I437 Final Project Report: GRAND Decoding

Xu Pengfei
Student ID: 2510082

JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY
1990

December 5, 2025

Contents

1 Introduction

In this report, I introduce the Guessing Random Additive Noise Decoding (GRAND) algorithm for hard-decision decoding over Additive white Gaussian noise(AWGN) channels and discrete memoryless channels (DMCs). I also study an important soft-input variant, **Soft Maximum Likelihood Decoding GRAND with Abandonment (SGRAND-AB)**.

To better understand these decoding algorithms, I implemented them for two linear block codes:

- The Extended Hamming code
- The Golay code

Since the Golay family includes both the perfect binary Golay code and the extended binary Golay code, I provide context for these codes throughout the simulation.

For the channel models, I use a Binary Symmetric Channel (BSC) and a Binary-Input Additive White Gaussian Noise (BI-AWGN) channel. In the DMC (BSC) case, I implement and compare:

- `SyndromeDecoder`
- `MLDecoder` (Maximum Likelihood)
- `GRANDDecoder`

In the BI-AWGN case, I compare:

- `MLDecoder`
- `GRANDDecoder` (Hard decision)
- `SGRANDDecoder` (Soft decision)

To support these experiments, I designed a modular simulation framework. The framework separates the code, decoder, channel, and runner into independent modules so that they can be combined arbitrarily for different simulations.

2 Decoding

2.1 Guessing Random Additive Noise Decoding (GRAND)

GRAND is a hard-decision decoder that searches for the **most likely noise pattern**, rather than directly for the most likely codeword. Starting from the hard-demodulated received word, GRAND tries possible noise patterns one by one, in order of their probability under the assumed channel model.

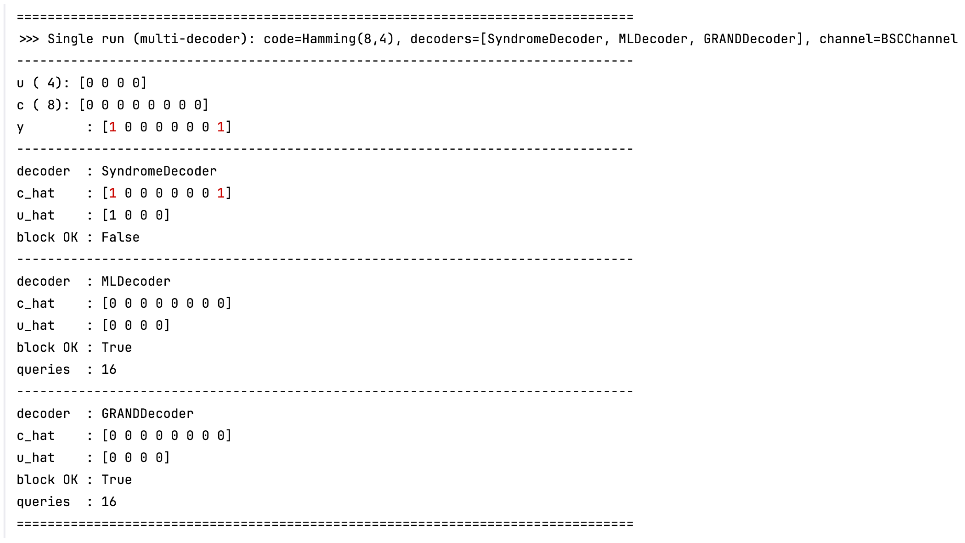
For each candidate noise pattern, it “subtracts” (inverts) that noise from the received word and checks whether the resulting vector is a valid codeword in the codebook. GRAND stops as soon as it finds a valid codeword. GRAND with abandonment (GRAND-AB) behaves the same way; however, if the number of codebook queries exceeds a preset limit, it stops early and outputs an erasure.

