```
# Python simulation of a simplified DMT (OFDM) system with convolutional coding + interleaving
# Run this cell in a Jupyter notebook.

import numpy as np
import matplotlib.pyplot as plt
from math import log10
import pandas as pd

# -- Utility: convolutional encoder (rate 1/2, constraint length 7, polynomials 171 and 133 octal) --
def conv_encode(bits):
    g1 = 0o171
    g2 = 0o133
    K = 7
    state = 0
    encoded = []
    for b in bits:
        state = ((state << 1) | int(b)) & ((1 << K) - 1)
        out1 = bin(state & g1).count("1") % 2
        out2 = bin(state & g2).count("1") % 2
        encoded.append(out1)
        encoded.append(out2)
    return np.array(encoded, dtype=np.int8)

# -- Utility: hard-decision Viterbi decoder for the above convolutional code --
def viterbi_decode(rx_bits):
    g1 = 0o171
    g2 = 0o133
    K = 7
    n_states = 1 << (K-1)
```

```
next_state = np.zeros((n_states,2), dtype=int)
out_bits = np.zeros((n_states,2,2), dtype=int)
for s in range(n_states):
    for b in [0,1]:
        st = ((s << 1) | b) & ((1 << (K-1)) - 1)
        full = ((s << 1) | b)
        o1 = bin(((full) & g1)).count("1") % 2
        o2 = bin(((full) & g2)).count("1") % 2
        next_state[s,b] = st
        out_bits[s,b,0] = o1
        out_bits[s,b,1] = o2

L = len(rx_bits) // 2
big = 10**9
metrics = np.full(n_states, big, dtype=np.int32)
metrics[0] = 0
survivors = np.zeros((L, n_states), dtype=np.int32)

for t in range(L):
    rx0 = rx_bits[2*t]
    rx1 = rx_bits[2*t+1]
    new_metrics = np.full(n_states, big, dtype=np.int32)
    new_survivors = np.zeros(n_states, dtype=np.int32)
    for s in range(n_states):
        if metrics[s] >= big:
            continue
        for b in [0,1]:
            ns = next_state[s,b]
            o1 = out_bits[s,b,0]
```

```
            o2 = out_bits[s,b,1]
            bm = (o1 != rx0) + (o2 != rx1)
            metric = metrics[s] + bm
            if metric < new_metrics[ns]:
                new_metrics[ns] = metric
                new_survivors[ns] = s | (b<< (K-1))
        metrics = new_metrics
        survivors[t,:] = new_survivors

    state = np.argmin(metrics)
    decoded_bits = []
    for t in range(L-1, -1, -1):
        info = survivors[t, state]
        prev_state = info & ((1 << (K-1)) - 1)
        input_bit = (info >> (K-1)) & 1
        decoded_bits.append(input_bit)
        state = prev_state
    decoded_bits.reverse()
    return np.array(decoded_bits, dtype=np.int8)

# -- Interleaver: simple block interleaver for burst resilience --
def block_interleave(bits, rows):
    cols = len(bits) // rows
    mat = bits.reshape((rows, cols))
    return mat.T.flatten()

def block_deinterleave(bits, rows):
    cols = len(bits) // rows
    mat = bits.reshape((cols, rows)).T
```

```
    return mat.flatten()

# -- QPSK mapping/demapping --
def qpsk_mod(bits):
    bits = bits.reshape(-1,2)
    mapping = {
        (0,0): 1+1j,
        (0,1): -1+1j,
        (1,1): -1-1j,
        (1,0): 1-1j
    }
    sym = np.array([mapping[tuple(b)] for b in bits]) / np.sqrt(2)
    return sym

def qpsk_demod(symbols):
    bits = np.zeros((len(symbols),2), dtype=np.int8)
    bits[:,0] = (np.real(symbols) < 0).astype(np.int8)
    bits[:,1] = (np.imag(symbols) < 0).astype(np.int8)
    return bits.flatten()

# -- OFDM (DMT) functions: IFFT/FFT, add CP --
def ofdm_modulate(symbols, N, cp_len):
    M = len(symbols) // N
    out = []
    idx = 0
    for m in range(M):
        X = np.zeros(N, dtype=complex)
        X[:] = symbols[idx:idx+N]
        idx += N
```

```
        x = np.fft.ifft(X)
        x_cp = np.concatenate([x[-cp_len:], x])
