Data Communications

Semester Project

Seshal Jain, 191112436, CSE3

Contents

1 Assignment 1

1.1 Sine wave of given frequency and time period

A sine wave of frequency f can be represented by

$$y = \sin(2\pi ft)$$

```
import numpy as np
import matplotlib.pyplot as plot
import ipywidgets as widgets

def plot_sin_wave(freq, time_period):
    time = np.arange(time_period[0], time_period[1], 0.01)
    amplitude = np.sin(2 * np.pi * freq * time)

    plot.figure(figsize=(15, 4))
    plot.title('Sine wave')
    plot.xlabel('Time')
    plot.ylabel('Amplitude')
    plot.grid(True, which='both')
    plot.axhline(y=0, color='k')
    plot.plot(time, amplitude)

tp = widgets.IntRangeSlider(
    value=(5, 15),
    min=0, max=20, step=1,
    description='Time Period'
)

f = widgets.FloatSlider(
    value=1,
    min=1, max=5, step=1,
```

```
    description='Frequency'
)

widgets.interact(plot_sin_wave, freq=f, time_period=tp)
```

The generated wave is a sine wave for the input frequency and time period. The wave gets more populated as the frequency increases, and time period spaces out the wave.

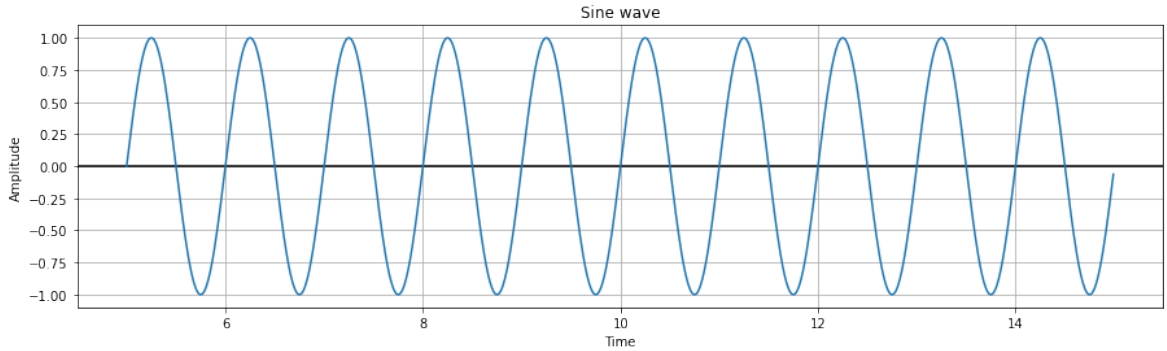

Figure 1: Sine wave of frequency 1 and time period [5, 10]

1.2 Square wave from sine waves

Using the Fourier Series, a function can be represented as a sum of sine and cosine functions, over a given time period.

According to the Fourier Series,

$$f(x) = \frac{A_0}{2} + \sum_{n=1}^{\infty} A_n \cos(nx) + B_n \sin(nx)$$

where

$$A_0 = \frac{1}{\pi} \int_0^{2\pi} f(x)dx$$
$$A_n = \frac{1}{\pi} \int_0^{2\pi} f(x)\cos(nx)dx$$
$$B_n = \frac{1}{\pi} \int_0^{2\pi} f(x)\sin(nx)dx$$

Calculating the Fourier Series for a square wave,

$$f(x) = \begin{cases} 1 & 0 \leq x \leq \pi \\ -1 & \pi x \leq 2\pi \end{cases}$$

Which yields the values:

$$A_0 = 0$$
$$A_n = 0$$
$$B_n = \begin{cases} 0 & \text{when n is even} \\ \frac{4}{n\pi} & \text{when n is odd} \end{cases}$$

Using these values,

$$f(x) = \frac{4}{\pi} \sum_{n=1}^{\infty} \frac{\sin((2n-1)\pi x)}{(2n-1)}$$

```
import numpy as np
import matplotlib.pyplot as plot
import ipywidgets as widgets

def plot_square_wave(n):
    time = np.arange(0, 10, 0.01)
    final_amplitude = np.sin(2 * np.pi * time)

    plot.figure(figsize=(15,4))
    plot.title('Square wave')
    plot.xlabel('Time')
    plot.ylabel('Amplitude')
    plot.grid(True, which='both')
    plot.axhline(y=0, color='k')

    for i in range(3, n + 1, 2):
        amplitude = np.sin(2 * np.pi * i * time) / i
        plot.plot(time, amplitude, ',')
        final_amplitude += amplitude

    plot.plot(time, final_amplitude, color='k')

n = widgets.IntSlider(
    value=5,
    min=1, max=15, step=2,
    description='Harmonics'
)

widgets.interact(plot_square_wave, n=n)
```

The series of odd harmonics leads to a generated wave similar in shape to that of an ideal square wave. As we **increase the number of harmonics, we get closer to an ideal square wave**, but because of the infinite nature of the Fourier series, we never get a perfect square wave.

