Digital Communications Laboratory Report

Digital Modulation

by JIE TIAN

Rankine 329

January 24, 2024

I. Importing libraries and setting input data

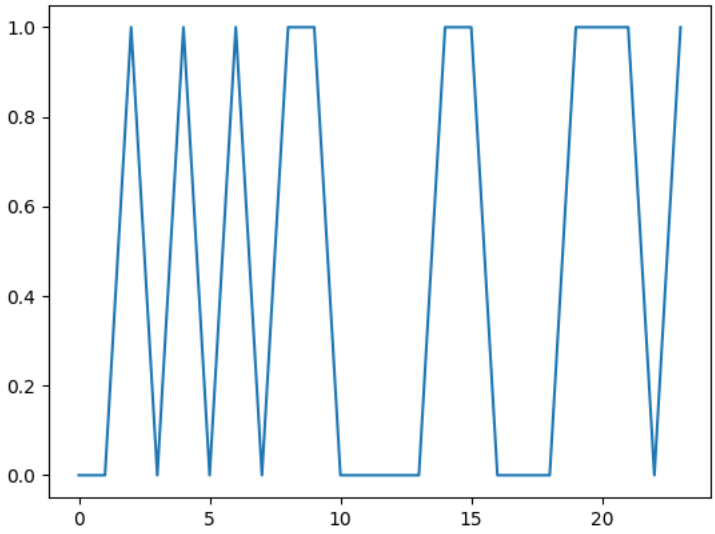
Before we start the BPSK and QPSK modulation, we need to import the necessary python libraries into the project, we can use the functions in these libraries to draw graphs, do Fast Fourier Transforms, filter, and some array operations. The code below shows the libraries we need to use in this project.

```
import numpy as np

from matplotlib import pyplot as plt

from scipy import fft

from scipy import signal
```

For the input data, we will use the 24-bit binary number to represent our student number. The following function converts our student number to a 24-bit binary number for our later modulation.

```
def bin_array(num, m):
    """Convert a positive integer num into an m-bit bit vector"""
    return np.array(list(np.binary_repr(num).zfill(m))).astype(np.bool_)

# import 24 bit digital data
id_num = 2802461
Nbits = 24
tx_bin = bin_array(id_num, Nbits)
```

The following is a 24-bit binary display of the student number: 2802461.



