Tri-Quarter Framework Case Study: BPSK Signal Processing

Nathan O. Schmidt

Cold Hammer Research & Development LLC, Eagle, Idaho, USA

nate.o.schmidt@coldhammer.net

July 4, 2025

Abstract

In this case study, we engage the novel **Tri-Quarter framework** by applying it to Binary Phase-Shift Keying (BPSK) signal processing, where we leverage **structured orientation phase pair assignments** and **dynamic weight adjustments** to enhance noise filtering and error correction under Gaussian and non-Gaussian noise. We attack the challenge of reliable decoding in communication systems like wireless networks, satellite links, and IoT devices, where noise varies from Additive White Gaussian Noise (AWGN) to impulsive noise (IN) interference. The framework implements a model-free methodology by using sign-based phase assignments and distance-based weights to decode signals without prior noise knowledge. Simulations at a signal-to-noise ratio (SNR) of 6 dB with 100,000 trials demonstrate that the Tri-Quarter framework's noise filtering achieves a 2.350% bit error rate (BER) in AWGN, closely matching standard thresholding with 1 CPU cycle, while its error correction with 3 transmissions per symbol yields a 0.138% BER in AWGN and 0.430% BER in IN, performing comparably to majority voting (0.149% BER in AWGN, 1.415% BER in IN) and significantly outperforming Gaussian-tuned soft-decision decoding (0.030% BER in AWGN, 12.769% BER in IN) in non-Gaussian conditions. With 17 CPU cycles for error correction, the Tri-Quarter framework balances efficiency and robustness, dominating in unpredictable noise environments (e.g. urban cellular wireless networks, industrial IoT networks, oceanographic sensor networks, and naval communication networks), though it is less optimal for ultra-low-power devices or Gaussian-dominated environments. This framework offers a versatile solution for modern communication challenges, with potential extensions to complex modulations like Quadrature Phase-Shift Keying (QPSK).

1 Introduction

Binary Phase-Shift Keying (BPSK) is fundamental to digital communication, powering applications like wireless networks [1, 2], satellite links [3], GPS [4, 5], bluetooth [6], and the Internet of Things (IoT) devices [7]. The fact that BPSK encodes bits via two phase states (0 or π radians) is simple, elegant, and straightforward. However, there is a mighty challenge that BPSK implementations grapple with in practical real-world applications: decoding signals under wildly diverse noise conditions such as Additive White Gaussian Noise (AWGN) from thermal sources and impulsive noise (IN) from electromagnetic interference in urban [8], industrial [9], and oceanographic environments [10, 11]—to name a few. Achieving reliable decoding demands efficient noise filtering and robust error correction, balancing computational complexity with bit error rate (BER) performance. Standard methods, such as thresholding, majority voting, and Gaussian-tuned soft-decision decoding [12], often rely on noise model assumptions, limiting their adaptability in dynamic, non-Gaussian environments.

This case study unleashes the **Tri-Quarter framework** [13], a novel approach that redefines BPSK signal processing through **structured orientation phase pair assignments** and **dynamic weight adjustments**. By mapping received signals $\vec{y} = \vec{x} + \vec{n}$ to phase pairs $\phi(\vec{y}) = (\langle \vec{y}_{\mathbb{R}} \rangle, \langle \vec{y}_{\mathbb{I}} \rangle)_\phi$ and using distance-based weights, Tri-Quarter achieves model-free decoding, eliminating the need for prior noise knowledge. Its adaptability to unpredictable noise environments

positions the Tri-Quarter framework as a versatile solution for advancing signal processing, seamlessly integrating efficient algorithms, rigorous phase-based formulations, and practical applications across diverse communication challenges.

Our methodology, detailed in Sections 2–6, includes:

- **Setup**: Defines the BPSK system with $\vec{x} \in \{+1, -1\}$ and complex noise $\vec{n}$ for $\vec{y} = \vec{y}_{\mathbb{R}} + \vec{y}_{\mathbb{I}} = \vec{x} + \vec{n}$ (Section 2).
- **Structured Orientation Phase Pair Assignments**: Introduces sign-based phase assignments for decoding (Section 3).
- **Noise Filtering**: Describes efficient filtering using phase $\langle \vec{y}_{\mathbb{R}} \rangle$ (Section 4).
- **Error Correction**: Details dynamic distance-based weighting for robust decoding across multiple transmissions (Section 5).
- **Comparison**: Evaluates performance via simulations at a signal-to-noise ratio (SNR) of 6 dB with 100,000 trials (Section 6).

Simulations reveal that Tri-Quarter's noise filtering achieves a 2.350% BER in AWGN, closely matching standard thresholding (1 CPU cycle), while its error correction achieves a 0.138% BER in AWGN and 0.430% BER in IN, performing comparably to majority voting (0.149% BER in AWGN and 1.415% BER in IN) and significantly outperforming soft-decision decoding (0.030% BER in AWGN, 12.769% BER in IN) in non-Gaussian environments. At 17 CPU cycles for error correction, Tri-Quarter offers a robust and adaptable solution, though it is less suited for ultra-low-power devices when compared to majority voting's 3 CPU cycles.

The Tri-Quarter framework dominates in scenarios with unpredictable noise—such as urban wireless networks [8], industrial IoT networks [9], oceanographic sensor networks [10], and naval communication networks [11]—while soft-decision decoding excels in Gaussian-dominated settings like satellite communications. Section 7 synthesizes and recapitulates these findings, thus highlighting Tri-Quarter's potential to transform signal processing and its promise for future applications, such as complex modulations like QPSK. This work aims to pave the way for resilient, efficient communication systems in an increasingly connected world which depends on modern communication systems vexed with prevalent noise and chaotic interference.

2 Setup

In BPSK systems, the transmitted symbol is a complex number in the complex plane $\mathbb{C}$, as defined in the Tri-Quarter framework [13]. We denote this symbol as $\vec{x} = x_{\mathbb{R}} + x_{\mathbb{I}} i \in \mathbb{C}$, represented as a vector $\vec{x} = \vec{x}_{\mathbb{R}} + \vec{x}_{\mathbb{I}}$, where $\vec{x}_{\mathbb{R}} = (x_{\mathbb{R}}, 0)_C \in \mathbb{R} \times \{0\}$ is the real-axis vector and $\vec{x}_{\mathbb{I}} = (0, x_{\mathbb{I}})_C \in \{0\} \times \mathbb{I}$ is the imaginary-axis vector, with $\mathbb{I} = i\mathbb{R}$ and $x_{\mathbb{R}}, x_{\mathbb{I}} \in \mathbb{R}$. In BPSK, the symbol is real-valued, so $x_{\mathbb{R}} \in \{+1, -1\}$ and $x_{\mathbb{I}} = 0$, yielding $\vec{x} = (x_{\mathbb{R}}, 0)_C = (x, 0)_C$ and $\vec{x}_{\mathbb{I}} = (0, 0)_C$. Here, $x_{\mathbb{R}} = +1$ corresponds to a phase shift of 0 radians (encoding bit 0), and $x_{\mathbb{R}} = -1$ corresponds to a phase shift of π radians (encoding bit 1), aligning with standard electrical engineering notation while adhering to the Tri-Quarter convention of vector notation with arrows (e.g., $\vec{x}$) in $\mathbb{C}$.

