EXP 4 data conditioning and reconditioning

```
import numpy as np

import matplotlib.pyplot as plt

# Input binary data

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

n = 100  # Samples per bit

# Time vector for the entire signal

T = len(bits)  # Total duration in bits

t = np.arange(0, T, 1/n)

# NRZ-L encoding (Non-Return-to-Zero Level)

nrz_l = np.zeros(len(t))

for i in range(len(bits)):

    nrz_l[i*n:(i+1)*n] = bits[i]

# NRZ-M encoding (Non-Return-to-Zero Mark)

nrz_m = np.zeros(len(t))

current_level = 0  # Start at low level

for i in range(len(bits)):

    if bits[i] == 1:

        current_level = 1 - current_level  # Toggle level on '1'

    nrz_m[i*n:(i+1)*n] = current_level

# NRZ-S encoding (Non-Return-to-Zero Space)

nrz_s = np.zeros(len(t))

current_level = 1  # Start at high level for '0'

for i in range(len(bits)):

    if bits[i] == 0:

        current_level = 0  # Go high for '0'

    else:

        current_level = 1  # Go low for '1'

    nrz_s[i*n:(i+1)*n] = current_level

# Plotting

plt.figure(figsize=(12, 8))
```

```
# NRZ-L plot
plt.subplot(3, 1, 1)
plt.title('NRZ Level (NRZ-L)')
plt.plot(t, nrz_l,color='b')
plt.ylim([-0.5, 1.5])
# NRZ-M plot
plt.subplot(3, 1, 2)
plt.title('NRZ Mark (NRZ-M)')
plt.plot(t, nrz_m,color='y')
plt.ylim([-0.5, 1.5])
# NRZ-S plot
plt.subplot(3, 1, 3)
plt.title('NRZ Space (NRZ-S)')
plt.plot(t, nrz_s,color='g')
plt.ylim([-0.5, 1.5])
plt.xlabel('Time (s)')
plt.show()
```

EXP 5 ASK

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import pi
# Carrier frequency and input data
fl = 2
data = np.array([1, 1, 0, 0, 1, 0, 1])
t = np.arange(0, 1, 1/500)  # Time array
# Carrier signal
xl = np.sin(2 * pi * fl * t)
# Prepare the figure and axes
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8))
lw, fs = 3, 18  # Line width and font size
```

```
for i in range(len(data)):
    z = i  # Increment the time offset for each symbol
# Carrier signal
    ax1.plot(t + z, xl, linewidth=lw)
# ASK signal (Amplitude Shift Keying)
    ax2.plot(t + z, xl * data[i], linewidth=lw)
# Carrier signal plot settings
ax1.set_ylabel('Carrier signal', fontsize=fs)
ax1.grid()
# ASK signal plot settings
ax2.set_ylabel('ASK signal', fontsize=fs)
ax2.grid()
plt.tight_layout()  # Adjust layout to prevent overlap
plt.show()
```

exp 6 PSK

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import pi
# Carrier frequency and input data
fl = 2
data = np.array([1, 1, 0, 0, 1, 0, 1])
t = np.arange(0, 1, 1/500)  # Time array
# Carrier signal
xl = np.sin(2 * pi * fl * t)
# Prepare the figure and axes
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8))
lw, fs = 3, 18  # Line width and font size
for i in range(len(data)):
```

```
    z = i  # Increment the time offset for each symbol
    # Carrier signal
    ax1.plot(t + z, xl, linewidth=lw)
    # PSK signal (Phase Shift Keying)
    if data[i] == 1:
        psk_signal = np.sin(2 * pi * fl * t)  # No phase shift for '1'
    else:
        psk_signal = np.sin(2 * pi * fl * t + pi)  # 180-degree phase shift for '0'
    ax2.plot(t + z, psk_signal, linewidth=lw)
# Carrier signal plot settings
ax1.set_ylabel('Carrier signal', fontsize=fs)
ax1.grid()
# PSK signal plot settings
ax2.set_ylabel('PSK signal', fontsize=fs)
ax2.grid()
plt.tight_layout()  # Adjust layout to prevent overlap
plt.show()
```

EXP 7 FSK

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import pi
# Carrier frequency and input data
f1 = 2  # Frequency for '1'
f0 = 1  # Frequency for '0'
data = np.array([1, 1, 0, 0, 1, 0, 1])
t = np.arange(0, 1, 1/500)  # Time array
# Carrier signal (using frequency f1 as the base carrier)
xl = np.sin(2 * pi * f1 * t)
# Prepare the figure and axes
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8))
```

```
lw, fs = 3, 18  # Line width and font size
for i in range(len(data)):
    z = i  # Increment the time offset for each symbol
    ax1.plot(t + z, xl, linewidth=lw)
    if data[i] == 1:
        fsk_signal = np.sin(2 * pi * f1 * t)  # Frequency for '1'
    else:
        fsk_signal = np.sin(2 * pi * f0 * t)  # Frequency for '0'
    ax2.plot(t + z, fsk_signal, linewidth=lw)
ax1.set_ylabel('Carrier signal', fontsize=fs)
ax1.grid()
ax2.set_ylabel('FSK signal', fontsize=fs)
ax2.grid()
plt.tight_layout()  # Adjust layout to prevent overlap
plt.show()
```

EXP 2 HOFFMAN CODE

```
import heapq
def build_huffman_tree(probabilities):
    heap = [[weight, [symbol, ""]] for symbol, weight in probabilities.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])  # Corrected line
    return sorted(heapq.heappop(heap)[1:], key=lambda p: (len(p[-1]), p))
def main():
    n = int(input("Enter the number of symbols: "))
```

```
    probabilities = {}
    for i in range(n):
        symbol = input(f"Enter symbol {i+1}: ")
        probability = float(input(f"Enter probability of {symbol}: "))
        probabilities[symbol] = probability
    huffman_tree = build_huffman_tree(probabilities)
    print("\nHuffman Codes:")
    for symbol, code in huffman_tree:
        print(f"Symbol: {symbol}, Code: {code}")
if __name__ == "__main__":
    main()
```

EXP 1 SHANNON FANO CODE

```
import math
# Get the symbols from the user
symbols = input("Enter the symbols (space-separated): ").split()
# Get the corresponding probabilities from the user
probs = input("Enter the probabilities (space-separated, e.g., '0.5 0.25 ...'): ").split()
# Convert the probabilities to floats and assign them to the symbols
symbol_probs = {}
for i in range(len(symbols)):
    symbol_probs[symbols[i]] = float(probs[i])
# Sort symbols by probabilities in descending order
sorted_symbols = sorted(symbol_probs.keys(), key=lambda x: symbol_probs[x], reverse=True)
# Shannon-Fano encoding function
def shannon_fano_encode(symbols):
    if len(symbols) == 1:
        return {symbols[0]: ''}

    split = len(symbols) // 2
```

```python
    left = shannon_fano_encode(symbols[:split])
    right = shannon_fano_encode(symbols[split:])

    for s in left:
        left[s] = '0' + left[s]
    for s in right:
        right[s] = '1' + right[s]

    return {**left, **right}
# Calculate entropy
entropy = sum([-symbol_probs[s] * math.log2(symbol_probs[s]) for s in symbol_probs])
# Shannon-Fano coding
shannon_fano_code = shannon_fano_encode(sorted_symbols)
# Print results
print("\nSymbol Probabilities:")
for symbol, prob in symbol_probs.items():
    print(f"{symbol}: {prob:.3f}")
print("\nShannon Fano Codes:")
for symbol, code in shannon_fano_code.items():
    print(f"{symbol}: {code}")
print("\nEntropy:", entropy)
```