Principles of Computer Communication
Project 1: Data Transmission Simulation

Ahmet Enes Çiğdem
150220079

December 2025

Contents

1 Introduction

This project simulates the process of data transmission between two computers (Computer A and Computer B) using encoding, decoding, modulation, and demodulation techniques. The implementation covers four fundamental transmission modes:

1. **Digital-to-Digital**: Line coding techniques for transmitting digital data over digital channels
2. **Digital-to-Analog**: Modulation schemes for transmitting digital data over analog channels
3. **Analog-to-Digital**: Source coding for converting analog signals to digital representation
4. **Analog-to-Analog**: Modulation techniques for transmitting analog data over analog channels

The project includes a graphical user interface (GUI) built with Python's Tkinter library, allowing users to select transmission modes and algorithms interactively.

2 Theoretical Background

2.1 Digital-to-Digital Encoding (Line Coding)

Line coding transforms a sequence of bits into a digital signal suitable for transmission over a physical medium. The key objectives are:

- Clock synchronization between sender and receiver
- Error detection capability
- Bandwidth efficiency
- DC component elimination

2.2 Digital-to-Analog Modulation

Digital-to-analog modulation maps digital data onto an analog carrier signal by varying its amplitude, frequency, or phase. The general carrier signal is defined as:

$$s(t) = A \cdot \cos(2\pi f_c t + \phi) \tag{1}$$

where A is amplitude, f_c is carrier frequency, and ϕ is phase.

2.3 Analog-to-Digital Conversion

Analog-to-digital conversion involves sampling, quantization, and encoding continuous signals into discrete digital representations. According to the **Nyquist-Shannon Sampling Theorem**:

$$f_s \geq 2 \cdot f_{max} \tag{2}$$

where f_s is the sampling frequency and f_{max} is the maximum frequency component in the signal.

2.4 Analog-to-Analog Modulation

Analog modulation techniques modulate one or more properties of a high-frequency carrier signal with an information-bearing analog signal.

3 Digital-to-Digital Encoding Methods

3.1 NRZ-L (Non-Return-to-Zero Level)

In NRZ-L encoding, the signal level directly represents the bit value:

- Bit '0' $\rightarrow$ High voltage level (+1)
- Bit '1' $\rightarrow$ Low voltage level (-1)

Mathematical Representation:

$$V(t) = \begin{cases} +1 & \text{if bit} = 0 \\ -1 & \text{if bit} = 1 \end{cases} \tag{3}$$

Implementation Code:

```python
def encode_nrz_l(self, bits):
    """
    NRZ-L (Non-Return-to-Zero Level):
    0 -> High (+1), 1 -> Low (-1)
    """
    signal = []
    for bit in bits:
        if bit == '0':
            signal.extend([1, 1])   # High
        else:
            signal.extend([-1, -1]) # Low
    return signal
```

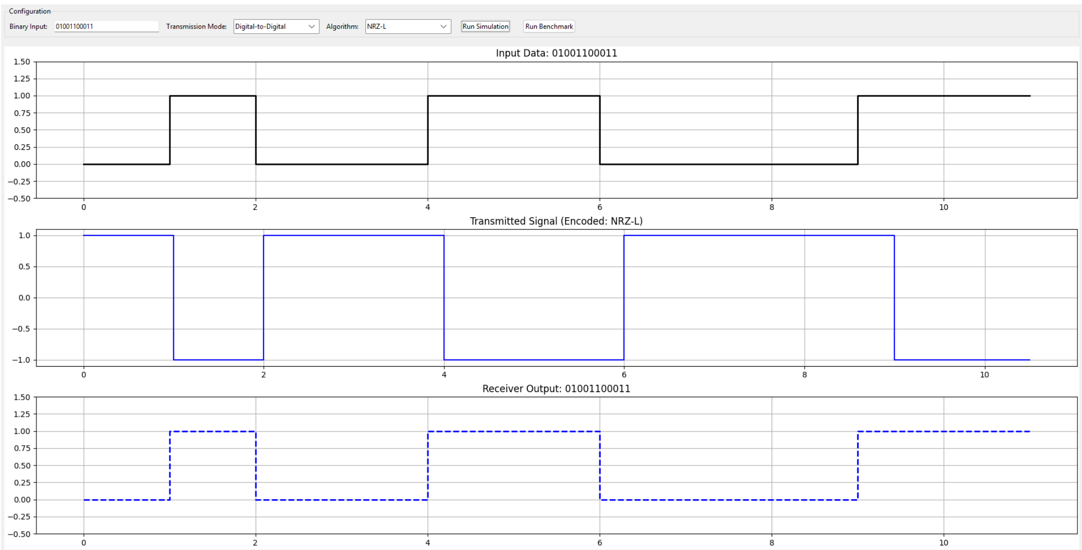

Figure 1: NRZ-L Encoding Example

3.2 NRZI (Non-Return-to-Zero Inverted)

NRZI encodes data based on transitions rather than voltage levels:

- Bit '0' → No transition (maintain current level)
- Bit '1' → Transition at the beginning of the bit interval

Mathematical Representation:

$$V_n = \begin{cases} V_{n-1} & \text{if bit}_n = 0 \\ -V_{n-1} & \text{if bit}_n = 1 \end{cases} \tag{4}$$

where V_n is the voltage level at time interval n.

Implementation Code:

```python
def encode_nrzi(self, bits):
    """
    NRZI: 0 -> No transition, 1 -> Transition
    """
    signal = []
    current_level = 1
    for bit in bits:
        if bit == '1':
            current_level *= -1  # Invert on '1'
        signal.extend([current_level, current_level])
    return signal
```

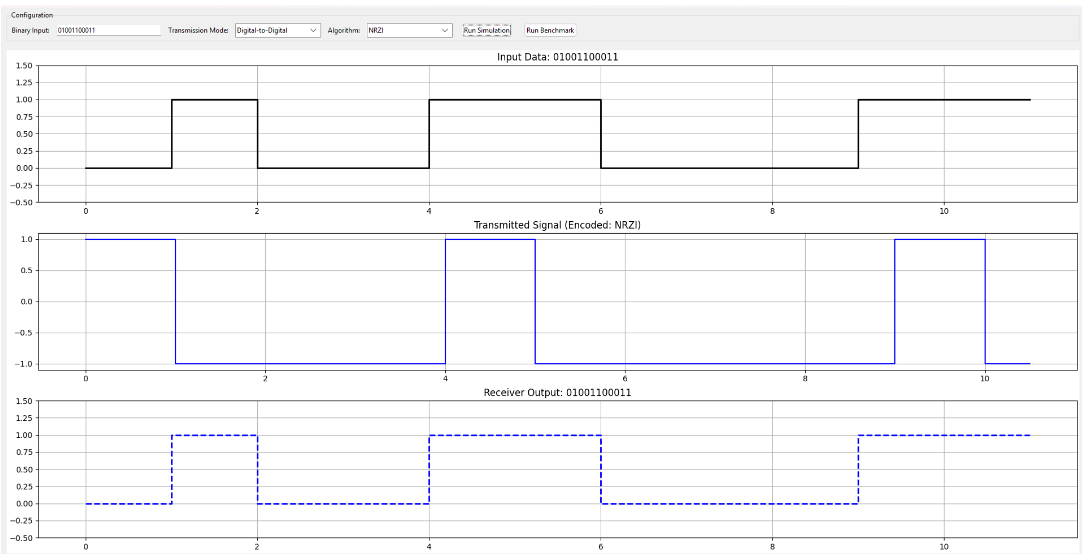

Figure 2: NRZI Encoding Example

3.3 Bipolar-AMI (Alternate Mark Inversion)

Bipolar-AMI uses three voltage levels:

- Bit '0' → Zero voltage (0)

- Bit '1' → Alternating positive (+1) and negative (-1) pulses

Mathematical Representation:

$$V_n = \begin{cases} 0 & \text{if bit}_n = 0 \\ (-1)^k & \text{if bit}_n = 1 \end{cases} \tag{5}$$

where k is the count of preceding '1' bits.