2.1.1 Example Simulation

We use the Extended Hamming (8,4) code as an example. The simulation code is shown below:

```
hamming = ExtendedHamming84()
bsc = BSCChannel(p=0.2)
run_single(code=hamming, channel=bsc,
           decoder_cls=[SyndromeDecoder, MLDecoder, GRANDDecoder])
```

Listing 1: *Running GRAND on Hamming Code*



```
================================================================================
>>> Single run (multi-decoder): code=Hamming(8,4), decoders=[SyndromeDecoder, MLDecoder, GRANDDecoder], channel=BSCChannel
--------------------------------------------------------------------------------
u ( 4): [0 0 0 0]
c ( 8): [0 0 0 0 0 0 0 0]
y        : [1 0 0 0 0 0 0 1]
--------------------------------------------------------------------------------
decoder  : SyndromeDecoder
c_hat    : [1 0 0 0 0 0 0 1]
u_hat    : [1 0 0 0]
block OK : False
--------------------------------------------------------------------------------
decoder  : MLDecoder
c_hat    : [0 0 0 0 0 0 0 0]
u_hat    : [0 0 0 0]
block OK : True
queries  : 16
--------------------------------------------------------------------------------
decoder  : GRANDDecoder
c_hat    : [0 0 0 0 0 0 0 0]
u_hat    : [0 0 0 0]
block OK : True
queries  : 16
================================================================================
```

Figure 1: *Output of the single run simulation.*

The Extended Hamming (8,4) code has a minimum distance of $d_{\min} = 4$. Therefore, the guaranteed error-correction radius is:

$$t = \left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor = 1. \tag{1}$$

In this particular run, two bit errors occur. Since $t = 1$, the syndrome decoder cannot guaranty correction. However, both the MLDecoder and GRANDDecoder successfully decode the block.

For GRAND, the decoding process proceeds by testing noise patterns in order of increasing Hamming weight:

$$e = \mathbf{0}, \quad \text{all weight-1 patterns}, \quad \text{all weight-2 patterns}, \ \ldots$$

When GRAND reaches the weight-2 patterns, it eventually tries the **true noise pattern**:

$$\mathbf{e} = [1\,0\,0\,0\,0\,0\,0\,1].$$

Thus, even though the error weight exceeds the algebraic correction radius, GRAND succeeds. Note that our implementation uses a simple abandonment rule limited to weight-3 patterns.

2.2 Soft Maximum Likelihood Decoding (SGRAND)

Rather than using only hard decisions like the hard GRAND algorithm, Soft GRAND (SGRAND) leverages soft information to achieve better performance. In our Monte Carlo experiments, SGRAND achieves almost the same Word Error Rate (WER) as ML decoding over a range of SNR values.

2.2.1 The Euclidean Distance Metric

```python
def calc_euclidean_distance(self, y: np.ndarray, theta_y: np.ndarray, e: np.
    ndarray) -> float:
    bits = (theta_y - e) & 1
    x = 1.0 - 2.0 * bits
    return np.sum((x - y) ** 2)
```

Listing 2: *Euclidean Distance Calculation*

We replace the raw probability p with a log-likelihood metric. For an Additive White Gaussian Noise (AWGN) channel:

$$\Pr(\mathbf{y}|\mathbf{c}) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x_i - y_i)^2}{2\sigma^2}} \tag{2}$$

Maximizing the likelihood is equivalent to maximizing the log-likelihood:

$$\ln \Pr(\mathbf{y}|\mathbf{c}) = -\frac{1}{2\sigma^2} \sum (x_i - y_i)^2 + n \ln \frac{1}{\sqrt{2\pi\sigma^2}} \tag{3}$$

