Proximity proofs

Thomas Coratger

1 An Introduction to Linear Codes

The journey into modern coding theory began in 1948 with Claude Shannon's groundbreaking paper, "A Mathematical Theory of Communication." He established that even over a noisy channel — like the vastness of space between a satellite and Earth, or a scratched compact disc — it's possible to transmit information with remarkable reliability. The key is to strategically add redundancy to the original message before sending it. This process, called **encoding**, transforms a message into a more robust form called a **codeword**. A **linear code** is a way to do this, using the structured language of linear algebra.

1.1 The Core Concepts

Before we build our first code, let's establish the fundamental concepts that allow us to measure information and errors.

1.1.1 The Alphabet, Codewords, and the Code Itself

First, we need an alphabet. In linear codes, this isn't just a set of symbols but a rich algebraic structure called a **finite field**, denoted $\mathbb{F}_q$. This is a set with q elements where we can perform addition, subtraction, multiplication, and division with well-defined rules.

Example: The Binary Field $\mathbb{F}_2$

The simplest alphabet is the binary field $\mathbb{F}_2 = \{0, 1\}$, where addition and multiplication are performed modulo 2. For instance, $1 + 1 = 0$.

With our alphabet, we can form strings. A **codeword** is a single vector (or string) of a fixed length, say n, composed of symbols from our alphabet. It represents one specific message after it has been encoded.

Example: A Binary Codeword

The vector $\mathbf{c} = (1, 0, 1, 1, 0)$ is a single codeword of length $n = 5$ in the space $\mathbb{F}_2^5$.

Finally, the **code** is the complete collection of all possible valid codewords. Think of it as a dictionary: each codeword is a valid entry, and the code, denoted $\mathcal{C}$, is the entire dictionary. The goal of decoding is to determine which "dictionary entry" was sent.

Example: A Simple Code

While $(1, 0, 1)$ is a single codeword, the set $\mathcal{C} = \{(0, 0, 0), (1, 0, 1), (0, 1, 1), (1, 1, 0)\}$ is the entire code.

1.1.2 Measuring Errors: Hamming Weight and Distance

To correct errors, we first need to quantify them. The Hamming metric provides an intuitive way to do this.

Hamming Weight The **Hamming weight** of a vector $\mathbf{u}$, denoted $w(\mathbf{u})$ or $|\mathbf{u}|_0$, is the number of its non-zero coordinates. It's a simple measure of how many positions are "active."

Example: Hamming Weight

For the vector $\mathbf{u} = (0, 1, 0, 1, 1) \in \mathbb{F}_2^5$, we count the number of ones. The Hamming weight is $w(\mathbf{u}) = 3$.

Hamming Distance The **Hamming distance** between two vectors $\mathbf{u}$ and $\mathbf{v}$, denoted $d(\mathbf{u}, \mathbf{v})$, is the number of positions in which their symbols differ. This is our fundamental measure of error — a distance of t means t symbols were flipped during transmission. For linear codes, there's a crucial link between distance and weight: the distance between two codewords is simply the weight of their difference: $d(\mathbf{u}, \mathbf{v}) = w(\mathbf{u} - \mathbf{v})$.

Example: Hamming Distance

Let $\mathbf{u} = (1, \mathbf{0}, 1, \mathbf{1})$ and $\mathbf{v} = (1, \mathbf{1}, 1, \mathbf{0})$. They differ in the 2nd and 4th positions, so their Hamming distance is $d(\mathbf{u}, \mathbf{v}) = 2$.

1.1.3 Minimum Distance

The single most important parameter determining a code's error-correction capability is its **minimum distance**, denoted ℓ. This is the smallest Hamming distance between any pair of distinct codewords in the entire code. A large minimum distance means the codewords are well-separated, making it harder for noise to change one codeword into another.

For linear codes, this simplifies beautifully: the minimum distance of the code is equal to the minimum Hamming weight of all its **non-zero** codewords.

$$\ell = \min_{\mathbf{u}, \mathbf{v} \in \mathcal{C}, \mathbf{u} \neq \mathbf{v}} d(\mathbf{u}, \mathbf{v}) = \min_{\mathbf{w} \in \mathcal{C}, \mathbf{w} \neq \mathbf{0}} w(\mathbf{w})$$

This is because if $\mathbf{u}$ and $\mathbf{v}$ are distinct codewords, their difference $\mathbf{u} - \mathbf{v}$ is also a non-zero codeword (due to the linearity of the code). So, finding the minimum separation between all pairs is the same as finding the "lightest" non-zero codeword.

1.2 Formal Definition of a Linear Code

With these concepts, we can state the formal definition. An $[n, k, \ell]_q$ linear code $\mathcal{C}$ is a k-dimensional linear subspace of the vector space $\mathbb{F}_q^n$ with minimum distance ℓ.

- n: The **block length**. This is the total length of each codeword.
- k: The **dimension**. This is the number of original message symbols. Since $\mathcal{C}$ is a k-dimensional subspace over $\mathbb{F}_q$, it contains exactly $|\mathcal{C}| = q^k$ unique codewords. The ratio $\rho = k/n$ is the **rate** of the code, measuring how much of the transmitted data is useful information.
- ℓ: The **minimum distance**, which determines the error-correcting capability. A code with minimum distance ℓ can detect up to $\ell - 1$ errors and correct up to $t = \lfloor(\ell - 1)/2\rfloor$ errors.
- q: The **size of the finite field** (the alphabet).

1.3 Constructing a Linear Code: The Generator Matrix

Because a linear code is a vector subspace, we can describe the entire set of q^k codewords using a small basis of just k vectors. We arrange these basis vectors as the rows of a $k \times n$ matrix called the **generator matrix**, G.

This provides a simple recipe for encoding: to encode a message vector $\mathbf{m} \in \mathbb{F}_q^k$, we compute the codeword $\mathbf{c}$ via matrix multiplication:

$$\mathbf{c} = \mathbf{m} \cdot G$$

The full code $\mathcal{C}$ is the set of all linear combinations of the rows of G.

Example: Constructing a $[3, 2, 2]_2$ Linear Code

Let's build a binary code using the following 2×3 generator matrix (we will see the details later on this generator matrix, the goal here is simply to see a first full example of a linear code):

$$G = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

Here, $n = 3$ and $k = 2$. We can generate all $2^k = 4$ codewords by multiplying each possible message vector $\mathbf{m} \in \mathbb{F}_2^2$ by G:

- For $\mathbf{m} = (0, 0)$: $\mathbf{c} = (0, 0) \cdot G = 0 \cdot (1, 0, 1) + 0 \cdot (0, 1, 1) = (\mathbf{0}, \mathbf{0}, \mathbf{0})$
- For $\mathbf{m} = (1, 0)$: $\mathbf{c} = (1, 0) \cdot G = 1 \cdot (1, 0, 1) + 0 \cdot (0, 1, 1) = (\mathbf{1}, \mathbf{0}, \mathbf{1})$
- For $\mathbf{m} = (0, 1)$: $\mathbf{c} = (0, 1) \cdot G = 0 \cdot (1, 0, 1) + 1 \cdot (0, 1, 1) = (\mathbf{0}, \mathbf{1}, \mathbf{1})$
- For $\mathbf{m} = (1, 1)$: $\mathbf{c} = (1, 1) \cdot G = 1 \cdot (1, 0, 1) + 1 \cdot (0, 1, 1) = (1 + 0, 0 + 1, 1 + 1) = (\mathbf{1}, \mathbf{1}, \mathbf{0})$

The complete code is $\mathcal{C} = \{(0, 0, 0), (1, 0, 1), (0, 1, 1), (1, 1, 0)\}$.
To find its minimum distance d, we find the weights of the non-zero codewords:

- $w(1, 0, 1) = 2$
- $w(0, 1, 1) = 2$
- $w(1, 1, 0) = 2$

The minimum non-zero weight is 2, so $\ell = 2$. This is a $[3, 2, 2]_2$ linear code.

1.4 How Good is a Code? The Singleton Bound

Now that we can build a code, a question arises: for a given length n, how much information (k) can we pack in while ensuring a certain error-correction capability (ℓ)? There is a fundamental trade-off, captured by the **Singleton Bound**.

For any $[n, k, \ell]_q$ code, the parameters must satisfy:

$$k \leq n - \ell + 1$$

This inequality sets a hard limit on the efficiency of any code. It tells us that we cannot arbitrarily increase both the dimension k (the amount of information) and the minimum distance ℓ (error-correction power) for a fixed block length n. Improving one often comes at the cost of the other.

1.4.1 Maximum Distance Separable (MDS) Codes

The most efficient codes are those that achieve equality in the Singleton Bound. A code is called a **Maximum Distance Separable (MDS) code** if its parameters satisfy:

$$k = n - \ell + 1$$

MDS codes are optimal because they offer the largest possible minimum distance for a given length and dimension, perfectly balancing information rate and error resilience.

Example: Checking for MDS Property

Let's check our $[3, 2, 2]_2$ example code against the Singleton Bound.

- The parameters are $n = 3$, $k = 2$, and $\ell = 2$.
- The bound states $k \leq n - \ell + 1$.
- Substituting the values: $2 \leq 3 - 2 + 1 \Rightarrow 2 \leq 2$.

The bound is met with equality! Our simple example code is an MDS code, meaning it's maximally efficient for its length and dimension.

1.5 The Encoding Process and Code Rate

The generator matrix provides a simple and powerful algebraic recipe for converting any message into its corresponding protected codeword. This process is called encoding. Here, we'll detail how it works and introduce the concept of code rate, a key measure of a code's efficiency.

1.5.1 The Encoding Map

The process of encoding is a linear map from the k-dimensional space of messages to the n-dimensional space of codewords:

$$\text{encode} : \mathbb{F}_q^k \to \mathbb{F}_q^n$$

Given a message vector $\mathbf{m} = (m_1, m_2, \ldots, m_k) \in \mathbb{F}_q^k$, the encoding is performed by multiplying it by the $k \times n$ generator matrix G:

$$\mathbf{c} = \mathbf{m} \cdot G$$

The resulting codeword $\mathbf{c}$ is a vector of length n that now includes structured redundancy.

A particularly useful type of generator matrix is one in **standard form**, $G = [I_k|A]$, where I_k is the $k \times k$ identity matrix. An encoder using such a matrix is called a **systematic encoder**, because the original message $\mathbf{m}$ appears unchanged in the first k positions of the codeword $\mathbf{c}$. The remaining $n - k$ positions are the added redundancy, often called parity check symbols.

Example: Encoding a Single Message

Let's use our $[3, 2, 2]_2$ code from the previous example, which has the generator matrix:

$$G = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

Suppose we want to encode the message $\mathbf{m} = (1, 1) \in \mathbb{F}_2^2$. We compute the product $\mathbf{m} \cdot G$:

$$\begin{aligned} \mathbf{c} &= (1, 1) \cdot \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \\ &= (1 \cdot 1 + 1 \cdot 0, \quad 1 \cdot 0 + 1 \cdot 1, \quad 1 \cdot 1 + 1 \cdot 1) \\ &= (1, 1, 0) \end{aligned}$$

The message $(1, 1)$ is successfully encoded into the codeword $(1, 1, 0) \in \mathcal{C}$. Notice that since this G is not in standard form, the original message does not appear directly in the codeword.

1.5.2 Measuring Efficiency: The Code Rate

The **rate** of a code, denoted by ρ (or often R), measures its efficiency. It's the ratio of the message length to the codeword length:

$$\rho = \frac{k}{n}$$

The rate is a value between 0 and 1. It tells us what fraction of each transmitted symbol is actual information versus redundancy.

- A **high rate** (e.g., $\rho = 0.9$) means the code is very efficient, with little redundancy. This allows for faster data transmission but offers less protection against errors.
- A **low rate** (e.g., $\rho = 0.1$) means the code is less efficient, with a lot of redundancy. This slows down data transmission but provides much stronger error correction.

The choice of rate is always a trade-off between efficiency and reliability, dictated by the noise level of the communication channel.

Example: Comparing Code Rates

Let's compare two different codes.

1. A code $\mathcal{C}_1$ has parameters $[n = 10, k = 5]$. It encodes 5-symbol messages into 10-symbol codewords. Its rate is:
$$\rho_1 = \frac{k}{n} = \frac{5}{10} = 0.5$$
Exactly half of the transmitted data is information, and half is redundancy.
2. The famous $[23, 12, 7]_2$ Golay code, $\mathcal{C}_2$, encodes 12-bit messages into 23-bit codewords. Its rate is:
$$\rho_2 = \frac{k}{n} = \frac{12}{23} \approx 0.52$$
This code is slightly more efficient than $\mathcal{C}_1$ and is renowned for its powerful error-correcting capabilities, making it a classic example of a well-balanced code.

1.6 The Unique Decoding Regime

Now that we can construct and evaluate codes, we arrive at the central question of decoding: if we receive a corrupted vector $\mathbf{y}$, when can we determine the original codeword $\mathbf{c}$ with absolute certainty? This is the domain of **unique decoding**.

The key insight is that the minimum distance ℓ creates a "safe zone" or protective bubble around each codeword. As long as the number of errors is small enough that the received vector $\mathbf{y}$ remains inside the original codeword's bubble, we can reverse the damage unambiguously.

1.6.1 The Unique Decoding Radius

Because the minimum distance between any two distinct codewords is ℓ, their protective bubbles can have a radius of up to (but not including) $\ell/2$ before they risk touching or overlapping. This critical value defines the code's error-correction limit.

The maximum number of errors that a code is guaranteed to correct is denoted by t. It is defined as:
$$t = \left\lfloor \frac{\ell - 1}{2} \right\rfloor$$
Any received vector with t or fewer errors will always be closer to the original codeword than to any other. This value t is often called the **packing radius** or the **unique decoding radius** of the code.

The Uniqueness Guarantee Why can a received word only be "very close" to one codeword at most? The guarantee comes from a proof by contradiction that uses the triangle inequality.

Let's imagine for a moment that a received word $\mathbf{w}$ *is* very close to two different codewords, $\mathbf{u}$ and $\mathbf{v}$. To be "very close" means the distance is less than half the minimum separation, $\ell/2$. So we assume:

- The number of errors between $\mathbf{w}$ and $\mathbf{u}$ is small: $d(\mathbf{w}, \mathbf{u}) < \ell/2$.
- The number of errors between $\mathbf{w}$ and $\mathbf{v}$ is also small: $d(\mathbf{w}, \mathbf{v}) < \ell/2$.