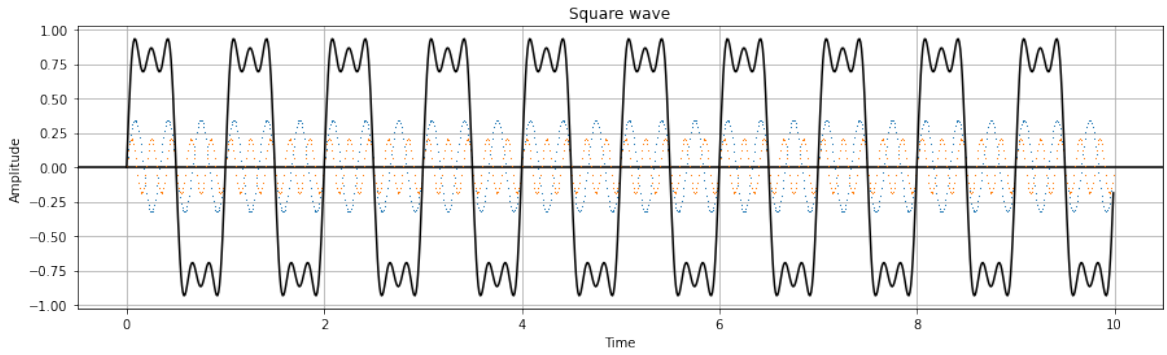

Figure 2: Square wave developed from 5 harmonics

2 Assignment 2

2.1 NRZ-L Encoding

NRZ-L Encoding is a line encoding technique, specifically a serializer line code used to send information bitwise. Conventionally, `1` is represented by one physical level `-1`, while `0` is represented by another level `1`.

In bipolar NRZ-L encoding, the signal essentially *swings* from one level to another.

```
def plot_nrzl(s, noise, freq):
    sampling_freq = freq // BITRATE

    noise_signal = np.random.normal(0, noise, sampling_freq * len(s))

    input_signal = []
    encoded_signal = []
    encoded_signal_with_noise = []
    decoded_signal = []
    decoded_signal_from_noise = []

    input_signal = list(map(int, list(s)))

    for bit in input_signal:
        if bit == 0:
            encoded_signal.append(1)
        else:
            encoded_signal.append(-1)

    encoded_signal_with_noise = [
        i for i in encoded_signal for j in range(sampling_freq)] + noise_signal

    for i in range(len(s)):
        bit_sum = 0
        for j in range(sampling_freq):
            bit_sum += encoded_signal_with_noise[i * sampling_freq + j]
        if bit_sum // sampling_freq < 0:
            decoded_signal_from_noise.append(0)
        else:
            decoded_signal_from_noise.append(1)

    for i in range(len(s)):
        if decoded_signal_from_noise[i] == 1:
            decoded_signal.append(0)
        else:
            decoded_signal.append(1)

    bit_error_count = 0

    for i in range(len(s)):
        if input_signal[i] != decoded_signal[i]:
            bit_error_count += 1

    bit_error_rate = bit_error_count / len(s) * 100

    time = np.arange(0, sampling_freq * len(s))
```

```
47
48      (fig, axes) = plot.subplots(ncols=2, sharex=True, sharey=True,
49                                  figsize=(15, 4), squeeze=False)
50
51      axes[0][0].step(time, [bit for bit in encoded_signal for j in range(
52          sampling_freq)], 'r', label='NRZL')
53      axes[0][0].plot(time, encoded_signal_with_noise, 'g',
54                      label='NRZL w/ Noise')
55      axes[0][1].step(time, [bit for bit in decoded_signal for j in range(
56          sampling_freq)], 'b', label='Decoded signal')
57
58      axes[0][0].legend()
59      axes[0][1].legend()
60
61      print("Bit errors =", bit_error_count)
62      print("BER =", bit_error_rate)
```

2.2 NRZ-I Encoding

NRZ-I Encoding is another serialiser line encoding technique, used to send information bitwise.

The two-level NRZ-I signal distinguishes data bits by the presence or absence of a transition, meaning that a `1` is represented by a transition from the previous encoded bit, while `0` is represented by no transition.

