Signal Encoding and Modulation Techniques

Technical Report - BLG 337E Principles of Computer Communication

Authors:

Mustafa Bozdoğan 150210007
Enes Saçak 150210014

Instructor:
Prof. Dr. Abdül Halim Zaim

Teaching Assistants:
Gülizar Kondel Büşra Bayram

January 2026
Istanbul Technical University

Abstract

This report presents a comprehensive simulation and analysis of data transmission between two computers using encoding, modulation, and demodulation techniques. The implementation covers four fundamental transmission modes: Digital-to-Digital line coding, Digital-to-Analog modulation, Analog-to-Digital conversion, and Analog-to-Analog modulation. We implement 16 different algorithms based on William Stallings' "Data and Computer Communications" curriculum, develop an interactive web-based simulator using Streamlit, and perform AI-assisted optimization with detailed benchmarking analysis.

Contents

1 Introduction

Data communication between computers requires the conversion of data into signals suitable for transmission over various media. This report presents the implementation of a signal encoding and modulation simulator that demonstrates four fundamental transmission modes:

1. **Digital-to-Digital (Line Coding)**: Converting digital data to digital signals
2. **Digital-to-Analog (Modulation)**: Converting digital data to analog signals
3. **Analog-to-Digital (Digitization)**: Converting analog signals to digital data
4. **Analog-to-Analog (Modulation)**: Converting analog signals to different analog signals

The simulator implements algorithms as defined in William Stallings' textbook and provides an interactive interface for visualizing the encoding/modulation processes.

2 Theoretical Background

2.1 Digital-to-Digital Encoding (Line Coding)

Line coding converts digital data into digital signals suitable for transmission over a physical medium. The main objectives are synchronization, error detection, and spectral efficiency.

2.1.1 NRZ (Non-Return to Zero)

NRZ-L (Level): The signal level directly represents the bit value.

- Binary 0 → Positive voltage (+V)
- Binary 1 → Negative voltage (-V)

NRZI (Invert on ones): Uses signal transitions to represent data.

- Binary 1 → Transition at the beginning of the bit period
- Binary 0 → No transition

Advantages: Simple implementation, efficient bandwidth usage.
Disadvantages: No self-clocking capability, DC component issues with long sequences of same bits.

2.1.2 Bipolar-AMI (Alternate Mark Inversion)

A three-level encoding scheme:

- Binary 0 → Zero voltage (no signal)
- Binary 1 → Alternating positive and negative voltage

The alternating polarity for 1s eliminates the DC component and provides error detection capability. However, long sequences of zeros cause synchronization loss.

2.1.3 Manchester Encoding (IEEE 802.3)

Self-clocking code with guaranteed mid-bit transitions:

- Binary 0 → High-to-low transition at mid-bit
- Binary 1 → Low-to-high transition at mid-bit

Advantages: Self-synchronizing, no DC component, excellent for Ethernet.
Disadvantages: Requires twice the bandwidth of NRZ.

2.1.4 Differential Manchester

Similar to Manchester but uses transitions differently:

- Always has a mid-bit transition (for clocking)
- Binary 0 → Transition at the beginning of bit period
- Binary 1 → No transition at the beginning

2.2 Scrambling Techniques: B8ZS and HDB3

Scrambling techniques address the synchronization problem of AMI encoding when long sequences of zeros occur.

2.2.1 B8ZS (Bipolar with 8-Zero Substitution) - North American Standard

B8ZS replaces 8 consecutive zeros with a special violation pattern:
Substitution Rule:

$$8 \text{ zeros} \rightarrow 000VB0VB$$

where:

- V = Violation (same polarity as last pulse, violates AMI rule)
- B = Bipolar (alternating polarity, follows AMI rule)

The receiver detects these deliberate violations and restores the original zeros. This maintains synchronization while preserving the error detection capability of bipolar encoding.

2.2.2 HDB3 (High Density Bipolar 3) - European Standard

HDB3 substitutes 4 consecutive zeros based on the count of 1s since the last substitution:
Substitution Rules:

- Odd number of 1s since last substitution: $0000 \rightarrow 000V$
- Even number of 1s since last substitution: $0000 \rightarrow B00V$

The additional B pulse ensures that violations can always be detected by maintaining proper polarity counting. HDB3 is used in E-carrier systems (E1, E2, etc.) in Europe.

2.3 Digital-to-Analog Modulation

Digital-to-analog modulation converts digital data to analog signals for transmission over analog channels such as telephone lines or wireless media.

2.3.1 ASK (Amplitude Shift Keying)

The amplitude of the carrier signal represents the digital data:

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

where $A(t) = 1$ for binary 1 and $A(t) = 0$ for binary 0.
ASK is simple but susceptible to noise that affects signal amplitude.

2.3.2 FSK (Frequency Shift Keying)

The frequency of the carrier represents the data:

$$s(t) = \cos(2\pi f_i t) \tag{2}$$

where f_1 is used for binary 0 and f_2 for binary 1. FSK is more robust against noise than ASK.

