NATIONAL INSTITUTE OF TECHNOLOGY , SRINAGAR

ASSIGNMENT OF DATA COMMUNICATION

MADE BY :

KHUSHI RENU (2021BITE075)

DANISHTA AKHTAR (2021BITE087)

SUBMITTED TO – DR. IQRA ALTAF

Objective:

Implement digital signal generator

- Line coding schemes to be implemented: NRZ-L, NRZ-I, Manchester, Differential Manchester, AMI and scrambling schemes: B8ZS, HDB3.
- Pulse code modulation (PCM) or Delta modulation (DM)

SPECIFICATION REPORT

LANGUAGE AND LIBRARIES USED:

Language: Python

Libraries: NumPy

ASSUMPTIONS CONSIDERED:

The input is either digital or analog. If digital, it is expected to be a sequence of 0s and 1s. If analog, a random analog signal is generated (replace it with actual analog input).

The PCM encoding assumes a linear quantization approach with a specified number of quantization levels.

For AMI encoding, scrambling can be optionally applied. Scrambling types include B8ZS (Bipolar with 8 Zero Substitution) and HDB3 (High-Density Bipolar of Order 3).

The line encoding functions include NRZ-L, NRZ-I, Manchester, Differential Manchester, and AMI.

HOW TO RUN THE CODE:

Run the script in a Python environment.

Enter the input type when prompted ("analog" or "digital").

If digital, enter the digital data (0s and 1s) when prompted.

If analog, a random analog signal is generated.

Choose the encoding type when prompted ("NRZ-L", "NRZ-I", "Manchester", "Differential Manchester", or "AMI").

For AMI encoding, you'll be prompted to specify if scrambling is desired and choose the scrambling type.

The script will print the digital data stream, the longest palindrome in the encoded data, and the digital signal produced.

If scrambling is applied, it will also print the scrambled signal.

```
import numpy as np
def nrz_l_encoding(data):
    return np.array(data)
def nrz_i_encoding(data):
    encoded_data = []
    current_level = 1
    for bit in data:
        encoded_data.extend([current_level] * len(bit))
        current_level *= -1
    return np.array(encoded_data)
def manchester_encoding(data):
    encoded_data = []
    for bit in data:
        encoded_data.extend([1, -1] if bit == 1 else [-1, 1])
    return np.array(encoded_data)
def differential_manchester_encoding(data):
    encoded_data = [1, -1]
    current_level = -1
    for bit in data:
        if bit == 1:
            current_level *= -1
        encoded_data.extend([current_level, -current_level])
    return np.array(encoded_data)
def ami_encoding(data, scrambling_type=None):
    encoded_data = []
    previous_level = 0
    for bit in data:
        if bit == 0:
            encoded_data.append(0)
        else:
            previous_level = previous_level * -1
            encoded_data.append(previous_level)
    if scrambling_type:
        encoded_data = apply_scrambling(encoded_data, scrambling_type)
    return np.array(encoded_data)
def pcm(data, quantization_levels=16):
    # Implement PCM encoding logic
    max_value = max(data)
    min_value = min(data)
    step_size = (max_value - min_value) / quantization_levels
    encoded_data = []
```

```
    for value in data:
        quantized_value = round((value - min_value) / step_size)
        encoded_data.append(quantized_value)
    return np.array(encoded_data)
def apply_scrambling(data, scrambling_type):
    return data
# Get user input
input_type = input("Enter input type (analog/digital): ").lower()
if input_type == "digital":
    data = list(map(int, input("Enter digital data (0s and 1s): ")))
else:
    analog_signal = np.random.randn(100)  # Replace this with actual analog input
    pcm_data = pcm(analog_signal)
    data = pcm_data
# Line encoding options
encoding_type = input("Choose encoding type (NRZ-L, NRZ-I, Manchester, Differential
Manchester, AMI): ").lower()
if encoding_type == "ami":
    scrambling_required = input("Do you want scrambling? (yes/no): ").lower()
    if scrambling_required == "yes":
        scrambling_type = input("Choose scrambling type (B8ZS, HDB3): ").lower()
    else:
        scrambling_type = None
    encoded_data = ami_encoding(data, scrambling_type)
else:
    encoded_data = globals()[f"{encoding_type}_encoding"](data)
# Find longest palindrome
binary_string = ''.join(map(str, encoded_data))
longest_palindrome = max((binary_string[i:j] for i in range(len(binary_string))
                          for j in range(i + 1, len(binary_string) + 1) if binary_string[i:j] ==
binary_string[i:j][::-1]),
                         key=len)
# Print results
print("Digital Data Stream:", data)
print("Longest Palindrome:", longest_palindrome)
print("Digital Signal Produced:", encoded_data)
if 'scrambling_type' in locals():
    scrambled_signal = apply_scrambling(encoded_data, scrambling_type)
    print("Scrambled Signal Produced:", scrambled_signal)
```