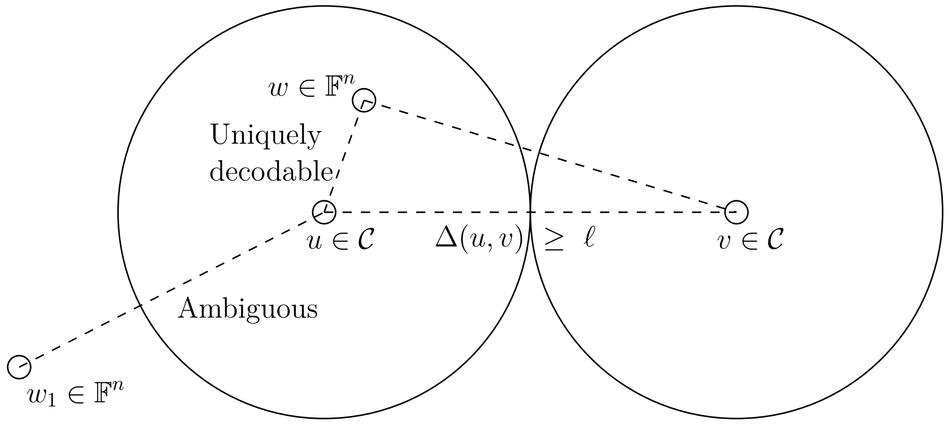


Figure 1: The unique decoding radius creates non-overlapping "safe zones" around codewords. A received $w \in \mathbb{F}^n$ inside a zone is uniquely decodable, while $w_1 \in \mathbb{F}^n$ outside is ambiguous.

The triangle inequality tells us that the direct path between two points is always the shortest. Going from **u** to **v** via the "detour" of **w** cannot be shorter:

$$\underbrace{d(\mathbf{u},\mathbf{v})}_{\text{Direct distance}} \leq \underbrace{d(\mathbf{u},\mathbf{w})}_{\text{Detour part 1}} + \underbrace{d(\mathbf{w},\mathbf{v})}_{\text{Detour part 2}}$$

Now, let's see what our assumption implies. Since each part of the detour is shorter than $\ell/2$, their sum must be less than ℓ:

$$d(\mathbf{u},\mathbf{v}) < \underbrace{\frac{\ell}{2}}_{\text{An upper bound for } d(\mathbf{u},\mathbf{w})} + \underbrace{\frac{\ell}{2}}_{\text{An upper bound for } d(\mathbf{v},\mathbf{w})} = \ell$$

This conclusion, $d(\mathbf{u},\mathbf{v}) < \ell$, says that the distance between codewords **u** and **v** is less than the *minimum possible distance* ℓ. This is a logical impossibility.

Therefore, our initial assumption was wrong. A received word **w** cannot be simultaneously "very close" (unique decoding regime) to two different codewords.

Example: Unique Decoding in Action

Case 1: An Ambiguous Code ($t = 0$) Let's first revisit our $[3, 2, 2]_2$ code.

- Parameters: $n = 3$, $k = 2$, $\ell = 2$.
- The code is $\mathcal{C} = \{(0, 0, 0), (1, 0, 1), (0, 1, 1), (1, 1, 0)\}$.
- Unique decoding radius: $t = \lfloor(\ell - 1)/2\rfloor = \lfloor(2 - 1)/2\rfloor = 0$.

A radius of $t = 0$ means this code can **guarantee to correct 0 errors**. It can detect 1 error, but cannot reliably correct it. Suppose the received vector is $\mathbf{w} = (1, 0, 0)$. The number of errors is 1, which is greater than t. As expected, decoding is ambiguous:

- $d(\mathbf{w}, (0, 0, 0)) = 1$
- $d(\mathbf{w}, (1, 0, 1)) = 1$
- $d(\mathbf{w}, (1, 1, 0)) = 1$

The received vector $\mathbf{w}$ is equally close to three different codewords, so the decoder fails.

Case 2: A Uniquely Decodable Code ($t = 1$) Now, let's consider a code designed for error correction: the $[3, 1, 3]_2$ repetition code.

- Parameters: $n = 3$, $k = 1$, $\ell = 3$.
- The code is $\mathcal{C} = \{(0, 0, 0), (1, 1, 1)\}$.
- Unique decoding radius: $t = \lfloor(3 - 1)/2\rfloor = 1$.

A radius of $t = 1$ means this code is **guaranteed to correct any single-bit error**. Suppose we send $\mathbf{c} = (1, 1, 1)$ and a single error occurs, flipping the first bit. The received vector is $\mathbf{w} = (0, 1, 1)$. The number of errors is 1, which is within our decoding radius ($1 \leq t$). Let's find the closest codeword to $\mathbf{w}$:

- $d(\mathbf{w}, (0, 0, 0)) = w(0, 1, 1) = 2$
- $d(\mathbf{w}, (1, 1, 1)) = w(1, 0, 0) = 1$

The received vector $\mathbf{w}$ is uniquely closest to $(1, 1, 1)$. The decoder correctly identifies the original codeword, successfully correcting the error.

1.7 Why Unique Decoding Isn't Enough

Unique decoding provides a wonderful guarantee, but it comes with a strict limitation: it only works if the number of errors is very small. If the noise on a channel is higher, a received word $\mathbf{w}$ is highly likely to fall outside of any "safe zone." This leads to a crucial question: just how many words are actually uniquely decodable?

As we will see, the answer is a surprisingly small fraction. This limitation is the primary motivation for more advanced techniques like **list decoding**.

Counting the "Safe" Words To understand the problem, let's count how many vectors are in one of these uniquely decodable bubbles. A **Hamming ball** of radius t around a codeword $\mathbf{u}$, denoted $B(\mathbf{u}, t)$, is the set of all vectors $\mathbf{w}$ that are at a distance of t or less from $\mathbf{u}$.

The number of vectors in this ball is the sum of the counts for every possible number of errors, from 0 up to t. To understand the formula's origin, let's first figure out how to count the number of vectors with *exactly* i errors. This is a two-step process:

1. **Choose the error positions:** First, where did the errors occur? We need to choose which i of

the n total positions in the vector were corrupted. The number of ways to do this is given by the binomial coefficient $\binom{n}{i}$.

2. **Choose the new symbols:** For each of the i positions that have an error, the original symbol must be changed to a different one. Our alphabet is the field $\mathbb{F}_q$, which has q total symbols. This means for each error, there are $(q-1)$ other symbols to choose from. Since there are i errors, there are $(q-1)^i$ ways to choose the new symbols.

Therefore, the total number of vectors with exactly i errors is the product of these two parts: $\binom{n}{i}(q-1)^i$. To get the total size of the Hamming ball, we simply sum the counts for all possible numbers of errors, from $i = 0$ (no errors) up to the maximum we can correct, t:

$$|B(\mathbf{u}, t)| = \sum_{i=0}^{t} \underbrace{\binom{n}{i}}_{\text{Ways to choose } i \text{ error positions}} \times \underbrace{(q-1)^i}_{\text{Ways to choose new symbols for them}}$$

This formula counts every possible vector that can be reached from $\mathbf{u}$ by introducing at most t errors.

A Quick Refresher on the Binomial Coefficient $\binom{n}{k}$

The notation $\binom{n}{k}$, read as "n choose k", represents the number of ways to select k items from a set of n distinct items, where the order of selection doesn't matter.
The general formula is:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

where $n!$ (n-factorial) is $n \times (n-1) \times \cdots \times 1$. In our main formula, $\binom{n}{i}$ is simply counting all the possible ways that i errors can be distributed among the n positions of a codeword.

Example: Counting Words in a Hamming Ball

Let's use a code over the ternary field $\mathbb{F}_3$ (so $q = 3$) with block length $n = 10$. Suppose its minimum distance is $\ell = 7$, which gives a unique decoding radius of $t = \lfloor (7-1)/2 \rfloor = 3$.
How many words are in the Hamming ball of radius 3 around a single codeword $\mathbf{u}$?

$$|B(\mathbf{u}, 3)| = \sum_{i=0}^{3} \binom{10}{i}(3-1)^i = \binom{10}{0}2^0 + \binom{10}{1}2^1 + \binom{10}{2}2^2 + \binom{10}{3}2^3$$

Let's calculate term by term:

- 0 errors: $\binom{10}{0} \cdot 1 = 1 \cdot 1 = 1$ (just the codeword itself)
- 1 error: $\binom{10}{1} \cdot 2 = 10 \cdot 2 = 20$
- 2 errors: $\binom{10}{2} \cdot 4 = 45 \cdot 4 = 180$
- 3 errors: $\binom{10}{3} \cdot 8 = 120 \cdot 8 = 960$

The total number of words in this single decoding bubble is $1 + 20 + 180 + 960 = \mathbf{1,161}$.
Now, let's compare this to the total number of possible vectors in the entire space $\mathbb{F}_3^{10}$. There are $3^{10} = \mathbf{59,049}$ possible vectors.
So, the "safe zone" for a single codeword covers only $1,161/59,049 \approx 1.9\%$ of the entire space. Even if we sum this over all codewords in a high-rate code, the total uniquely decodable volume remains a tiny fraction of the whole space.

The example and the demonstration above reveal something important. The set of all uniquely decodable words (the union of all Hamming balls around all codewords) is almost always *exponentially smaller* than the total space of all possible received words, $\mathbb{F}_q^n$.

This means that if a received word **w** suffers from more than t errors, it will land in the vast and ambiguous space between the decoding bubbles. In this scenario, asking the decoder for a single, unique answer is asking the wrong question.

This is precisely why we need **list decoding**. Instead of demanding a single answer, we relax the question to: can you give me a short list of *all* codewords that are reasonably close to the received word?

This approach acknowledges that ambiguity is likely and provides a way to manage it, forming the basis for many modern and powerful decoding algorithms.

1.8 List Decoding

We've seen that the guarantee of unique decoding is powerful but fragile. It only applies within a small "safe zone" around each codeword. If a received word **w** suffers from more errors than the unique decoding radius t, it lands in the vast, ambiguous space where it might be close to several codewords at once.

Instead of giving up, we can change our goal. This leads to the idea of **list decoding**.

1.8.1 Motivation: A More Realistic Question

The problem with unique decoding is that it insists on a single, unambiguous answer. When that's not possible, it fails. List decoding takes a more different approach by asking a more flexible question: Instead of the *one* true codeword, can you give me a short *list* of all plausible candidates?

The goal is no longer to find a single answer but to efficiently produce a small list of all codewords $\mathbf{u} \in \mathcal{C}$ that are within a certain distance e of the received word **w**.

The immediate question, however, is: how long could this list be? If it's potentially enormous, it's not very useful. This is where a remarkable result, the Johnson bound, provides a crucial guarantee.

1.8.2 The Johnson Bound: A Limit on Ambiguity

The **Johnson bound** is a universal "safety net" for decoding. It tells us that even when we are outside the unique decoding radius, the number of nearby codewords doesn't immediately explode. For a significant range of errors, the list of candidates remains small and manageable.

To state the bound formally, we use the *relative* parameters of the code:

- The **relative minimum distance**, $\mu = \ell/n$.
- The **relative decoding radius** (or fraction of errors), $\delta = e/n$, where e is the number of errors we want to tolerate.

The Johnson Bound

Let $\mathcal{C}$ be any code with relative minimum distance μ. For any received word **w**, the number of codewords within a relative distance δ is bounded, as long as δ is less than the **Johnson radius**:

$$\delta < 1 - \sqrt{1-\mu}$$

Inside this radius, the size of the list of candidate codewords is guaranteed to be small. Specifically, for binary codes ($q = 2$), the list size is bounded by approximately n. For general q, the bound is more complex but remains polynomial in n.

This is a powerful result because the bound depends only on the distance parameters μ and δ, not on the specific structure of the code.

The Johnson bound reveals that the ambiguity of decoding grows in a controlled way. It doesn't become chaotic as soon as we leave the unique decoding radius.

1.8.3 The Three Regimes of Decoding

We can now summarize the entire decoding landscape based on the fraction of errors, δ, we are trying to correct.

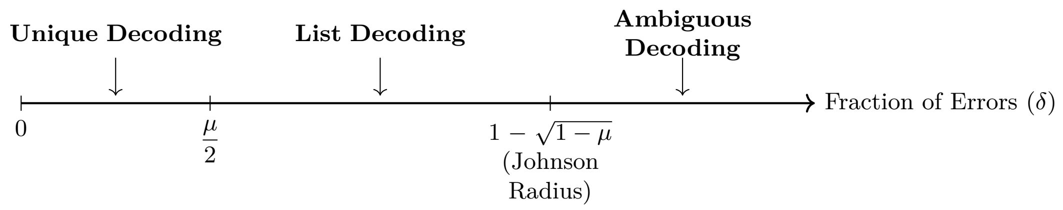


Figure 2: The three decoding regimes based on the fraction of errors δ.

1. **Unique Decoding Regime ($\delta < \mu/2$):** This is the "safe zone" we've already discussed. The list of candidates has at most one codeword.
2. **List Decoding Regime ($\mu/2 \leq \delta < 1-\sqrt{1-\mu}$):** Here, we can no longer guarantee a unique answer, but the Johnson bound assures us that the list of potential candidates is small. Efficient algorithms exist that can find this list.
3. **Ambiguous Decoding Regime ($\delta \geq 1-\sqrt{1-\mu}$):** Beyond the Johnson radius, the list of candidates could be exponentially large, and decoding becomes computationally infeasible for general codes.

The existence of the Johnson bound is crucial for modern coding theory. It proves that we can correct many more errors than the unique decoding radius suggests, as long as we are willing to accept a small list of answers instead of just one.

1.8.4 A Note on Terminology: δ-close and δ-far

To simplify our discussion of list decoding, it's helpful to have clear language to describe whether a received word is a plausible candidate for decoding or not. We use the terms **δ-close** and **δ-far** for this purpose.

Definitions

Let δ be a given fraction of errors (our relative decoding radius).

- We say a received word $\mathbf{w}$ is **δ-close** to the code $\mathcal{C}$ if there is *at least one* codeword $\mathbf{u} \in \mathcal{C}$ such that $d(\mathbf{w}, \mathbf{u}) \leq \delta n$. This means the list of potential candidates is not empty.
- We say a received word $\mathbf{w}$ is **δ-far** from the code $\mathcal{C}$ if *every* codeword $\mathbf{u} \in \mathcal{C}$ has a distance greater than δn from it, i.e., $d(\mathbf{w}, \mathbf{u}) > \delta n$. In this case, the list of candidates is empty, and a list decoder would return nothing.

This terminology provides a clean shorthand for discussing decoding problems. Instead of describing the conditions every time, we can simply ask: "Is the received word $\mathbf{w}$ δ-close to our code?" This captures the essential geometric question of whether $\mathbf{w}$ lies within the decoding radius of any codeword in $\mathcal{C}$.

2 Reed-Solomon Codes: Correcting Errors with Polynomials

Having established the fundamentals of linear codes, we now turn to one of the most powerful and widely used families of error-correcting codes in history: **Reed-Solomon (RS) codes**. Invented in 1960 by Irving S. Reed and Gustave Solomon, these codes are at the heart of technologies we use every day, from storing data on CDs and Blu-ray discs to ensuring the integrity of QR codes and transmitting data from deep space probes.