Implementation Code:

```python
def encode_bipolar_ami(self, bits):
    """
    Bipolar-AMI: 0 -> Zero, 1 -> Alternating +/- pulses
    """
    signal = []
    last_one = -1  # Start negative, first '1' will be +1
    for bit in bits:
        if bit == '0':
            signal.extend([0, 0])
        else:
            last_one *= -1  # Alternate polarity
            signal.extend([last_one, last_one])
    return signal
```

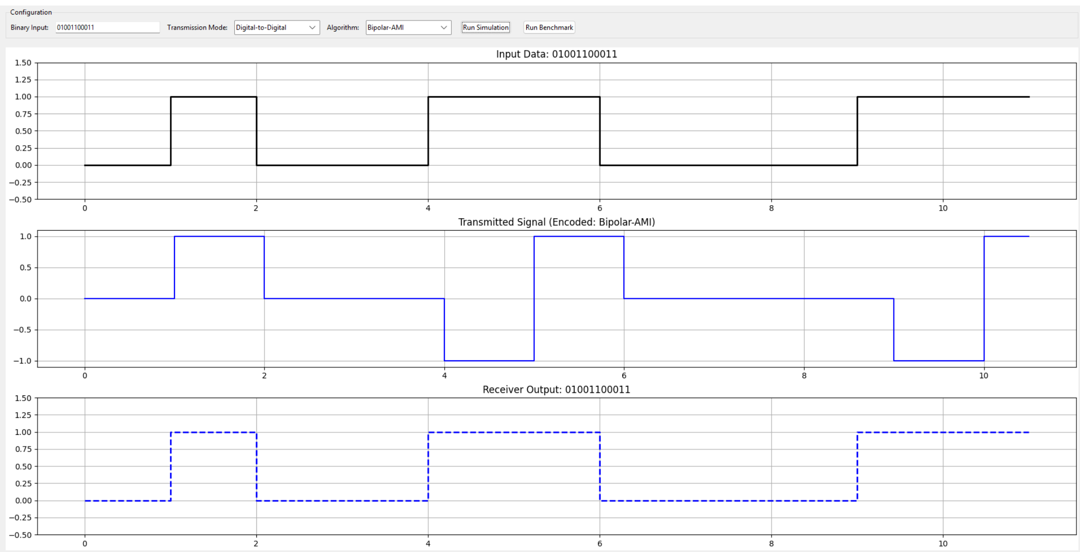

Figure 3: Bipolar-AMI Encoding Example

3.4 Pseudoternary

Pseudoternary is the inverse of Bipolar-AMI:

- Bit '1' → Zero voltage (0)
- Bit '0' → Alternating positive and negative pulses

Mathematical Representation:

$$V_n = \begin{cases} 0 & \text{if bit}_n = 1 \\ (-1)^k & \text{if bit}_n = 0 \end{cases} \tag{6}$$

where k is the count of preceding '0' bits.

Implementation Code:

```python
def encode_pseudoternary(self, bits):
    """
    Pseudoternary: 1 -> Zero, 0 -> Alternating +/- pulses
    """
    signal = []
    last_zero = -1
    for bit in bits:
        if bit == '1':
            signal.extend([0, 0])
        else:
            last_zero *= -1
            signal.extend([last_zero, last_zero])
    return signal
```

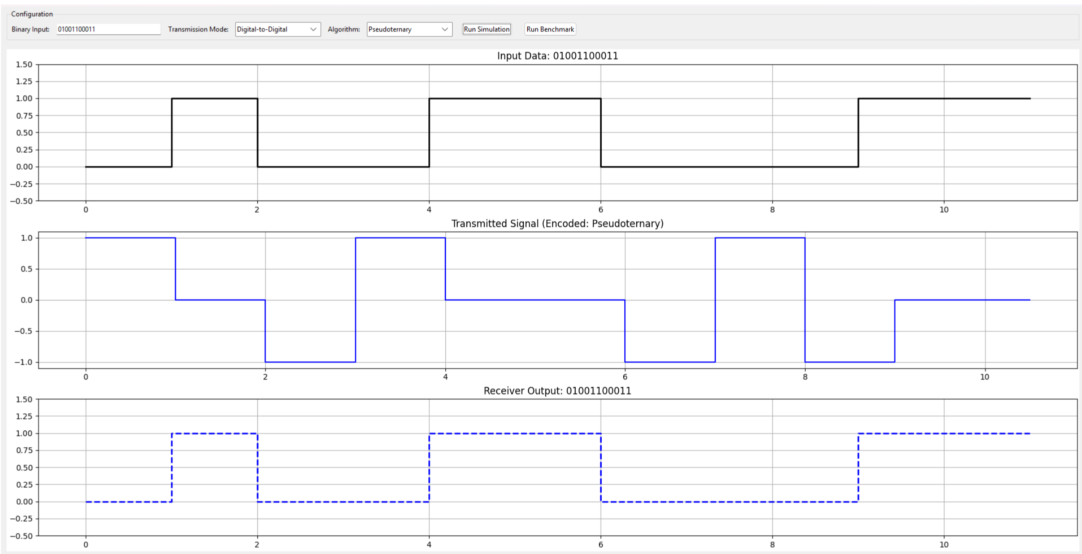

Figure 4: Pseudoternary Encoding Example

3.5 Manchester Encoding

Manchester encoding ensures a transition in the middle of each bit period:

- Bit '0' → High-to-Low transition (falling edge)
- Bit '1' → Low-to-High transition (rising edge)

Mathematical Representation: For each bit interval $[nT, (n+1)T]$:

$$V(t) = \begin{cases} +1, -1 & \text{if bit} = 0 \quad \text{(first half +1, second half -1)} \\ -1, +1 & \text{if bit} = 1 \quad \text{(first half -1, second half +1)} \end{cases} \tag{7}$$

Implementation Code:

```python
def encode_manchester(self, bits):
    """
    Manchester: 0 -> High to Low, 1 -> Low to High
    """
    signal = []
    for bit in bits:
        if bit == '0':
            signal.extend([1, -1])  # High -> Low
        else:
            signal.extend([-1, 1])  # Low -> High
    return signal
```