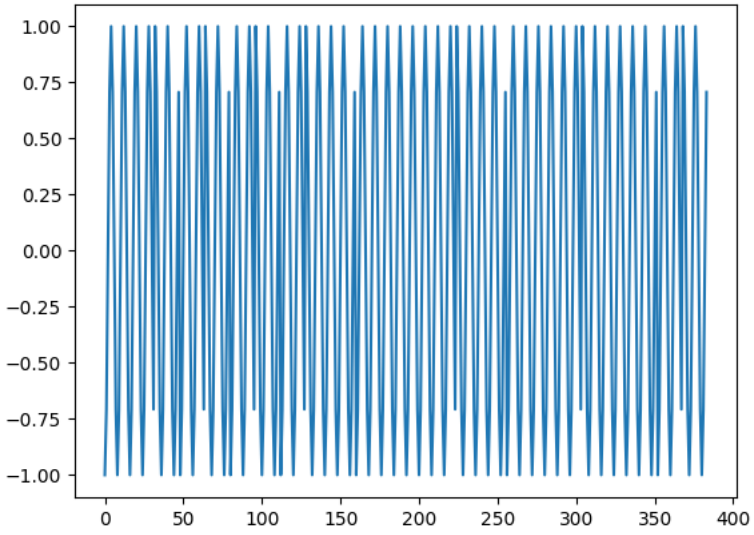
II. Binary Phase Shift Keying

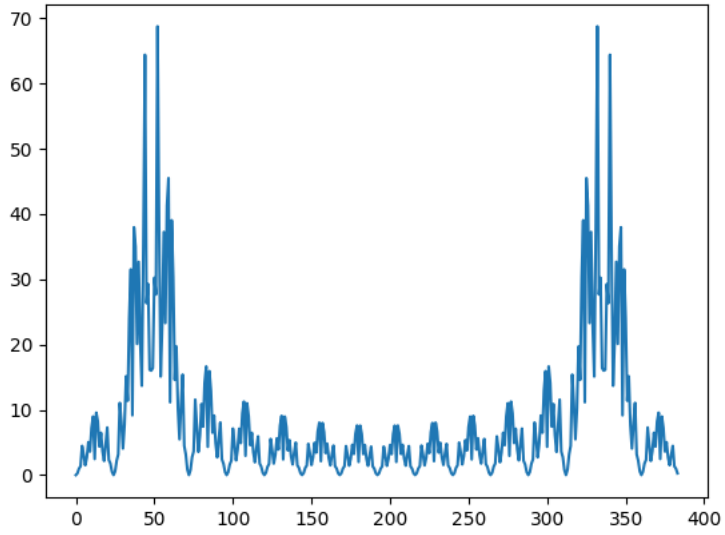
1. Modulation

In BPSK modulation, we distinguish between different signals by changing the phase. When the input is 0, the phase of the carrier signal is 0; when the input is 1, the phase of the carrier signal is pi. Since the carrier is a cos function when the input is 0, we get a cos waveform, and when the input is 1, we get a flipped cos waveform. The following code implements the modulation process and initializes some important variables, such as fc and bit_period.

```
# initialize constants and variables
fc = 0.125
bit_period = 16
tx_mod = np.empty(0)

# BPSK modulation
for i in range(Nbits):
    for j in range(bit_period):
        tx_mod = np.append(tx_mod, (2 * tx_bin[i] - 1) *
                           np.cos(2 * np.pi * fc * (i * bit_period + j)))
```

The following figures show the modulated waveform and the spectrum after the Fast Fourier Transform. We can observe the π phase jumps in the modulated signal, and the sidebands in the spectrum.



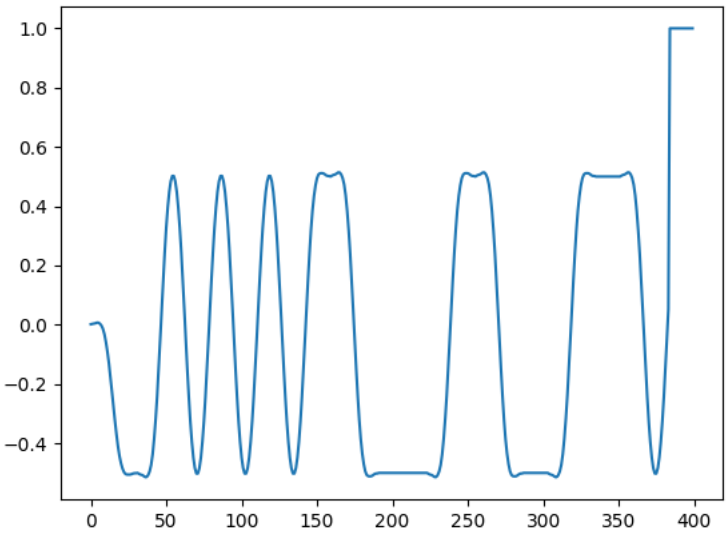


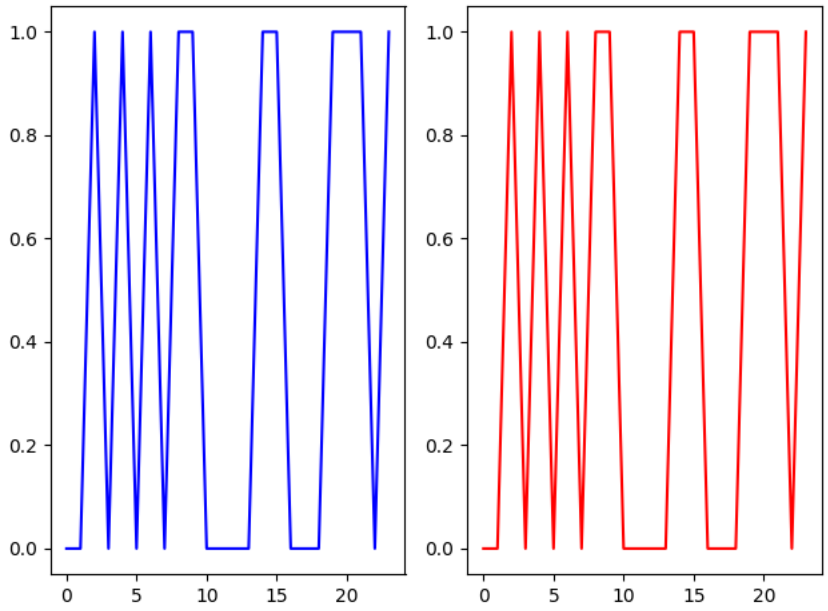
2. Demodulation

The first step in demodulation is to multiply the modulated signal by the carrier wave again. Next, we need to pass this mixed signal through a low-pass filter. scipy.signal library has routines for defining and using digital filtering. The following code shows the details of the implementation.

```
# Demodulation
rx_mixed = np.empty(0)
for i in range(Nbits):
    for j in range(bit_period):
        rx_mixed = np.append(rx_mixed, tx_mod[i * bit_period + j] *
                             np.cos(2 * np.pi * fc * (i * bit_period + j)))

rx_lpf = signal.lfilter(b1, 1, rx_mixed)
rx_lpf = np.append(rx_lpf, np.ones(numtaps // 2))
```

The following figures show the waveform of the filtered signal and a comparison of the restored data with the original data.