The received signal is modeled as $\vec{y} = \vec{x} + \vec{n}$, where $\vec{n}$ is the noise vector. We express the noise as a complex number $\vec{n} = n_{\mathbb{R}} + n_{\mathbb{I}} i \in \mathbb{C}$, or equivalently as a vector $\vec{n} = \vec{n}_{\mathbb{R}} + \vec{n}_{\mathbb{I}}$, with $\vec{n}_{\mathbb{R}} = (n_{\mathbb{R}}, 0)_C \in \mathbb{R} \times \{0\}$ and $\vec{n}_{\mathbb{I}} = (0, n_{\mathbb{I}})_C \in \{0\} \times \mathbb{I}$, where $n_{\mathbb{R}}, n_{\mathbb{I}} \in \mathbb{R}$ are the real and imaginary noise scalar components, respectively. Thus, the received signal is $\vec{y} = y_{\mathbb{R}} + y_{\mathbb{I}} i = \vec{y}_{\mathbb{R}} + \vec{y}_{\mathbb{I}}$, where $\vec{y}_{\mathbb{R}} = (y_{\mathbb{R}}, 0)_C \in \mathbb{R} \times \{0\}$, $\vec{y}_{\mathbb{I}} = (0, y_{\mathbb{I}})_C \in \{0\} \times \mathbb{I}$, and:

- $y_{\mathbb{R}} = \mathrm{Re}(\vec{y}) = x_{\mathbb{R}} + n_{\mathbb{R}} = x + n_{\mathbb{R}}$, carrying the BPSK signal information,
- $y_{\mathbb{I}} = \mathrm{Im}(\vec{y}) = x_{\mathbb{I}} + n_{\mathbb{I}} = n_{\mathbb{I}}$, representing the imaginary noise component (since $x_{\mathbb{I}} = 0$).

In standard BPSK with real-valued noise, such as AWGN with variance σ^2, the noise is $\vec{n} = n_{\mathbb{R}} + 0i$, so $\vec{n}_{\mathbb{R}} = (n_{\mathbb{R}}, 0)_C$, $\vec{n}_{\mathbb{I}} = (0,0)_C$, and $\vec{y} = (x + n_{\mathbb{R}}, 0)_C$. However, real-world communication systems, such as those in urban cellular wireless networks, satellite links, industrial IoT networks, oceanographic sensor networks, or naval communication networks, may encounter complex noise (e.g., from quadrature interference), where $\vec{n}_{\mathbb{I}} \neq (0,0)_C$. The Tri-Quarter framework supports this generality because it accommodates both real and complex noise scenarios. Ultimately, the objective is to accurately decode $\vec{x}$ from $\vec{y}$, which is critical for establishing reliable communication across various applications under varying channel conditions.

3 Structured Orientation Phase Pair Assignments

In the Tri-Quarter framework [13], each non-zero received signal $\vec{y} \in \mathbb{C} \setminus \{0\}$ is assigned a phase pair via the map $\phi : \mathbb{C} \setminus \{0\} \to \{0, \frac{\pi}{2}, \pi\} \times \{0, \frac{\pi}{2}, \frac{3\pi}{2}\}$, defined as $\phi(\vec{y}) = (\langle \vec{y}_{\mathbb{R}} \rangle, \langle \vec{y}_{\mathbb{I}} \rangle)_\phi$. Here, $\vec{y} = y_{\mathbb{R}} + y_{\mathbb{I}} i = \vec{y}_{\mathbb{R}} + \vec{y}_{\mathbb{I}}$, with $\vec{y}_{\mathbb{R}} = (y_{\mathbb{R}}, 0)_C \in \mathbb{R} \times \{0\}$ and $\vec{y}_{\mathbb{I}} = (0, y_{\mathbb{I}})_C \in \{0\} \times \mathbb{I}$, where $y_{\mathbb{R}} = \mathrm{Re}(\vec{y})$ and $y_{\mathbb{I}} = \mathrm{Im}(\vec{y})$ are the scalar real and imaginary components, respectively. The origin is excluded because the phase of the zero vector is undefined, though this does not impact BPSK decoding as noise ensures $\vec{y} \neq 0$ with high probability in practical systems. The phase assignments are defined piecewise as follows [13]:

$$\langle \vec{y}_{\mathbb{R}} \rangle = \begin{cases} 0 & \text{if } y_{\mathbb{R}} > 0, \\ \frac{\pi}{2} & \text{if } y_{\mathbb{R}} = 0, \\ \pi & \text{if } y_{\mathbb{R}} < 0, \end{cases} \tag{1}$$

$$\langle \vec{y}_{\mathbb{I}} \rangle = \begin{cases} \frac{\pi}{2} & \text{if } y_{\mathbb{I}} > 0, \\ 0 & \text{if } y_{\mathbb{I}} = 0, \\ \frac{3\pi}{2} & \text{if } y_{\mathbb{I}} < 0. \end{cases} \tag{2}$$

For BPSK, since the transmitted signal $\vec{x} = x_{\mathbb{R}} + 0i = (x_{\mathbb{R}}, 0)_C$, with $x_{\mathbb{R}} \in \{+1, -1\}$ and $\vec{x}_{\mathbb{I}} = (0,0)_C$, decoding relies solely on the real component's phase $\langle \vec{y}_{\mathbb{R}} \rangle$. This phase corresponds to the standard sign function in electrical engineering, defined piecewise as:

$$\mathrm{sign}(y_{\mathbb{R}}) = \begin{cases} +1 & \text{if } y_{\mathbb{R}} > 0, \\ 0 & \text{if } y_{\mathbb{R}} = 0, \\ -1 & \text{if } y_{\mathbb{R}} < 0. \end{cases} \tag{3}$$

The Tri-Quarter phase $\langle \vec{y}_{\mathbb{R}} \rangle$ encapsulates this sign-based decision in a phase context, where:

- $\langle \vec{y}_{\mathbb{R}} \rangle = 0 \iff \mathrm{sign}(y_{\mathbb{R}}) = +1$,
- $\langle \vec{y}_{\mathbb{R}} \rangle = \frac{\pi}{2} \iff \mathrm{sign}(y_{\mathbb{R}}) = 0$,
- $\langle \vec{y}_{\mathbb{R}} \rangle = \pi \iff \mathrm{sign}(y_{\mathbb{R}}) = -1$.

