Communication Channel Report

James Henrik Middleton (2785209M)

December 2025

```python
def read_text(path):
    with open(path, "r", encoding="utf-8") as f:
        s = f.read()

    data = s.encode("utf-8")  # convert to real bytes

    b = ''.join(format(byte, '08b') for byte in data)
    vector = [int(bit) for bit in b]

    return vector


def modulate_bask(vector, samples_per_bit=1000, freq=1):
    t = np.linspace(0, 1, samples_per_bit)          # time for one bit
    carrier = np.sin(2 * np.pi * freq * t)          # one-bit carrier

    full_carrier = np.tile(carrier, len(vector))    # repeat carrier for each bit
    modulation_signal = np.repeat(vector, samples_per_bit)

    modulated = full_carrier * modulation_signal

    # Build full time axis for entire signal
    total_time = np.linspace(0, len(vector), len(modulated))

    return total_time, modulated


def demodulate_bask(modulated_signal, samples_per_bit, threshold=0.4):
    # Threshold => threshold separating rounding for 0 and 1
    num_bits = len(modulated_signal) // samples_per_bit
    recovered = []

    for i in range(num_bits):
        block = modulated_signal[i*samples_per_bit : (i+1)*samples_per_bit]
        amplitude = np.mean(np.abs(block))           # envelope detection
        recovered.append(1 if amplitude > threshold else 0)

    return recovered
```

```python
import numpy as np
```

```python
import numpy as np

def fir_bandpass_filter(lowcut, highcut, fs, numtaps=501):
    # Normalize frequencies (0..0.5)
    f1 = lowcut / fs
    f2 = highcut / fs

    # Filter tap index centered at 0
    n = np.arange(numtaps) - (numtaps - 1) / 2

    # Bandpass = High LPF minus Low LPF
    h = 2*f2 * np.sinc(2*f2*n) - 2*f1 * np.sinc(2*f1*n)

    # Apply Hamming window
    window = np.hamming(numtaps)
    h *= window

    # Normalize to unity gain at passband center
    # Calculate frequency response at center frequency
    center_freq = (f1 + f2) / 2
    omega = 2 * np.pi * center_freq
    freq_response = np.sum(h * np.exp(-1j * omega * np.arange(numtaps)))
    h /= np.abs(freq_response)

    return h


def apply_fir(signal, filt):
    """Apply FIR filter via convolution (no SciPy)."""
    return np.convolve(signal, filt, mode='same')
```

```
def bits_to_text(bits):
    if len(bits) % 8 != 0:
        raise ValueError("Binary length is not a multiple of 8")

    # group into bytes
    byte_values = [
        int(''.join(str(bit) for bit in bits[i:i+8]), 2)
        for i in range(0, len(bits), 8)
    ]

    data = bytes(byte_values)
    return data.decode("utf-8")


def plot_bits(time, modulated_signal, samples_per_bit, bits=5):
    # Plot the first N bits of the modulated signal.
    N = bits * samples_per_bit

    plt.figure(figsize=(10,4))
    plt.plot(time[:N], modulated_signal[:N])
    plt.title(f"First {bits} BASK-modulated bits")
    plt.xlabel("Time")
    plt.ylabel("Amplitude")
    plt.grid(True)
    plt.show()


# Define the SNR values you want to test
snr_values = [24, 18, 12, 8, 6]
# Define Original Data as a vector of Binary digits
vector = read_text("/home/james/University/Communication_Systems/Communication_Channel_Report/Input.txt")
# Modulate a sin wave using this data
time, modulated = modulate_bask(vector, samples_per_bit=1000, freq=5)
# Simulate Noise in the signal
noisy_signals = snr(snr_values, modulated)
# Recovered Data
recovered = demodulate_bask(modulated, samples_per_bit=1000)
recovered_text = bits_to_text(recovered)

# Error Checking
print("Original bits:", len(vector))
print("Recovered bits:", len(recovered))
print("Difference:", len(recovered) - len(vector))
print("Original first 40 bits:", vector[:40])
print("Recovered first 40 bits:", recovered[:40])