The genius of Reed-Solomon codes lies in a beautifully simple idea: **representing messages as polynomials**. By leveraging the unique properties of polynomials, we can add redundancy in a highly structured way, allowing us to correct not just single-bit flips, but entire chunks of corrupted data.

2.1 The Polynomial Toolkit

To understand RS codes, we first need to be comfortable with the tool they are built from: polynomials.

2.1.1 What is a Polynomial?

A polynomial is a mathematical expression consisting of variables and coefficients. For our purposes, a message polynomial $P(x)$ is a function of a variable x defined as:

$$P(x) = m_0 + m_1 x + m_2 x^2 + \cdots + m_{k-1} x^{k-1}$$

The values $m_0, m_1, \ldots, m_{k-1}$ are the **coefficients**, and the highest power of x with a non-zero coefficient is the **degree** of the polynomial. While we often think of coefficients and variables as real numbers, the real power of polynomials in coding theory comes from defining them over **finite fields**. This ensures all our calculations are exact and produce results that stay within our chosen alphabet.

2.1.2 The Fundamental Property: Uniqueness

The single most important property of polynomials for error correction is their uniqueness. A polynomial of degree $k-1$ is uniquely determined by any k of its points.

This means if you know the value of a degree-$(k-1)$ polynomial at k distinct points, you can perfectly reconstruct the one and only polynomial that passes through them. For example, two points define a unique line (degree 1), and three points define a unique parabola (degree 2).

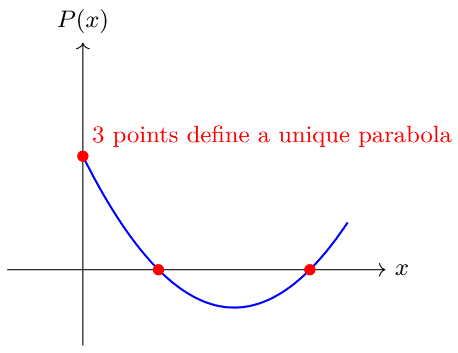


Figure 3: The uniqueness property of polynomials.

This property is what allows us to recover lost information. If a message is encoded as points on a polynomial, we only need to recover a sufficient number of those points (any k of them) to reconstruct the entire original message.

Example: Reconstructing a Polynomial

Suppose we are told that four points, $(-1, 2)$, $(0, 3)$, $(1, -5)$, and $(2, 1)$, lie on a polynomial of degree 3. Our message is encoded in the coefficients (m_0, m_1, m_2, m_3) of this polynomial:

$$P(x) = m_0 + m_1x + m_2x^2 + m_3x^3$$

By plugging in each point, we get a system of four linear equations with four unknowns:

$$\begin{aligned} P(-1) = m_0 - m_1 + m_2 - m_3 &= 2 \\ P(0) = m_0 &= 3 \\ P(1) = m_0 + m_1 + m_2 + m_3 &= -5 \\ P(2) = m_0 + 2m_1 + 4m_2 + 8m_3 &= 1 \end{aligned}$$

Solving this system yields a unique solution for the coefficients: $(m_0, m_1, m_2, m_3) = (3, -4.5, -5.5, 2)$. The fractions highlight a key issue with using real numbers, which we'll address later using finite fields. For now, the crucial insight is that the four points uniquely locked in the polynomial and thus the message.

2.2 The Core Idea: Encoding by Oversampling

The most intuitive way to understand RS encoding is to view it as "oversampling" a function. We take a message, turn it into a polynomial, and then evaluate that polynomial at more points than are strictly necessary. These extra points are our redundancy.

This leads to a **systematic** encoding, where the original message is embedded directly within the final codeword. Here's the process:

1. **Interpret the Message as Points:** Take a message block of k symbols, $\mathbf{m} = (m_0, m_1, \ldots, m_{k-1})$. Treat these as the y-values of a polynomial $P(x)$ at k predefined evaluation points, say $x = 0, 1, \ldots, k-1$. This gives us k points: $(0, m_0), (1, m_1), \ldots, (k-1, m_{k-1})$.
2. **Interpolate to Find the Polynomial:** Find the unique polynomial $P(x)$ of degree at most $k - 1$ that passes through these k points.
3. **Evaluate to Generate Redundancy:** "Oversample" the function by evaluating $P(x)$ at $n - k$ new points, say $x = k, k + 1, \ldots, n - 1$. These new values are our parity symbols.
4. **Form the Codeword:** The final codeword $\mathbf{c}$ is the list of all n evaluations. The first k symbols are the original message, and the next $n - k$ symbols are the redundancy.

Example: Systematic RS Encoding

Let's encode a message $\mathbf{m} = (3, 1)$ with $k = 2$ into a codeword of length $n = 4$ over a finite field (we'll use integers for simplicity, but imagine the math is modulo a prime, like 5).

1. **Message as Points:** We treat the message as the values of a polynomial $P(x)$ at $x = 0$ and $x = 1$. Our points are $(0, 3)$ and $(1, 1)$.
2. **Interpolate:** We need to find the unique line (degree $k - 1 = 1$) of the form $P(x) = a_0 + a_1 x$ that passes through these points.
   - $P(0) = a_0 = 3$
   - $P(1) = a_0 + a_1 = 1 \implies 3 + a_1 = 1 \implies a_1 = -2$

   So, our message polynomial is $P(x) = 3 - 2x$.
3. **Generate Redundancy:** We oversample by evaluating $P(x)$ at two new points, $x = 2$ and $x = 3$.
   - $P(2) = 3 - 2(2) = -1$
   - $P(3) = 3 - 2(3) = -3$
4. **Form the Codeword:** We concatenate the original message values with the new redundant values.

$$\mathbf{c} = (\underbrace{3, 1}_{\text{Message}}, \underbrace{-1, -3}_{\text{Redundancy}})$$

The final codeword is $(3, 1, -1, -3)$. If this is transmitted and one of the symbols is lost (an erasure), we can still recover the polynomial $P(x) = 3 - 2x$ from any of the remaining three points and regenerate the missing value.

2.3 Why Finite Fields are Essential

Throughout our examples, we've used integers for simplicity, but you may have noticed a problem: interpolation can produce fractions. Furthermore, the values of the evaluated points can become very large. Digital systems, however, work with fixed-size data types (like bytes) and cannot handle arbitrary fractions or infinite growth. **Finite fields** (or Galois Fields, GF) solve both problems perfectly.

1. **No Fractions:** In a finite field, division is well-defined and always results in another element of the field. All calculations are exact.
2. **Closure:** The result of any addition or multiplication is always another element within the finite set of the field. This prevents numbers from growing too large and ensures that symbols always fit within their designated size (e.g., 8 bits for $\mathbb{F}_{2^8}$).

By constructing polynomials over a finite field like $\mathbb{F}_{256}$, Reed-Solomon codes become perfectly suited for modern digital storage and communication systems.

3 Proximity Gaps: An All-or-Nothing Property of Codes

We've explored how codes are built and how they can be used to correct a limited number of errors. But in advanced applications like modern proof systems (e.g., STARKs) and secure multi-party computation, we face a more complex challenge: verifying the integrity of not just one, but a whole *batch* of alleged codewords simultaneously. This leads to a profound structural question about codes, the answer to which has significant real-world consequences.

3.1 The Motivation: Verifying a Batch of Codewords

Imagine a scenario where a prover claims that a large set of vectors $\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_l\}$ are all valid (or at least very close to valid) Reed-Solomon codewords. Verifying each vector individually would be slow and inefficient. A much faster approach is to ask the prover to combine them into a single vector using random coefficients r_i:

$$\mathbf{u}' = r_0\mathbf{u}_0 + r_1\mathbf{u}_1 + \cdots + r_l\mathbf{u}_l$$

The verifier then only needs to check if this single random combination, $\mathbf{u}'$, is close to the code.

This raises a critical security question: **Is this test sound?** If a malicious prover provides just one vector $\mathbf{u}_i$ that is very far from the code, can they cleverly construct the other vectors such that their random combination $\mathbf{u}'$ almost always looks valid? If so, the test is useless.

For this test to be reliable, we need a guarantee that if even one vector is "far," the random combination will also be "far" with high probability. This is precisely what the theory of **proximity gaps** provides.

3.2 The Proximity Gap Phenomenon

A proximity gap is a powerful "all-or-nothing" principle. It states that for certain codes and structured collections of vectors (like lines or planes, known as affine spaces), there is no middle ground. Every vector in the collection is "close" to being a valid codeword, or almost none are.

Analogy: The Proximity Gap

Scenario 1: "All-In" (Correlated Errors) Imagine two slightly corrupted vectors, $\mathbf{u}_0$ and $\mathbf{u}_1$, where the errors happen to occur in the **same position**.

- Error in $\mathbf{u}_0$: $(0, 0, 5, 0)$
- Error in $\mathbf{u}_1$: $(0, 0, 2, 0)$

Any vector on the line between them, $\mathbf{u}_z = \mathbf{u}_0 + z \cdot \mathbf{u}_1$, will have a combined error of $(0, 0, 5 + 2z, 0)$. Notice that for any value of z, the error is still confined to a single position. If one error is correctable, they all are. Thus, **all** vectors on the line are "close" to the code.

Scenario 2: "All-Out" (Uncorrelated Errors) Now, imagine the errors are in **different positions**.

- Error in $\mathbf{u}_0$: $(5, 0, 0, 0)$
- Error in $\mathbf{u}_1$: $(0, 0, 2, 0)$

The combined error is $(5, 0, 2z, 0)$. Unless $z = 0$, this vector has two errors. If the code can only correct one error, then every vector on the line (except possibly $\mathbf{u}_0$ itself) is now "far" from the code. Thus, **almost none** of the vectors are "close".

The Insight: The rigid algebraic structure of polynomials prevents widespread, random cancellation of errors. The errors either align perfectly from the start (correlated), or they don't, in which case their combination almost always results in more errors. There is no scenario where roughly half the combinations are close and half are far. This is the essence of the proximity gap.

3.2.1 Formal Definition

This "all-or-nothing" idea is captured formally in the definition of a proximity gap.

Definition: The Proximity Gap for Reed-Solomon Codes

Let our code V define a property (the property of being a valid codeword), and let $\mathcal{S}$ be a collection of sets of vectors (for example, all affine spaces in $\mathbb{F}_q^n$). We say that this collection $\mathcal{S}$

displays a (δ, ε)-proximity gap with respect to the code V if for every set S in the collection, exactly one of the following is true:

1. **All-In:** Every vector in the affine space S is δ-close to the code V.
$$\Pr_{\mathbf{u}\in S}[d(\mathbf{u}, V) \le \delta n] = 1$$
2. **All-Out:** At most a tiny fraction, ε, of vectors in S are δ-close to the code V.
$$\Pr_{\mathbf{u}\in S}[d(\mathbf{u}, V) \le \delta n] \le \varepsilon$$

The theorem provides the precise formulas for the error parameter ε based on the proximity parameter δ.

Parameters of the Gap for a Reed-Solomon Code $V = \mathbf{RS}[\mathbb{F}_q, D, k]$ Let V be a Reed-Solomon code defined by polynomials of degree at most k evaluated on a domain D of size n.

- The **rate** of the code is $\rho = (k+1)/n$.
- The **proximity parameter** δ is the fraction of errors. The gap holds for any δ in the range $(0, 1-\sqrt{\rho})$.
- The **error parameter** ε is a piecewise function that depends on which regime δ is in:

1. **Unique Decoding Bound:** For a small fraction of errors, where $\delta \in \left(0, \frac{1-\rho}{2}\right)$, the error parameter is:
$$\varepsilon = \varepsilon_U(q, n) := \frac{n}{q}$$
This gives a very strong guarantee when the field size q is larger than the block length n.
2. **Johnson Bound:** For a larger fraction of errors, where $\delta \in \left[\frac{1-\rho}{2}, 1-\sqrt{\rho}\right)$, we first define a "safety margin" $\eta := (1-\sqrt{\rho}) - \delta$. The error parameter is then:
$$\varepsilon = \varepsilon_J(q, n, \rho, \delta) := \frac{(k+1)^2}{\left(2\min\left(\eta, \frac{\sqrt{\rho}}{20}\right)\right)^7 q} = O\left(\frac{1}{(\eta\rho)^{O(1)}} \cdot \frac{n^2}{q}\right)$$
This shows the gap still holds, but requires a larger field size (roughly $q > n^2$) to ensure ε remains small.

3.3 A more visual intuition

To build a more intuitive understanding of the proximity gap, it helps to visualize the abstract spaces and relationships involved. While we are working in a high-dimensional vector space $\mathbb{F}_q^n$, we can represent the key concepts with a simplified 2D diagram.

In this visual explanation, we will represent the entire Reed-Solomon code V as a set of discrete points. The affine space S (which in the FRI protocol is a line of vectors like $\{\mathbf{u}_0 + z \cdot \mathbf{u}_1\}$) will be represented as a continuous 2D plane.

The concept of being "δ-close" can be visualized as a region, or Hamming ball, around each valid codeword. Any vector that falls inside the ball around a codeword $\mathbf{v}$ is considered close to it. The proximity gap theorem, then, is a statement about the nature of the intersection between the affine space S (our plane) and the union of all these Hamming balls. The theorem explores: what fraction of the plane S is covered by these "close" regions?

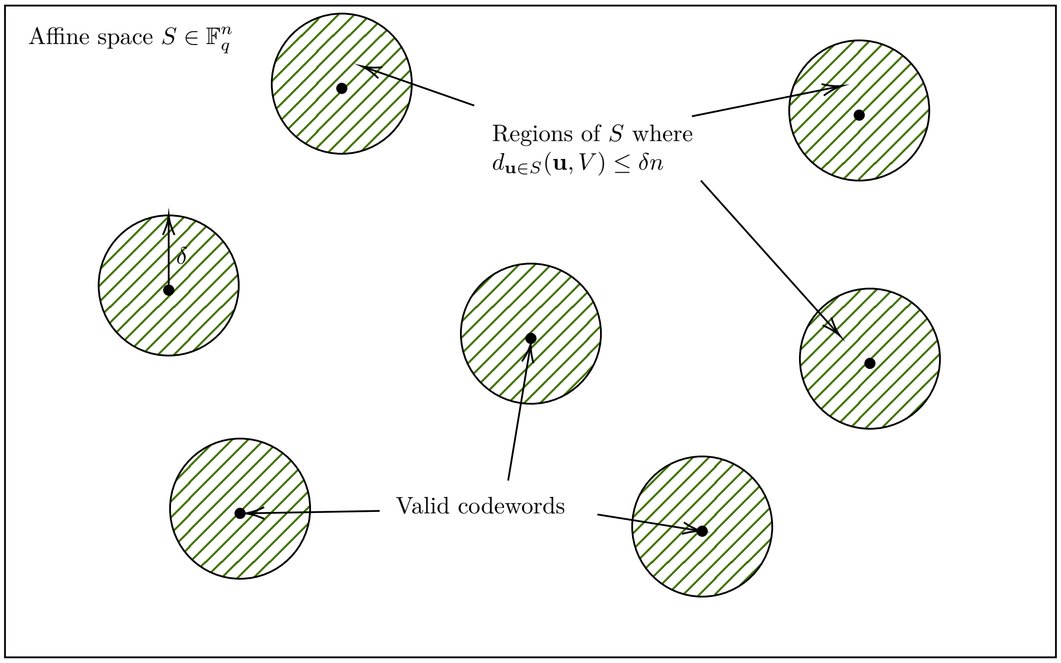

Affine space $S \in \mathbb{F}_q^n$
Regions of S where $d_{\mathbf{u}\in S}(\mathbf{u}, V) \leq \delta n$
δ
Valid codewords

Figure 4: A simplified, two-dimensional view of the setup for the Proximity Gap Theorem. The large rectangle represents an affine space S within the high-dimensional space $\mathbb{F}_q^n$. The shaded circles are Hamming balls, each containing all the vectors that are δ-close to a specific Reed-Solomon codeword. The theorem analyzes what fraction of the space S is covered by these balls.