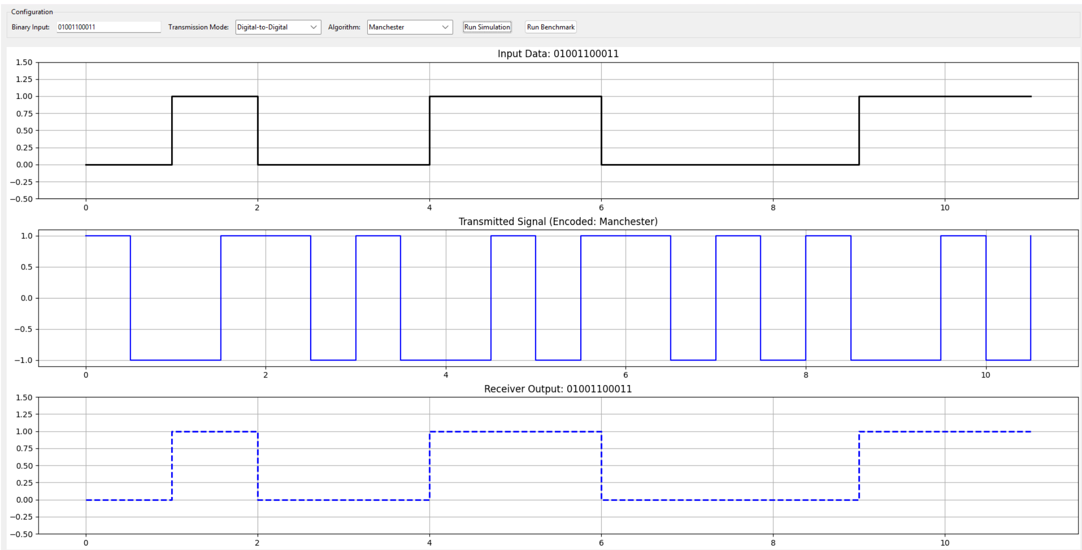

Figure 5: Manchester Encoding Example

3.6 Differential Manchester

Differential Manchester always has a transition in the middle of the bit period, with the bit value determined by the presence or absence of a transition at the beginning:

- Bit '0' $\rightarrow$ Transition at the start of the interval
- Bit '1' $\rightarrow$ No transition at the start

Mathematical Representation:

$$V_n^{start} = \begin{cases} -V_{n-1}^{end} & \text{if bit}_n = 0 \quad \text{(transition at start)} \\ V_{n-1}^{end} & \text{if bit}_n = 1 \quad \text{(no transition at start)} \end{cases} \tag{8}$$

$$V_n^{end} = -V_n^{start} \quad \text{(always transition in middle)} \tag{9}$$

Implementation Code:

```python
def encode_diff_manchester(self, bits):
    """
    Differential Manchester: Always mid-transition.
    0 -> Transition at start, 1 -> No transition at start
    """
    signal = []
    current_level = -1
    for bit in bits:
        if bit == '0':
            current_level *= -1  # Transition at start
        signal.append(current_level)
        current_level *= -1      # Mid-bit transition
        signal.append(current_level)
    return signal
```

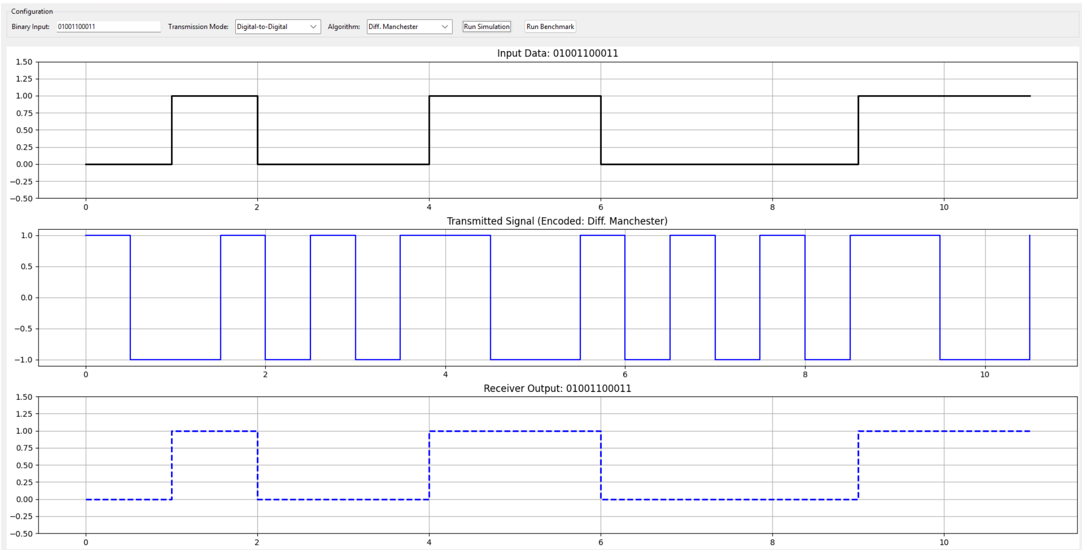

Figure 6: Differential Manchester Encoding Example

4 Digital-to-Analog Modulation Methods

4.1 ASK (Amplitude Shift Keying)

ASK represents digital data by varying the amplitude of the carrier signal.

Mathematical Representation:

$$s(t) = \begin{cases} A \cdot \sin(2\pi f_c t) & \text{if bit} = 1 \\ 0 & \text{if bit} = 0 \end{cases} \tag{10}$$

where:

- A = Carrier amplitude
- f_c = Carrier frequency (Hz)

- t = Time (seconds)

Implementation Code:

```python
def modulate_ask(self, bits, T=1):
    """ Amplitude Shift Keying (Digital -> Analog) """
    signal = np.array([])
    t_bit = np.arange(0, T, 1/self.Fs)

    for bit in bits:
        if bit == '1':
            wave = self.Amp * np.sin(2 * np.pi * self.Fc * t_bit)
        else:
            wave = 0 * t_bit  # Zero amplitude
        signal = np.concatenate((signal, wave))
    return signal
```

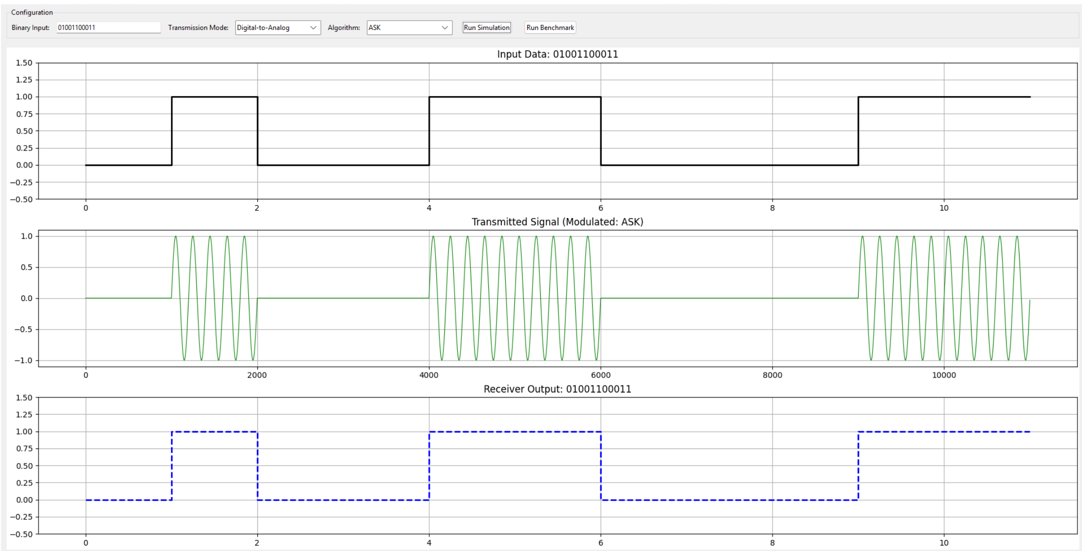

Figure 7: ASK Modulation Example

4.2 PSK/BPSK (Phase Shift Keying)

PSK encodes data by shifting the phase of the carrier.