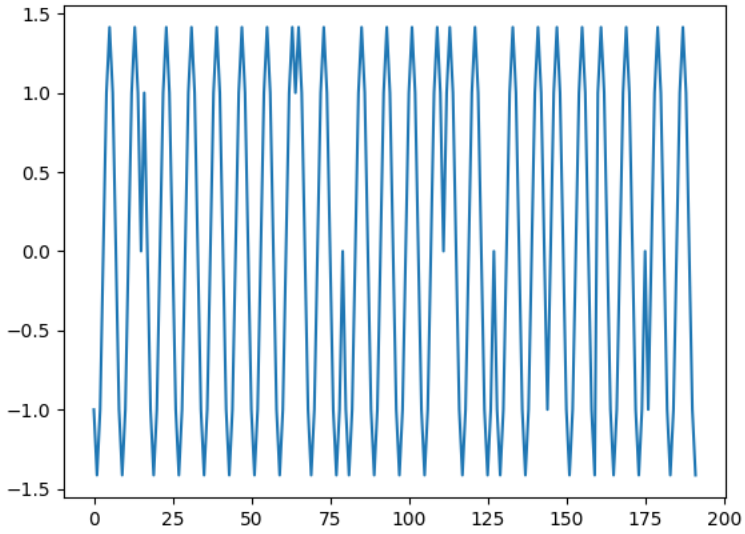
III. Quadrature Phase Shift Keying

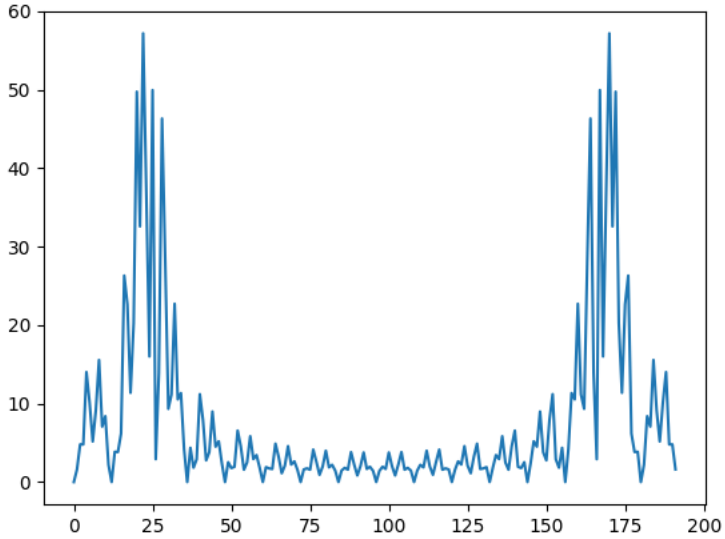
1. Modulation

In QPSK modulation we need to add an orthogonal carrier wave to increase the efficiency of signal transmission, using BPSK we can carry two binary signals in one waveform. The following is the code implementation.

```
# QPSK modulation
for i in range(0, Nbits, 2):
    for j in range(bit  period):
        tx_mod = np.append(tx_mod, (2 * tx_bin[i] - 1) *
                           np.cos(2 * np.pi * fc * (i * bit_period + j)) +
                           (2 * tx_bin[i + 1] - 1) * np.sin(2 * np.pi * fc * (i *
bit_period + j)))
```

The following figures show the modulated waveform and the spectrum after the Fast Fourier Transform.