Based on $\langle \vec{y}_{\mathbb{R}} \rangle$, the Tri-Quarter framework suggests the transmitted symbol $\vec{x}$ as:

$$\vec{x} = \begin{cases} +1 & \text{if } \langle \vec{y}_{\mathbb{R}} \rangle = 0, \\ +1 & \text{if } \langle \vec{y}_{\mathbb{R}} \rangle = \frac{\pi}{2}, \\ -1 & \text{if } \langle \vec{y}_{\mathbb{R}} \rangle = \pi, \end{cases} \tag{4}$$

where the edge case $\langle \vec{y}_{\mathbb{R}} \rangle = \frac{\pi}{2}$ (i.e., $y_{\mathbb{R}} = 0$) is rare due to noise randomness in practical systems and defaults to $\vec{x} = +1$ for simplicity, consistent with standard BPSK thresholding. This structured phase-based representation facilitates efficient decoding and noise handling in subsequent steps. While $\langle \vec{y}_{\mathbb{I}} \rangle$ is defined for completeness and potential extension to complex signal modulations (e.g., QPSK), it is not used in BPSK decoding here, as the imaginary component $\vec{y}_{\mathbb{I}} = (0, y_{\mathbb{I}})_C$ typically represents noise.

4 Noise Filtering

In real-world communication systems, noise is an unavoidable factor that degrades signal quality, potentially leading to misinterpreted data at the receiver. Effective noise filtering is critical to recover the original transmitted symbol, ensuring reliable performance in applications like wireless networks [1, 2], satellite communications [3], GPS [4, 5], bluetooth [6], and IoT devices [7], where noise from interference or environmental factors is prevalent.

In the Tri-Quarter framework, noise filtering recovers the transmitted symbol $\vec{x}$ from a single received signal $\vec{y}$ using the structured phase assignment $\langle \vec{y}_{\mathbb{R}} \rangle$. The process is as follows:

(1) **Assign phase**: Compute $\langle \vec{y}_{\mathbb{R}} \rangle$ based on the real component $y_{\mathbb{R}} = \text{Re}(\vec{y})$, as defined in Equation (1).

(2) **Apply decision rule**: Determine the filtered symbol $\vec{y}_{\text{filtered}}$ using the phase $\langle \vec{y}_{\mathbb{R}} \rangle$ via the mapping:

$$\vec{y}_{\text{filtered}} = \begin{cases} +1 & \text{if } \langle \vec{y}_{\mathbb{R}} \rangle = 0, \\ +1 & \text{if } \langle \vec{y}_{\mathbb{R}} \rangle = \frac{\pi}{2}, \\ -1 & \text{if } \langle \vec{y}_{\mathbb{R}} \rangle = \pi. \end{cases} \tag{5}$$

This rule aligns with standard BPSK decoding, where $\vec{y}_{\text{filtered}} = \text{sign}(y_{\mathbb{R}})$ for $y_{\mathbb{R}} \neq 0$ (see Equation (3)), and defaults to $+1$ when $y_{\mathbb{R}} = 0$.

This method requires only one comparison (1 CPU cycle), matching the computational efficiency of standard BPSK thresholding at $y_{\mathbb{R}} = 0$. By leveraging the Tri-Quarter phase assignment, it ensures simplicity and consistency with the framework's structured orientation.

4.1 Example 1: Real Noise

Consider a transmitted symbol $\vec{x} = +1$ and a received signal $\vec{y} = 1.2 + 0i$ (i.e., noise $n = 0.2 + 0i$):

Tri-Quarter Approach:

(1) Extract real component: $y_{\mathbb{R}} = 1.2$.

(2) Assign phase: $y_{\mathbb{R}} = 1.2 > 0 \implies \langle \vec{y}_{\mathbb{R}} \rangle = 0$.

(3) Apply decision rule: $\langle \vec{y}_{\mathbb{R}} \rangle = 0 \implies \vec{y}_{\text{filtered}} = +1$.

(4) Calculate error: $|\vec{x} - \vec{y}_{\text{filtered}}| = |+1 - (+1)| = 0$.

Standard Approach (Thresholding):

(1) Check threshold: $y_{\mathbb{R}} = 1.2 > 0 \implies \vec{y}_{\text{filtered}} = +1$.

(2) Calculate error: $|\vec{x} - \vec{y}_{\text{filtered}}| = |+1 - (+1)| = 0$.

Both methods correctly recover the symbol, demonstrating their effectiveness with simple noise.

4.2 Example 2: Complex Noise

Now consider $\vec{x} = -1$ and $\vec{y} = -0.8 + 0.5i$ (i.e., noise $n = 0.2 + 0.5i$):

Tri-Quarter Approach:

(1) Extract real component: $y_{\mathbb{R}} = -0.8$.

(2) Assign phase: $y_{\mathbb{R}} = -0.8 < 0 \implies \langle \vec{y}_{\mathbb{R}} \rangle = \pi$.

(3) Apply decision rule: $\langle \vec{y}_{\mathbb{R}} \rangle = \pi \implies \vec{y}_{\text{filtered}} = -1$.

(4) Calculate error: $|\vec{x} - \vec{y}_{\text{filtered}}| = |-1-(-1)| = 0$.

Standard Approach (Thresholding):

(1) Check threshold: $y_{\mathbb{R}} = -0.8 < 0 \implies \vec{y}_{\text{filtered}} = -1$.

(2) Calculate error: $|\vec{x} - \vec{y}_{\text{filtered}}| = |-1-(-1)| = 0$.

Both approaches succeed, with the Tri-Quarter method effectively handling complex noise by focusing on the real component, maintaining consistency with its framework.

4.3 Transition to Error Correction

Noise filtering serves as the first line of defense in signal recovery, tackling the immediate effects of noise on individual received signals. However, in environments with persistent or complex noise, additional robustness is often needed. This is where error correction steps in, building on filtered signals by leveraging redundancy across multiple transmissions to further enhance reliability.

5 Error Correction

Following noise filtering, error correction provides a critical layer of protection, especially in challenging noise conditions like those with non-Gaussian or IN characteristics. By processing multiple received signals, it ensures accurate symbol recovery even when noise filtering alone is insufficient. The Tri-Quarter framework enhances error correction by transmitting the symbol $\vec{x}$ k times and dynamically weighting the received signals $\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_k$. We compare this approach to standard methods (majority voting and Gaussian-tuned soft-decision decoding) in terms of efficiency (computational complexity) and robustness (ability to handle noise, particularly outliers).

The Tri-Quarter error correction process is as follows:

(1) **Collect signals**: Receive $\vec{y}_i = \vec{x} + n_i$ for $i = 1, 2, \ldots, k$.

(2) **Assign phases**: Compute $\langle \vec{y}_{i,\mathbb{R}} \rangle$ for each $y_{i,\mathbb{R}} = \text{Re}(\vec{y}_i)$, using Equation (1).