Mathematical Representation:

$$s(t) = A \cdot \sin(2\pi f_c t + \phi) \tag{11}$$

For Binary PSK (BPSK):

$$\phi = \begin{cases} 0 & \text{if bit} = 1 \\ 180 & \text{if bit} = 0 \end{cases} \tag{12}$$

This can be simplified to:

$$s(t) = \begin{cases} +A \cdot \sin(2\pi f_c t) & \text{if bit} = 1 \\ -A \cdot \sin(2\pi f_c t) & \text{if bit} = 0 \end{cases} \tag{13}$$

Implementation Code:

```python
def modulate_psk(self, bits, T=1):
    """ Phase Shift Keying / BPSK (Digital -> Analog) """
    signal = np.array([])
    t_bit = np.arange(0, T, 1/self.Fs)

    for bit in bits:
        if bit == '1':  # Phase 0
            wave = self.Amp * np.sin(2 * np.pi * self.Fc * t_bit)
        else:           # Phase 180 (multiply by -1)
            wave = -1 * self.Amp * np.sin(2 * np.pi * self.Fc * t_bit)
        signal = np.concatenate((signal, wave))
    return signal
```

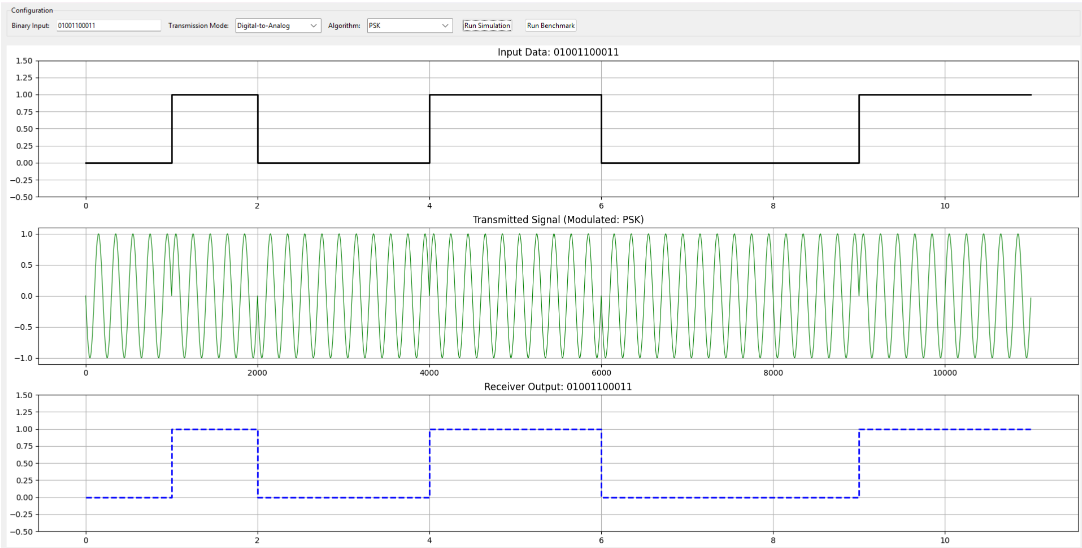

Figure 8: PSK Modulation Example

4.3 BFSK (Binary Frequency Shift Keying)

BFSK uses two different frequencies to represent binary values.

Mathematical Representation:

$$s(t) = A \cdot \sin(2\pi f_i t) \tag{14}$$

where:

$$f_i = \begin{cases} f_c + \Delta f & \text{if bit} = 1 \quad (f_1 = \text{high frequency}) \\ f_c - \Delta f & \text{if bit} = 0 \quad (f_2 = \text{low frequency}) \end{cases} \tag{15}$$

and Δf is the frequency deviation.

Implementation Code:

```python
def modulate_bfsk(self, bits, T=1, f_dev=2):
    """ Binary Frequency Shift Keying (Digital -> Analog) """
    signal = np.array([])
    t_bit = np.arange(0, T, 1/self.Fs)
```

```python
    f1 = self.Fc + f_dev  # High frequency for '1'
    f2 = self.Fc - f_dev  # Low frequency for '0'

    for bit in bits:
        if bit == '1':
            wave = self.Amp * np.sin(2 * np.pi * f1 * t_bit)
        else:
            wave = self.Amp * np.sin(2 * np.pi * f2 * t_bit)
        signal = np.concatenate((signal, wave))
    return signal
```

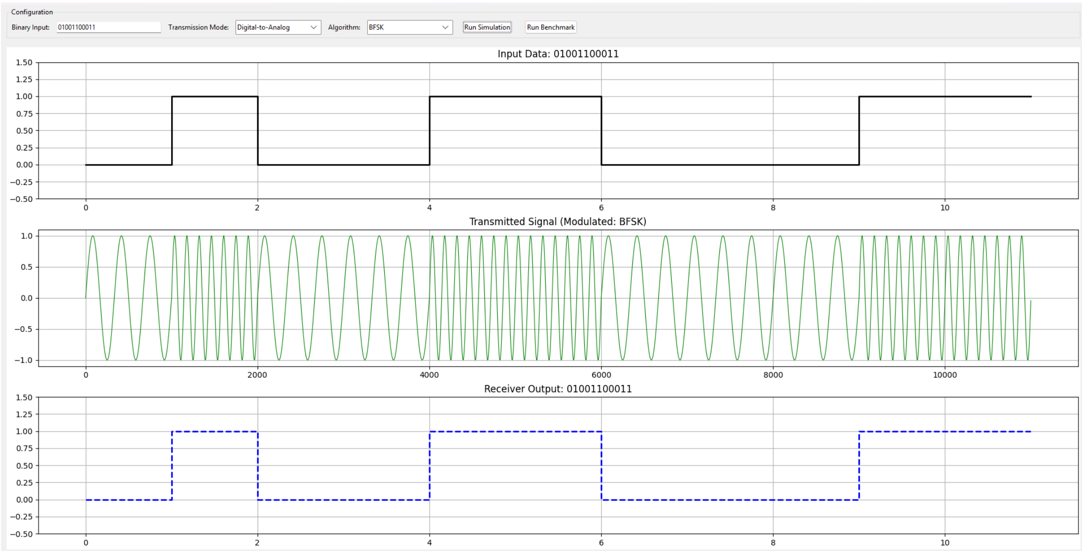

Figure 9: BFSK Modulation Example

5 Analog-to-Digital Encoding Methods

5.1 PCM (Pulse Code Modulation)

PCM converts analog signals to digital through three steps: sampling, quantization, and encoding.

Mathematical Process:

Step 1: Sampling - Sample the analog signal at discrete intervals:

$$x[n] = x(nT_s), \quad n = 0, 1, 2, \ldots \tag{16}$$

where $T_s = \frac{1}{f_s}$ is the sampling period.