3.3.1 Honest prover case

The "All-In" scenario of the proximity gap corresponds to the case of an honest prover. Here, the prover's initial claim is true: the batch of vectors he provides, $\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_l\}$, are close to valid Reed-Solomon codewords. This means every vector $\mathbf{u}_i$ is close to an element of the code space V.

The property that helps us in this situation is the linearity of Reed-Solomon codes. The set of all valid codewords, V, is a linear subspace, which means it is closed under linear combinations. When the verifier tests the prover, they are effectively sampling a random vector $\mathbf{u}'$ from the affine space S generated by the prover's vectors:

$$S = \left\{ \sum_{i=0}^{l} r_i \mathbf{u}_i \mid r_i \in \mathbb{F}_q \right\}$$

Since each initial vector $\mathbf{u}_i$ is an element of the linear subspace V, any linear combination of them is guaranteed to also be close to a valid codeword in V.

Consequence for the Proximity Test

Because every possible combined vector $\mathbf{u}' \in S$ is itself a perfect codeword, its distance to the code space V is zero.

$$d(\mathbf{u}', V) = 0 \quad \text{for all } \mathbf{u}' \in S$$

This is true for *any* choice of random coefficients. Therefore, the probability that a random vector from S is δ-close to the code is 1.

$$\Pr_{\mathbf{u}' \in S}[d(\mathbf{u}', V) \leq \delta n] = 1$$

This perfectly matches the **"All-In"** case of the proximity gap definition.

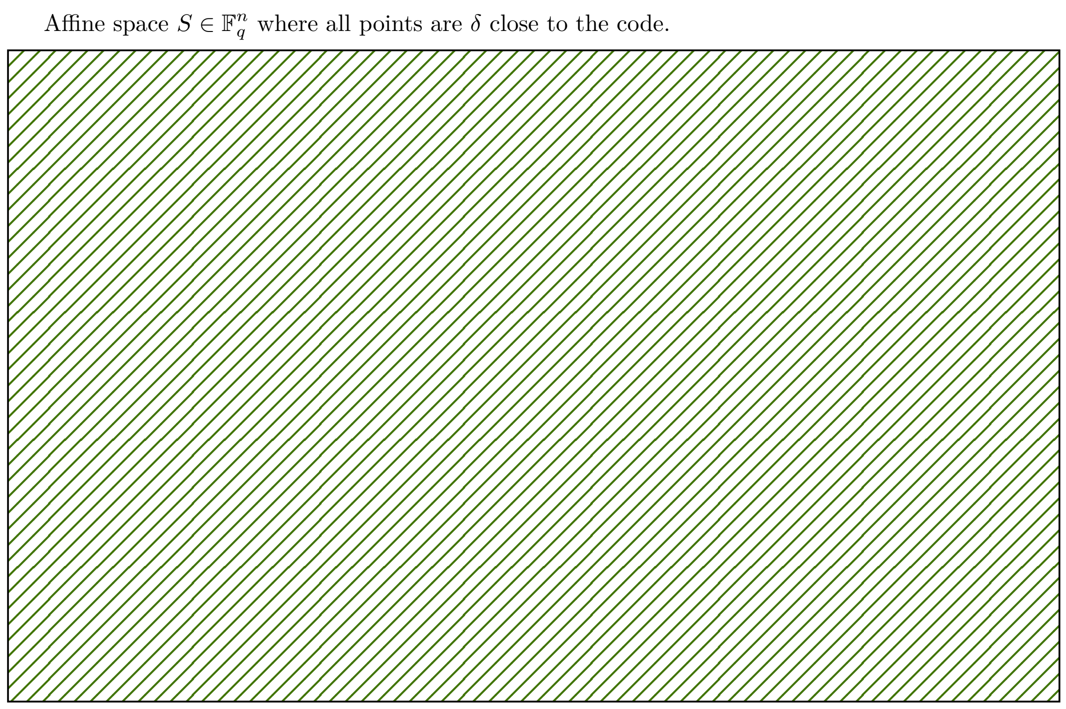

Affine space $S \in \mathbb{F}_q^n$ where all points are δ close to the code.

Figure 5: Visual representation of the "All-In" case. For an honest prover, the affine space S generated by their valid codewords is entirely contained within the code space V. Consequently, every vector in S has a distance of 0 to the code and is therefore δ-close, as indicated by the fully shaded region.

3.3.2 Malicious prover

Now, we consider the more complex case of a malicious prover. Here, the prover's claim is false: at least one of the vectors he provides in the batch $\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_l\}$ is far from the code space V.

The prover's goal is to construct the other vectors in such a way that when the verifier chooses a random linear combination, the resulting vector $\mathbf{u}'$ appears to be close to a valid codeword, allowing the prover to cheat.

This is where the proximity gap provides its strong security guarantee. Because at least one of the initial vectors is far from V, the generated affine space S cannot be "All-In." The proximity gap theorem therefore forces it into the only other possible state: the **"All-Out"** case.

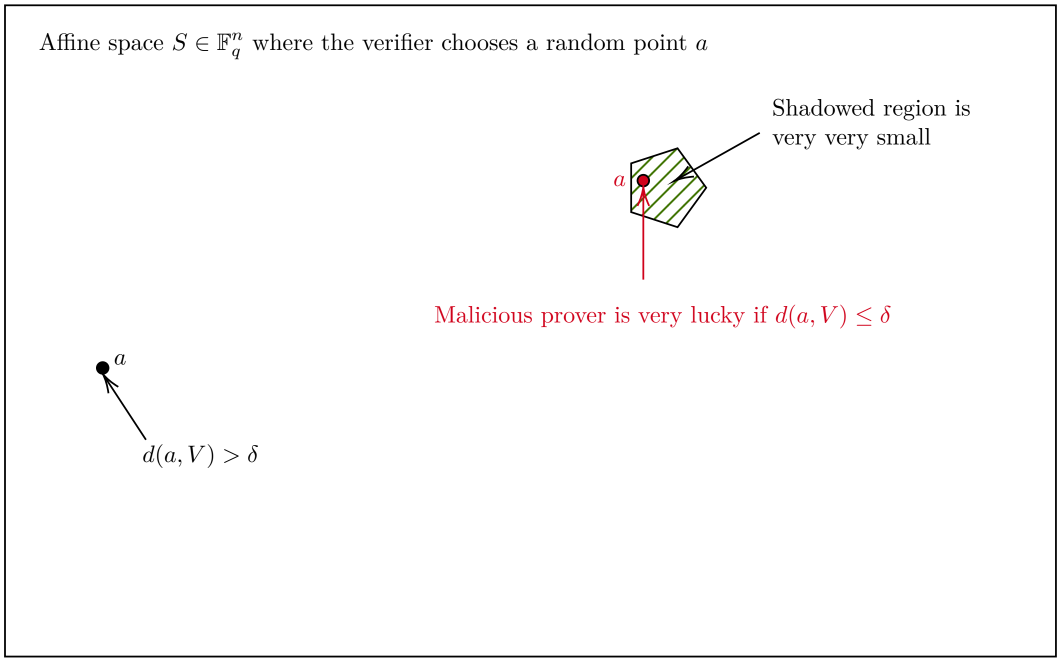
This means that the set of vectors within S that happen to be δ-close to the code is exceptionally small. The probability that a random linear combination falls into this "lucky" region is negligible, at most ε.

The Verifier's Two Hurdles for a Malicious Prover

For a malicious prover to succeed, they must pass two consecutive, low-probability checks:

1. **Hurdle 1: The Random Combination Check (Commit Phase)**
   The verifier selects random coefficients to form the vector $\mathbf{u}'$. The prover's first hope is to get lucky—that this specific combination lands in the tiny, ε-sized region of vectors that are δ-close to the code. The proximity gap theorem tells us the probability of this is negligible:
   $$\Pr[\text{Prover passes Hurdle 1}] \leq \varepsilon$$
2. **Hurdle 2: The Query Check (Query Phase)**
   In the far more likely event (with probability $1 - \varepsilon$), the vector $\mathbf{u}'$ is δ-far from the code. Now the prover faces a second hurdle. The verifier performs a small number of random queries on $\mathbf{u}'$. Since $\mathbf{u}'$ is δ-far, it must disagree with any valid codeword in at least δn positions. A few random queries are therefore very likely to hit one of these disagreements, causing the verifier to reject. The probability of passing this check is also very low.

The total probability of a malicious prover succeeding is bounded by the chance of passing Hurdle 1 **or** Hurdle 2. Because both probabilities are extremely small, the overall soundness of the protocol is very high.



Affine space $S \in \mathbb{F}_q^n$ where the verifier chooses a random point a
Shadowed region is very very small
a
Malicious prover is very lucky if $d(a, V) \leq \delta$
a
$d(a, V) > \delta$

Figure 6: Visual representation of the "All-Out" proximity gap scenario. When at least one initial vector is far from the code, the resulting affine space S is also "far." The set of vectors that are δ-close to the code (the shaded region) is extremely small, with a measure of at most ε. A random sample $\mathbf{u}'$ from S is therefore highly unlikely to fall into this lucky region.

3.4 The Main Result: Reed-Solomon Codes Have a Proximity Gap

The central theorem of the Ben-Sasson et al. paper[1] is that Reed-Solomon codes exhibit exactly this property. This provides the theoretical foundation needed to trust the random linear combination test.

The result is split into two regimes, depending on the fraction of errors δ we are willing to tolerate, compared to the code's rate $\rho = k/n$.

Theorem (Informal): Proximity Gap for RS Codes

The collection of all affine spaces has a strong proximity gap with respect to any Reed-Solomon code V. This holds for any fraction of errors δ up to the Johnson bound.

1. The Unique Decoding Regime ($\delta < \frac{1-\rho}{2}$) When the number of errors is small, the gap is extremely sharp. If an affine space is not "All-In", the fraction of close words is at most $\varepsilon = n/q$. For this to be a strong guarantee, we need the field size q to be larger than the block length n.

2. The List Decoding (Johnson) Regime ($\frac{1-\rho}{2} \leq \delta < 1 - \sqrt{\rho}$) Remarkably, the gap still holds for a much larger number of errors. The guarantee is slightly weaker, with the fraction of close words being at most $\varepsilon \approx O(n^2/q)$. To ensure this is a small number, we need a larger field, with q being roughly quadratic in n.

3.5 Application: Soundness of the FRI Protocol

The theory of proximity gaps has a direct and significant application in analyzing the security of modern proof systems. Specifically, it provides the mathematical foundation for the soundness of the FRI protocol[2] (Fast Reed-Solomon Interactive Oracle Proof of Proximity), which is a core component of transparent and quantum-resistant proof systems like STARKs.

3.5.1 The FRI Protocol: A Recursive Low-Degree Test

The primary function of the FRI protocol is to perform a **low-degree test**. In this context, a prover possesses a large dataset, represented as a function $f^{(0)}$ over a domain D_0, and asserts that this function is the evaluation of a low-degree polynomial $P_0(x)$ (i.e., that $f^{(0)}$ is a Reed-Solomon codeword). A verifier must be able to check this claim with high confidence by querying only a small number of points from the dataset.

FRI accomplishes this through a recursive process known as "folding." In each round, the prover reduces a given function to a new, smaller function, effectively halving its associated degree and domain size.

The FRI Folding Step

In each round i, the protocol begins with a function $f^{(i)}$ that is asserted to be of a certain low degree.

1. The prover splits $f^{(i)}$ into two components, often based on the even and odd coefficients of its polynomial representation:
$$f^{(i)}(x) = f^{(i)}_{\text{even}}(x^2) + x \cdot f^{(i)}_{\text{odd}}(x^2)$$
2. The verifier provides a random challenge scalar, $z_i \in \mathbb{F}_q$.
3. The prover uses this challenge to form a random linear combination of the two components. This defines the function for the next round:
$$f^{(i+1)}(y) = f^{(i)}_{\text{even}}(y) + z_i \cdot f^{(i)}_{\text{odd}}(y)$$
The domain of $f^{(i+1)}$ is the set of squared elements from the previous domain, thus its size is halved.

This recursive process continues until the function is small enough for the verifier to check completely by reading all of its values.

$$\begin{tikzcd}
 & f^{(0)}_{\text{even}}(x) + z_0 \cdot f^{(0)}_{\text{odd}}(x) \arrow[d, dashed] & f^{(1)}_{\text{even}}(x) + z_1 \cdot f^{(1)}_{\text{odd}}(x) \arrow[d, dashed] & f^{(2)}_{\text{even}}(x) + z_2 \cdot f^{(2)}_{\text{odd}}(x) \arrow[d, dashed] \\
f^{(0)}(x) \arrow[r] \arrow[ur, dashed] & f^{(1)}(x) \arrow[r] \arrow[ur, dashed] & f^{(2)}(x) \arrow[r] \arrow[ur, dashed] & f^{(3)}(x) \\
\text{degree} < k & \text{degree} < k/2 & \text{degree} < k/4 & \text{degree} < k/8
\end{tikzcd}$$

Figure 7: The recursive structure of the FRI protocol's commit phase. In each round i, the function $f^{(i)}$ is split into even and odd components, which are then combined with a random challenge z_i to produce the next function $f^{(i+1)}$, halving the effective degree at each step.

3.5.2 The Soundness Argument

The security analysis of the FRI protocol involves two properties:

- **Completeness:** An honest prover, starting with a true low-degree polynomial $f^{(0)}$, will always pass the verification. This property is straightforward to show, as the folding process preserves the low-degree nature of the function at each step. This is often referred to as the "top-down" direction of the argument.
- **Soundness:** A dishonest prover, starting with a function $f^{(0)}$ that is far from any low-degree polynomial, will be caught with high probability. Proving this is more complex. It requires a "bottom-up" argument.