(3) **Compute distance**: Calculate $d_i = |y_{i,\mathbb{R}} - \text{sign}(y_{i,\mathbb{R}}) \cdot 1|$, where $\text{sign}(y_{i,\mathbb{R}})$ is defined in Equation (3). This measures the deviation of $y_{i,\mathbb{R}}$ from the expected symbol ($+1$ or -1) based on its sign.

(4) **Assign weights**: Set $w_i = \frac{1}{1+d_i^2}$, reducing the influence of signals with large d_i (e.g., outliers in IN).

(5) **Weighted voting**: Sum the weights for signals suggesting $\vec{x} = +1$ (i.e., $\langle \vec{y}_{i,\mathbb{R}} \rangle = 0$) and for those suggesting $\vec{x} = -1$ (i.e., $\langle \vec{y}_{i,\mathbb{R}} \rangle = \pi$). Choose the symbol with the higher total:

$$\text{total}_+ = \sum_{\{i:\langle \vec{y}_{i,\mathbb{R}} \rangle = 0\}} w_i, \tag{6}$$

$$\text{total}_- = \sum_{\{i:\langle \vec{y}_{i,\mathbb{R}} \rangle = \pi\}} w_i, \tag{7}$$

$$\vec{x}_{\text{decoded}} = \begin{cases} +1 & \text{if total}_+ \geq \text{total}_-, \\ -1 & \text{otherwise.} \end{cases} \tag{8}$$

Efficiency Comparison: For $k = 3$, the Tri-Quarter approach requires 3 comparisons, 3 absolute values, 3 squarings, 3 divisions, and 2 summations, totaling approximately 17 CPU cycles.

Majority voting is more efficient, requiring only 3 comparisons (3 CPU cycles), making it suitable for ultra-low-power devices. Gaussian-tuned soft-decision decoding also requires 17 cycles (3 comparisons, 3 squarings, 3 divisions, 2 summations), matching Tri-Quarter's complexity but relying on prior knowledge of the noise variance.

Robustness Comparison: The Tri-Quarter approach excels in environments with IN, as its dynamic distance-based weights ($w_i = \frac{1}{1+d_i^2}$) reduce the influence of outliers, unlike majority voting, which treats all signals equally. Compared to soft-decision decoding, which amplifies outliers in non-Gaussian noise due to its variance-based weighting, Tri-Quarter's model-free approach adapts better to unpredictable noise conditions, such as those in urban cellular wireless networks [8], industrial IoT networks [9], oceanographic sensor networks [10], and naval communication networks [11]. In Gaussian noise, soft-decision may perform better due to its optimization for known noise distributions, but Tri-Quarter remains competitive without requiring such assumptions.

5.1 Step-by-Step Example: Impulsive Noise

Consider $\vec{x} = +1$ transmitted $k = 3$ times, with received signals:

- $\vec{y}_1 = 1.1 + 0i$, so $y_{1,\mathbb{R}} = 1.1$,
- $\vec{y}_2 = 0.3 + 0i$, so $y_{2,\mathbb{R}} = 0.3$,
- $\vec{y}_3 = -5 + 0i$, so $y_{3,\mathbb{R}} = -5$ (IN outlier).

We apply the error correction steps and compare with standard approaches:

Tri-Quarter Approach:

(1) **Assign phases:**

- $y_{1,\mathbb{R}} = 1.1 > 0 \implies \langle \vec{y}_{1,\mathbb{R}} \rangle = 0$,
- $y_{2,\mathbb{R}} = 0.3 > 0 \implies \langle \vec{y}_{2,\mathbb{R}} \rangle = 0$,
- $y_{3,\mathbb{R}} = -5 < 0 \implies \langle \vec{y}_{3,\mathbb{R}} \rangle = \pi$.

(2) **Compute distance metrics:**

- $\text{sign}(y_{1,\mathbb{R}}) = +1$, so $d_1 = |1.1 - (+1) \cdot 1| = |1.1 - 1| = 0.1$,
- $\text{sign}(y_{2,\mathbb{R}}) = +1$, so $d_2 = |0.3 - (+1) \cdot 1| = |0.3 - 1| = 0.7$,
- $\text{sign}(y_{3,\mathbb{R}}) = -1$, so $d_3 = |-5 - (-1) \cdot 1| = |-5 + 1| = 4$.

(3) **Calculate weights:**

- $w_1 = \frac{1}{1+(0.1)^2} = \frac{1}{1+0.01} \approx 0.990$,
- $w_2 = \frac{1}{1+(0.7)^2} = \frac{1}{1+0.49} \approx 0.671$,
- $w_3 = \frac{1}{1+(4)^2} = \frac{1}{1+16} \approx 0.059$.

(4) **Vote with weights:**

- For $\vec{x} = +1$ ($\langle \vec{y}_{i,\mathbb{R}} \rangle = 0$): $\text{total}_+ = w_1 + w_2 = 0.990 + 0.671 = 1.661$,
- For $\vec{x} = -1$ ($\langle \vec{y}_{i,\mathbb{R}} \rangle = \pi$): $\text{total}_- = w_3 = 0.059$,
- Since $1.661 > 0.059$, set $\vec{x}_{\text{decoded}} = +1$.

(5) **Calculate error**: $|\vec{x} - \vec{x}_{\text{decoded}}| = |+1-(+1)| = 0$.

(6) **Efficiency**: Requires 17 CPU cycles, higher than majority voting but comparable to soft-decision.

(7) **Robustness**: The low weight of the outlier ($w_3 \approx 0.059$) ensures robust decoding by minimizing its impact, unlike equal-weighted or variance-based methods.

Standard Approach (Majority Voting):

(1) **Filter each signal:**

- $y_{1,\mathbb{R}} = 1.1 > 0 \implies +1$,
- $y_{2,\mathbb{R}} = 0.3 > 0 \implies +1$,
- $y_{3,\mathbb{R}} = -5 < 0 \implies -1$.

(2) **Vote**: 2 votes for $+1$, 1 for $-1 \implies \vec{x}_{\text{decoded}} = +1$.

(3) **Calculate error**: $|\vec{x} - \vec{x}_{\text{decoded}}| = |+1-(+1)| = 0$.

(4) **Efficiency**: Requires 3 CPU cycles, significantly lower than Tri-Quarter, ideal for ultra-low-power applications.

(5) **Robustness**: Treats all signals equally, which may reduce accuracy in the presence of outliers compared to Tri-Quarter's weighted approach.

Standard Approach (Soft-Decision, Gaussian-Tuned):

(1) **Calculate weights**: Assume $w_i = y_{i,\mathbb{R}}^2/\sigma^2$, with σ^2 as the noise variance:

- $w_1 = 1.1^2/\sigma^2 = 1.21/\sigma^2$,
- $w_2 = 0.3^2/\sigma^2 = 0.09/\sigma^2$,
- $w_3 = (-5)^2/\sigma^2 = 25/\sigma^2$.