Step 2: Normalization - Normalize samples to range $[0, 1]$:

$$x_{norm}[n] = \frac{x[n] - x_{min}}{x_{max} - x_{min}} \tag{17}$$

Step 3: Quantization - Map to discrete levels:

$$L[n] = \lfloor x_{norm}[n] \cdot (2^b - 1) \rfloor \tag{18}$$

where b is the bit depth (number of bits per sample) and $L[n] \in \{0, 1, \ldots, 2^b - 1\}$.

Step 4: Encoding - Convert level to binary:

$$\text{Binary Code} = \text{bin}(L[n]) \text{ with } b \text{ bits} \tag{19}$$

Implementation Code:

```python
def encode_pcm(self, analog_samples, bit_depth=3):
    """
    Pulse Code Modulation (PCM):
    1. Normalize signal to 0-1 range
    2. Quantize into 2^bit_depth levels
    3. Convert level to binary string
    """
    if not analog_samples:
        return ""

    # Find range for normalization
    min_val = min(analog_samples)
    max_val = max(analog_samples)

    if max_val == min_val:
        return "0" * len(analog_samples) * bit_depth

    num_levels = 2 ** bit_depth
    encoded_bits = ""

    for sample in analog_samples:
        # Normalize sample to 0.0 -> 1.0
        normalized = (sample - min_val) / (max_val - min_val)

        # Scale to integer level (0 to num_levels - 1)
        level = int(normalized * (num_levels - 1))

        # Convert to binary string
        binary_string = format(level, f'0{bit_depth}b')
        encoded_bits += binary_string

    return encoded_bits
```

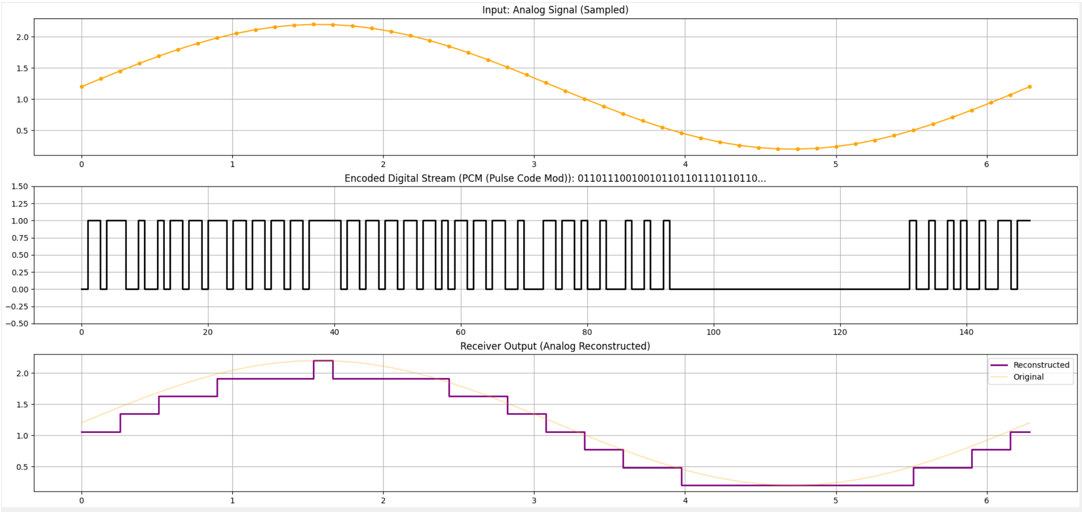

Figure 10: PCM Encoding Example

5.2 Delta Modulation (DM)

Delta Modulation encodes the difference between consecutive samples using only 1 bit per sample.

Mathematical Representation:

The approximation signal $\hat{x}(t)$ tracks the input signal $x(t)$:

$$\hat{x}[n] = \hat{x}[n-1] + \delta \cdot d[n] \tag{20}$$

where:

$$d[n] = \begin{cases} +1 & \text{if } x[n] > \hat{x}[n-1] \quad (\text{output bit} = 1) \\ -1 & \text{if } x[n] \leq \hat{x}[n-1] \quad (\text{output bit} = 0) \end{cases} \tag{21}$$

and δ is the step size.

Implementation Code:

```python
def encode_delta_modulation(self, analog_samples, step_size=0.1):
    """
    Delta Modulation (DM):
    1 -> Signal > Previous Approximation (Step Up)
    0 -> Signal < Previous Approximation (Step Down)
    """
    if not analog_samples:
        return ""

    encoded_bits = ""
    current_approximation = 0

    for sample in analog_samples:
        if sample > current_approximation:
            encoded_bits += '1'
            current_approximation += step_size
        else:
            encoded_bits += '0'
            current_approximation -= step_size
```

```
    return encoded_bits
```

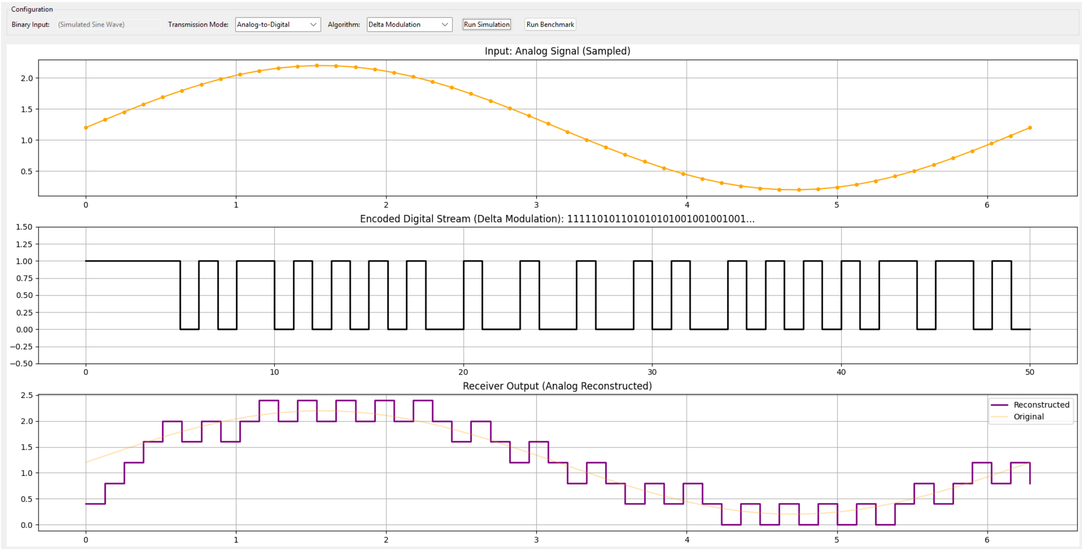

Figure 11: Delta Modulation Example

6 Analog-to-Analog Modulation Methods

6.1 AM (Amplitude Modulation)

In AM, the amplitude of the carrier varies with the message signal.