The core of the soundness proof is to show that the low-degree property propagates backwards up the recursive ladder. If the verifier is convinced that the final function $f^{(r)}$ is low-degree, it must follow that the initial function $f^{(0)}$ was also close to being low-degree.

3.5.3 The Role of the Proximity Gap in the Soundness Proof

The potential vulnerability in the soundness argument lies in the folding step. A malicious prover could attempt to construct functions $f^{(i)}_{\text{even}}$ and $f^{(i)}_{\text{odd}}$ that are both far from being low-degree, but whose random combination,

$$f^{(i+1)} = f^{(i)}_{\text{even}} + z_i \cdot f^{(i)}_{\text{odd}}$$

is close to a low-degree polynomial for a non-negligible fraction of the verifier's random challenges z_i.

This is precisely the scenario that the proximity gap theorem for Reed-Solomon codes demonstrates is not possible. The set of functions $\{f^{(i)}_{\text{even}} + z \cdot f^{(i)}_{\text{odd}} \mid z \in \mathbb{F}_q\}$ forms an affine line. The proximity gap theorem dictates that such a line must be either "All-In" (all functions are close to the code) or "All-Out" (almost no functions are close). A middle ground, where a significant fraction but not all functions are close, does not exist.

How the Proximity Gap Secures FRI

The proximity gap provides the critical logical step in the bottom-up soundness argument.

1. The verifier confirms that the final function, $f^{(r)}$, is a low-degree polynomial.
2. The verifier then proceeds backwards to round i. He is convinced that $f^{(i+1)}$ has high agreement with a low-degree polynomial (a codeword).
3. This means that for the prover's chosen $f^{(i)}_{\text{even}}$ and $f^{(i)}_{\text{odd}}$, their combination using the random challenge z_i was close to a codeword.
4. For the prover's attack to be successful, this would need to hold true for a significant fraction of possible challenges z_i.
5. The Proximity Gap Theorem states that if the line of functions $\{f^{(i)}_{\text{even}} + z \cdot f^{(i)}_{\text{odd}}\}$ is close to the code for more than a tiny fraction ε of challenges, it must be the "All-In" case.
6. The "All-In" case implies a strong structural property known as **correlated agreement**: the functions $f^{(i)}_{\text{even}}$ and $f^{(i)}_{\text{odd}}$ must themselves be close to low-degree polynomials, with their errors located in the same positions.
7. This correlated agreement is the necessary condition to prove that the parent function, $f^{(i)}$, was also close to being a low-degree polynomial.

By providing this inferential link at each recursive step, the proximity gap guarantees that the low-degree property propagates from the final function all the way back to the initial one. This ensures the soundness of the random linear combination check at the heart of FRI, which in turn allows the protocol to achieve high security with few queries, leading to smaller proof sizes and faster verification times.

3.5.4 From Tighter Soundness to Smaller Proofs

The primary goal of cryptographic proof systems like STARKs is to achieve a high level of security (e.g., 128 bits) as efficiently as possible. Efficiency is measured by the proof size and verifier computation time. The soundness analysis of FRI, backed by the proximity gap theorem, directly impacts this efficiency.

The connection is based on the relationship between security, queries, and proof size:

- **Proof Size depends on Queries:** A significant portion of a STARK proof consists of Merkle authentication paths that substantiate the prover's answers to the verifier's random queries. Therefore, the total proof size is directly proportional to the number of queries performed.

- **Queries depend on Soundness:** To reach a target security level, the verifier must perform enough queries to reduce the probability of accepting a false proof to a negligible amount (e.g., 2^{-128}). The number of queries needed depends on the *soundness error* of the protocol — that is, how effective each individual query is at catching a malicious prover.

This is where the proximity gap theorem provides a crucial improvement. Older analyses of the FRI protocol had looser bounds on the soundness error, suggesting that each query was only moderately effective at detecting fraud. To compensate, a large number of queries were required to achieve the desired security level, leading to larger proofs.

The proximity gap theorem provides a **tighter soundness analysis**. It proves that the random linear combination check at the heart of FRI is exceptionally reliable. This means each query is far more powerful at detecting inconsistencies than previously understood.

In summary, because the proximity gap provides a stronger guarantee on the soundness of FRI, fewer queries are needed to achieve the same level of cryptographic security. This directly translates to fewer Merkle paths in the final STARK proof, resulting in a smaller proof size and faster verification time.

4 Proof of the Proximity Gap (Unique Decoding Regime)

This section details the proof for the proximity gap of Reed-Solomon codes in the unique decoding regime. The proof demonstrates that if a significant fraction of vectors on an affine line are close to a Reed-Solomon code, then the vectors defining the line must share a strong algebraic structure. This structure, in turn, forces the entire line to be close to the code.

4.1 The Goal and the Strategy

The formal statement we aim to prove, based on Theorem 4.1 in Ben-Sasson et al.[1], is as follows:

Theorem to Prove (Correlated Agreement)

Let V be a Reed-Solomon code of degree k over a domain $D \subset \mathbb{F}_q$. Let $\{\mathbf{u}_0 + z \cdot \mathbf{u}_1 \mid z \in \mathbb{F}_q\}$ be an affine line of vectors.
If the set $S = \{z \in \mathbb{F}_q \mid d(\mathbf{u}_0 + z \cdot \mathbf{u}_1, V) \leq \delta n\}$ is sufficiently large (specifically, $|S| > k + 1$), then there must exist two codewords, $\mathbf{v}_0, \mathbf{v}_1 \in V$, such that $\mathbf{u}_0$ and $\mathbf{v}_0$ agree on a large fraction of coordinates, and $\mathbf{u}_1$ and $\mathbf{v}_1$ agree on that *same* large fraction of coordinates. This property is known as **correlated agreement**.

The Proof Strategy The core of the proof is a shift in perspective. Instead of analyzing each of the $|S|$ "close" vectors individually, we analyze the entire line simultaneously using a higher level of algebraic abstraction.

1. **Create a Formal Object:** We represent the entire affine line as a single *formal vector* $\mathbf{w}(x)$ whose entries are linear polynomials in a formal variable Z. This vector is defined over a **function field** $\mathbb{F}_q(Z)$.
$$\mathbf{w}(x) = \mathbf{u}_0(x) + Z \cdot \mathbf{u}_1(x)$$

2. **Decode the Formal Object:** We run a classic Reed-Solomon decoding algorithm, the Berlekamp-Welch algorithm, on this single formal vector $\mathbf{w}$.
3. **Deduce Structure:** We show that the success of the decoder on many concrete instances (for $Z = z$ where $z \in S$) implies that the decoder must also succeed on the formal object. The output of this formal decoding reveals the correlated agreement between the original vectors $\mathbf{u}_0$ and $\mathbf{u}_1$.

4.2 The Toolkit: Algebraic Decoders and Lemmas

The proof relies on two key algebraic tools.

4.2.1 Tool 1: The Berlekamp-Welch Algorithm

The Berlekamp-Welch (BW) algorithm is a method for the unique decoding of Reed-Solomon codes. Its objective is to recover the original low-degree polynomial $P(X)$ from a received word $\mathbf{w}$ that may contain a limited number of errors at unknown locations.

The main challenge in error correction is that the locations of the errors are unknown. The BW algorithm's sidesteps this problem by creating a new equation that holds true for *all* positions, whether they are corrupted or not. It achieves this by turning the non-linear problem of finding error locations into a system of linear equations.

The Auxiliary Polynomials The algorithm introduces two auxiliary polynomials whose coefficients are the unknowns we need to solve for:

- An **Error-Locator Polynomial**, $A(X)$. This is a polynomial of degree t (the number of errors the code can correct) whose roots are the locations of the corrupted symbols. It is defined as:

$$A(X) = (X - e_1)(X - e_2) \dots (X - e_t)$$

  where $e_1, \dots, e_t$ are the positions of the errors. Crucially, we do not know these positions in advance; the algorithm will find the coefficients of $A(X)$ for us, which implicitly reveals the error locations. The defining property of this polynomial is that $A(x) = 0$ if and only if position x contains an error.

- A **Codeword-Evaluator Polynomial**, $B(X)$. This polynomial is defined as the product of the original (and unknown) message polynomial $P(X)$ and the error-locator polynomial $A(X)$:

$$B(X) = P(X) \cdot A(X)$$

The Key Equation Using these definitions, the BW algorithm establishes an equation that is valid for every point x in the evaluation domain D, where $\mathbf{w}(x)$ is the received symbol at that position:

$$A(x) \cdot \mathbf{w}(x) = B(x)$$

This identity holds universally due to the following case analysis:

1. **Case 1: Position x is correct.** By definition, the received symbol is the same as the original, so $\mathbf{w}(x) = P(x)$. The equation becomes $A(x) \cdot P(x) = B(x)$, which is true by the definition of $B(X)$.
2. **Case 2: Position x is corrupted.** In this case, $\mathbf{w}(x) \neq P(x)$. However, because x is an error location, it must be a root of the error-locator polynomial, meaning $A(x) = 0$. The equation becomes $0 \cdot \mathbf{w}(x) = 0$, which is always true.

Then, the master equation, $A(x)\mathbf{w}(x) = B(x)$, holds for every position, effectively masking the errors.

Solving the System The equation $A(x)\mathbf{w}(x) = B(x)$ is a *linear* equation in the unknown coefficients of $A(X)$ and $B(X)$. By evaluating this equation at every point $x \in D$ (of which there are n), we generate a system of n linear equations. The number of unknowns is the sum of the unknown coefficients in $A(X)$ and $B(X)$. For a code that corrects t errors, this gives us just enough equations to solve for the unknowns uniquely.

The Berlekamp-Welch Algorithm: A Summary

The goal is to recover the original polynomial $P(X)$ from a received word $\mathbf{w}$.

1. **Set up the System:** Construct a system of n linear equations by asserting that $A(x) \cdot \mathbf{w}(x) = B(x)$ for every position x in the domain. The variables are the unknown coefficients of the polynomials $A(X)$ and $B(X)$.
2. **Solve for the Polynomials:** Solve the linear system to find the unique coefficients of the error-locator $A(X)$ and the evaluator $B(X)$.
3. **Recover the Message:** The original message polynomial is recovered by performing polynomial division:
$$P(X) = \frac{B(X)}{A(X)}$$

If $\mathbf{w}$ is indeed close to a low-degree polynomial, a non-zero solution for $A(X)$ and $B(X)$ is guaranteed to exist, and $A(X)$ is guaranteed to divide $B(X)$ without a remainder.

A Closer Look: The System of Equations The equation $A(x) \cdot \mathbf{w}(x) = B(x)$ is linear in the unknown coefficients of $A(X)$ and $B(X)$. To understand the setup, let's count these unknown coefficients.

Counting the Unknowns Assume the code can correct up to t errors and the message polynomial $P(X)$ has degree less than k.

- For $A(X)$: This polynomial must have a root at each of the t error locations. Its degree is therefore exactly t. A polynomial of degree t has $t+1$ coefficients. By convention, we set the leading coefficient to 1 (making it *monic*), which leaves exactly t unknown coefficients.
$$A(X) = \underbrace{1 \cdot X^t}_{\text{Known}} + \underbrace{a_{t-1}X^{t-1} + \cdots + a_0}_{\text{t unknown coefficients}}$$
- For $B(X)$: This polynomial is the product $B(X) = P(X) \cdot A(X)$. The degree of a product is the sum of the degrees:
$$\deg(B) = \deg(P) + \deg(A) \le (k-1) + t = k + t - 1$$
A polynomial of degree $k+t-1$ has $k+t$ coefficients in total:
$$B(X) = \underbrace{b_{k+t-1}X^{k+t-1} + \cdots + b_0}_{\text{k+t unknown coefficients}}$$

In total, we have $t + (k+t) = k + 2t$ unknown coefficients to solve for.

Constructing the Equations For each of the n received points $(x_i, \mathbf{w}_i)$, we write down our key equation. This gives us n linear equations, one of which is shown below:
$$\underbrace{(b_{k+t-1}x_i^{k+t-1} + \cdots + b_0)}_{\text{Linear expression in unknown } b_j\text{'s}} = \mathbf{w}_i \cdot \underbrace{(x_i^t + a_{t-1}x_i^{t-1} + \cdots + a_0)}_{\text{Linear expression in unknown } a_j\text{'s}}$$

Since the code is designed such that $n \ge k + 2t$, we have at least as many equations as unknowns, which allows us to find a unique solution for the coefficients.

4.2.2 Tool 2: The Polishchuk-Spielman Lemma

This lemma provides the logical bridge from a property observed in many specific instances to a property that must hold formally and universally. It allows us to "lift" a conclusion from a collection of concrete observations to a single, abstract algebraic truth.

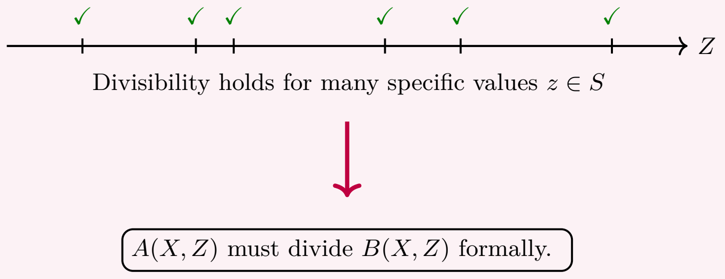
Intuition: From Slices to the Whole Imagine a two-variable polynomial, $B(X, Z)$, as a surface. When we fix the value of Z to a specific constant, z, we are taking a vertical "slice" of that surface, which results in a simple one-variable curve, $B(X, z)$.

The situation in our proof is that we have two such surfaces, $A(X, Z)$ and $B(X, Z)$. We don't know their formal relationship, but we do observe that for many different slices (i.e., for many values of z), the curve $A(X, z)$ divides the curve $B(X, z)$. It is highly unlikely for this divisibility to occur repeatedly by coincidence. The Polishchuk-Spielman lemma formalizes this intuition, stating that if this pattern holds for a sufficient number of slices, it cannot be an accident.

The Polishchuk-Spielman Lemma (Principle)

If the univariate polynomial $A(X, z)$ divides $B(X, z)$ for a sufficiently large number of distinct values of z, then the bivariate polynomial $A(X, Z)$ must formally divide $B(X, Z)$. This means there exists a third polynomial, $P(X, Z)$, such that:

$$B(X, Z) = A(X, Z) \cdot P(X, Z)$$


✓ ✓ ✓ ✓ ✓ ✓
Z
Divisibility holds for many specific values $z \in S$
$A(X, Z)$ must divide $B(X, Z)$ formally.

This principle allows us to deduce a strong, formal algebraic property from a collection of individual observations.

4.3 The Proof

With these tools, the proof proceeds as follows.