2.3.3 PSK (Phase Shift Keying)

The phase of the carrier represents the data.
BPSK (Binary PSK):

$$s(t) = \cos(2\pi f_c t + \phi) \tag{3}$$

where $\phi = 0$ for binary 0 and $\phi = 180$ for binary 1.
DPSK (Differential PSK): Phase change represents the data rather than absolute phase, making it easier to implement without requiring carrier phase recovery.

2.3.4 QAM (Quadrature Amplitude Modulation)

QAM combines ASK and PSK to achieve higher data rates:

$$s(t) = I \cdot \cos(2\pi f_c t) + Q \cdot \sin(2\pi f_c t) \tag{4}$$

where I and Q are the in-phase and quadrature components representing multiple bits per symbol. 4-QAM (QPSK) uses 2 bits per symbol, 16-QAM uses 4 bits, and 64-QAM uses 6 bits.

2.4 Analog-to-Digital Conversion and the Sampling Theorem

2.4.1 The Nyquist Sampling Theorem

The foundation of analog-to-digital conversion is the **Nyquist-Shannon Sampling Theorem**:

> *A continuous-time signal can be completely represented by its samples if the sampling frequency f_s is at least twice the highest frequency component f_{max} of the signal:*

$$\boxed{f_s \geq 2 \cdot f_{max}} \tag{5}$$

If this condition is violated, **aliasing** occurs, where high-frequency components appear as lower frequencies in the sampled signal, causing irreversible distortion.

2.4.2 PCM (Pulse Code Modulation)

PCM is the standard method for digitizing analog signals and involves three steps:

1. **Sampling**: Sample the analog signal at rate $f_s \geq 2f_{max}$
2. **Quantization**: Map continuous amplitude values to discrete levels

$$L = 2^n \tag{6}$$

   where n is the number of bits and L is the number of quantization levels.

3. **Encoding**: Convert quantized values to binary representation

Quantization Signal-to-Noise Ratio:

$$SNR_{dB} = 6.02n + 1.76 \tag{7}$$

Each additional bit increases the SNR by approximately 6 dB.

2.4.3 Delta Modulation

Delta Modulation is a simpler 1-bit encoding using staircase approximation:

- If signal $>$ approximation: output 1, increase approximation by step size δ
- If signal $\leq$ approximation: output 0, decrease approximation by step size δ

Types of Distortion:

- **Slope Overload**: Step size too small to track rapid signal changes
- **Granular Noise**: Step size too large for slow-varying signals

2.5 Analog-to-Analog Modulation

2.5.1 AM (Amplitude Modulation)

In AM, the amplitude of a high-frequency carrier is varied according to the message signal:

$$s(t) = [1 + m \cdot x(t)] \cdot \cos(2\pi f_c t) \tag{8}$$

where:

- m = modulation index ($0 < m \leq 1$)
- $x(t)$ = normalized message signal ($-1 \leq x(t) \leq 1$)
- f_c = carrier frequency

Bandwidth: $BW = 2 \cdot f_m$ where f_m is the message bandwidth.

2.5.2 FM (Frequency Modulation)

In FM, the frequency of the carrier varies with the message signal:

$$s(t) = \cos\left(2\pi f_c t + 2\pi k_f \int_0^t x(\tau)d\tau\right) \tag{9}$$

where k_f is the frequency deviation constant.
Carson's Rule for Bandwidth:

$$BW \approx 2(\Delta f + f_m) \tag{10}$$

where Δf is the peak frequency deviation.
FM provides better noise immunity than AM at the cost of increased bandwidth.

3 Implementation

3.1 System Architecture

The implementation follows a modular object-oriented design:

Listing 1: Class Structure

```
# Encoder Classes
class DigitalToDigitalEncoder:
    ALGORITHMS = ['NRZ-L', 'NRZI', 'Bipolar-AMI',
                  'Manchester', 'Differential Manchester',
                  'B8ZS', 'HDB3']
    def encode(data, algorithm) -> (time, signal)

class DigitalToAnalogModulator:
    ALGORITHMS = ['ASK', 'BFSK', 'BPSK', 'DPSK', 'QAM']
    def modulate(data, algorithm) -> (time, signal)

class AnalogToDigitalConverter:
    ALGORITHMS = ['PCM', 'Delta Modulation']
    def convert(time, signal, algorithm) -> (samples, bits,
       quantized)

class AnalogToAnalogModulator:
    ALGORITHMS = ['AM', 'FM']
    def modulate(time, signal, algorithm) -> (time, modulated)
```

3.2 Project Structure

```
signal-encoding-simulator/
 app.py                      # Streamlit dashboard
 requirements.txt            # Dependencies
 encoders/
    digital_to_digital.py    # 7 line coding algorithms
    digital_to_analog.py     # 5 modulation algorithms
```

```
    analog_to_digital.py    # PCM, Delta Modulation
    analog_to_analog.py     # AM, FM
visualization/
    plotter.py              # Matplotlib visualization
benchmarks/
    version_a.py            # Original implementation
    version_b.py            # Runtime optimized
    version_c.py            # Memory optimized
    run_benchmark.py        # Benchmark runner
report/
     blg337e_1.pdf          # This report
```