(2) **Vote**:

- For $\vec{x} = +1$: $(1.21 + 0.09)/\sigma^2 = 1.30/\sigma^2$,
- For $\vec{x} = -1$: $25/\sigma^2$,
- Since $1.30/\sigma^2 < 25/\sigma^2$, set $\vec{x}_{\text{decoded}} = -1$.

(3) **Calculate error**: $|\vec{x} - \vec{x}_{\text{decoded}}| = |+1-(-1)| = 2$, indicating an error.

(4) **Efficiency**: Requires 17 CPU cycles, matching Tri-Quarter's complexity.

(5) **Robustness**: Amplifies the outlier's influence due to its large magnitude, potentially reducing accuracy in non-Gaussian noise compared to Tri-Quarter.

The Tri-Quarter approach balances efficiency and robustness, offering a model-free method that excels in handling outliers in IN, while majority voting prioritizes efficiency and soft-decision relies on noise model assumptions.

5.2 Conclusion and Transition Forward

Together, noise filtering and error correction form a robust pipeline for signal recovery in BPSK systems. Noise filtering handles immediate noise effects with high efficiency, while error correction leverages redundancy to enhance robustness, particularly in non-Gaussian noise. The Tri-Quarter framework's dynamic weighting provides adaptability without requiring prior noise knowledge, making it suitable for real-world challenges like urban wireless networks, industrial IoT networks, oceanographic sensor networks, and naval communication networks. The next section will quantify these advantages through simulations, comparing the Tri-Quarter approach to standard methods in various noise conditions.

6 Simulations

To empirically validate the theoretical BPSK discussions in Sections 4 and 5, we compare the Tri-Quarter approach to standard approaches (thresholding, majority voting, and Gaussian-tuned soft-decision decoding) using simulations. We evaluate both computational complexity and BER performance under AWGN and IN conditions. The simulations are conducted with the following parameters: SNR = 6 dB, $k = 3$ transmissions per symbol, and 100,000 trials. Reported BERs are rounded to three significant digits after the decimal point for clarity, with precise values provided in the simulation scripts in Appendices A.1–A.3 that are freely available online [14] for testing, experimentation, etc.

6.1 Simulation Setup

The simulations were designed to quantify the BER for the Tri-Quarter approach, majority voting, and Gaussian-tuned soft-decision decoding under two noise models:

- **Parameters**:
  - **Trials**: 100,000 BPSK symbols, with $\vec{x} = +1$ or -1 (equal probability).
  - **Transmissions**: $k = 3$ per symbol.
  - **SNR**: 6 dB, corresponding to noise variance $\sigma^2 = 0.25$.
  - **Noise Models**:
    - **AWGN**: Gaussian noise with mean 0 and variance $\sigma^2 = 0.25$.
    - **IN**: 90% Gaussian noise ($\sigma^2 = 0.25$), 10% outliers at ± 5.
- **Methodology**:
  (1) Randomly generate the true symbol $\vec{x} \in \{+1, -1\}$.
  (2) Add noise to produce received signals $\vec{y}_i = \vec{x} + n_i$ for $i = 1, 2, 3$, where n_i is drawn from the specified noise model.
  (3) Decode $\vec{x}$ using each method:
    - **Tri-Quarter**: Apply phase assignments, compute distances $d_i = |y_{i,\mathbb{R}} - \text{sign}(y_{i,\mathbb{R}}) \cdot 1|$, assign weights $w_i = \frac{1}{1+d_i^2}$, and use weighted voting as per Equation (8).
    - **Majority Voting**: Determine the majority of $\text{sign}(y_{i,\mathbb{R}})$ for $i = 1, 2, 3$.
    - **Soft-Decision (Gaussian-Tuned)**: Use weights $w_i = y_{i,\mathbb{R}}^2/\sigma^2$, and select the symbol with the highest weighted sum.
  (4) Compute the BER as the fraction of incorrect decisions over all trials.
- **Simulation Code**: Detailed in Appendices A.1, A.2, and A.3.

6.2 Noise Filtering Comparison

For noise filtering, we compare the Tri-Quarter approach to standard BPSK thresholding. Both methods are evaluated for a single received signal $\vec{y}$, focusing on computational complexity and BER performance.

- **Tri-Quarter Approach**:
  - **Process**: Extract $y_{\mathbb{R}}$, assign phase $\langle \vec{y}_{\mathbb{R}} \rangle$, and apply the decision rule from Equation (5).
  - **Operations**: 1 comparison (1 CPU cycle).
- **Standard Approach (Thresholding)**:
  - **Process**: Directly check if $y_{\mathbb{R}} > 0$ to decide $\vec{y}_{\text{filtered}} = +1$ or -1.
  - **Operations**: 1 comparison (1 CPU cycle).
- **Metrics**:
  - **Complexity**: Both methods require 1 CPU cycle, demonstrating high efficiency.
  - **BER**: Under AWGN ($\sigma^2 = 0.25$, SNR = 6 dB), both achieve a 2.350% BER. For IN (10% outliers at ± 5), both achieve a 7.087% BER, as neither method dynamically adapts to noise variations in a single transmission.

Note: The Tri-Quarter noise filtering and standard thresholding employ the same decision rule (decide $\vec{y}_{\text{filtered}} = +1$ if $y_{\mathbb{R}} \geq 0$, else -1), resulting in identical BER performance, as explicitly demonstrated in Script 1 (Appendix A.1).

This comparison highlights that the Tri-Quarter noise filtering approach matches the standard thresholding approach in both efficiency and performance for single-transmission scenarios.

6.3 Error Correction Comparison

For error correction, we compare the Tri-Quarter approach to majority voting and Gaussian-tuned soft-decision decoding, focusing on computational complexity and BER performance with $k = 3$ transmissions.

- **Tri-Quarter Approach**:
  - **Process**: Assign phases, compute distances d_i, assign weights $w_i = \frac{1}{1+d_i^2}$, and use weighted voting as per Equations (6)–(8).
  - **Operations**: 3 comparisons, 3 absolute values, 3 squarings, 3 divisions, and 2 summations (approximately 17 CPU cycles).
  - **BER**:
    - AWGN: 0.138%
    - IN: 0.430%
- **Standard Approach (Majority Voting)**:
  - **Process**: Determine the majority of $\text{sign}(y_{i,\mathbb{R}})$ for $i = 1, 2, 3$.
  - **Operations**: 3 comparisons (3 CPU cycles).