4.3.1 Step 1: Constructing a Formal Solution with Berlekamp-Welch

Our first step is to apply the Berlekamp-Welch algorithm to our formal word $\mathbf{w}(x) = \mathbf{u}_0(x) + Z \cdot \mathbf{u}_1(x)$. This requires us to find two formal polynomials, $A(X, Z)$ and $B(X, Z)$, that satisfy the key equation for all $x \in D$:

$$A(x, Z) \cdot \big(\mathbf{u}_0(x) + Z \cdot \mathbf{u}_1(x)\big) = B(x, Z)$$

The challenge is to prove that a non-trivial solution (where A and B are not both the zero polynomial) is guaranteed to exist.

From Polynomial Equation to Linear System The equation above might seem complex, but it's actually a system of linear equations in disguise. The true *unknowns* are the coefficients of the polynomials $A(X, Z)$ and $B(X, Z)$. These coefficients are not simple numbers; they are polynomials in the formal variable Z. Let's denote them $A_j(Z)$ and $B_j(Z)$.

For each evaluation point $x_i \in D$, we get one linear equation. If we have n points in our domain, we have a system of n linear equations. This can be written in matrix form:

$$\mathbf{M}(Z) \cdot \mathbf{c}(Z) = \mathbf{0}$$

Here, $\mathbf{c}(Z)$ is a vector containing all the unknown coefficient polynomials, and $\mathbf{M}(Z)$ is a large matrix whose entries are themselves polynomials in Z.

Illustration: A Small Example of the Matrix $\mathbf{M}(Z)$

Let's consider a small code with the following parameters:

- A message polynomial of degree at most $k-1=1$ (so dimension $k=2$).
- The ability to correct $t=1$ error.
- An evaluation domain of $n=4$ points, $\{x_1, x_2, x_3, x_4\}$.

The Berlekamp-Welch algorithm needs to find the unknown polynomials $A(X,Z)$ and $B(X,Z)$. Let's determine their form:

- $A(X,Z)$ is the error-locator of degree $t=1$. It has the form $A(X,Z) = A_1(Z)X + A_0(Z)$.
- $B(X,Z)$ is the product $P(X)A(X,Z)$. Its degree is at most $(k-1)+t = 1+1 = 2$. It has the form $B(X,Z) = B_2(Z)X^2 + B_1(Z)X + B_0(Z)$.

The unknowns are the five coefficient functions: $\{A_1(Z), A_0(Z), B_2(Z), B_1(Z), B_0(Z)\}$.
The key equation is $A(x,Z)\cdot \mathbf{w}(x,Z) - B(x,Z) = 0$. For a single point x_i, this expands to:

$$\underbrace{(A_1(Z)x_i + A_0(Z))}_{\text{Evaluated } A} \cdot \underbrace{(\mathbf{u}_0(x_i) + Z\mathbf{u}_1(x_i))}_{\text{Formal word } \mathbf{w}_i(Z)} - \underbrace{\left(B_2(Z)x_i^2 + B_1(Z)x_i + B_0(Z)\right)}_{\text{Evaluated } B} = 0$$

By rearranging and grouping the five unknown coefficient functions, we get a linear equation. We generate one such equation for each of the $n=4$ points. This gives us a system of 4 linear equations for 5 unknowns, which can be written in matrix form $\mathbf{M}(Z)\cdot\mathbf{c}(Z) = \mathbf{0}$:

$$\underbrace{\begin{pmatrix} x_1\mathbf{w}_1(Z) & \mathbf{w}_1(Z) & -x_1^2 & -x_1 & -1 \\ x_2\mathbf{w}_2(Z) & \mathbf{w}_2(Z) & -x_2^2 & -x_2 & -1 \\ x_3\mathbf{w}_3(Z) & \mathbf{w}_3(Z) & -x_3^2 & -x_3 & -1 \\ x_4\mathbf{w}_4(Z) & \mathbf{w}_4(Z) & -x_4^2 & -x_4 & -1 \end{pmatrix}}_{\text{The } 4\times 5 \text{ matrix } \mathbf{M}(Z)} \cdot \begin{pmatrix} A_1(Z) \\ A_0(Z) \\ B_2(Z) \\ B_1(Z) \\ B_0(Z) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

In this specific example, since the matrix has more columns (5 unknowns) than rows (4 equations), linear algebra guarantees that a non-trivial (non-zero) solution must exist.
In the general proof, the matrix is not always "wide". For cases where the number of equations is greater than or equal to the number of unknowns, the more powerful determinant argument (using the "Too Many Roots" principle) is required to prove that the matrix is rank-deficient and that a non-trivial formal solution still exists.

Example: The General Case (A "Tall" or Square Matrix)

Let's now consider a more general case where the simple "more columns than rows" argument does not apply. Imagine a code with these parameters:

- Message polynomial of degree at most $k-1=1$ (dimension $k=2$).
- Ability to correct $t=1$ error.
- An evaluation domain of $n=5$ points, $\{x_1, \ldots, x_5\}$.

Counting Unknowns and Equations The unknown polynomials are:

- $A(X,Z) = A_1(Z)X + A_0(Z)$, with 2 unknown coefficient functions.
- $B(X,Z)$, with $\deg(B) \le (k-1)+t = 1+1 = 2$. So, $B(X,Z) = B_2(Z)X^2 + B_1(Z)X + B_0(Z)$, with 3 unknown coefficient functions.

The total number of unknown functions is $2+3=5$. The number of equations is the number of evaluation points, $n=5$.

The Matrix System The key equation is $A(x,Z)\cdot\mathbf{w}(x,Z)-B(x,Z)=0$. For a single point x_i, this expands to:

$$\underbrace{(A_1(Z)x_i+A_0(Z))}_{\text{Evaluated } A}\cdot\underbrace{(\mathbf{u}_0(x_i)+Z\mathbf{u}_1(x_i))}_{\text{Formal word } \mathbf{w}_i(Z)}-\underbrace{(B_2(Z)x_i^2+B_1(Z)x_i+B_0(Z))}_{\text{Evaluated } B}=0$$

Writing one such equation for each of the 5 points gives us a **square** 5×5 system of linear equations, $\mathbf{M}(Z)\cdot\mathbf{c}(Z)=\mathbf{0}$:

$$\underbrace{\begin{pmatrix} x_1\mathbf{w}_1(Z) & \mathbf{w}_1(Z) & -x_1^2 & -x_1 & -1 \\ x_2\mathbf{w}_2(Z) & \mathbf{w}_2(Z) & -x_2^2 & -x_2 & -1 \\ x_3\mathbf{w}_3(Z) & \mathbf{w}_3(Z) & -x_3^2 & -x_3 & -1 \\ x_4\mathbf{w}_4(Z) & \mathbf{w}_4(Z) & -x_4^2 & -x_4 & -1 \\ x_5\mathbf{w}_5(Z) & \mathbf{w}_5(Z) & -x_5^2 & -x_5 & -1 \end{pmatrix}}_{\text{The } 5\times 5 \text{ matrix } \mathbf{M}(Z)}\cdot\begin{pmatrix} A_1(Z) \\ A_0(Z) \\ B_2(Z) \\ B_1(Z) \\ B_0(Z) \end{pmatrix}=\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The Need for the Determinant Argument In this case, because the matrix is square, a non-trivial solution is **not** automatically guaranteed. From linear algebra, a non-trivial solution exists if and only if the determinant of the matrix is zero:

$$\det(\mathbf{M}(Z))=0$$

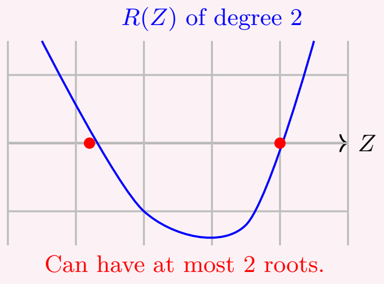
This is precisely why the proof must show that this determinant polynomial, $R(Z)=\det(\mathbf{M}(Z))$, is identically zero. This is accomplished using the "Too Many Roots" principle, which handles all cases, whether the matrix is wide, square, or tall.

Proving a Solution Exists: The Determinant Argument From linear algebra, a homogeneous system of equations like $\mathbf{M}(Z)\cdot\mathbf{c}(Z)=\mathbf{0}$ has a non-trivial solution if and only if its matrix $\mathbf{M}(Z)$ is **rank-deficient**. This means the rows (or columns) are not all linearly independent; there is some redundancy in the equations. The standard way to check for rank deficiency is to compute the determinants of all the largest possible square submatrices (known as maximal minors). If all of these determinants are zero, the matrix is rank-deficient and a solution is guaranteed.

Our goal is therefore to prove that for our formal matrix $\mathbf{M}(Z)$, the determinant of every maximal square submatrix is identically the zero polynomial. We can prove this for any such determinant, which we'll call $R(Z)$, by using what I will call the "Too Many Roots" principle (for ease of understanding and identification).

The "Too Many Roots" Principle

A non-zero, one-variable polynomial of degree d can have at most d distinct roots.



The Implication: If we find that a polynomial of degree at most d actually has $d+1$ or more roots, it must be the **zero polynomial**, $R(Z)\equiv 0$.

The proof follows a clear, three-step logical chain:

1. **The Determinant is a Low-Degree Polynomial:** Since the entries of the formal matrix $\mathbf{M}(Z)$ are polynomials in Z, the determinant of any of its square submatrices, $R(Z)$, is also a polynomial in Z. A careful analysis of the matrix structure, as shown in the paper, reveals that the degree of this polynomial is low. Specifically, it is at most $t+1$, where t is the number of errors the code can correct.
$$\deg(R(Z)) \leq t+1$$

2. **The "Good" Set Provides Many Roots:** This is the important step that connects our initial assumption to the formal system. We assumed that for every z in our "good" set S, the concrete word $\mathbf{u}_0 + z \cdot \mathbf{u}_1$ is close to a codeword. The Berlekamp-Welch algorithm guarantees that the corresponding concrete system, $\mathbf{M}(z) \cdot \mathbf{c} = \mathbf{0}$, has a non-trivial solution for each of these z's.

   This can only be true if each concrete matrix $\mathbf{M}(z)$ is rank-deficient, which means the determinant of our chosen square submatrix, when evaluated at z, must be zero.
$$\underbrace{R(z) = 0}_{\text{The determinant is zero...}} \qquad \underbrace{\text{for all } z \in S}_{\text{...at every "good" point.}}$$

   This gives us a powerful piece of information: every single element in the set S must be a root of the polynomial $R(Z)$.

3. **Applying the Principle to Prove $R(Z) \equiv 0$:** The final step brings the previous two points together. The setup of the theorem requires that the size of our "good" set is greater than the maximum possible degree of the determinant polynomial, i.e., $|S| > t+1$. We are therefore faced with the following situation:
$$\text{A polynomial } R(Z) \text{ with } \underbrace{\deg(R(Z)) \leq t+1}_{\text{Low Degree}} \text{ that has } \underbrace{|S| > t+1}_{\text{Many Roots}}$$

   According to the "Too Many Roots" principle, this is impossible unless $R(Z)$ is the zero polynomial.

Since this argument holds for any maximal square submatrix of $\mathbf{M}(Z)$, we conclude that the formal matrix is rank-deficient. Therefore, a formal, non-trivial solution for $A(X, Z)$ and $B(X, Z)$ is guaranteed to exist.

Full Example: Proving a Formal Solution Exists

Let's walk through the entire process with a numerical example.

1. The Setup Suppose we are working over the finite field $\mathbb{F}_5 = \{0, 1, 2, 3, 4\}$. We have an affine line of words defined by $\mathbf{u}_0$ and $\mathbf{u}_1$ over the domain $D = \{0, 1, 2, 3\}$ $(n = 4)$.

- $\mathbf{u}_0 = (1, 0, 3, 0)$
- $\mathbf{u}_1 = (3, 2, 2, 1)$

Our assumption is that for a "good" set of values S, the word $\mathbf{w}_z = \mathbf{u}_0 + z \cdot \mathbf{u}_1$ is close to the Reed-Solomon code of degree $k-1 = 1$. Let's say we have found three such values, $S = \{1, 2, 3\}$. (This corresponds to $t = 1$ error correction, where we need $|S| > t+1$, so $3 > 2$).

2. The Formal Matrix $\mathbf{M}(Z)$ We set up the Berlekamp-Welch system for our formal word $\mathbf{w}(x, Z) = \mathbf{u}_0(x) + Z\mathbf{u}_1(x)$. With $n = 4, k = 2, t = 1$, we get a 4×4 square system $\mathbf{M}(Z) \cdot \mathbf{c}(Z) = \mathbf{0}$. The entries of $\mathbf{M}(Z)$ are polynomials in Z:
$$\mathbf{M}(Z) = \begin{pmatrix} x_1\mathbf{w}_1(Z) & \mathbf{w}_1(Z) & -x_1 & -1 \\ x_2\mathbf{w}_2(Z) & \mathbf{w}_2(Z) & -x_2 & -1 \\ x_3\mathbf{w}_3(Z) & \mathbf{w}_3(Z) & -x_3 & -1 \\ x_4\mathbf{w}_4(Z) & \mathbf{w}_4(Z) & -x_4 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 1+3Z & 0 & -1 \\ 1(0+2Z) & 0+2Z & -1 & -1 \\ 2(3+2Z) & 3+2Z & -2 & -1 \\ 3(0+1Z) & 0+1Z & -3 & -1 \end{pmatrix}$$

3. The Determinant as a Polynomial $R(Z)$ To prove a formal solution exists, we must show that $\det(\mathbf{M}(Z)) = 0$. This determinant, $R(Z)$, is a polynomial in Z. By inspecting the matrix, we can see the highest power of Z in any entry is 1. The determinant calculation involves products of entries, and in a 4×4 matrix, the highest possible degree is bounded. In this case, it can be shown that $\deg(R(Z)) \leq t + 1 = 2$.

4. Finding Roots of $R(Z)$ using the set S This is the key step. We know that for every $z \in S = \{1, 2, 3\}$, the concrete system has a solution, which means the determinant of the concrete matrix $\mathbf{M}(z)$ must be zero. Let's check for $z = 1$:

$$\mathbf{M}(1) = \begin{pmatrix} 0 & 4 & 0 & -1 \\ 2 & 2 & -1 & -1 \\ 0 & 0 & -2 & -1 \\ 3 & 1 & -3 & -1 \end{pmatrix} \pmod 5$$

The determinant of this matrix is indeed 0. Therefore, $R(1) = 0$, and $Z = 1$ is a root of $R(Z)$. If we do the same for $z = 2$ and $z = 3$, we would find that $\det(\mathbf{M}(2)) = 0$ and $\det(\mathbf{M}(3)) = 0$. So, $Z = 2$ and $Z = 3$ are also roots.

5. Applying the "Too Many Roots" Principle We have now established the following facts:

- $R(Z)$ is a polynomial with $\deg(R(Z)) \leq 2$.
- $R(Z)$ has at least 3 roots (at $Z = 1, 2, 3$).

Since the number of roots (3) is greater than the maximum possible degree (2), the "Too Many Roots" principle tells us that $R(Z)$ cannot be a non-zero polynomial. It must be identically zero: $R(Z) \equiv 0$. Because $\det(\mathbf{M}(Z))$ is formally zero, a non-trivial formal solution for $A(X, Z)$ and $B(X, Z)$ is guaranteed to exist.