        out.append(x_cp)
    return np.concatenate(out)

def ofdm_demodulate(signal, N, cp_len):
    sym_per_ofdm = N
    ofdm_len = N + cp_len
    M = len(signal) // ofdm_len
    symbols = []
    for m in range(M):
        x_cp = signal[m*ofdm_len:(m+1)*ofdm_len]
        x = x_cp[cp_len:]
        X = np.fft.fft(x)
        symbols.append(X)
    return np.concatenate(symbols)

# -- Channel: AWGN + impulse noise --
def add_awgn(signal, snr_db):
    sig_pow = np.mean(np.abs(signal)**2)
    snr = 10**(snr_db/10.0)
    noise_pow = sig_pow / snr
    noise = np.sqrt(noise_pow/2) * (np.random.randn(*signal.shape) +
1j*np.random.randn(*signal.shape))
    return signal + noise

def add_impulse_noise(signal, p_impulse=0.001, amp=10.0):
    noisy = signal.copy()
    n = len(signal)
```

```
    mask = np.random.rand(n) < p_impulse
    noisy[mask] += amp * (np.random.randn(np.sum(mask)) + 1j*np.random.randn(np.sum(mask)))
    return noisy

# -- Simulation parameters --
np.random.seed(0)
N = 64
cp = 16
ofdm_symbols = 100
M_qpsk = 2
bits_per_ofdm = N * M_qpsk
total_bits = bits_per_ofdm * ofdm_symbols

interleaver_rows = 8
p_impulse = 0.002
impulse_amp = 20.0

EbN0_dBs = np.array([0, 2, 4, 6, 8, 10])
ber_uncoded = []
ber_coded = []

code_rate = 1/2
for EbN0_db in EbN0_dBs:
    EsN0_uncoded_db = EbN0_db + 10*np.log10(1.0 / M_qpsk)
    EsN0_db = EbN0_db + 10*np.log10(code_rate / M_qpsk)

    # UNCODED
    bits = np.random.randint(0,2,total_bits)
    symbols = qpsk_mod(bits)
```

```
tx_signal = ofdm_modulate(symbols, N, cp)
tx_signal = add_impulse_noise(tx_signal, p_impulse=p_impulse, amp=impulse_amp)
rx_signal = add_awgn(tx_signal, EsN0_uncoded_db)
rx_symbols = ofdm_demodulate(rx_signal, N, cp)
rx_bits_hat = qpsk_demod(rx_symbols)
ber_u = np.mean(bits != rx_bits_hat[:len(bits)])
ber_uncoded.append(ber_u)

# CODED
bits = np.random.randint(0,2,total_bits)
encoded = conv_encode(bits)
if len(encoded) % interleaver_rows != 0:
    pad = interleaver_rows - (len(encoded) % interleaver_rows)
    encoded = np.concatenate([encoded, np.zeros(pad, dtype=np.int8)])
inter = block_interleave(encoded, interleaver_rows)
sym = qpsk_mod(inter)
tx_signal = ofdm_modulate(sym, N, cp)
tx_signal = add_impulse_noise(tx_signal, p_impulse=p_impulse, amp=impulse_amp)
rx_signal = add_awgn(tx_signal, EsN0_db)
rx_symbols = ofdm_demodulate(rx_signal, N, cp)
rx_bits_hard = qpsk_demod(rx_symbols)
if len(rx_bits_hard) % interleaver_rows != 0:
    pad = interleaver_rows - (len(rx_bits_hard) % interleaver_rows)
    rx_bits_hard = np.concatenate([rx_bits_hard, np.zeros(pad, dtype=np.int8)])
deinter = block_deinterleave(rx_bits_hard, interleaver_rows)
deinter = deinter[:len(encoded)]
decoded = viterbi_decode(deinter)
decoded = decoded[:len(bits)]
ber_c = np.mean(bits != decoded)
```

```
    ber_coded.append(ber_c)
    print(f"Eb/N0={EbN0_db} dB -> Uncoded BER={ber_u:.4e}, Coded BER={ber_c:.4e}")

# Plot BER curves
plt.figure()
plt.semilogy(EbN0_dBs, ber_uncoded, marker='o', label='Uncoded OFDM (QPSK)')
plt.semilogy(EbN0_dBs, ber_coded, marker='s', label='Conv-coded (R=1/2) + Interleaver')
plt.xlabel('Eb/N0 (dB)')
plt.ylabel('BER (log scale)')
plt.title('BER vs Eb/N0 — simple DMT/OFDM simulation (with impulse noise)')
plt.grid(True)
plt.legend()
plt.show()

# Show numeric table
df = pd.DataFrame({
    'Eb/N0 (dB)': EbN0_dBs,
    'BER Uncoded': ber_uncoded,
    'BER Coded': ber_coded
})
try:
    import caas_jupyter_tools as cjt
    cjt.display_dataframe_to_user("BER results", df)
except Exception:
    print("\nDataFrame:\n", df)
print("\nSimulation summary:")
print(df.to_string(index=False))
```