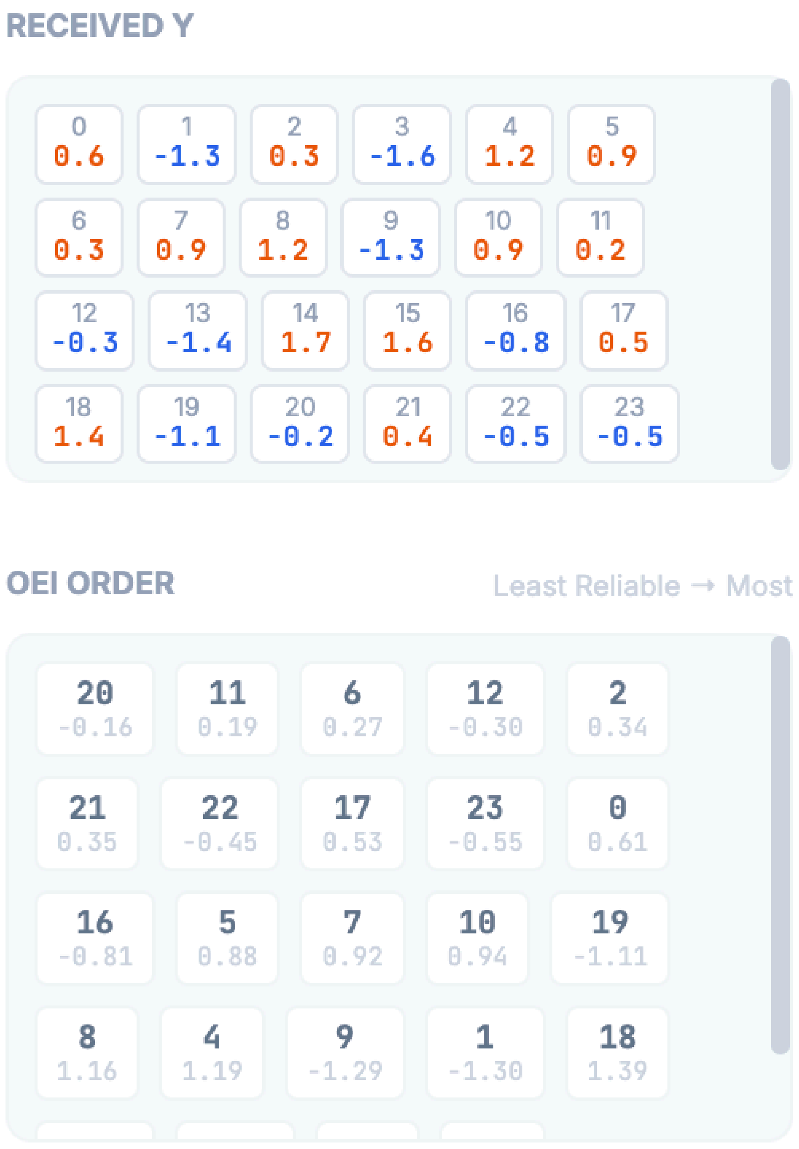
In optimizing over $\mathbf{c}$, the independent terms $1/(2\sigma^2)$ and $n \ln 2\pi\sigma^2$ may be ignored. Finally

$$\hat{\mathbf{c}} = \arg\min_{\mathbf{c}\in\mathcal{C}} ||\mathbf{x} - \mathbf{y}||^2 = \arg\min_{\mathbf{c}\in\mathcal{C}} ||\mathbf{1} - 2\mathbf{c} - \mathbf{y}||^2 \tag{4}$$

A smaller distance indicates a higher likelihood that the candidate is the correct codeword. our *calc_euclidean_distance* function implements this minimization target.

2.2.2 Algorithm Explanation

1. Compute Ordered Error Indices (OEI) For BPSK, soft values y_i that are closer to 0 are less reliable because they can be flipped more easily. Values that are far from 0 are more reliable. The OEI is therefore obtained by sorting the bit positions in ascending order of $|y_i|$, from the least reliable bit to the most reliable bit.



RECEIVED Y

0: 0.6	1: -1.3	2: 0.3	3: -1.6	4: 1.2	5: 0.9
6: 0.3	7: 0.9	8: 1.2	9: -1.3	10: 0.9	11: 0.2
12: -0.3	13: -1.4	14: 1.7	15: 1.6	16: -0.8	17: 0.5
18: 1.4	19: -1.1	20: -0.2	21: 0.4	22: -0.5	23: -0.5

OEI ORDER Least Reliable → Most

20: -0.16	11: 0.19	6: 0.27	12: -0.30	2: 0.34
21: 0.35	22: -0.45	17: 0.53	23: -0.55	0: 0.61
16: -0.81	5: 0.88	7: 0.92	10: 0.94	19: -1.11
8: 1.16	4: 1.19	9: -1.29	1: -1.30	18: 1.39

Figure 2: *Example of Ordered Error Indices (OEI).*

2.2.3 Priority Queue Search

Next, It check each candidate codeword by removing error vectors in order of their likelihood, starting from the most probable one.

To perform this search, we use a priority queue implemented as a binary **min-heap**. In a min-heap, the root always contains the smallest key; therefore, popping the heap always returns the candidate with the **minimum Euclidean distance**.

Since minimizing the squared Euclidean distance is equivalent to maximizing the likelihood for BPSK over an AWGN channel, this mechanism allows us to explore error vectors in descending likelihood order.

The search process proceeds as follows:

1. For each popped node, we subtract the error vector from the hard decision $\theta(\mathbf{y})$ and check whether the resulting vector is a valid codeword.

2. If it is valid, we have found the most probable error pattern and can output the decoded codeword $\hat{\mathbf{c}}$.
3. Otherwise, we generate the next most likely candidates and insert them into the min-heap.