errors = sum(1 for a,b in zip(vector, recovered) if a!=b)
print("Total bit errors:", errors)

# Display Results
print("Recovered text:", recovered_text)
plot_bits(time, modulated, samples_per_bit=1000, bits=100)

# Display Noisy signals
num_plots = len(noisy_signals)
plt.figure(figsize=(12, 3 * num_plots))

for i, (snr, noisy) in enumerate(noisy_signals.items(), start=1):
    plt.subplot(num_plots, 1, i)
    plt.plot(time[:50000], noisy[:50000])
    plt.title(f"BASK Signal with AWGN (SNR = {snr} dB)")
    plt.xlabel("Time")
    plt.ylabel("Amplitude")
    plt.grid(True)

plt.tight_layout()
plt.show()


fs = 1000      # sampling rate
fc = 5         # carrier frequency

lowcut = 4.5       # narrow band around fc (20% of original BW)
highcut = 5.5

# Design the filter
bp_filter = fir_bandpass_filter(lowcut, highcut, fs, numtaps=801)

# Display Filtered Noisy Signals
num_plots = len(noisy_signals)
plt.figure(figsize=(12, 3 * num_plots))

for i, (snr, noisy) in enumerate(noisy_signals.items(), start=1):
    # Apply the filter to your BASK signal
    filtered_signal = apply_fir(noisy, bp_filter)
    plt.subplot(num_plots, 1, i)
    plt.plot(time[:50000], noisy[:50000], label="Noisy")
    plt.plot(time[:50000], filtered_signal[:50000], label="Filtered", linewidth=2)

    plt.grid(True)
    plt.legend()
    plt.title(f"Bandpass Filter (SNR = {snr} dB)")
```

```
plt.tight_layout()
plt.show()


import random
import numpy as np

def gen_rand_vector():
    # 1. Generate random 100-bit sequence
    bit_sequence = [random.randint(0,1) for _ in range(100)]
    samples_per_bit = 10
    # 2. Expand each bit into samples, convert to float
    vector = np.array([float(bit) for bit in bit_sequence for _ in range(samples_per_bit)])
    return vector

# 3. Compute RMS voltage of the waveform
def vrms(waveform):
    v_rms_carrier = np.sqrt(np.mean(waveform**2))
    print(f"Signal RMS voltage = {v_rms_carrier:.5f}")
    return v_rms_carrier

# Function to add AWGN at given SNR_dB and rescale the signal
def add_awgn_and_rescale(carrier, snr_db, v_rms_carrier):
    # compute noise RMS
    v_rms_noise = v_rms_carrier / (10**(snr_db / 20.0))
    print(f"For SNR={snr_db} dB → Noise RMS = {v_rms_noise:.5f}")
    # generate noise (Gaussian with zero mean)
    noise = np.random.randn(len(carrier)) * v_rms_noise
    # add noise to signal
    noisy = carrier + noise
    # rescale so max(|value|) = 1
    max_abs = np.max(np.abs(noisy))
    if max_abs == 0:
        rescaled = noisy
    else:
        rescaled = noisy / max_abs
    return rescaled

# Function to just add AWGN at given SNR_dB
def add_awgn(carrier, snr_db, v_rms_carrier):
    v_rms_noise = v_rms_carrier / (10**(snr_db / 20.0))
    print(f"For SNR={snr_db} dB → Noise RMS = {v_rms_noise:.5f}")
    noise = np.random.randn(len(carrier)) * v_rms_noise
    noisy = carrier + noise
    return noisy  # no rescale here


# 5. Generate noisy waveforms & save CSVs
def snr(snr_values, vector):
    if(not snr_values):
        raise FileNotFoundError("SNR Values not found. Abort.")

    # Compute RMS voltage of the waveform
    v_rms_carrier = vrms(vector)

    noisy_waves = {}
    for snr in snr_values:
        noisy_wave = add_awgn(vector, snr, v_rms_carrier)
        # fname = f"waveform_SNR_{snr}dB.csv"
        # with open(fname, "w", newline="") as f:
        #     writer = csv.writer(f)
        #     for v in noisy_wave:
        #         writer.writerow([v])
        # print(f"Saved {fname}")
        noisy_waves[snr] = noisy_wave
    return noisy_waves


# Define SNR values
snr_values = [24, 18, 12, 8, 6]
# Define vector
vector = gen_rand_vector()
# Generate noisy waveforms & save CSVs
noisy_waves = snr(snr_values, vector)
```