Experiment No. : 03

Experiment Name: Line Coding Techniques

Objectives:

- To understand various line coding techniques used in digital communication.
- To implement Unipolar NRZ, Polar NRZ, Manchester, and Bipolar NRZ line coding schemes.
- To analyze and compare the performance of these line coding methods.
- To evaluate the spectral characteristics of each coding scheme.
- To determine the bandwidth requirements for the different line codes.
- To study the error detection capabilities of the implemented line coding techniques.

Theory:

Line coding is the process of converting data to signal.

Significance:

1. Self-synchronization.
2. Low probability of error.
3. This is the spectrum that is suitable for the channel.
4. Noise immunity.
5. Better suited for processing.
6. Simple to measure & evaluate.

Properties of Line Coding:

1. Error detection capability.
2. Transparency.
3. Transmission bandwidth should be as small as possible.

Types of Line Coding:

Unipolar Signaling:

In positive unipolar signaling, the binary 1's are represented by a high level (+A Volts) and binary 0's by a zero level. This type of signaling is called on-off keying.

Polar Signaling:

Binary 1's and 0's are represented by equal positive and negative levels.

Bipolar Signaling:

1's are represented by alternatively positive or negative values; 0's are represented by a zero level.

Manchester Signaling:

Each 1 is represented by a positive half bit period pulse, followed by a negative half bit period pulse. The reverse thing happens for a binary 0.

Implementation Step:

Unipolar Line Coding:

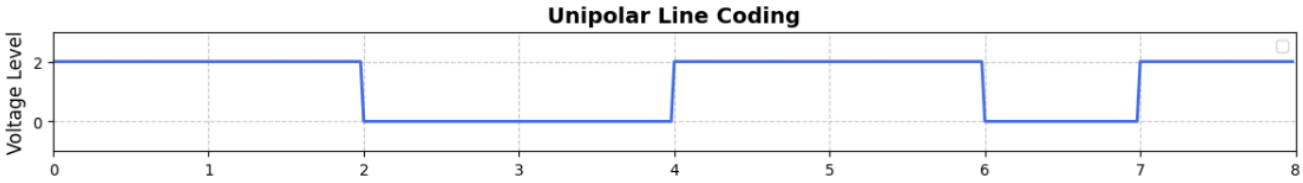
```
import numpy as np
import matplotlib.pyplot as plt

n = np.array([1, 1, 0, 0, 1, 1, 0, 1])

# Unipolar Line Coding
nn = np.where(n == 1, 2, 0)
t = np.arange(0, len(n), 0.02)
y = np.zeros_like(t)

i = 0
for j in range(len(t)):
    if t[j] >= i + 1:
        i += 1
    if i < len(nn):
        y[j] = nn[i]

plt.figure(figsize=(12, 6))
plt.subplot(4, 1, 1)
plt.plot(t, y, color='royalblue', linestyle='-', linewidth=2)
plt.ylabel('Voltage Level', fontsize=12)
plt.title('Unipolar Line Coding', fontsize=14, fontweight='bold')
plt.grid(True, linestyle='--', alpha=0.7)
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.legend(loc='upper right')
plt.axis([0, len(n), -1, 3])
plt.tight_layout()
plt.show()
```

Explanation:

The code implements Unipolar Line Coding for a binary sequence n = np.array([1, 1, 0, 0, 1, 1, 0, 1]). In this scheme, binary 1 is represented by a high

voltage (2) and binary 0 by 0, achieved using np.where(n == 1, 2, 0). A time vector t is created for plotting, and the output signal y is generated by assigning corresponding voltage levels based on the time index. The resulting waveform is plotted to visualize the unipolar representation of the binary input.

Output:



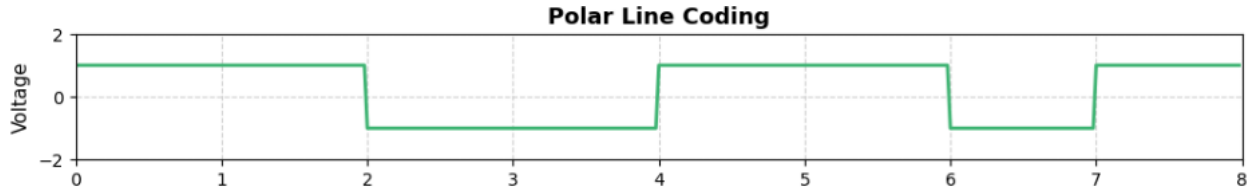
Polar Line Coding:

```
# --- Polar Line Coding ---
nn = np.where(n == 1, 1, -1)
y = np.zeros_like(t)
i = 0
for j in range(len(t)):
    if t[j] >= i + 1:
        i += 1
    if i < len(nn):
        y[j] = nn[i]

plt.subplot(4, 1, 2)
plt.plot(t, y, color='mediumseagreen', linewidth=2)
plt.title('Polar Line Coding', fontsize=13, fontweight='bold')
plt.ylabel('Voltage', fontsize=11)
plt.grid(True, linestyle='--', alpha=0.6)
plt.axis([0, len(n), -2, 2])
```

Explanation :

In the Polar line coding scheme, binary digits are mapped such that a binary 1 is represented by a positive voltage (e.g., +1) and a binary 0 is represented by a negative voltage (e.g., -1). In the code, this mapping is performed using nn = np.where(n == 1, 1, -1), which transforms the original binary input array into corresponding polar levels. A time vector t is reused from the unipolar section, and an output signal array y is initialized. A loop is used to assign each polar voltage level to the appropriate portion of the time vector, ensuring each bit occupies one unit of time. The final waveform is plotted using matplotlib, providing a visual representation of the polar encoding.