In practice, generating candidates implies modifying the error vector $\vec{e}$ along the Ordered Error Indices (OEI):

- **Append step**: Set the next least-reliable bit to 1.
- **Shift step**: After setting that bit, unset the previous bit to form another valid child pattern.

Both of these child error patterns are pushed into the min-heap for further exploration.

2.2.4 Example: Understanding the SGRAND Search Tree

We use a Hamming(8,4) code with 7 queries as a simple example to illustrate how SGRAND explores the error space.

The decoder begins with the zero error pattern Err : $\varnothing$. Since

$$\mathbf{c} = \theta(\mathbf{y}) \oplus \text{Err} \tag{5}$$

It is not a valid codeword; it expands the node.

The first OEI position is **2**, so the decoder adds: Err : $\{2\}$. There is no previously flipped bit, so only this one child is created.

Next, Err : $\{2\}$ is also invalid. The next OEI position is **5**, so the decoder generates two new nodes:

- Add bit 5 to the current pattern: Err : $\{2, 5\}$
- Add bit 5 and remove the previous bit 2: Err : $\{5\}$

The figure below shows this search process.

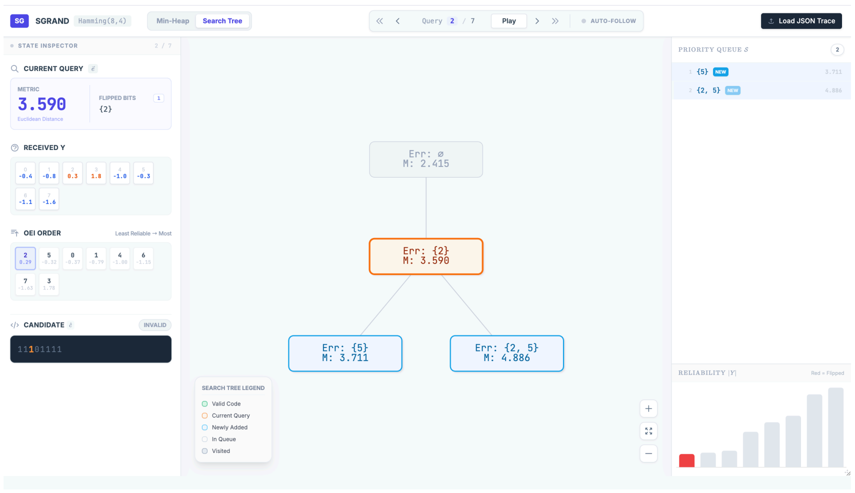


Figure 3: *Search tree expansion: generating children from error pattern {2}.*

Similarly, since Err : $\{5\}$ has the smallest metric and is still invalid, we expand this node. The next OEI position is **0**, so we create two children, Err : $\{5, 0\}$ and Err : $\{0\}$, and add them to the tree.

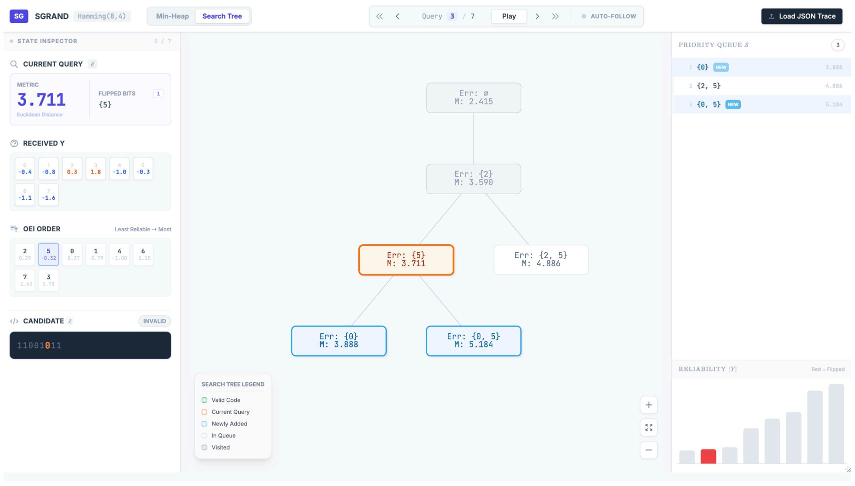


Figure 4: *Search tree expansion: generating children from error pattern {5}.*

Since the pop operation removes a node from the priority queue, the actual min-heap at this step looks as follows:

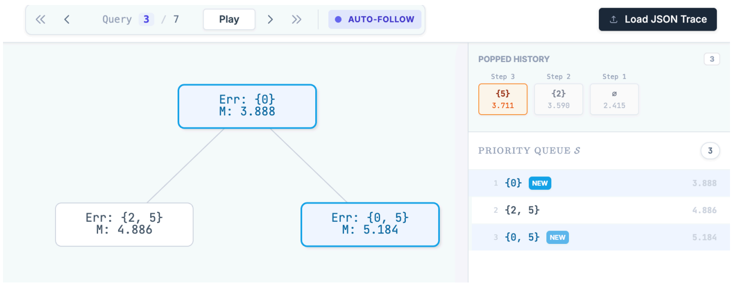
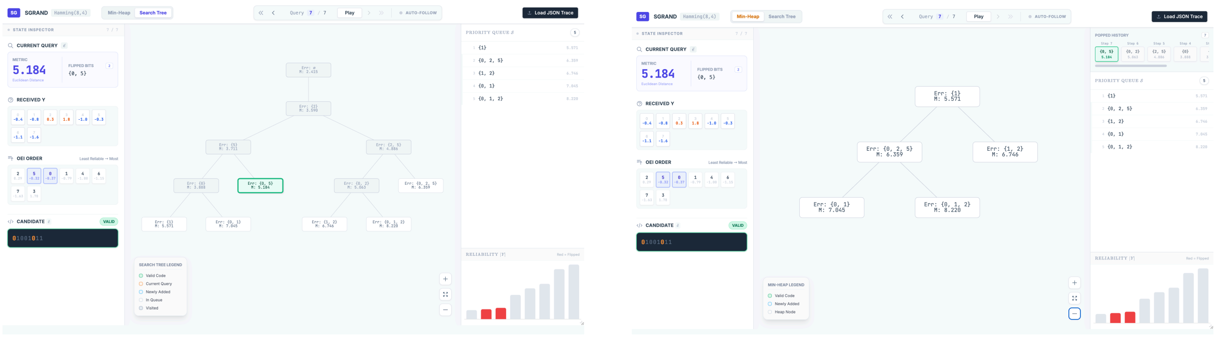


Figure 5: *Snapshot of the min-heap priority queue during search.*

In the end, the decoder finds an error vector that produces a valid codeword → `01001011`. The search tree figure below summarizes this whole process.



(a) *Complete search tree structure.* **(b)** *Heap evolution and pop history.*

Figure 6: *Overview of the SGRAND decoding example.*

You can try more complex examples with the visualization tool to better understand how this process works.

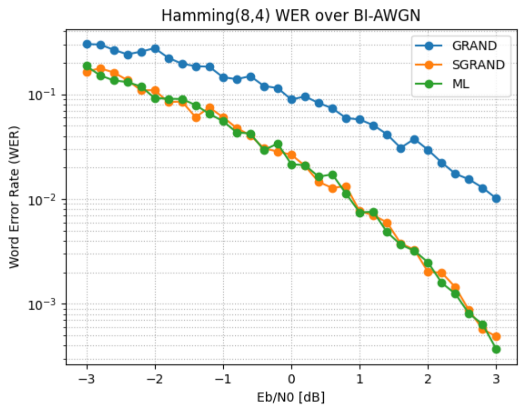
Finally, the **Abandonment** part is straightforward: we simply count every code check performed. When the count exceeds the `max_queries` setting, the loop breaks, and the decoder reports an erasure.

3 Evaluation

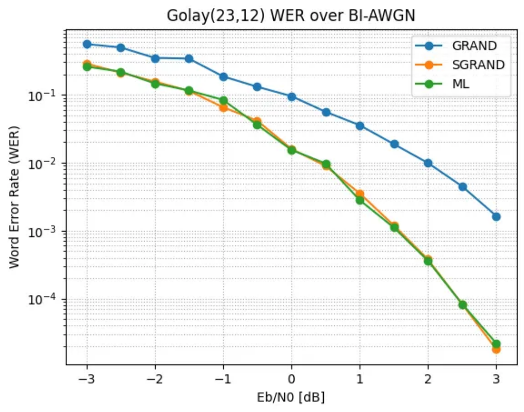
3.1 WER over BI-AWGN

Setup:

- SNR Range: $E_b/N_0 \in [-3.0, 3.0]$ dB, step ≤ 0.5 dB.
- Termination: 100 block errors or 2,000,000 simulations.
- Decoders: ML, Hard GRAND, Soft GRAND (SGRAND).



