https://ideone.com/NIqeh6

#6 Write a program to implement a CRC (Cyclic Redundancy Code) error detection model.

```
def xor(a, b):
    # Perform XOR operation
    result = []
    for i in range(1, len(b)):
        if a[i] == b[i]:
            result.append('0')
        else:
            result.append('1')
    return ''.join(result)

def mod2div(dividend, divisor):
    # Perform Modulo-2 division
    pick = len(divisor)
    tmp = dividend[:pick]

    while pick < len(dividend):
        if tmp[0] == '1':
            tmp = xor(divisor, tmp) + dividend[pick]
        else:
            tmp = xor('0'*pick, tmp) + dividend[pick]

        pick += 1

    if tmp[0] == '1':
        tmp = xor(divisor, tmp)
    else:
        tmp = xor('0'*pick, tmp)

    return tmp

def encode_data(data, key):
    # Append zeros to the data
    l_key = len(key)
    appended_data = data + '0'*(l_key-1)
    remainder = mod2div(appended_data, key)

    # Append the remainder to the original data
    codeword = data + remainder
    return codeword
```

```
def decode_data(codeword, key):
    remainder = mod2div(codeword, key)
    return remainder

# Example usage
data = "11010011101100"
key = "1011"

print("Original Data: ", data)
codeword = encode_data(data, key)
print("Encoded Data: ", codeword)

# Simulating transmission with error (no error introduced in this case)
received_codeword = codeword
print("Received Data: ", received_codeword)

# Checking for errors
remainder = decode_data(received_codeword, key)
if '1' in remainder:
    print("Error detected in the received data.")
else:
    print("No error detected in the received data.")
```

#4 Write a program to simulate the digital-digital conversion techniques NRZ, NRZ-L, Manchester and Differential Manchester.

```
import numpy as np
import matplotlib.pyplot as plt
def NRZ_encoding(bits):
  return np.array([1 if bit=='1' else 0 for bit in bits])
def NRZL_encoding(bits):
  return np.array([1 if bit=='0' else -1 for bit in bits])
def Manchester_encoding(bits):
  result=[]
  for bit in bits:
    current_label=1
    if bit=='1':
      result.extend([-1,1])
    else:
      result.extend([1,-1])
  return result
def differential_encoding(bits):
  result=[]
```

```
  current_label=1
  for bit in bits:
    if bit=='0':
      result.extend([current_label,-current_label])
    else:
      current_label=-current_label
      result.extend([current_label,-current_label])
  return result


def plot_graph(signal,title,duration=1):
  time = np.arange(0, len(signal) * duration ,duration)
  plt.figure(figsize=(10,4))
  plt.step(time,signal,where="post")
  plt.ylabel("label")
  plt.xlabel("time")
  plt.ylim(-1.5,1.5)
  plt.title(title)
  plt.grid(True)
  plt.show()
bits="1011001"
# bits = np.random.choice(['0', '1'], size=7)
# print(bits)
nrz=NRZ_encoding(bits)
plot_graph(nrz,"NRZ Encoding")
nrzL=NRZL_encoding(bits)
plot_graph(nrzL,"NRZL Encoding")
Manchester=Manchester_encoding(bits)
plot_graph(Manchester,"MANCHESTER Encoding",.5)
DIFFERENTIAL=differential_encoding(bits)
plot_graph(DIFFERENTIAL,"DIFFERENTIAL Encoding",.5)
```

#5 Write a program for 4 x 8 block even parity error detection.

```
import numpy as np

def calculate_parity(data):
    rows, cols = data.shape
    parity_data = np.zeros((rows + 1, cols + 1), dtype=int)

    parity_data[:rows, :cols] = data
```

```
    for i in range(rows):
        parity_data[i, -1] = np.sum(data[i, :]) % 2

    for j in range(cols):
        parity_data[-1, j] = np.sum(data[:, j]) % 2

    parity_data[-1, -1] = np.sum(parity_data[:-1, :-1]) % 2

    return parity_data

def detect_error(parity_data):
    rows, cols = parity_data.shape

    # Check row parity
    row_parity_errors = []
    for i in range(rows - 1):
        if np.sum(parity_data[i, :-1]) % 2 != parity_data[i, -1]:
            row_parity_errors.append(i)

    # Check column parity
    col_parity_errors = []
    for j in range(cols - 1):
        if np.sum(parity_data[:-1, j]) % 2 != parity_data[-1, j]:
            col_parity_errors.append(j)

    if row_parity_errors or col_parity_errors:
        print(f"Row parity errors at rows: {row_parity_errors}")
        print(f"Column parity errors at columns: {col_parity_errors}")
    else:
        print("No errors detected")

def print_block(data):
    for row in data:
        print(" ".join(str(bit) for bit in row))
    print()

# Example usage
data = np.array([
    [1, 0, 1, 1, 0, 0, 1, 0],
    [0, 1, 1, 0, 1, 1, 0, 1],
    [1, 1, 0, 0, 1, 0, 1, 1],
    [0, 0, 1, 1, 0, 1, 0, 0]
], dtype=int)

print("Original Data:")
print_block(data)

# Calculate and print parity data
```

```
parity_data = calculate_parity(data)
print("Parity Data:")
print_block(parity_data)

# Introduce an error for testing
parity_data[1, 2] ^= 1  # Flip a bit to introduce an error
print("Parity Data with Error:")
print_block(parity_data)

# Detect and print errors
detect_error(parity_data)
```