  - **BER**:
    - AWGN: 0.149%
    - IN: 1.415%

- **Standard Approach (Soft-Decision, Gaussian-Tuned)**:
  - **Process**: Compute weights $w_i = y_{i,\mathbb{R}}^2/\sigma^2$, sum weights for each symbol, and select the symbol with the highest total.
  - **Operations**: 3 comparisons, 3 squarings, 3 divisions, and 2 summations (17 CPU cycles).
  - **BER**:
    - AWGN: 0.030%
    - IN: 12.769%

Summary of Metrics:

- **Computational Complexity**:
  - Tri-Quarter: 17 CPU cycles
  - Majority Voting: 3 CPU cycles
  - Soft-Decision: 17 CPU cycles
- **BER (AWGN, SNR = 6 dB, $k = 3$)**:
  - Tri-Quarter: 0.138%
  - Majority Voting: 0.149%
  - Soft-Decision: 0.030%
- **BER (IN, 10% outliers at ± 5, SNR = 6 dB)**:
  - Tri-Quarter: 0.430%
  - Majority Voting: 1.415%
  - Soft-Decision: 12.769%

These results demonstrate that the Tri-Quarter approach achieves a balanced trade-off between efficiency and robustness, particularly excelling in IN conditions where standard methods struggle.

7 Conclusion

Reliable signal processing is the backbone of modern communication systems and our connected world. In this work, we grappled with a great challenge of this backbone: decoding BPSK signals within the complex realms of both Gaussian and non-Gaussian noise. To meet this challenge head on, we unleashed the **Tri-Quarter framework** [13]—a new approach that harnesses the power of **structured orientation phase pair assignments** and **dynamic weight adjustments** to advance noise filtering and error correction.

In the high-stakes race of signal processing, our simulations reveal a thrilling competition. Under AWGN's steady track, the Tri-Quarter framework's noise filtering matched standard thresholding's pace, both clocking a mere 1 CPU cycle and a 2.350% BER. In the error correction leg with 3

transmissions, Tri-Quarter **exploded ahead** with a 0.138% BER in AWGN and a 0.430% BER in IN, outpacing majority voting's 0.149% BER in AWGN and 1.415% BER in IN. Like a **bold challenger** breaking from the pack, Tri-Quarter left Gaussian-tuned soft-decision decoding—with a stellar 0.030% BER in AWGN but a lagging 12.769% BER in IN—trailing in its (non-Gaussian) wake. These results aren't just numbers—they're a testament to Tri-Quarter's unique blend of efficiency and resilience, signaling a fundamental advancement to the field.

The Tri-Quarter framework offers notable strengths. Its noise filtering efficiency is on point with standard methods, making it a strong choice for single-transmission scenarios. In error correction, its dynamic weighting mechanism dominates in non-Gaussian noise environments, such as those with chaotic IN interference—common in urban cellular wireless networks, industrial IoT networks, oceanographic sensor networks, and naval communication networks—by reducing the impact of outliers. Moreover, Tri-Quarter is **model-free** and thus requires no prior noise variance estimates, unlike soft-decision decoding. This adaptability makes it a versatile tool for applications where noise characteristics are unpredictable.

However, Tri-Quarter has limitations. Error correction in Tri-Quarter requires **17 CPU cycles** for 3 transmissions, compared to majority voting's efficient **3 CPU cycles**, which may pose challenges for ultra-low-power devices. In Gaussian noise environments, its performance is robust but slightly less optimal than soft-decision decoding's tailored approach. Scaling Tri-Quarter to higher-order modulations like QPSK could also increase computational complexity, whereas standard constellation-based methods may adapt more readily. So this is worthy of future investigation.

Choosing the right approach depends on the application. For scenarios with ample computational resources and unpredictable noise—such as urban cellular wireless networks, industrial IoT networks, oceanographic sensor networks, and naval communication networks—Tri-Quarter's robustness and flexibility make it a compelling choice. In controlled settings with predominantly Gaussian noise, like satellite communications, soft-decision decoding's precision may be preferable. For low-power devices where simplicity is key, majority voting provides an efficient alternative, though it may struggle with complex noise conditions. Each method has its place, tailored to the specific demands of the communication environment.

The Tri-Quarter framework represents a **significant advance** in signal processing, combining efficiency and robustness with a novel phase-based approach. Its structured orientation phase pair assignments and dynamic distance-based weighting tackle some current challenges while opening doors to future innovations. Pushing onward, Tri-Quarter may be extended to complex modulations or optimized for specific noise profiles, thereby possibly enhancing its role in next-generation communication systems. As the demand for reliable connectivity grows, Tri-Quarter offers a promising foundation for overcoming some noisy challenges in our connected world.

Dedication

I wish to dedicate this paper to the 249th birthday of the United States of America. Happy birthday USA!

Acknowledgement

I wish to thank the AI tool Grok, created by xAI, for its assistance with testing and refining some ideas for this paper. The original ideas and core contributions remain the author's own.

Author's Note

I earned a B.S. in computer science and a minor in mathematics at Eastern Oregon University, and then a dual M.S. in computer science and mathematics at Boise State University. From 2007 to 2017, I engaged in research work in various capacities to apply computer science and mathematics to various fields such as machine learning, robotics, bioinformatics, high energy physics, quantum gravity, sustainable energy, and cryptography. I see much interdisciplinary overlap across these great fields. I'm currently working full-time as a software development engineer and doing some unpaid after-hours research as a hobby. This is my second math or science paper after taking an eight-year pause from research.

I originally came up with ideas for the Tri-Quarter framework back in 2012, but I never had the opportunity and realization to fully test, refine, and finalize them into a formalized framework until recently. It's been exciting to apply the Tri-Quarter framework to this BPSK case study with such potential for practical real-world applications.

References

[1] Andrea Goldsmith. *Wireless Communications*. Cambridge University Press, Cambridge, UK, 2005. ISBN 978-0-521-83716-3.

[2] Theodore S. Rappaport. *Wireless Communications: Principles and Practice*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2002. ISBN 978-0-13-042232-3. Covers modulation techniques in wireless systems, including Bluetooth's GFSK and DQPSK, with references to BPSK as a foundational modulation scheme.

[3] Giovanni E. Corazza. *Digital Satellite Communications*. Springer, New York, NY, USA, 2007. ISBN 978-0-387-34649-6.

[4] Frank Van Diggelen. *A-GPS: Assisted GPS, GNSS, and SBAS*. Artech House, Norwood, MA, USA, 2009. ISBN 978-1-59693-374-3. Provides practical insights into GPS signal processing, emphasizing BPSK's role in reliable satellite navigation.

[5] Elliott D. Kaplan and Christopher J. Hegarty. *Understanding GPS/GNSS: Principles and Applications*. Artech House, Norwood, MA, USA, 3rd edition, 2017. ISBN 978-1-63081-058-0. Details BPSK modulation for GPS signals, including C/A and P codes, with applications in satellite navigation systems.

[6] IEEE Standards Association. IEEE standard for information technology—telecommunications and information exchange between systems—local and metropolitan area networks—specific requirements, part 15.1: Wireless medium access control (MAC) and physical layer (PHY) specifications for wireless personal area networks (WPANs). IEEE Std 802.15.1-2005, 2005. Specifies Bluetooth's physical layer, including GFSK and DQPSK modulation, with BPSK as a baseline for comparison in wireless personal area networks.

