(1, dc // p)
```

```
for layer in range(p):
    perm = np.random.permutation(n)
    for row_idx in range(rows_per_layer):
        row = layer * rows_per_layer + row_idx
        group_size = n // rows_per_layer
        start = row_idx * group_size
        end = start + group_size
        # if group_size is 0 (shouldn't happen for sane params) fallback to entire perm
        if group_size <= 0:
            chosen = perm
        else:
            chosen = perm[start:end]
        indices = chosen[:ones_per_row_layer]
        H[row, indices] = 1

# Adjust column degrees to be close to dv (force if necessary)
col_degrees = H.sum(axis=0)
# add ones where degree < dv
for col in range(n):
    while col_degrees[col] < dv:
        row_choices = np.where(H[:, col] == 0)[0]
        if len(row_choices) == 0:
            break
        r = np.random.choice(row_choices)
```

```python
            H[r, col] = 1
            col_degrees[col] += 1
    # remove ones where degree > dv
    for col in range(n):
        while col_degrees[col] > dv:
            rows_with_one = np.where(H[:, col] == 1)[0]
            if len(rows_with_one) == 0:
                break
            r = np.random.choice(rows_with_one)
            H[r, col] = 0
            col_degrees[col] -= 1

    return H


# -------------------------
# Syndrome computation utility
# -------------------------
def syndrome(H: np.ndarray, v: np.ndarray) -> np.ndarray:
    """Compute syndrome s = H @ v (mod 2)."""
    return (H.dot(v) % 2).astype(int)
```

```python
# -------------------------
# Simple iterative Bit-Flipping decoder (Gallager A/B style)
# -------------------------
def bit_flipping_decoder(H: np.ndarray,
                         r: np.ndarray,
                         s_alice: np.ndarray,
                         max_iters: int = 50,
                         flip_threshold: int = None) -> Tuple[np.ndarray, bool, int]:
    """
    Attempt to find error vector e such that H @ (r ^ e) == s_alice (mod 2).
    We solve H @ e == s_alice ^ H @ r.
    Uses a majority bit-flip approach.
    Returns (e_est, success_flag, iterations_used)
    """
    m, n = H.shape
    s_r = syndrome(H, r)
    s_e = (s_alice ^ s_r) % 2  # desired syndrome for error vector e

    var_to_checks = [np.where(H[:, j] == 1)[0] for j in range(n)]
    check_to_vars = [np.where(H[i, :] == 1)[0] for i in range(m)]

    if flip_threshold is None:
        var_deg = np.array([len(lst) for lst in var_to_checks])
        flip_threshold = np.ceil(var_deg / 2).astype(int)
```

```python
e = np.zeros(n, dtype=int)

for it in range(1, max_iters + 1):
    cur_s = syndrome(H, e)
    residual = (s_e ^ cur_s) % 2
    if not residual.any():
        return e, True, it - 1

    unsatisfied_checks = np.where(residual == 1)[0]

    unsat_count = np.zeros(n, dtype=int)
    for chk in unsatisfied_checks:
        vars_in_check = check_to_vars[chk]
        unsat_count[vars_in_check] += 1

    flips = unsat_count > flip_threshold
    if not flips.any():
        max_idx = int(np.argmax(unsat_count))
        if unsat_count[max_idx] == 0:
            break
        flips[max_idx] = True

    e = (e ^ flips.astype(int)) % 2
```

```python
    final_res = (s_e ^ syndrome(H, e)) % 2
    success = not final_res.any()
    return e, success, max_iters


# -------------------------
# Integration utility: reconcile_keys_ldpc
# -------------------------
def reconcile_keys_ldpc(alice_bits: np.ndarray,
                        bob_bits: np.ndarray,
                        H: np.ndarray,
                        max_iters: int = 50) -> Tuple[np.ndarray, np.ndarray, dict]:
    """
    Alice computes syndrome s = H @ alice_bits and sends to Bob.
    Bob runs decoder to estimate error e and recovers alice_est = bob_bits ^ e_est.
    Returns (alice_bits, alice_est, stats)
    """
    if alice_bits.shape != bob_bits.shape:
        raise ValueError("Alice and Bob key lengths mismatch")

    n = alice_bits.size
    s_alice = syndrome(H, alice_bits)
    e_est, success, iters = bit_flipping_decoder(H, bob_bits, s_alice, max_iters=max_iters)
    alice_est = (bob_bits ^ e_est) % 2
```

```python
    alice_est = (bob_bits ^ e_est) % 2
    stats = {
        'n': n,
        'success': success,
        'iterations': iters,
        'errors_before': int(np.sum(alice_bits != bob_bits)),
        'errors_after': int(np.sum(alice_bits != alice_est))
    }
    return alice_bits, alice_est, stats


# -------------------------
# Demo harness: generate synthetic keys, apply errors, run ldpc reconciliation
# -------------------------
def demo_reconciliation(n: int = 1024,
                        dv: int = 4,
                        dc: int = 8,
                        noise_prob: float = 0.02,
                        seed: int = 42):
    """
    End-to-end demo:
      - build H for n variable nodes; compute m from dv/dc relation
      - create random alice key
      - simulate Bob receiving Alice's key through bit-flip channel (with noise_prob)
      - run LDPC reconciliation
      - print stats
    """
```

```python
np.random.seed(seed)
random.seed(seed)

# choose m to satisfy n * dv == m * dc  => m = n * dv / dc
if (n * dv) % dc != 0:
    raise ValueError("n * dv must be divisible by dc for regular LDPC. adjust parameters.")
m = (n * dv) // dc

# Additional check: ensure m divisible by dv for Gallager layers
if m % dv != 0:
    raise ValueError(f"Computed m={m} is not divisible by dv={dv}. Choose dv/dc that produce m % dv == 0.")

print(f"Building LDPC H (m={m}, n={n}, dv={dv}, dc={dc}) ...")
H = gallager_H(n=n, m=m, dv=dv, dc=dc, seed=seed)

alice = np.random.randint(0, 2, size=n, dtype=int)
flips = (np.random.random(size=n) < noise_prob).astype(int)
bob = (alice ^ flips) % 2

errs_before = np.sum(alice != bob)
qber_est = errs_before / n
print(f"Simulated channel noise prob={noise_prob:.4f} -> errors before = {errs_before} (QBER ~ {qber_est:.4%})")

alice_bits, alice_est, stats = reconcile_keys_ldpc(alice, bob, H, max_iters=200)
print("Reconciliation stats:", stats)
if stats['success']:
```

```python
    if stats['success']:
        print("Reconciliation SUCCESS: Bob recovered Alice's key exactly.")
    else:
        print("Reconciliation FAILED: some mismatches remain.")
    print(f"errors after = {stats['errors_after']}, iterations = {stats['iterations']}")

    s_alice = syndrome(H, alice)
    s_recovered = syndrome(H, alice_est)
    print("Syndrome match after reconciliation:", np.array_equal(s_alice, s_recovered))

    return {
        'H': H,
        'alice': alice,
        'bob_before': bob,
        'alice_est': alice_est,
        'stats': stats
    }


# If run as script, do a demo with compatible dv/dc
if __name__ == "__main__":
    # NOTE: dv=4, dc=8 chosen so that m = n*dv/dc = 1024*4/8 = 512 and m % dv == 0
    res = demo_reconciliation(n=1024, dv=4, dc=8, noise_prob=0.02, seed=123)
```

Thank You