(a) *WER of Hamming(8,4).*



(b) *WER of Golay(23,12).*

Figure 7: *Word Error Rate performance over BI-AWGN.*

The hard GRAND decoder is clearly the worst.

The SGRAND curve almost overlaps with the ML decoder, showing that SGRAND achieves near-ML performance, while hard GRAND suffers a clear loss.

3.2 Query Complexity

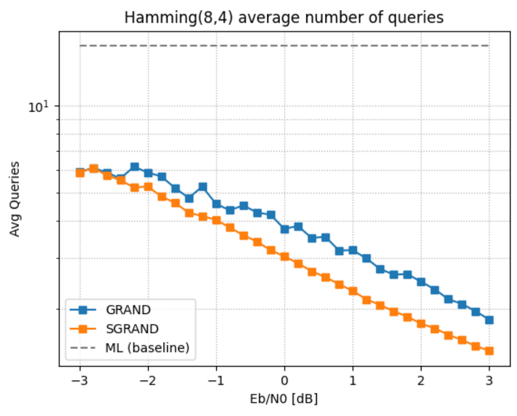
I also count the number of check queries to evaluate complexity. The results are shown in Figure 8.

In the top two plots (Average Queries vs E_b/N_0):

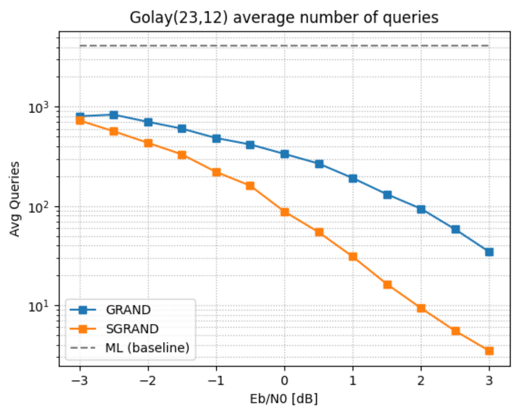
- The ML baseline is simply 2^k (exhaustive search), shown as the dotted horizontal line.
- SGRAND always needs fewer queries than hard GRAND for the same SNR, so it is more efficient.

The bottom two plots show Average Queries vs WER. To reach the same WER, SGRAND consistently requires fewer queries than GRAND.

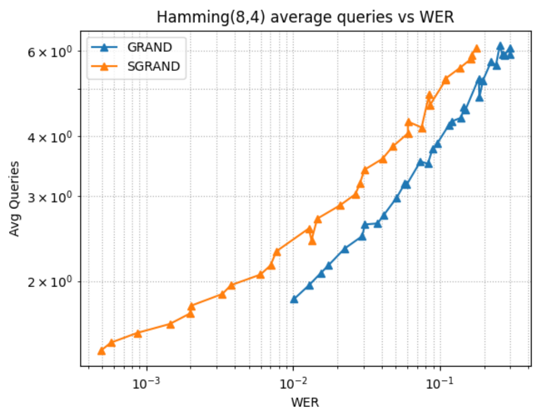
By comparing the two codes horizontally, we see that the average number of queries for Golay(23,12) is **2–3 orders of magnitude higher** than for Hamming(8,4). This shows that the code length has a strong impact on the decoding complexity of GRAND and SGRAND.