NRZ-I encoding is used in USBs, but the opposite convention i.e. “change on 0” is used for encoding.

```
1   def plot_nrzi(s, noise, freq):
2       sampling_freq = freq // BITRATE
3
4       noise_signal = np.random.normal(0, noise, sampling_freq * len(s))
5
6       input_signal = []
7       encoded_signal = []
8       encoded_signal_with_noise = []
9       decoded_signal = []
10      decoded_signal_from_noise = []
11
12      input_signal = list(map(int, list(s)))
13
14      last_bit = 0
15
16      for bit in input_signal:
17          if bit == 1:
18              last_bit = (1 if last_bit == 0 else 0)
19          encoded_signal.append(last_bit)
20
21      encoded_signal_with_noise = [i for i in encoded_signal for j in
22                                   range(sampling_freq)] + noise_signal
23
24      for i in range(len(s)):
25          bit_sum = 0
26          for j in range(sampling_freq):
27              bit_sum += encoded_signal_with_noise[i * sampling_freq + j]
28          if bit_sum // sampling_freq < 0.5:
29              decoded_signal_from_noise.append(0)
30          else:
```

```
            decoded_signal_from_noise.append(1)

    for i in range(1, len(s)):
        if decoded_signal_from_noise[i] == decoded_signal_from_noise[i - 1]:
            decoded_signal.append(0)
        else:
            decoded_signal.append(1)

    if encoded_signal_with_noise[0] == 0:
        decoded_signal.insert(0, 0)
    else:
        decoded_signal.insert(0, 1)

    bit_error_count = 0

    for i in range(len(s)):
        if input_signal[i] != decoded_signal[i]:
            bit_error_count += 1

    bit_error_rate = bit_error_count / len(s) * 100

    time = np.arange(0, sampling_freq * len(s))

    (fig, axes) = plot.subplots(ncols=2, sharex=True, sharey=True,
                                figsize=(15, 4), squeeze=False)

    axes[0][0].step(time, [bit for bit in encoded_signal for j in
                           range(sampling_freq)], 'r', label='NRZI')
    axes[0][0].plot(time, encoded_signal_with_noise, 'g',
                    label='NRZI w/ Noise')
    axes[0][1].step(time, [bit for bit in decoded_signal for j in
                           range(sampling_freq)], 'b', label='Decoded signal')

    axes[0][0].legend()
    axes[0][1].legend()

    print("Bit errors =", bit_error_count)
    print("BER =", bit_error_rate)
```

2.3 Input Data

2.4 Plots

```
bitstring = widgets.Text(
    value="1101010010l0",
    description='Input bitstring'
)

nrzl_n = widgets.FloatSlider(
    value=2,
    min=0, max=4, step=0.25,
    description='Noise Threshold'
)

nrzi_n = widgets.FloatSlider(
```

```
    value=0.75,
    min=0, max=1.5, step=0.1,
    description='Noise Threshold'
)

s_f = widgets.IntSlider(
    value=150, step=10,
    min=20, max=200,
    description='Sampling Frequency'
)

widgets.interact(plot_input, s=bitstring, freq=s_f)
widgets.interact(plot_nrzl, s=bitstring, noise=nrzl_n, freq=s_f)
widgets.interact(plot_nrzi, s=bitstring, noise=nrzi_n, freq=s_f)
```