Output:



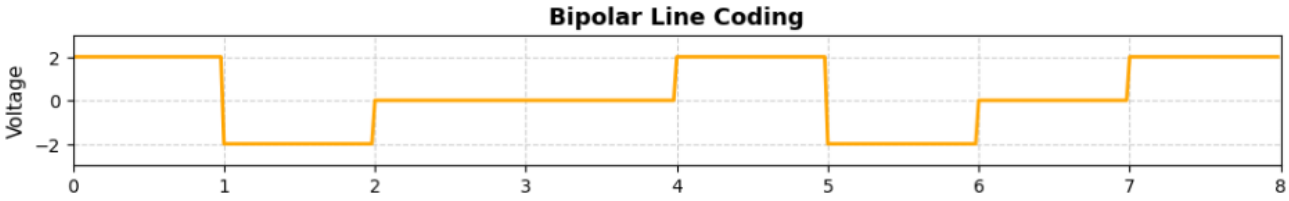
Bipolar Line Coding:

```
# --- Bipolar Line Coding ---
nn = np.zeros_like(n, dtype=int)
sign = 1
for ii in range(len(n)):
    if n[ii] == 1:
        nn[ii] = 2 * sign
        sign *= -1
    else:
        nn[ii] = 0

y = np.zeros_like(t)
i = 0
for j in range(len(t)):
    if t[j] >= i + 1:
        i += 1
    if i < len(nn):
        y[j] = nn[i]
plt.figure(figsize=(12, 6))
plt.subplot(4, 1, 1)

plt.plot(t, y, color='orange', linewidth=2)
plt.title('Bipolar Line Coding', fontsize=13, fontweight='bold')
plt.ylabel('Voltage', fontsize=11)
plt.grid(True, linestyle='--', alpha=0.6)
plt.axis([0, len(n), -3, 3])
```

Explanation :

Bipolar line coding alternates the polarity of the voltage for every binary 1, while binary 0 is represented by a zero voltage level. This helps reduce DC bias in the transmitted signal. In the code, a sign variable is initialized as 1 and flipped with each occurrence of a 1 in the input array. This is implemented using a loop that assigns either +2 or -2 to the output array *nn*, while 0 is assigned for binary zeros. A second loop then maps these values to a continuous time-domain signal y for plotting. The resulting bipolar waveform alternates between positive and negative levels for consecutive ones and shows zero for zeros.

Output:



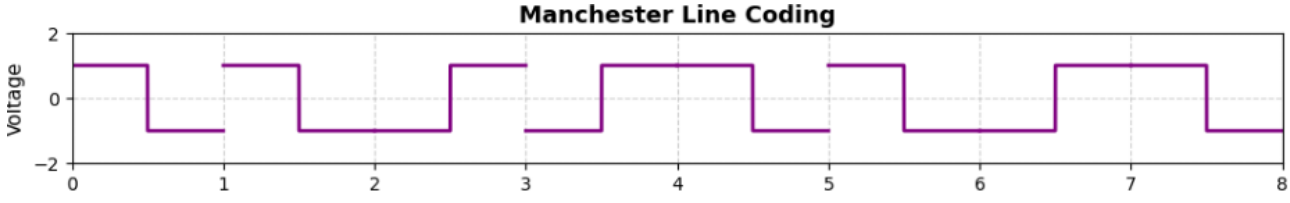
Manchester Line Coding:

```
# --- Manchester Line Coding ---
plt.figure(figsize=(12, 6))
plt.subplot(4, 1, 1)
for i in range(len(n)):
    t_seg = np.arange(i, i + 1, 0.001)
    if n[i] == 1:
        y_seg = np.where((t_seg - i) < 0.5, 1, -1)
    else:
        y_seg = np.where((t_seg - i) < 0.5, -1, 1)
    plt.plot(t_seg, y_seg, color='purple', linewidth=2)

plt.title('Manchester Line Coding', fontsize=13, fontweight='bold')
plt.ylabel('Voltage', fontsize=11)
plt.grid(True, linestyle='--', alpha=0.6)
plt.axis([0, len(n), -2, 2])
```

Explanation :

Manchester coding combines clock and data information by representing each bit with a voltage transition. A binary 1 is encoded as a high-to-low transition, and a binary 0 as a low-to-high transition, both within one bit period. In the code, this is implemented by iterating through each bit in the input sequence. For each bit, a time segment t_seg is generated, and a corresponding signal segment y_seg is created using np.where() to apply the appropriate voltage level before and after the midpoint (0.5) of the bit duration. The resulting segments are plotted sequentially to produce the full Manchester-coded waveform.

Output:



Discussion:

Spectral Characteristics: Among the line coding techniques, Manchester coding requires the highest bandwidth because it introduces a transition in the middle of every bit period. In contrast, Unipolar NRZ and Polar NRZ have lower bandwidth requirements due to fewer signal transitions. Bipolar NRZ provides a middle ground, offering a moderate bandwidth requirement along with additional benefits.

Power Efficiency: Unipolar and Polar NRZ schemes are simpler to implement but suffer from the presence of a DC component, which can degrade signal transmission over long distances. Bipolar NRZ and Manchester coding, on the other hand, eliminate the DC component by alternating signal levels or using transitions, making them more power-efficient and suitable for long-distance communication.

Error Detection: Bipolar NRZ inherently supports basic error detection; an error is suspected if two consecutive ones have the same polarity. Manchester coding also aids in error reduction by ensuring regular transitions, which enhance synchronization and reduce the likelihood of bit errors.

Practical Applications: Manchester coding is commonly used in systems like Ethernet and RFID, where precise timing and synchronization are critical. Bipolar NRZ is often applied in telecommunication networks that require error detection and efficient transmission over long distances.