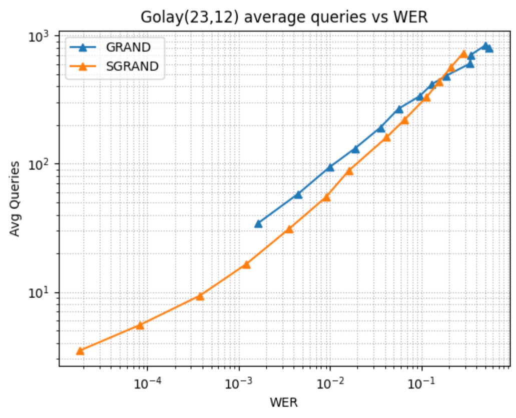
(a) *Avg Queries vs SNR (Hamming).*



(b) *Avg Queries vs SNR (Golay).*



(c) *Avg Queries vs WER (Hamming).*



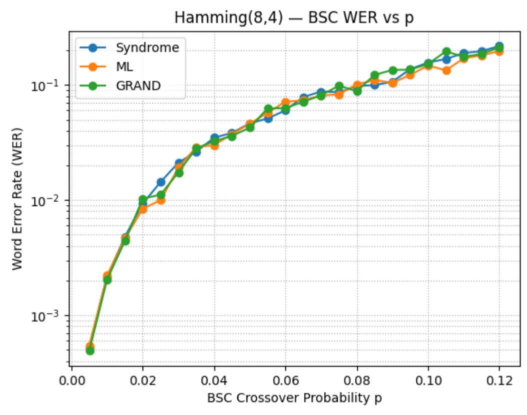
(d) *Avg Queries vs WER (Golay).*

Figure 8: *Analysis of computational complexity (number of queries).*

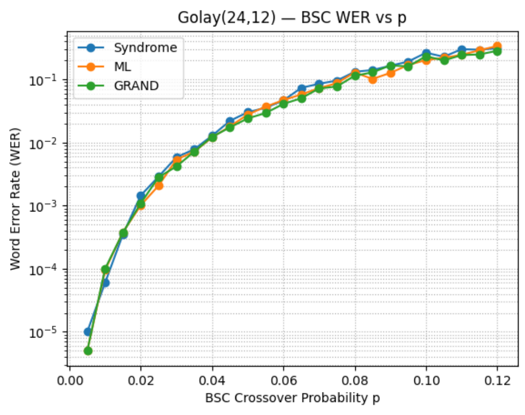
3.3 WER over BSC

Setup:

- Probability $p \in [0.005, 0.12]$.
- Decoders: Syndrome, ML, GRAND.



(a) *WER vs p (Hamming).*



(b) *WER vs p (Golay).*

Figure 9: *Performance over Binary Symmetric Channel.*

The GRAND and ML curves almost overlap, and the syndrome decoder performs similarly well.

4 Conclusion

4.1 Strengths

1. GRAND is proven to achieve **maximum-likelihood decoding** for *any* block code.
2. Strong performance for short and medium length codes.
3. Can set a maximum number of queries (Abandonment).
4. Friendly to hardware parallelization.

4.2 Weaknesses

1. At low SNR, complexity grows exponentially.
2. Not suitable for long codes due to the search space size.

5 Implementation Details

The full implementation, including all decoders and simulation scripts, is available on GitHub: `https://github.com/xuing/Understand-SGrand-Decoding-through-visualization`

You can view the SGRAND decoding visualization process using the provided tool:

`https://www.jaist.ac.jp/~s2510082/sgrand_visualization.html`

Load one of the example JSON traces, or generate new traces using the `generate_SGRAND_trace` function in the Python codebase.

The simulation framework is implemented in Python. The codebase follows an object-oriented design, allowing for the modular testing of different linear block codes and decoders.

5.1 The Linear Block Code Base Class

The core logic is encapsulated in the `LinearBlockCode` class. This parent class defines the fundamental attributes of a block code, including the codeword length n, information length k, and the field order (set to 2).

Key Operations:

- **Encoding:**
$$\mathbf{c} = \mathbf{u}\mathbf{G} \pmod 2 \tag{6}$$
- **Syndrome Calculation:**
$$\mathbf{s} = \mathbf{c}\mathbf{H}^T \pmod 2 \tag{7}$$
The method `is_codeword` returns `True` if $\mathbf{s} = \mathbf{0}$.

5.2 Extended Hamming Code

The `ExtendedHamming84` class implements the $(8, 4)$ extended Hamming code. Since the dimensions are small, the Generator matrix **G** and Parity-check matrix **H** are hard-coded directly as `numpy` arrays. This code has a minimum distance $d_{\text{min}} = 4$ and corrects $t = 1$ error.

5.3 Golay Codes Construction

The Golay codes are implemented dynamically using generator polynomials rather than hard-coded matrices.

5.3.1 Binary Golay Code (23, 12)

The `Golay23` class constructs the perfect binary Golay code. The generator matrix **G** is formed by cyclically shifting the coefficients of the generator polynomial:

$$g(x) = x^{11} + x^9 + x^7 + x^6 + x^5 + x + 1 \tag{8}$$

This creates a cyclic structure where each row of **G** is a right-cyclic shift of the previous row.