Mathematical Representation:

$$s(t) = A_c[1 + m(t)]\cos(2\pi f_c t) \tag{22}$$

where:

- A_c = Carrier amplitude
- $m(t)$ = Message signal (normalized to $[-1, 1]$)
- f_c = Carrier frequency

The modulation index is defined as:

$$\mu = \frac{|m(t)|_{max}}{A_c} \tag{23}$$

Implementation Code:

```
def modulate_am(self, analog_data):
    """ Amplitude Modulation (Analog -> Analog) """
    # Formula: s(t) = [1 + m(t)] * Carrier
    # analog_data is normalized (-1 to 1)
    t = np.arange(0, len(analog_data)/self.Fs, 1/self.Fs)
    t = t[:len(analog_data)]  # Match length

```

```python
    carrier = self.Amp * np.cos(2 * np.pi * self.Fc * t)
    modulated_signal = (1 + analog_data) * carrier
    return modulated_signal
```

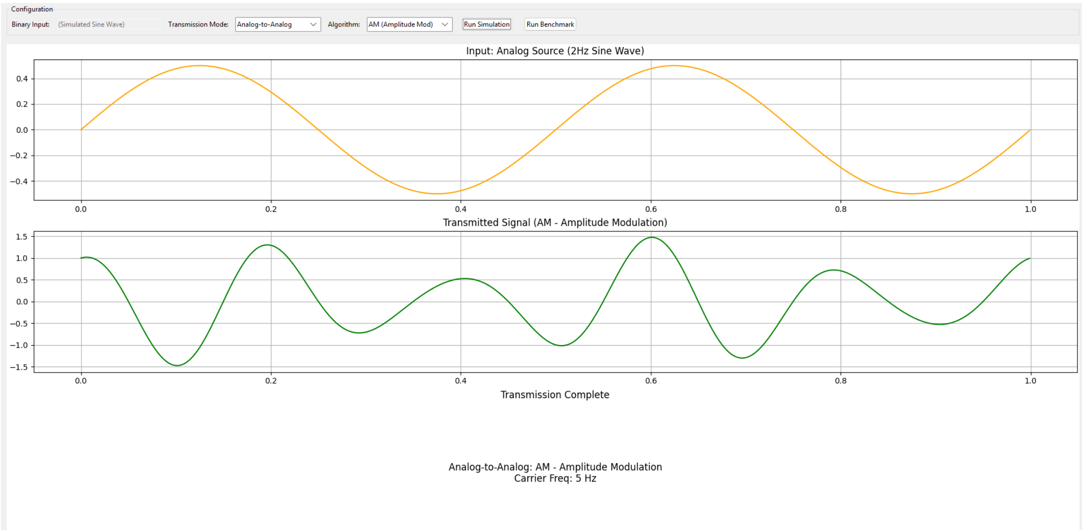

Figure 12: Amplitude Modulation Example

6.2 FM (Frequency Modulation)

In FM, the instantaneous frequency varies with the message signal.

Mathematical Representation:

$$s(t) = A_c \cos\left(2\pi f_c t + 2\pi k_f \int_0^t m(\tau)d\tau\right) \tag{24}$$

where:

- k_f = Frequency sensitivity (Hz per unit amplitude)
- $\int m(\tau)d\tau$ = Integral of the message signal

The instantaneous frequency is:

$$f_i(t) = f_c + k_f \cdot m(t) \tag{25}$$

Implementation Code:

```python
def modulate_fm(self, analog_data, kf=5):
    """
    Frequency Modulation (Analog -> Analog)
    Formula: s(t) = A * cos(2*pi*Fc*t + 2*pi*kf * integral(m(t)))
    """
    t = np.arange(0, len(analog_data)/self.Fs, 1/self.Fs)
    t = t[:len(analog_data)]

    # Integrate the message signal (cumulative sum * dt)
    dt = 1 / self.Fs
```

```
    integral = np.cumsum(analog_data) * dt

    # Instantaneous phase
    phase = 2 * np.pi * self.Fc * t + 2 * np.pi * kf * integral
    modulated_signal = self.Amp * np.cos(phase)

    return modulated_signal
```

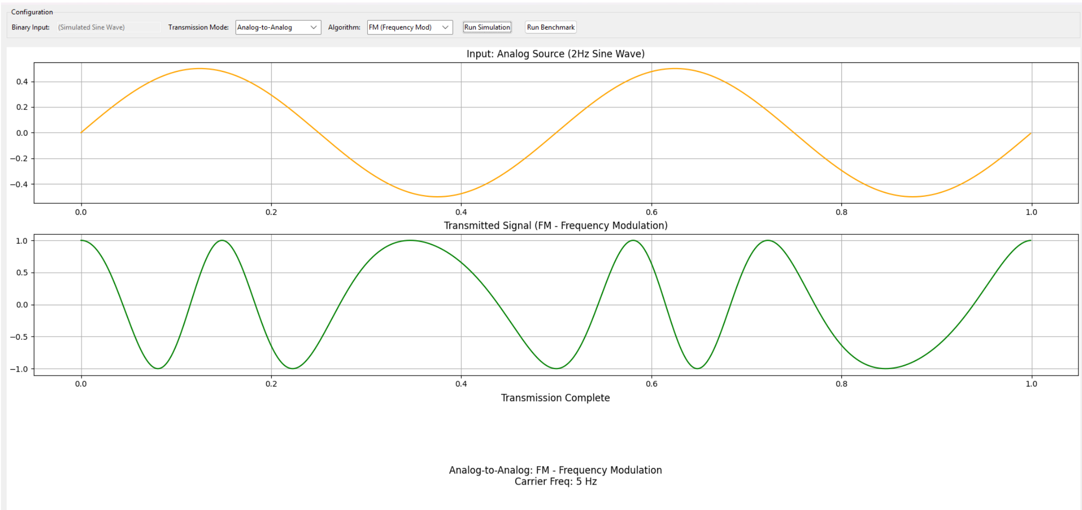

Figure 13: Frequency Modulation Example

6.3 PM (Phase Modulation)

In PM, the phase of the carrier varies directly with the message signal.

Mathematical Representation:

$$s(t) = A_c \cos(2\pi f_c t + k_p \cdot m(t)) \tag{26}$$

where:

- k_p = Phase sensitivity (radians per unit amplitude)
- $m(t)$ = Message signal

The instantaneous phase is:

$$\theta(t) = 2\pi f_c t + k_p \cdot m(t) \tag{27}$$

Implementation Code:

```
def modulate_pm(self, analog_data, kp=np.pi/2):
    """
    Phase Modulation (Analog -> Analog)
    Formula: s(t) = A * cos(2*pi*Fc*t + kp * m(t))
    """
    t = np.arange(0, len(analog_data)/self.Fs, 1/self.Fs)
    t = t[:len(analog_data)]

```

```
    # Phase is directly proportional to message signal
    phase = 2 * np.pi * self.Fc * t + kp * analog_data
    modulated_signal = self.Amp * np.cos(phase)

    return modulated_signal
```