4.3.2 Step 2: Lifting Divisibility and Deriving the Gap

From Step 1, we have successfully constructed two non-zero formal polynomials, $A(X, Z)$ and $B(X, Z)$. The next, and most critical, part of the proof is to show that they have the right relationship: that $A(X, Z)$ formally divides $B(X, Z)$.

We begin with our central assumption: for every z in our "good" set S, the concrete word $\mathbf{w}_z = \mathbf{u}_0 + z\mathbf{u}_1$ is close to a unique codeword $P_z(X)$. As a consequence of the Berlekamp-Welch algorithm, this means that for each of these $z \in S$:

$$\frac{B(X, z)}{A(X, z)} = P_z(X)$$

This observation (that $A(X, z)$ divides $B(X, z)$ for all $|S|$ values of z) is the input to the Polishchuk-Spielman lemma. The lemma's role is to "lift" this property from the concrete instances to the formal object, but its power is conditional.

The Lemma's Condition and the Origin of the Gap

The Polishchuk-Spielman lemma only works if a specific inequality, which balances the complexity of the problem across its two variables (X and Z), is satisfied. Informally, the condition is:

$$\underbrace{\frac{k + t}{n}}_{\text{Complexity from X}} + \underbrace{\frac{t + 1}{|S|}}_{\text{Complexity from Z}} < 1$$

This inequality creates a "balancing act". The first term is fixed by the code's parameters. For the sum to be less than 1, the second term must be small, which requires its denominator, $|S|$, to be large. But how large?

Deriving the Threshold We can find the required size of $|S|$ by solving the inequality. First, we isolate the term containing $|S|$:

$$\frac{t+1}{|S|} < 1 - \frac{k+t}{n} \implies \frac{t+1}{|S|} < \frac{n-k-t}{n}$$

Now, we can solve for $|S|$ by inverting both sides (which flips the inequality sign):

$$|S| > n \cdot \underbrace{\frac{t+1}{n-k-t}}_{\text{This fraction is close to 1}}$$

The crucial insight comes from the parameters of any code in the unique decoding regime. For such codes, the block length n must be greater than the message size k plus twice the number of correctable errors t (i.e., $n > k + 2t$). This implies that $n - k - t > t$.
Because the denominator $(n-k-t)$ is always slightly larger than the numerator (t), the fraction is always less than or equal to 1. In the "worst-case" scenario (where the code's parameters are tightest), this fraction is almost exactly 1. The proof therefore uses the clean, slightly stronger condition that $|S|$ must be greater than n itself. This ensures the lemma's inequality is always satisfied.
This leads directly to the proximity gap:

1. The lemma provides the engine for the proof, but requires the condition $|S| > n$ to be met.

2. The proof therefore establishes the following powerful implication:

$$\text{If } |S| > n \quad \implies \quad \text{(Lemma applies, forcing correlated agreement)} \quad \implies \quad |S| = q$$

3. This implication, If $|S| > n$ then $|S| = q$, when stated in terms of probabilities, creates the "forbidden zone" $(n/q, 1)$ and gives us the error parameter $\varepsilon = n/q$.

The proximity gap is therefore a direct consequence of the algebraic constraints imposed by the lemma.

With this understanding, the final step of the proof is clear. Our initial assumption in the proximity gap theorem is that we are in a situation where the fraction of good words is greater than $\varepsilon = n/q$. This is equivalent to assuming $|S| > n$.

Because this condition is met, we satisfy the requirements of the Polishchuk-Spielman lemma. This allows us to perform the crucial "lift" and conclude that because divisibility holds for many specific slices, it must hold for the formal object itself. Therefore, $A(X, Z)$ must formally divide $B(X, Z)$.

4.3.3 Step 3: Unveiling the Correlated Agreement

The result of Step 2 is a formal decoded polynomial, $P(X, Z) = B(X, Z)/A(X, Z)$. The Polishchuk-Spielman lemma provides one more piece of information: this resulting polynomial must have a simple structure. Specifically, it must be linear in Z. This allows us to write it as:

$$P(X, Z) = \mathbf{v}_0(X) + Z \cdot \mathbf{v}_1(X)$$

where $\mathbf{v}_0(X)$ and $\mathbf{v}_1(X)$ are themselves low-degree polynomials (and thus represent valid Reed-Solomon codewords).

The final step is to relate this "perfect" formal line back to our original, potentially corrupted formal word $\mathbf{w}(x, Z)$. The Berlekamp-Welch properties guarantee that $\mathbf{w}(x, Z)$ is close to $P(X, Z)$; that is, they must agree on a large fraction of coordinates $x \in D$. For any such coordinate x where they agree, the following formal identity in Z must hold:

$$\underbrace{\mathbf{u}_0(x) + Z \cdot \mathbf{u}_1(x)}_{\text{Original Formal Word}} = \underbrace{\mathbf{v}_0(x) + Z \cdot \mathbf{v}_1(x)}_{\text{Decoded Formal Codeword}}$$

Since this is an equality between two linear polynomials in the formal variable Z, their corresponding coefficients must be identical. This forces two conditions to be true simultaneously for this large set of agreeing positions:

1. $\mathbf{u}_0(x) = \mathbf{v}_0(x)$
2. $\mathbf{u}_1(x) = \mathbf{v}_1(x)$

4.3.4 Conclusion of the Proof

This result is precisely the **correlated agreement** we sought. It demonstrates that the errors in $\mathbf{u}_0$ (positions where it differs from the codeword $\mathbf{v}_0$) and the errors in $\mathbf{u}_1$ (where it differs from $\mathbf{v}_1$) must occur on the exact same, small set of coordinates.

If this correlated agreement holds, then for any $z \in \mathbb{F}_q$, the errors in the combined word $\mathbf{u}_0 + z\mathbf{u}_1$ are also confined to that same set of positions. Consequently, every vector on the line is close to the corresponding codeword $\mathbf{v}_0 + z\mathbf{v}_1$. This is the "All-In" case.

The proof has shown that if an affine line is not "All-Out," the underlying algebraic structure forces it into the "All-In" case. No middle ground is possible, which completes the proof of the proximity gap for the unique decoding regime.

5 Proof of the Proximity Gap (List Decoding Regime Up to the Johnson Bound)

TODO

6 Advanced Proximity Testing: The WHIR Protocol

While the FRI protocol and the proximity gap theorem represent a cornerstone of modern proof systems, the intense demands of real-world applications, particularly within the Ethereum ecosystem, are pushing the boundaries of what's possible. The relentless drive for scalability, post-quantum security, and efficiency has ignited a search for protocols that are not just faster, but fundamentally better suited to new computational paradigms, leading to a major architectural shift from univariate to multilinear polynomials. In this demanding landscape, the ultimate goal is to optimize the trifecta of performance: minimizing prover computation, shrinking proof sizes, and slashing the verifier's workload, as this directly translates into lower on-chain gas fees and higher network throughput.

One of the most promising recent developments in this area is WHIR, a recursive, hash-based proximity testing protocol. WHIR is designed for super-fast verification, often running in hundreds of microseconds, a significant improvement over the millisecond-scale verification times of many earlier systems. It achieves this by cleverly integrating constraint-checking directly into the proximity test itself.

WHIR can be seen as a sophisticated evolution of protocols like FRI and Basefold. It introduces a nice trade-off: the prover performs slightly more complex operations, and in return, the verifier can achieve the same level of security with significantly fewer queries. The reduction of both the proof size and the verifier work are the key to its attractiveness.

6.1 The Core Idea: Integrating Constraints and Folding

At its heart, WHIR is a proximity test for a special type of code called a **constrained Reed-Solomon code**. Instead of just proving that a function f is close to *some* low-degree polynomial, WHIR proves that f is close to a low-degree polynomial that also satisfies a specific algebraic constraint, such as an evaluation at a point or a more general sumcheck claim.

To achieve its speed, WHIR combines three ideas:

1. **Recursive Folding (inspired by STIR):** Like FRI, WHIR recursively reduces the size of the problem. However, each WHIR round can eliminate multiple variables at once (say, k variables), while only halving the domain size. This rapidly improves the code's rate, which is a key factor in reducing the number of queries needed for soundness.
2. **Integrated Sumcheck (inspired by Basefold):** The algebraic constraint is not checked separately at the end. Instead, it is folded down together with the function in each round, using a lightweight sumcheck protocol. This keeps the verifier's work minimal at each step.
3. **Out-of-Domain Sampling (inspired by DEEP-FRI):** To ensure soundness in the list-decoding regime, WHIR uses out-of-domain sampling. This technique forces a malicious prover to commit to a *single* polynomial candidate, collapsing any ambiguity that might arise from a list of potential codewords.

These components work together to create a protocol where each recursive step is both sound and extremely cheap for the verifier.

6.2 A Round of WHIR in Detail: From a Large Problem to a Small One

Let's break down the mechanics of a single WHIR reduction step, which recursively transforms a large, complex proximity problem into a smaller, more manageable one. Suppose the prover starts with a function f on a domain L and claims it is a codeword satisfying an initial algebraic property (e.g., a sumcheck claim). A WHIR round reduces this to a similar, but much smaller, problem about a new function g.

The entire process, illustrated in Figure 8, can be understood as a sequence of four main phases.

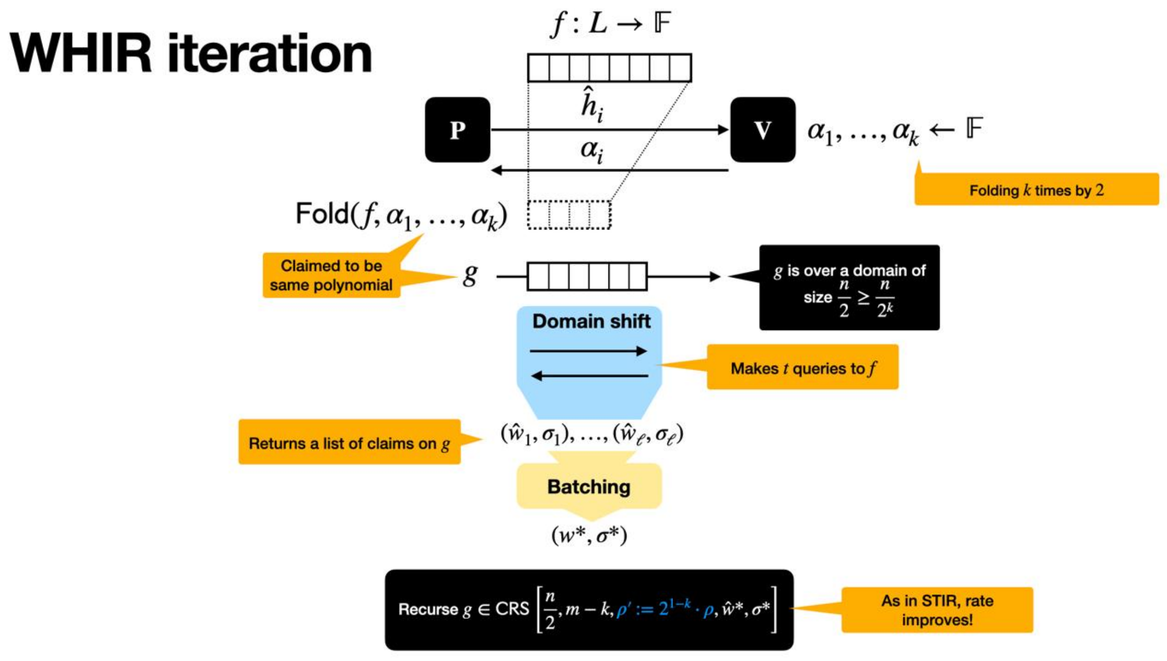

WHIR iteration
$f : L \to \mathbb{F}$
P
V
$\hat{h}_i$
α_i
$\alpha_1, \ldots, \alpha_k \leftarrow \mathbb{F}$
Folding k times by 2
Fold$(f, \alpha_1, \ldots, \alpha_k)$
Claimed to be same polynomial
g
g is over a domain of size $\frac{n}{2} \geq \frac{n}{2^k}$
Domain shift
Makes t queries to f
Returns a list of claims on g
$(\hat{w}_1, \sigma_1), \ldots, (\hat{w}_\ell, \sigma_\ell)$
Batching
(w^*, σ^*)
Recurse $g \in \text{CRS}\left[\frac{n}{2}, m-k, \rho' := 2^{1-k} \cdot \rho, \hat{w}^*, \sigma^*\right]$
As in STIR, rate improves!

Figure 8: A WHIR iteration

6.2.1 Phase 1: Interaction and Folding

The round begins with an interaction where the verifier provides the prover with randomness.

- **Prover-Verifier Exchange:** The verifier (V) sends a vector of k random challenges, $\alpha_1, \ldots, \alpha_k$, to the prover (P).

- **Prover's Task:** The prover uses these challenges for two parallel tasks:
    1. To run k rounds of a sumcheck protocol on the initial algebraic constraint, producing intermediate sumcheck polynomials $\hat{h}_i$.
    2. To compute a new, smaller function g by **folding** the original function f using the same challenges: $g = \text{Fold}(f, \alpha_1, \ldots, \alpha_k)$.

At the end of this phase, the prover commits to the resulting function g. This function lives on a new, smaller domain (a "domain shift") and is claimed to be the correct, folded representation of f.

6.2.2 Phase 2: Consistency Checks

The verifier now needs to check if the prover was honest. Was g really generated correctly from f? To do this, the verifier makes a small number of queries to the original function f to generate a "list of claims" that g must satisfy. This is a DEEP-style verification.

- **Out-of-Domain Query to Pinpoint a Polynomial:** The verifier picks a random point z_0 from **outside** g's primary commitment domain and asks the prover for the evaluation $v_0 = g(z_0)$. This crucial query forces the prover to commit to a *single* unique polynomial from any potential list of candidates, effectively collapsing list-decoding ambiguity.
- **Consistency Queries:** The verifier also picks several random points $z_1, \ldots, z_s$ where the value of g can be predicted from f. The verifier queries f to compute these expected values, creating a set of consistency checks that an honest g must pass.

6.2.3 Phase 3: Batching the Claims

At this point, the verifier has generated multiple algebraic claims that the polynomial underlying g must satisfy: (1) the specialized sumcheck claim, (2) the out-of-domain evaluation claim, and (3) several folding consistency claims. Checking them individually would be inefficient.

Instead, as shown in the "Batching" step of the diagram, the verifier uses an additional random challenge to combine all these separate claims $(\hat{w}_1, \sigma_1), \ldots, (\hat{w}_\ell, \sigma_\ell)$ into a single, compact constraint $(\hat{w}^*, \sigma^*)$. This is a technique that preserves soundness while dramatically reducing the verifier's work.