5.3.2 Extended Binary Golay Code (24, 12)

The `Golay24` class extends the $(23, 12)$ code to $(24, 12)$ by appending a parity bit. To facilitate encoding, the matrix is converted into systematic form $\mathbf{G}_{sys} = [\mathbf{I}_k|\mathbf{P}]$.

5.4 GRAND and SGRAND Decoders

The decoding logic is encapsulated in two classes inheriting from a base `ChannelDecoder`.

5.4.1 Hard-Decision GRAND

The `GRANDDecoder` implements the original hard-decision algorithm. It normalizes the input to binary values and uses `itertools` to systematically enumerate error patterns of increasing Hamming weight.

```python
class GRANDDecoder(ChannelDecoder):
    """
    Hard-decision GRAND: Enumerates error patterns by increasing Hamming weight.
    """
    def __init__(self, code: LinearBlockCode, max_weight: int = None):
        super().__init__(code)
        self.max_weight = max_weight if max_weight is not None else code.n

    def decode_to_codeword(self, y: np.ndarray):
        y = np.asarray(y)

        # Normalize input to {0, 1}
        if y.dtype.kind == "f":
            y = (y < 0).astype(int) # Soft input (AWGN)
        else:
            y = (y.astype(int) & 1) # Hard input (BSC)

        # 1. Check zero-error case
        if self.code.is_codeword(y):
            return y

        # 2. Enumerate error patterns up to max_weight
        for w in range(1, self.max_weight + 1):
            for idxs in itertools.combinations(range(self.code.n), w):
                c_test = y.copy()
                c_test[list(idxs)] ^= 1
                if self.code.is_codeword(c_test):
                    return c_test
        return y
```

Listing 3: *Hard-Decision GRAND Implementation*

5.4.2 Soft-Decision SGRAND

The `SGRANDDecoder` implements Soft-GRAND with abandonment. It utilizes a priority queue (min-heap) to order error patterns based on the Squared Euclidean Distance metric.

```
class SGRANDDecoder(ChannelDecoder):
    def __init__(self, code: LinearBlockCode, max_queries: int = None):
        super().__init__(code)
        self.max_queries = int(max_queries) if max_queries is not None else int(1
            e9)

    def calc_euclidean_distance(self, y: np.ndarray, theta_y: np.ndarray, e: np.
        ndarray) -> float:
        """
        Metric: Squared Euclidean Distance.
        Minimizing this is equivalent to maximizing log-likelihood for BPSK/AWGN.
        """
        bits = (theta_y - e) & 1
        x = 1.0 - 2.0 * bits
        return np.sum((x - y) ** 2)

    def compute_OEI(self, y: np.ndarray) -> np.ndarray:
        """Sort positions by reliability |y| (ascending)."""
        return np.argsort(np.abs(y))

    def expand_children(self, e: np.ndarray, oei: np.ndarray) -> List[np.ndarray]:
        """Expand child patterns based on the current error vector e."""
        n = self.code.n
        ones = np.where(e[oei] == 1)[0]
        j_star = (ones[-1] + 1) if len(ones) > 0 else 0

        children = []
        if j_star < n:
            # Child 1: Flip next least reliable bit
            e1 = e.copy()
            e1[oei[j_star]] ^= 1
            children.append(e1.copy())

            # Child 2: Shift the flip (unset previous, set current)
            if j_star > 0:
                e2 = e1.copy()
                e2[oei[j_star - 1]] ^= 1
                children.append(e2)
        return children

    def decode_to_codeword(self, y: np.ndarray):
        y = y.astype(float)
        theta_y = (y < 0).astype(int) # Hard decision estimate
        self.code.queries = 0

        # Step 1: Compute OEI
        oei = self.compute_OEI(y)

        # Step 2: Initialize Priority Queue with zero-error pattern
        S = [] # Min-heap
        e0 = np.zeros(self.code.n, dtype=int)
        heapq.heappush(S, (self.calc_euclidean_distance(y, theta_y, e0), e0))

        # Step 3: Main Search Loop
        while S and self.code.queries < self.max_queries:
            _, e = heapq.heappop(S)

            # Check validity
            c_test = (theta_y - e) & 1
            if self.code.is_codeword(c_test):
                return c_test

            # Expand next candidates
            for child_e in self.expand_children(e, oei):
                metric = self.calc_euclidean_distance(y, theta_y, child_e)
                heapq.heappush(S, (metric, child_e))

        print("SGRANDDecoder: Abandonment triggered.")
        return theta_y
```

Listing 4: *SGRAND Implementation with Priority Queue*