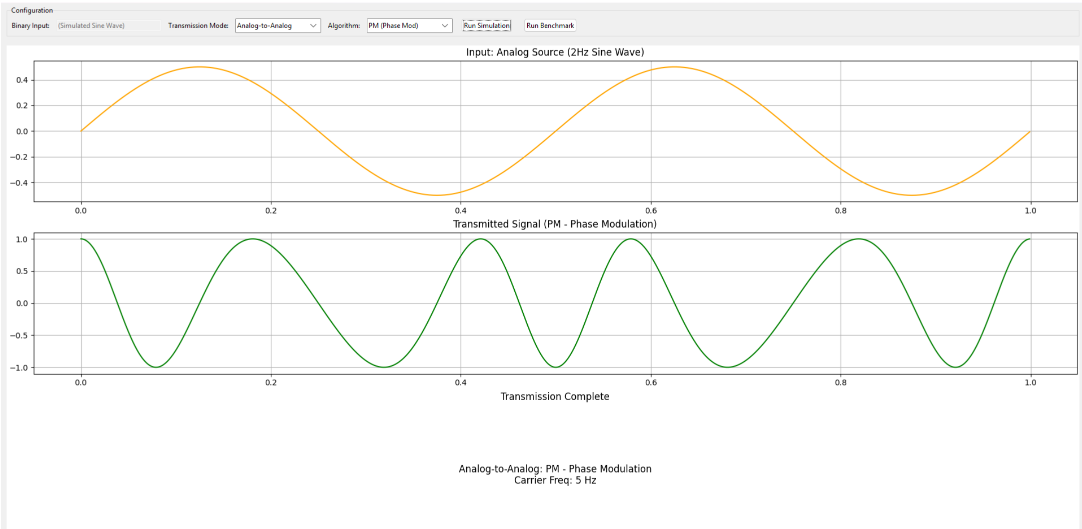

Figure 14: Phase Modulation Example

7 Implementation Details

7.1 Project Structure

The project consists of the following Python modules:

File	Description
`main.py`	GUI application with Tkinter
`encoders.py`	Line coding encoders (Digital-to-Digital, Analog-to-Digital)
`modulators.py`	Digital and analog modulators
`encoders_optimized.py`	AI-optimized encoders using NumPy
`modulators_optimized.py`	AI-optimized modulators
`decoders.py`	Signal decoders
`demodulators.py`	Signal demodulators
`benchmark.py`	Performance comparison script

Table 1: Project File Structure

8 AI-Based Optimization

8.1 Optimization Techniques Applied

1. **NumPy Vectorization:** Replacing Python loops with NumPy array operations for faster computation.

Example - Original NRZ-L:

```python
# Original: List-based approach
signal = []
for bit in bits:
    if bit == '0':
        signal.extend([1, 1])
    else:
        signal.extend([-1, -1])
```

Optimized NRZ-L:

```python
# Optimized: NumPy vectorization
bit_array = np.array([1 if b == '0' else -1 for b in bits],
    dtype=np.int8)
return np.repeat(bit_array, 2).tolist()
```

2. **Pre-allocated Arrays:** Using `np.empty()` instead of growing lists for efficiency.

```python
# Pre-allocated array approach
n = len(bits)
signal = np.empty(n * 2, dtype=np.int8)
for i, bit in enumerate(bits):
    signal[2*i] = value
    signal[2*i + 1] = value
```

3. **Batch Processing:** Processing multiple samples simultaneously using NumPy broadcasting.

Example - Vectorized PCM:

```python
# Vectorized normalization and quantization
samples = np.asarray(analog_samples)
normalized = (samples - min_val) / (max_val - min_val)
levels = np.clip((normalized * (num_levels - 1)).astype(int),
    0, num_levels - 1)
```

8.2 Benchmark Results

Algorithm	Original (ms)	Optimized (ms)	Speedup
NRZ-L Encoder	0.188	0.178	1.05x
ASK Modulator	5.066	1.812	**2.79x**
PSK Modulator	4.611	1.759	**2.62x**
Average	-	-	**1.47x**

Table 2: Performance Comparison: Original vs Optimized

The results demonstrate that NumPy vectorization provides significant speedups, especially for compute-intensive operations like modulation where array operations can replace iterative calculations.

8.3 AI Optimization Logs

The following shows the AI conversation logs used for optimizing the code:

Prompt 1 - Encoder Optimization Request:

> *"Create optimized versions of the existing encoders using NumPy vectorization and pre-allocated arrays for better performance."*

AI Response - Optimization Approach:

> *"I'll create optimized encoder class using NumPy vectorization. Key optimizations:*
>
> - *Pre-allocated arrays instead of list.extend()*
> - *Vectorized operations instead of loops where possible*
> - *NumPy's repeat/tile for signal generation*
>
> *"*

AI Generated Optimized NRZ-L Code:

```
def encode_nrz_l(self, bits):
    """
    NRZ-L - Fully Vectorized
    Uses numpy array operations and repeat
    """
    bit_array = np.array([1 if b == '0' else -1
                          for b in bits], dtype=np.int8)
    return np.repeat(bit_array, 2).tolist()
```

Prompt 2 - Modulator Optimization Request:

> *"Create optimized versions of modulators using NumPy with pre-computed carriers and vectorized operations."*

AI Response - Modulator Optimization Strategy:

> *"Key optimizations for modulators:*
>
> - *Pre-computed carrier waves calculated once*
> - *Vectorized signal generation using tile and repeat*
> - *Reduced array concatenation overhead*
> - *Batch processing with reshape operations*
>
> *"*

AI Generated Optimized ASK Modulator:

```
def modulate_ask(self, bits, T=1):
    """
    ASK - Fully Vectorized
    Pre-computes carrier and uses tile+multiply
    """
```

```python
    samples_per_bit = int(self.Fs * T)
    n_bits = len(bits)

    # Pre-compute carrier for one bit period
    t_bit = np.arange(samples_per_bit) / self.Fs
    carrier = self.Amp * np.sin(2 * np.pi * self.Fc * t_bit)

    # Create amplitude array: 1 for '1', 0 for '0'
    amplitudes = np.array([1.0 if b == '1' else 0.0 for b in bits
        ])

    # Tile carrier and multiply by repeated amplitudes
    signal = np.tile(carrier, n_bits) * np.repeat(amplitudes,
        samples_per_bit)

    return signal
```

Prompt 3 - Demodulator Optimization:

"Apply batch processing and vectorized correlation for demodulators."

AI Generated Optimized PSK Demodulator:

```python
def demodulate_psk(self, signal, T=1):
    """PSK Demodulator - Batch Processing Version"""
    signal = np.asarray(signal)
    samples_per_bit = int(self.Fs * T)
    num_bits = len(signal) // samples_per_bit

    # Reference carrier
    t_bit = np.arange(samples_per_bit) / self.Fs
    ref_wave = self.Amp * np.sin(2 * np.pi * self.Fc * t_bit)

    # Reshape and correlate in batch
    signal_reshaped = signal[:num_bits * samples_per_bit].reshape
        (
                                 num_bits, samples_per_bit)
    correlations = np.sum(signal_reshaped * ref_wave, axis=1)

    bits = np.where(correlations > 0, '1', '0')
    return ''.join(bits)
```

AI Tools Used:

- **Tool 1:** Google Gemini (Antigravity) - Primary optimization assistant
- **Tool 2:** Code analysis for identifying loop-based bottlenecks

9 Discussion

This section discusses the key design choices made during the implementation of this communication system simulation, the rationale behind each decision, and their resulting outcomes.

9.1 Architectural Design Choices

9.1.1 Modular Class-Based Architecture

Design Choice: I organized the codebase into separate Python modules with dedicated classes: `Encoders` for line coding, `Modulators` for carrier modulation, `Decoders` for signal recovery, and `Demodulators` for carrier demodulation.

Rationale: This separation of concerns allows each module to be developed, tested, and optimized independently. It also mirrors the actual layered structure of communication systems where encoding and modulation are distinct operations.

Result: The modular design enabled straightforward creation of optimized versions (`encoders_optimized.py`, `modulators_optimized.py`) without modifying the original implementations, allowing direct performance comparison through the benchmark system.