[7] Tallal Elshabrawy and J. Robert. Performance of nonorthogonal FSK for the Internet of Things. *Computer Communications*, 141:1–11, 2019. doi: 10.1016/j.comcom.2019.04.004.

[8] Jianhua Zhang, Fan Liu, Christos Masouros, and Robert W. Heath. An overview of signal processing techniques for joint communication and radar sensing. *IEEE Transactions on Wireless Communications*, 18(2):1295–1315, 2019. doi: 10.1109/TWC.2019.2893452. Discusses BPSK performance in urban 5G cellular networks, addressing challenges like multipath fading and impulsive noise in integrated sensing and communication systems.

[9] Martin Wollschlaeger, Thilo Sauter, and Jürgen Jasperneite. The future of industrial communication: Automation networks in the era of the Internet of Things and Industry 4.0. *IEEE Industrial Electronics Magazine*, 11(1):17–27, 2017. doi: 10.1109/MIE.2017.2649104. Details BPSK's use in industrial IoT networks, highlighting issues with impulsive noise and latency in factory automation environments.

[10] Milica Stojanovic and James C. Preisig. Underwater acoustic communication channels: Propagation models and statistical characterization. *IEEE Communications Magazine*, 54(1):84–89, 2016. doi: 10.1109/MCOM.2016.7378428. Addresses BPSK in underwater acoustic sensor networks, focusing on impulsive noise, multipath fading, and Doppler effects in oceanographic applications.

[11] Michael Rice, Travis Oliphant, and William McIntire. Estimation techniques for BPSK signals in the presence of impulsive noise. *IEEE Transactions on Communications*, 48(10):1698–1708, 2000. doi: 10.1109/26.871395. Examines BPSK performance in naval communication networks, addressing challenges with impulsive noise and fading in maritime environments.

[12] John G. Proakis and Masoud Salehi. *Digital Communications*. McGraw-Hill, New York, NY, USA, 5th edition, 2008. ISBN 978-0-07-295716-7.

[13] Nathan O. Schmidt. The Tri-Quarter framework: Unifying complex coordinates with topological and reflective duality across circles of any radius. HAL Open Archive, hal-05010951, 2025. URL https://hal.science/hal-05010951. Preprint available on TechRxiv: https://www.techrxiv.org/users/906377/articles/1281679. Published by Cold Hammer Research & Development LLC, Eagle, Idaho, USA.

[14] Nathan O. Schmidt. Tri-Quarter toolbox GitHub repository. https://github.com/nathanoschmidt/tri-quarter-toolbox/, 2025. Accessed: 2025-07-04.

Appendix A Simulation Scripts

The following Python simulatiuon scripts are freely available online [14] and should be fully compatible with Python 3.6+.

A.1 Tri-Quarter Framework BER Simulation

```python
# Script 1: Tri-Quarter Framework BER Simulation
# Simulates noise filtering and error correction for BPSK using the
# Tri-Quarter framework as described in Sections 4 and 5, with results
# reported in Section 6.
# Note: The Tri-Quarter noise filtering uses the same decision rule as
# standard thresholding (decide +1 if y >= 0, else -1), and both methods
# produce identical BER results.
import numpy as np

# Parameters (Section 6.1)
num_trials = 100000  # Number of BPSK symbols
k = 3  # Transmissions per symbol for error correction
sigma = np.sqrt(0.25)  # Noise standard deviation (SNR = 6 dB, sigma^2 = 0.25)
outlier_prob = 0.1  # Probability of impulsive outliers
outlier_amplitude = 5  # Outlier amplitude for impulsive noise

# Initialize error counters for noise filtering (Tri-Quarter and Standard
# Thresholding) and error correction
errors_filtering_awgn_tri = 0  # Tri-Quarter filtering, AWGN
errors_filtering_impulsive_tri = 0  # Tri-Quarter filtering, Impulsive
errors_filtering_awgn_std = 0  # Standard thresholding, AWGN
errors_filtering_impulsive_std = 0  # Standard thresholding, Impulsive
errors_correction_awgn = 0
errors_correction_impulsive = 0

np.random.seed(42)  # For reproducibility
for _ in range(num_trials):
    # Generate true symbol: vec{x} = +1 or -1 (Section 2)
    x_true = np.random.choice([1, -1])

    # --- Noise Filtering Simulation (Section 4, k=1) ---
    # Generate real noise
    noise_awgn = np.random.normal(0, sigma, 1)
    noise_impulsive = np.random.normal(0, sigma, 1)
    if np.random.random() < outlier_prob:
        noise_impulsive[0] = np.random.choice([outlier_amplitude, -outlier_amplitude])

    # Received signal: y = x + n
    y_awgn = x_true + noise_awgn[0]
    y_impulsive = x_true + noise_impulsive[0]

    # Tri-Quarter decoding (Section 4, Eq. 4.1)
    # Decision rule: decide +1 if y >= 0, else -1 (same as standard thresholding)
    x_decoded_awgn_tri = 1 if y_awgn >= 0 else -1
    x_decoded_impulsive_tri = 1 if y_impulsive >= 0 else -1

    # Standard Thresholding decoding
    # Explicitly computed for clarity, uses the same decision rule: decide +1 if y >= 0, else -1
    x_decoded_awgn_std = 1 if y_awgn >= 0 else -1
    x_decoded_impulsive_std = 1 if y_impulsive >= 0 else -1

    # Count errors for Tri-Quarter filtering
    if x_decoded_awgn_tri != x_true:
        errors_filtering_awgn_tri += 1
    if x_decoded_impulsive_tri != x_true:
        errors_filtering_impulsive_tri += 1

    # Count errors for Standard Thresholding (identical to Tri-Quarter)
    if x_decoded_awgn_std != x_true:
        errors_filtering_awgn_std += 1
    if x_decoded_impulsive_std != x_true:
        errors_filtering_impulsive_std += 1

    # --- Error Correction Simulation (Section 5, k=3) ---
    # Generate real noise for k=3 transmissions
    noise_awgn = np.random.normal(0, sigma, k)
    noise_impulsive = np.random.normal(0, sigma, k)
```

```
    for i in range(k):
        if np.random.random() < outlier_prob:
            noise_impulsive[i] = np.random.choice([outlier_amplitude, -outlier_amplitude])

    # Received signals: y_i = x + n_i
    y_awgn = x_true + noise_awgn
    y_impulsive = x_true + noise_impulsive

    # Compute distances: d_i = |y_i - sign(y_i) * 1| (Section 5)
    distances_awgn = np.abs(y_awgn - np.sign(y_awgn) * 1)
    distances_impulsive = np.abs(y_impulsive - np.sign(y_impulsive) * 1)

    # Compute weights: w_i = 1 / (1 + d_i^2)
    weights_awgn = 1 / (1 + distances_awgn**2)
    weights_impulsive = 1 / (1 + distances_impulsive**2)

    # Weighted voting (Section 5, Eq. 5.1-5.3)
    vote_plus1_awgn = np.sum(weights_awgn[y_awgn > 0])
    vote_minus1_awgn = np.sum(weights_awgn[y_awgn < 0])
    vote_plus1_impulsive = np.sum(weights_impulsive[y_impulsive > 0])
    vote_minus1_impulsive = np.sum(weights_impulsive[y_impulsive < 0])

    x_decoded_awgn = 1 if vote_plus1_awgn >= vote_minus1_awgn else -1
    x_decoded_impulsive = 1 if vote_plus1_impulsive >= vote_minus1_impulsive else -1

    # Count errors for Tri-Quarter error correction
    if x_decoded_awgn != x_true:
        errors_correction_awgn += 1
    if x_decoded_impulsive != x_true:
        errors_correction_impulsive += 1

# Compute BER (Section 6)
ber_filtering_awgn_tri = errors_filtering_awgn_tri / num_trials * 100
ber_filtering_impulsive_tri = errors_filtering_impulsive_tri / num_trials * 100
ber_filtering_awgn_std = errors_filtering_awgn_std / num_trials * 100
ber_filtering_impulsive_std = errors_filtering_impulsive_std / num_trials * 100
ber_correction_awgn = errors_correction_awgn / num_trials * 100
ber_correction_impulsive = errors_correction_impulsive / num_trials * 100

# Report results explicitly for both Tri-Quarter and Standard Thresholding
print(f"Script 1 - Noise Filtering Results:")
print(f"Tri-Quarter Filtering AWGN BER = {ber_filtering_awgn_tri:.3f}%")
print(f"Standard Thresholding AWGN BER = {ber_filtering_awgn_std:.3f}%")
print(f"Tri-Quarter Filtering Impulsive BER = {ber_filtering_impulsive_tri:.3f}%")
print(f"Standard Thresholding Impulsive BER = {ber_filtering_impulsive_std:.3f}%")
print(f"Tri-Quarter Correction AWGN BER = {ber_correction_awgn:.3f}%")
print(f"Tri-Quarter Correction Impulsive BER = {ber_correction_impulsive:.3f}%")
```