 # Simulate the analog-digital signal using Pulse Code Modulation (PCM)

```
sample_rate = 1000
time_vector = np.linspace(0, 1, sample_rate)
signal_freq = 10  # Hz
signal_amp = 1
continuous_analog_signal = signal_amp * np.sin(2 * np.pi * signal_freq * time_vector)

plt.figure(figsize=(14, 6))
plt.plot(time_vector, continuous_analog_signal, label='Continuous Signal')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.title('Continuous Analog Signal')
plt.legend()
plt.grid(True)
plt.show()
```

#2 Write a program to simulate the digital-digital conversion techniques (Line Coding and Decoding, Block Coding and Decoding Schemes)

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

# Line Coding and Decoding
def nrz_encoding(bits):
   return np.array([1 if bit == '1' else 0 for bit in bits])

def nrz_decoding(encoded):
   return ''.join(['1' if bit == 1 else '0' for bit in encoded])
```

```
def nrz_i_encoding(bits):
    signal = []
    current_level = 0
    for bit in bits:
        if bit == '1':
            current_level = 1 - current_level  # Flip level on 1
        signal.append(current_level)
    return np.array(signal)

def nrz_i_decoding(encoded):
    bits = []
    current_level = 0
    for level in encoded:
        if level == current_level:
            bits.append('0')
        else:
            bits.append('1')
            current_level = level
    return ''.join(bits)

def manchester_encoding(bits):
    signal = []
    for bit in bits:
        if bit == '1':
            signal.extend([1, 0])
        else:
            signal.extend([0, 1])
    return np.array(signal)

def manchester_decoding(encoded):
    bits = []
    for i in range(0, len(encoded), 2):
        if tuple(encoded[i:i+2]) == (1, 0):
            bits.append('1')
        elif tuple(encoded[i:i+2]) == (0, 1):
            bits.append('0')
        else:
            raise ValueError(f"Invalid Manchester encoded sequence: {encoded[i:i+2]}")
    return ''.join(bits)

# Block Coding and Decoding (4B/5B)
def fourb_fiveb_encoding(bits):
    mapping = {
        '0000': '11110', '0001': '01001', '0010': '10100', '0011': '10101',
        '0100': '01010', '0101': '01011', '0110': '01110', '0111': '01111',
        '1000': '10010', '1001': '10011', '1010': '10110', '1011': '10111',
        '1100': '11010', '1101': '11011', '1110': '11100', '1111': '11101'
```

```
    }
    # Pad the input bits to make it a multiple of 4
    padded_bits = bits + '0' * ((4 - len(bits) % 4) % 4)
    encoded = ''
    for i in range(0, len(padded_bits), 4):
        nibble = padded_bits[i:i+4]
        encoded += mapping[nibble]
    return encoded

def fourb_fiveb_decoding(encoded):
    reverse_mapping = {
        '11110': '0000', '01001': '0001', '10100': '0010', '10101': '0011',
        '01010': '0100', '01011': '0101', '01110': '0110', '01111': '0111',
        '10010': '1000', '10011': '1001', '10110': '1010', '10111': '1011',
        '11010': '1100', '11011': '1101', '11100': '1110', '11101': '1111'
    }
    decoded = ''
    for i in range(0, len(encoded), 5):
        quintet = encoded[i:i+5]
        decoded += reverse_mapping[quintet]
    return decoded

def plot_signal(signal, title):
    plt.figure(figsize=(12, 2))
    plt.step(range(len(signal)), signal, where='post')
    plt.ylim(-0.5, 1.5)
    plt.title(title)
    plt.xlabel('Bit Index')
    plt.ylabel('Level')
    plt.grid(True)
    plt.show()