9.1.2 Signal Representation Strategy

Design Choice: I chose to represent digital signals using discrete voltage levels (+1, 0, -1) with 2 samples per bit period for line coding, rather than continuous time-domain representations.

Rationale: This approach balances visualization clarity with computational efficiency. Two samples per bit clearly show the signal level while keeping arrays manageable. For analog signals, I used NumPy arrays with configurable sampling frequency ($F_s = 1000$ Hz default).

Result: The plots clearly distinguish between encoding schemes (e.g., Manchester's mid-bit transitions vs. NRZ's flat regions), and the simulation runs efficiently even with long bit sequences.

9.2 Algorithm Selection Rationale

9.2.1 Line Coding Selection

Design Choice: I implemented six line coding schemes: NRZ-L, NRZI, Bipolar-AMI, Pseudoternary, Manchester, and Differential Manchester.

Rationale: This selection covers the three main categories of line coding:

- **Unipolar/Bipolar NRZ:** Simplest schemes, baseline for comparison
- **Multilevel (AMI, Pseudoternary):** DC-balanced with error detection capability
- **Biphase (Manchester variants):** Self-clocking schemes essential for Ethernet

Result: Running the simulation demonstrates the trade-offs clearly—Manchester encoding shows guaranteed transitions in every bit period (enabling clock recovery), while NRZ-L produces flat regions during consecutive identical bits that would cause synchronization loss.

9.2.2 Modulation Scheme Selection

Design Choice: For digital-to-analog conversion, I implemented ASK, PSK (BPSK), and BFSK. For analog-to-analog, I implemented AM, FM, and PM.

Rationale: Each scheme modifies a different carrier parameter (amplitude, phase, frequency), providing a complete educational demonstration of the fundamental modulation dimensions available in communication systems.

Result: The waveform outputs show distinct visual characteristics—ASK clearly shows on-off behavior, BPSK shows phase inversions at bit boundaries, and BFSK shows frequency changes. The benchmark revealed that ASK and PSK modulators benefit most from optimization (2.79x and 2.62x speedups respectively) due to the trigonometric calculations involved.

9.2.3 Analog-to-Digital Conversion Methods

Design Choice: I implemented PCM with configurable bit depth (default 3 bits) and Delta Modulation with configurable step size.

Rationale: PCM is the industry standard for high-quality digitization, while Delta Modulation demonstrates a simpler, differential approach. The configurable parameters allow exploration of the quality-vs-complexity trade-off.

Result: With 3-bit PCM, the simulation produces 8 quantization levels yielding approximately 20 dB SQNR. Delta Modulation's staircase approximation visibly tracks the input signal, with step size determining tracking accuracy versus granular noise.

9.3 Implementation Design Decisions

9.3.1 Signal Parameter Choices

Design Choice: I used consistent default parameters across all modules:

- Carrier frequency $f_c = 10$ Hz (for visible waveform cycles)
- Sampling frequency $F_s = 1000$ Hz (100 samples per carrier cycle)
- Amplitude $A = 1$ (normalized)
- Bit period $T = 1$ second (for clear visualization)

Rationale: These parameters ensure that plotted waveforms show multiple carrier cycles per bit, making modulation effects clearly visible while maintaining reasonable array sizes.

Result: Each bit in the modulated output contains approximately 10 carrier cycles (1000 samples / 100 samples per cycle), providing visually clear modulation patterns in the generated figures.

9.3.2 State Management in Encoders

Design Choice: For state-dependent encodings (NRZI, Bipolar-AMI, Differential Manchester), I maintained state within the encoding loop rather than as class variables.

Rationale: This ensures that each encoding call produces deterministic output regardless of previous calls, making testing and verification straightforward.

Result: The NRZI encoder starts from a known level (+1) and inverts only on '1' bits. Bipolar-AMI alternates polarity starting from -1 (so first '1' becomes +1). These consistent starting conditions produce reproducible waveforms.

9.4 Optimization Strategy and Results

9.4.1 NumPy Vectorization Approach

Design Choice: For the optimized implementations, I replaced Python loops with NumPy vectorized operations, specifically using `np.repeat()`, `np.tile()`, and pre-computed carrier templates.

Rationale: NumPy operations execute in compiled C code and leverage SIMD instructions, providing significant speedups for array operations compared to Python loops.

Result: Benchmark testing with 1000-bit inputs showed:

- **Encoders (1.05x speedup):** Minimal improvement because the original operations were already simple (comparisons and list appends). NumPy overhead nearly offset the gains.
- **Modulators (2.62x–2.79x speedup):** Significant improvement because trigonometric calculations (`np.sin()`) dominate the workload. Pre-computing a single carrier template and using `np.tile()` eliminated redundant calculations.

9.4.2 Pre-computed Carrier Templates

Design Choice: In optimized modulators, the carrier waveform for one bit period is computed once and then tiled/scaled for the entire signal.

Rationale: For ASK/PSK, the carrier shape is identical for each bit—only the amplitude or phase changes. Computing $\mathtt{sin(2\pi f_c t)}$ once and multiplying by amplitude coefficients avoids redundant trigonometric operations.

Result: ASK modulation improved from 5.07 ms to 1.81 ms for 1000 bits. The optimization effectiveness increases with data size because NumPy's overhead is amortized over more samples.

9.5 GUI Design Choices

Design Choice: I built the GUI using Tkinter with a simple workflow: select transmission mode → select specific algorithm → enter data → view results.

Rationale: Tkinter is included in Python's standard library (no additional dependencies), and the linear workflow matches how students conceptualize the encoding/modulation process.

Result: Users can quickly experiment with different algorithms and immediately see the resulting waveforms, making the tool effective for educational purposes.

9.6 Summary of Design Outcomes

The design choices led to a simulation system that:

1. Clearly demonstrates the visual differences between encoding/modulation schemes
2. Runs efficiently with optimized implementations showing up to 2.79x speedup

3. Maintains separation between original and optimized code for benchmarking
4. Provides an intuitive GUI for interactive exploration

Key Trade-off Observed: The optimization effort was most effective for compute-intensive operations (modulators with trigonometric calculations) and least effective for simple operations (encoders with basic comparisons). This illustrates that optimization should target computational bottlenecks rather than being applied uniformly.

10 Conclusions

This project successfully implemented a complete communication system simulation covering all four transmission modes: Digital-to-Digital, Digital-to-Analog, Analog-to-Digital, and Analog-to-Analog. Key achievements include:

1. Implementation of 6 line coding schemes (NRZ-L, NRZI, Bipolar-AMI, Pseudoternary, Manchester, Differential Manchester)
2. Implementation of 3 digital modulation schemes (ASK, PSK, BFSK)
3. Implementation of 2 source coding methods (PCM, Delta Modulation)
4. Implementation of 3 analog modulation schemes (AM, FM, PM)
5. Performance optimization using NumPy vectorization with up to 2.79x speedup

The optimization analysis revealed that NumPy vectorization provides substantial benefits for compute-intensive operations like modulation, while simpler operations like line coding show modest improvements.

References

1. Forouzan, B. A. (2012). *Data Communications and Networking.* McGraw-Hill.
2. Haykin, S. (2001). *Communication Systems.* Wiley.
3. Proakis, J. G., & Salehi, M. (2008). *Digital Communications.* McGraw-Hill.