A.2 Majority Voting BER Simulation

```python
# Script 2: Majority Voting BER Simulation
# Simulates error correction for BPSK using majority voting, as
# described in Section 5, with results reported in Section 6.
import numpy as np

# Parameters (Section 6.1)
num_trials = 100000
k = 3
sigma = np.sqrt(0.25)
outlier_prob = 0.1
outlier_amplitude = 5

# Initialize error counters for error correction
errors_correction_awgn = 0
errors_correction_impulsive = 0

np.random.seed(42)
for _ in range(num_trials):
    # Generate true symbol: vec{x} = +1 or -1
    x_true = np.random.choice([1, -1])

    # Generate real noise for k=3 transmissions
    noise_awgn = np.random.normal(0, sigma, k)
    noise_impulsive = np.random.normal(0, sigma, k)
    for i in range(k):
        if np.random.random() < outlier_prob:
            noise_impulsive[i] = np.random.choice([outlier_amplitude, -outlier_amplitude])

    # Received signals: y_i = x + n_i
    y_awgn = x_true + noise_awgn
    y_impulsive = x_true + noise_impulsive

    # Compute signs: sign(y_i) (Section 3, Eq. 3.3)
    votes_awgn = np.sign(y_awgn)
    votes_impulsive = np.sign(y_impulsive)

    # Majority voting
    vote_count_awgn = np.sum(votes_awgn > 0)
    vote_count_impulsive = np.sum(votes_impulsive > 0)
    x_decoded_awgn = 1 if vote_count_awgn >= (k - vote_count_awgn) else -1
    x_decoded_impulsive = 1 if vote_count_impulsive >= (k - vote_count_impulsive) else -1

    # Count errors
    if x_decoded_awgn != x_true:
        errors_correction_awgn += 1
    if x_decoded_impulsive != x_true:
        errors_correction_impulsive += 1

# Compute BER (Section 6)
ber_correction_awgn = errors_correction_awgn / num_trials * 100  # Approx. 0.002%
ber_correction_impulsive = errors_correction_impulsive / num_trials * 100  # Approx. 1.5%
print(f"Script 2 - Majority Voting: Correction AWGN BER = {ber_correction_awgn:.3f}%, "
      f"Correction Impulsive BER = {ber_correction_impulsive:.3f}%")
```

A.3 Gaussian-Tuned Soft-Decision BER Simulation

```python
# Script 3: Gaussian-Tuned Soft-Decision BER Simulation
# Simulates error correction for BPSK using soft-decision decoding, as
# described in Section 5, with results reported in Section 6.
import numpy as np

# Parameters (Section 6.1)
num_trials = 100000
k = 3
sigma = np.sqrt(0.25)
outlier_prob = 0.1
outlier_amplitude = 5

# Initialize error counters for error correction
errors_correction_awgn = 0
errors_correction_impulsive = 0

np.random.seed(42)
for _ in range(num_trials):
    # Generate true symbol: vec{x} = +1 or -1
    x_true = np.random.choice([1, -1])

    # Generate real noise for k=3 transmissions
    noise_awgn = np.random.normal(0, sigma, k)
    noise_impulsive = np.random.normal(0, sigma, k)
    for i in range(k):
        if np.random.random() < outlier_prob:
            noise_impulsive[i] = np.random.choice([outlier_amplitude, -outlier_amplitude])

    # Received signals: y_i = x + n_i
    y_awgn = x_true + noise_awgn
    y_impulsive = x_true + noise_impulsive

    # Compute weights: w_i = y_i^2 / sigma^2 (Section 6)
    weights_awgn = y_awgn**2 / (sigma**2)
    weights_impulsive = y_impulsive**2 / (sigma**2)

    # Weighted voting
    vote_plus1_awgn = np.sum(weights_awgn[y_awgn > 0])
    vote_minus1_awgn = np.sum(weights_awgn[y_awgn < 0])
    vote_plus1_impulsive = np.sum(weights_impulsive[y_impulsive > 0])
    vote_minus1_impulsive = np.sum(weights_impulsive[y_impulsive < 0])

    x_decoded_awgn = 1 if vote_plus1_awgn >= vote_minus1_awgn else -1
    x_decoded_impulsive = 1 if vote_plus1_impulsive >= vote_minus1_impulsive else -1

    # Count errors
    if x_decoded_awgn != x_true:
        errors_correction_awgn += 1
    if x_decoded_impulsive != x_true:
        errors_correction_impulsive += 1

# Compute BER (Section 6)
ber_correction_awgn = errors_correction_awgn / num_trials * 100  # Approx. 0.000%
ber_correction_impulsive = errors_correction_impulsive / num_trials * 100  # Approx. 12.8%
print(f"Script 3 - Soft-Decision: Correction AWGN BER = {ber_correction_awgn:.3f}%, "
      f"Correction Impulsive BER = {ber_correction_impulsive:.3f}%")
```