6.2.4 Phase 4: The Recursive Claim

The round concludes by establishing a new, smaller proximity problem. The verifier's task is now reduced to checking if the function g is close to a codeword in a new Constrained Reed-Solomon (CRS) code, defined by the single batched constraint $(\hat{w}^*, \sigma^*)$:

$$g \in \text{CRS}\left[\frac{n}{2}, m-k, \rho', \hat{w}^*, \sigma^*\right]$$

As the diagram highlights, this new problem is not just smaller, but also has an improved rate. The new rate ρ' is given by $\rho' = 2^{1-k}\rho$. This is because we reduced the polynomial's complexity by a factor of 2^k (by eliminating k variables) while only halving the domain size. This rate improvement at each step is the engine behind WHIR's low query complexity and fast verification. The protocol now recurses on this new, more efficient claim.

6.3 Soundness of a WHIR Round

The security of WHIR comes from the combined strength of its three phases. If a malicious prover starts with a function f_0 that is far from any valid codeword, he will be caught with very high probability. The analysis follows a **round-by-round soundness** approach, where we show that if a prover can pass one phase of the protocol, it must be because their claims in the previous phase were already valid. A failure at any stage results in rejection.

Let's assume the initial claim is that the prover's function f_0 on domain D_0 is θ_0-close to a polynomial p_0 that satisfies an inner product claim $\langle P_0, R_0 \rangle_{H_n} = c_0$. A malicious prover starts with an f_0 that violates this.

6.3.1 Soundness of the Sumcheck and Commit Phase

This phase consists of k rounds of the sumcheck protocol. The soundness relies on a generalization of the proximity gap theorem, often called a **correlated agreement theorem with constraints**.

The core argument is that algebraic consistency is preserved backwards in time. If a prover can consistently provide sumcheck polynomials $q_i(X)$ that satisfy the verifier's checks, it implies their data from the previous step was already consistent.

Let's formalize this. In each step i (from 1 to k), the protocol reduces a claim about a (virtual) folded function f_{i-1} to a claim about $f_i = \text{fold}_{\lambda_i}(f_{i-1})$.

- **Initial claim (R_{i-1}):** f_{i-1} is θ_0-close to a polynomial p_{i-1} whose multilinear version P_{i-1} satisfies the $(i-1)$-th sumcheck claim.
- **New claim (R_i):** After receiving the prover's polynomial $q_{i-1}(X)$ and providing a challenge λ_i, the new folded function f_i is θ_0-close to a polynomial p_i satisfying the new, specialized sumcheck claim.

The correlated agreement theorem with constraints guarantees that if the new claim R_i holds with non-negligible probability over the choice of λ_i, then the initial claim R_{i-1} must have been true. A malicious prover starting with a false claim for f_0 cannot create a folded function f_k that satisfies its constraints, except with a tiny probability ε_i at each step.

The total soundness error for this phase is the maximum of these round errors.

$$\varepsilon_{\text{sumcheck}} = \max_{1 \le i \le k} \varepsilon_i$$

Crucially, this holds even though the intermediate functions $f_1, \dots, f_{k-1}$ are not explicitly committed.

6.3.2 Soundness of the Query Phase

This is where WHIR's unique security properties come into play. After the commit phase, a malicious prover may have provided a commitment f_1 over the domain D_1 that is not close to a single unique polynomial, but rather to a small *list* of them. The query phase is designed to make this ambiguity impossible by forcing the prover onto a single, verifiable path.

The Out-of-Domain Query (z_0): Collapsing the List The primary goal of the out-of-domain query is to defeat list decoding attacks. Let's assume the prover's commitment f_1 is θ_1-close to a list of candidate polynomials, $\Lambda(f_1, \theta_1)$. Let the size of this list be bounded by $\ell_1(\theta_1)$, as given by the Guruswami-Sudan bound.

Why Out-of-Domain? The Power of a New Data Point

A natural question is: why must the query z_0 be *outside* the domain D_1?

- **In-Domain queries are insufficient:** If the verifier chose a point $x_q \in D_1$, the prover's answer would simply be the already-committed value $f_1(x_q)$. By definition, every polynomial $p(X)$ on the candidate list $\Lambda(f_1, \theta_1)$ is already close to f_1, so they likely agree with f_1 at x_q. An in-domain query provides no new information to distinguish between the candidates on the list; it only re-confirms the data that created the ambiguity in the first place.
- **Out-of-Domain queries force a choice:** An OOD query at $z_0 \notin D_1$ forces the prover to provide a *new* piece of information: the value v_0. This value acts as a filter. Since distinct low-degree polynomials will almost certainly have different values at a random point, the answer v_0 effectively forces the prover to commit to the single polynomial on the list for which $p(z_0) = v_0$. Any other candidate is immediately invalidated. This is what makes the OOD query a powerful tool for collapsing the list of possibilities.

The verifier's strategy is to pick a random point z_0 from the very large set $\mathbb{F} \setminus D_1$ and ask for the evaluation $v_0 = f_1(z_0)$. A cheating prover gets caught if this single query is sufficient to distinguish between the different polynomial candidates on the list. The only way for the prover to succeed is if he gets lucky and at least two distinct polynomials on the list, say $p_a(X)$ and $p_b(X)$, happen to have the same value at z_0.

We can bound the probability of this lucky event using the Schwartz-Zippel Lemma. The difference $p_a(X) - p_b(X)$ is a non-zero polynomial of degree less than 2^{n-k}. The probability that it evaluates to zero at a random point z_0 is:

$$\Pr_{z_0 \in \mathbb{F} \setminus D_1}[p_a(z_0) = p_b(z_0)] = \Pr_{z_0 \in \mathbb{F} \setminus D_1}[(p_a - p_b)(z_0) = 0] \leq \frac{\deg(p_a - p_b)}{|\mathbb{F} \setminus D_1|} < \frac{2^{n-k}}{|\mathbb{F} \setminus D_1|}$$

To find the probability that *any* pair on the list agrees, we use a union bound. The number of distinct pairs of polynomials in a list of size $\ell_1(\theta_1)$ is precisely $\binom{\ell_1(\theta_1)}{2}$. Therefore, the total probability of failure for this step is:

$$\varepsilon_{\text{list-collapse}} \leq \binom{\ell_1(\theta_1)}{2} \cdot \frac{2^{n-k}}{|\mathbb{F} \setminus D_1|}$$

Connecting to the Paper's Formula

The paper [3] states the error term as $\frac{\ell_1(\theta_1)^2}{2} \cdot \frac{2^{n-k}}{|\mathbb{F} \setminus D_1|}$. The relationship is as follows:

$$\binom{\ell_1(\theta_1)}{2} = \frac{\ell_1(\theta_1) \cdot (\ell_1(\theta_1) - 1)}{2} = \frac{\ell_1(\theta_1)^2 - \ell_1(\theta_1)}{2}$$

For a large list size $\ell_1(\theta_1)$, the ℓ_1^2 term dominates, so $\frac{\ell_1^2 - \ell_1}{2} \approx \frac{\ell_1^2}{2}$. It is common in cryptographic proofs to use the simpler, slightly looser upper bound of $O(\ell^2)$ for clarity, as the asymptotic behavior is the same. The formula with the binomial coefficient is the more precise bound.

The Folding Consistency Queries ($z_1, \ldots, z_{s_0}$): Verifying the Fold Now that the out-of-domain query has forced the prover to implicitly commit to a *single* polynomial candidate, $p_1(X)$, the verifier must check if this candidate is the authentic folding of the original function f_0.

The verifier samples s_0 points z_j from the folded domain $\pi^k(D_0)$. For each point, the verifier computes the correct value $v_j = \text{fold}_{\lambda_1,\ldots,\lambda_k}(f_0)(z_j)$ by querying the oracle for f_0. If the prover's chosen polynomial p_1 is actually θ_0-far from this true folded function, then a random query z_j will land on a position where they disagree with probability at least θ_0.

The probability that a single query *fails* to detect a mismatch is at most $(1 - \theta_0)$. Therefore, the probability that all s_0 independent queries fail to detect the fraud is:

$$\varepsilon_{\text{sampling}} = (1 - \theta_0)^{s_0}$$

By choosing a moderate number of queries for s_0, this failure probability can be made exponentially small.

Total Soundness Error of the Query Phase The total soundness error of the query phase is the probability that the prover passes *either* by getting lucky on the list-collapsing step *or* by evading detection in the sampling step. Combining these, we get the final expression for the soundness error, as stated in the lemma:

$$\varepsilon_{k+1} = \underbrace{\binom{\ell_1(\theta_1)}{2} \cdot \frac{2^{n-k}}{|\mathbb{F} \setminus D_1|}}_{\text{Prob. that } z_0 \text{ fails to isolate a unique candidate}} + \underbrace{(1 - \theta_0)^{s_0}}_{\text{Prob. that sampling fails to detect inconsistency}}$$

This two defense makes the query phase extremely robust against a malicious prover, even one who is trying to leverage the ambiguities of list decoding.

6.3.3 Soundness of the Aggregation Step: Batching Claims for Efficiency

The final phase of a WHIR round is aggregation, also known as batching. After the prover has successfully navigated the query phase, the verifier is left with a single candidate polynomial, $p_1(X)$, and a list of properties that this polynomial must satisfy. Checking each property individually would be inefficient.

Instead, the verifier combines all these separate claims into a single, compact claim using a random challenge γ. The soundness of this step is guaranteed by the fact that it is extremely unlikely for multiple incorrect claims to "cancel each other out" and coincidentally satisfy a random combination.

The Claims to be Batched Let's first clearly restate the set of claims the verifier has accumulated about the prover's candidate polynomial, P_1.

1. **The Sumcheck Claim:** This is the folded-down algebraic constraint from the sumcheck phase.

$$\langle P_1, R_0(\lambda_1, \ldots, \lambda_k, \cdot)\rangle_{H_{n-k}} = q_{k-1}(\lambda_k)$$

2. **The Evaluation Claims:** These are the $s_0 + 1$ claims from the query phase.

$$\langle P_1, Q_{n-k}(z_j)\rangle_{H_{n-k}} = v_j \quad \text{for } j = 0, \ldots, s_0$$

The Aggregation Mechanism To combine these claims, the verifier uses a random challenge $\gamma \in \mathbb{F}$ to create a random linear combination. A single new kernel, R_1, is computed:

$$\underbrace{R_1}_{\text{New Batched Kernel}} = \underbrace{R_0(\lambda_1, \ldots, \lambda_k, \cdot)}_{\text{Sumcheck Kernel}} + \sum_{j=0}^{s_0} \underbrace{\gamma^{j+1}}_{\text{Random Weight}} \cdot \underbrace{Q_{n-k}(z_j)}_{\text{Evaluation Kernels}}$$

And a corresponding new target value, c_1, is also computed:

$$c_1 = q_{k-1}(\lambda_k) + \sum_{j=0}^{s_0} \gamma^{j+1} \cdot v_j$$

The original list of claims is now replaced by a single, equivalent claim for the next round: $\langle P_1, R_1\rangle_{H_{n-k}} = c_1$.

Why Batching is Secure: The Soundness Argument The security of this step relies on the Schwartz-Zippel Lemma. Suppose a malicious prover is trying to cheat, meaning his candidate polynomial P_1 violates at least one of the original claims. Let's define an "error polynomial" $E(\gamma)$ that is zero if and only if the batched claim holds:

$$E(\gamma) = (\langle P_1, R_0(\ldots)\rangle - q_{k-1}(\lambda_k)) + \sum_{j=0}^{s_0} \gamma^{j+1} \cdot (\langle P_1, Q_{n-k}(z_j)\rangle - v_j)$$

Since the prover is cheating, at least one of the terms in the parentheses is non-zero, which means $E(\gamma)$ is a *non-zero polynomial* in the variable γ.

The degree of this polynomial is at most $s_0 + 1$. By the Schwartz-Zippel Lemma, it can have at most $s_0 + 1$ roots. The probability that a random γ happens to be one of these roots is negligibly small. This is the core of the soundness argument.

Why Batching is Secure: The Soundness Argument

The security of this step comes from a core principle of algebra: a non-zero polynomial can only have a few roots. Let's see how this applies.

The Setup A malicious prover has a list of at most $\ell_1(\theta_1)$ candidate polynomials for f_1. For his cheat to succeed, at least one of these candidates must satisfy the verifier's single, batched claim for the randomly chosen challenge γ.

The Logic For any *single* candidate polynomial, the batched claim is an equation that can be viewed as a polynomial in the variable γ. Because the aggregation combines $s_0 + 1$ evaluation claims using powers of γ up to γ^{s_0+1}, this polynomial has a degree of $d = s_0 + 1$.
If the prover is cheating, at least one of the original claims for this candidate was false. This means the polynomial in γ is not identically zero. A non-zero polynomial of degree d can have at most d roots. Therefore, for each candidate polynomial, there are at most $d = s_0 + 1$ ”lucky” values of γ that would make the verifier accept it.

Bounding the Total Probability With a list of at most $\ell_1(\theta_1)$ candidate polynomials, the total number of ”lucky” γ values across all candidates is at most $\ell_1(\theta_1) \cdot (s_0+1)$. The probability that the verifier randomly picks one of these few lucky values from the entire field $\mathbb{F}$ is therefore very small. This gives us the soundness error for the aggregation step:

$$\varepsilon_{\text{agg}} \leq \frac{\ell_1(\theta_1) \cdot (s_0 + 1)}{|\mathbb{F}|}$$

The paper's stated bound of $\frac{\ell_1(\theta_1) \cdot s_0}{|\mathbb{F}|}$ reflects the same core idea: the error is proportional to the list size multiplied by the number of claims being batched. I need to check more precisely why such a difference is happening.

6.3.4 Global Soundness of a WHIR Round

By combining these three layers of defense, WHIR ensures that any attempt by a malicious prover to cheat is detected with very high probability. The total soundness error for a single round is the sum of the errors from each phase. A prover starting with an invalid function must first pass the sumcheck phase (with probability $\approx \varepsilon_{\text{sumcheck}}$), then the query phase by collapsing the list and passing sampling (with probability $\approx \varepsilon_{\text{query}}$), and finally the aggregation phase (with probability $\approx \varepsilon_{\text{agg}}$). The multi-stage verification, especially the powerful list-collapsing technique of the out-of-domain query, provides a robust defense that allows WHIR to achieve high security with far fewer queries than previous methods, which is the key to its exceptional verifier performance.

References

[1] E. Ben-Sasson, D. Carmon, Y. Ishai, S. Kopparty, and S. Saraf, “Proximity gaps for reed–solomon codes,” *Journal of the ACM*, vol. 70, no. 5, pp. 1–57, 2023.

[2] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, “Fast reed-solomon interactive oracle proofs of proximity,” in *45th international colloquium on automata, languages, and programming (icalp 2018)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 14–1.

[3] U. Haböck, “Basefold in the list decoding regime,” *Cryptology ePrint Archive*, 2024.