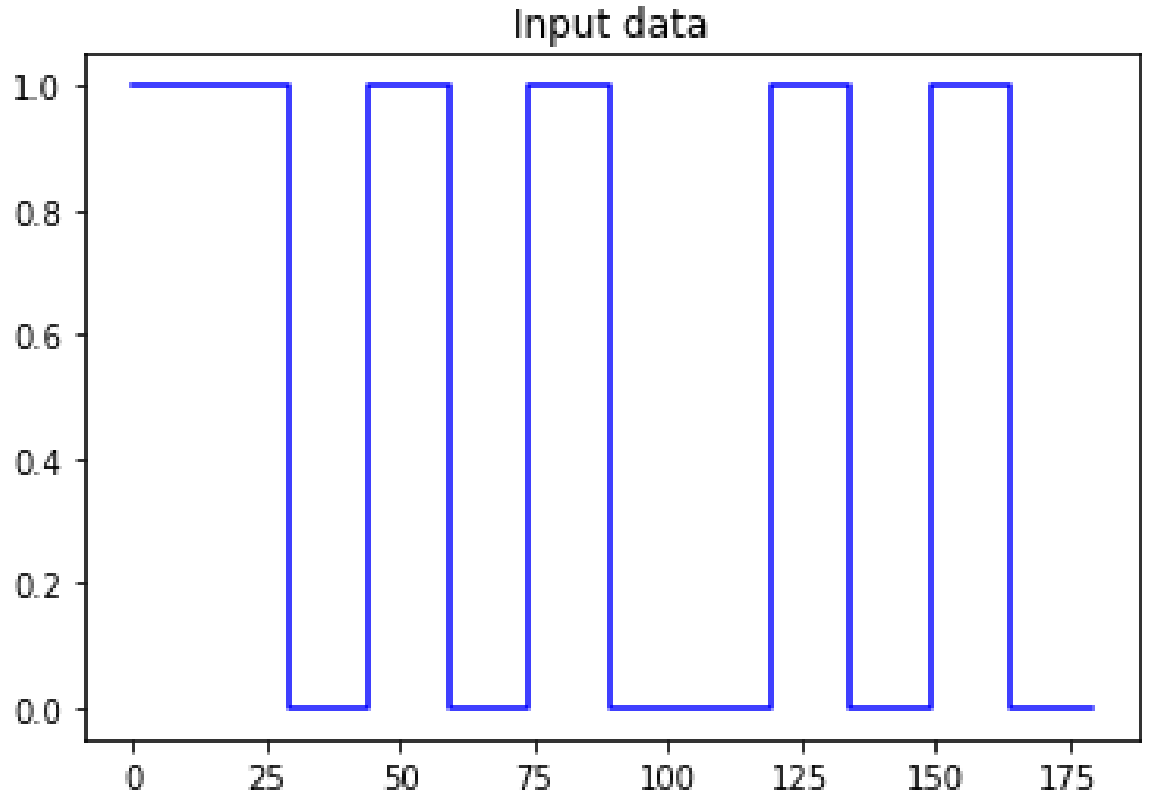

Figure 3: Input Data: 110101001010

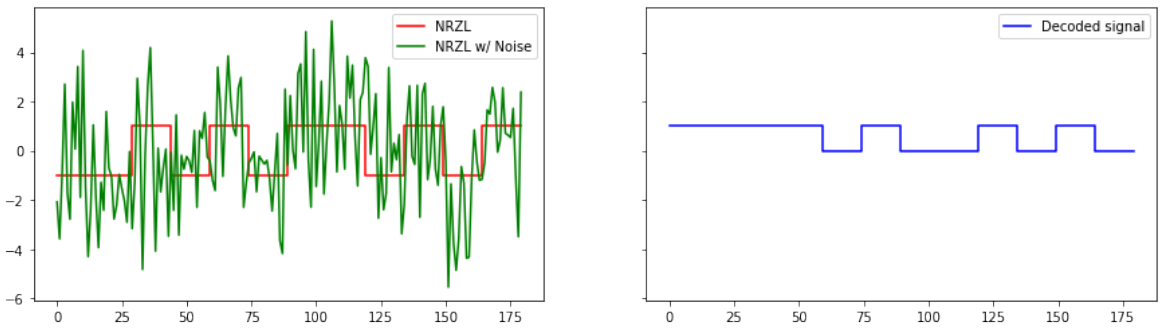

Figure 4: NRZL Encoding, BER = 8.33%

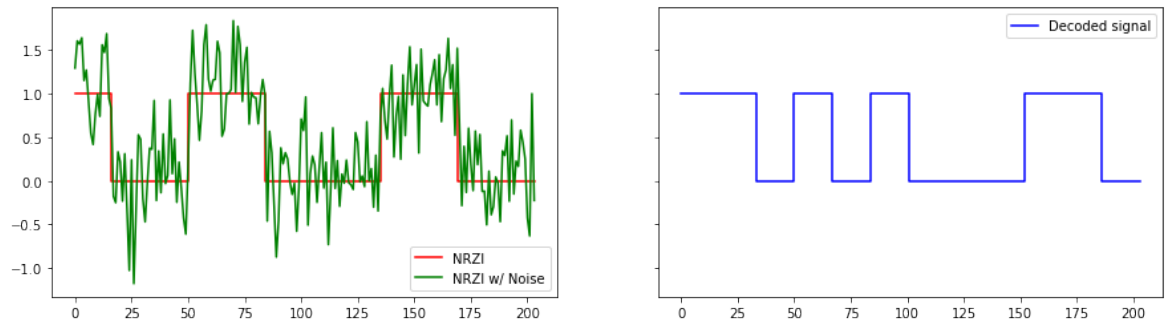

Figure 5: NRZI Encoding, BER = 16.66%

We can clearly see that both encodings require different thresholds for decoding, because NRZ-L encoding transitions from 0 to 1, requiring a threshold of 0.5, while NRZ-I required a threshold of 0.

The signal may become asynchronous without an explicit clock signal provided along with the encoding, especially in scenarios when long, unchanged bits are sent as input. Other types of encoding like the Manchester encoding overcome this problem, but require much more bandwidth for the same.

A greater noise threshold brings in more bit errors and subsequently a greater BER, while a higher sampling frequency reduces the BER.