3.3 User Interface

The simulator uses Streamlit to provide an interactive web-based dashboard with:

- Mode selection (4 transmission types)
- Algorithm selection within each mode
- Adjustable parameters (carrier frequency, sample rate, etc.)
- Real-time waveform visualization
- Encode/Decode verification with match indicators

4 AI-Based Optimization and Benchmarking

4.1 Optimization Strategies

Three versions of the core algorithms were developed:

4.1.1 Version A: Original Implementation

- Straightforward Python implementation using loops
- Clear, readable code following the algorithm definitions
- Serves as the baseline for comparison

4.1.2 Version B: Runtime Optimized

- NumPy vectorization replacing Python loops
- Use of `np.where()`, `np.repeat()`, broadcasting
- Pre-computed constants outside loops
- Minimized function call overhead

4.1.3 Version C: Memory Optimized

- `float32` instead of `float64` (50% memory reduction)
- Avoided intermediate array creation
- In-place operations where possible
- Comprehensive documentation for maintainability

4.2 Benchmark Results

Benchmarks were conducted using Python's `timeit` module for execution time and `tracemalloc` for memory profiling.

Table 1: Execution Time Comparison (milliseconds)

Data Size	Version A	Version B	Version C	C vs A
100 samples	0.745	0.632	0.319	-57.2%
500 samples	1.882	2.203	1.318	-30.0%
1000 samples	3.614	4.413	2.556	-29.3%
5000 samples	18.645	22.168	18.185	-2.5%

Table 2: Memory Usage Comparison (kilobytes)

Data Size	Version A	Version B	Version C	Improvement
100 samples	1,891.90	3,504.87	1,329.34	29.7%
500 samples	9,396.13	17,257.99	6,641.84	29.3%
1000 samples	18,790.66	34,449.40	13,282.47	29.3%
5000 samples	93,946.91	171,980.65	66,407.47	29.3%

4.3 Analysis

1. **Version B (Runtime Optimized):** While vectorization typically improves performance, in this case the overhead of creating intermediate arrays actually increased execution time for larger datasets. This demonstrates that vectorization is not always beneficial.
2. **Version C (Memory Optimized):** Consistently achieved approximately 29% memory reduction by using `float32` arrays instead of `float64`. This halves the memory requirement for numerical data while maintaining sufficient precision for signal processing visualization.
3. **Trade-offs:** The best choice depends on the use case:
   - For memory-constrained environments: Version C
   - For production code requiring maintainability: Version C (with comprehensive documentation)
   - For rapid prototyping: Version A

5 Conclusions

This project successfully implemented a comprehensive signal encoding and modulation simulator covering:

1. **16 Algorithms**: 7 line coding (NRZ-L, NRZI, AMI, Manchester, Diff. Manchester, B8ZS, HDB3), 5 digital modulation (ASK, BFSK, BPSK, DPSK, QAM), 2 digitization (PCM, DM), and 2 analog modulation (AM, FM) techniques.
2. **Interactive Web Interface**: A Streamlit-based dashboard allowing users to select transmission modes, algorithms, and parameters while visualizing the encoding/-modulation process in real-time.
3. **AI-Assisted Optimization**: Three versions of the implementation demonstrating different optimization strategies with measurable performance improvements.
4. **Key Technical Insights**:
   - The Nyquist Sampling Theorem ($f_s \geq 2f_{max}$) is fundamental to avoiding aliasing in A/D conversion
   - Scrambling techniques (B8ZS/HDB3) solve synchronization problems while maintaining error detection
   - Memory optimization using `float32` provides consistent 29% memory savings
   - Vectorization may increase overhead for certain operation patterns

The simulator is publicly accessible at:
https://signal-encoding-simulator-yuqjysbpdtyaloro8tp4bb.streamlit.app/

References

1. Stallings, W. (2014). *Data and Computer Communications* (10th ed.). Pearson.

2. Forouzan, B. A. (2013). *Data Communications and Networking* (5th ed.). McGraw-Hill.

3. IEEE 802.3 Standard for Ethernet.

4. ITU-T G.703 - Physical/Electrical Characteristics of Hierarchical Digital Interfaces.

5. Nyquist, H. (1928). "Certain Topics in Telegraph Transmission Theory." *Transactions of the AIEE*, 47(2), 617-644.

A Running the Simulator

Listing 2: Installation and Running

```
# Clone the repository
git clone https://github.com/nonamexishere/signal-encoding-
    simulator.git
cd signal-encoding-simulator

# Install dependencies
pip install -r requirements.txt

# Run the Streamlit dashboard
streamlit run app.py

# Run benchmarks
python benchmarks/run_benchmark.py
```