# Example usage
bits = "1101011011"

# NRZ-L Encoding and Decoding
nrz_l = nrz_encoding(bits)
plot_signal(nrz_l, "NRZ-L Encoding")
decoded_nrz_l = nrz_decoding(nrz_l)
print(f"NRZ-L Decoded: {decoded_nrz_l}")

# NRZ-I Encoding and Decoding
nrz_i = nrz_i_encoding(bits)
plot_signal(nrz_i, "NRZ-I Encoding")
decoded_nrz_i = nrz_i_decoding(nrz_i)
print(f"NRZ-I Decoded: {decoded_nrz_i}")

# Manchester Encoding and Decoding
```

```
manchester = manchester_encoding(bits)
plot_signal(manchester, "Manchester Encoding")
decoded_manchester = manchester_decoding(manchester)
print(f"Manchester Decoded: {decoded_manchester}")

# 4B/5B Encoding and Decoding
fourb_fiveb = fourb_fiveb_encoding(bits)
print(f"4B/5B Encoded: {fourb_fiveb}")
decoded_fourb_fiveb = fourb_fiveb_decoding(fourb_fiveb)
print(f"4B/5B Decoded: {decoded_fourb_fiveb}")
```

#3 Write a program to simulate the digital-analog conversion techniques (ASK, FSK, PSK)

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


###### Digital Signal #######
num_bits = 8
bit_duration = 0.1
sampling_rate = 1000
amplitude = 1


digital_data = np.random.choice([0, 1], num_bits)


time_intervals = np.linspace(0, num_bits * bit_duration, num_bits * sampling_rate // 10)


upsampled_data = np.repeat(digital_data, sampling_rate // 10)


plt.figure(figsize=(10, 4))
plt.step(time_intervals, upsampled_data, where='post', label='Digital Signal')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.title('Generated Digital Signal')
plt.ylim(-0.5, 1.5)
plt.grid(True)
plt.legend()
plt.show()
```

```
###### ASK #######

carrier_freq = 10  # Hz

# Carrier wave (sine wave with a given frequency and amplitude)
carrier_wave = amplitude * np.sin(2 * np.pi * carrier_freq * time_intervals)


ask_modulation = np.zeros_like(time_intervals)


# Modulate the amplitude based on the upsampled data
for i, bit in enumerate(upsampled_data):
    if bit == 1:
        ask_modulation[i] = 1 * carrier_wave[i]  # High amplitude for '1'
    else:
        ask_modulation[i] = 0 * carrier_wave[i]


# Plot the ASK signal
plt.figure(figsize=(10, 4))
plt.plot(time_intervals, ask_modulation, label='ASK Signal')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.title('Amplitude Shift Keying (ASK) Signal')
plt.grid(True)
plt.legend()
plt.show()






###### FSK #######

freq_0 = 5
freq_1 = 15


fsk_modulation = np.zeros_like(time_intervals)
for i in range(num_bits):
    if digital_data[i] == 0:
```

```
        fsk_modulation[i * sampling_rate // 10 : (i + 1) * sampling_rate // 10] = amplitude *
np.sin(2 * np.pi * freq_0 * time_intervals[i * sampling_rate // 10 : (i + 1) * sampling_rate // 10])
    else:
        fsk_modulation[i * sampling_rate // 10 : (i + 1) * sampling_rate // 10] = amplitude *
np.sin(2 * np.pi * freq_1 * time_intervals[i * sampling_rate // 10 : (i + 1) * sampling_rate // 10])


plt.figure(figsize=(10, 4))
plt.plot(time_intervals, fsk_modulation, label='FSK Signal')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.title('Frequency Shift Keying (FSK) Signal')
plt.grid(True)
plt.legend()
plt.show()




###### PSK #######

carrier_freq = 10  # Hz
carrier_wave = amplitude * np.sin(2 * np.pi * carrier_freq * time_intervals)


ask_modulation = (upsampled_data * 2 - 1) * carrier_wave


plt.figure(figsize=(10, 4))
plt.plot(time_intervals, ask_modulation, label='PSK Signal')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.title('Phase Shift Keying (PSK) Signal')
plt.grid(True)
plt.legend()
plt.show()
```