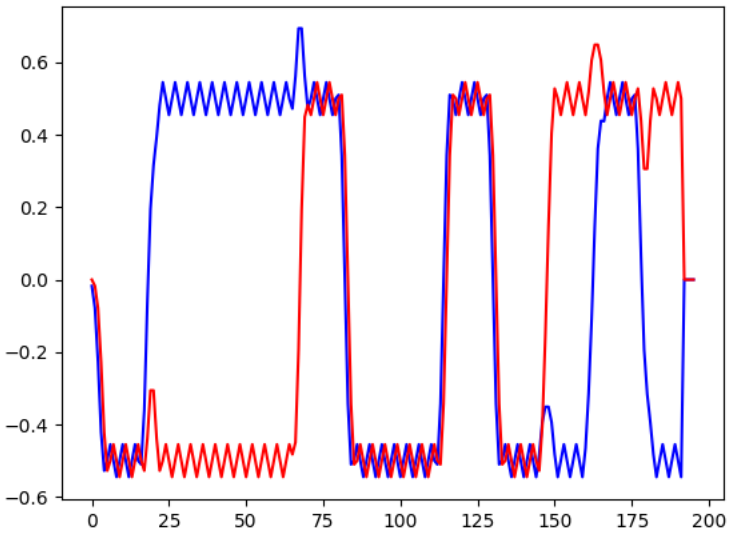


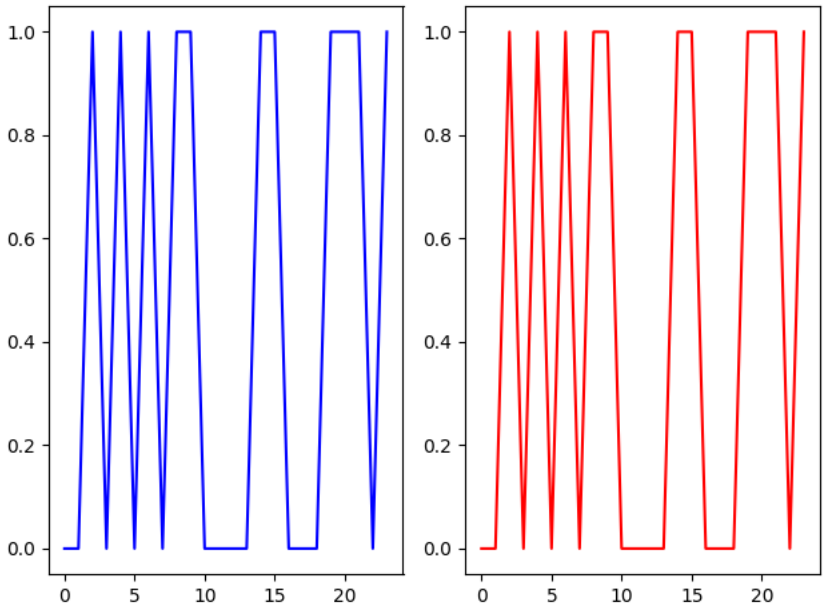
2. Demodulation

In the demodulation part, we need to demodulate the in-phase and quadrature parts respectively, so the signal tx_mod needs to be multiplied by the cos and sin functions respectively and then passed through the low-pass filter to filter the high-frequency part of the signal, the following is the code implementation.

```
# Demodulation
rx_mixed_i = np.empty(0)
rx_mixed_q = np.empty(0)

for i in range(0, Nbits, 2):
    for j in range(bit_period):
        rx_mixed_i = np.append(rx_mixed_i, tx_mod[(i // 2) * bit_period +
j] *
                               np.cos(2 * np.pi * fc * (i * bit_period + j)))
        rx_mixed_q = np.append(rx_mixed_q, tx_mod[(i // 2) * bit_period
+ j] *
                               np.sin(2 * np.pi * fc * (i * bit_period + j)))

rx_filt_i = signal.lfilter(b1, 1, rx_mixed_i)
rx_filt_i = np.append(rx_filt_i, np.zeros(numtaps // 2) / 2)
rx_filt_q = signal.lfilter(b1, 1, rx_mixed_q)
rx_filt_q = np.append(rx_filt_q, np.zeros(numtaps // 2) / 2)
```

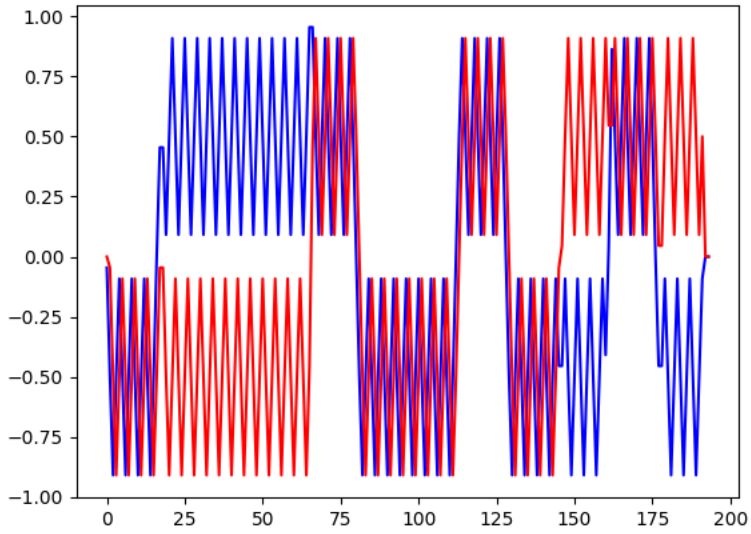
The following figures show the waveform of the filtered signal and a comparison of the restored data with the original data.

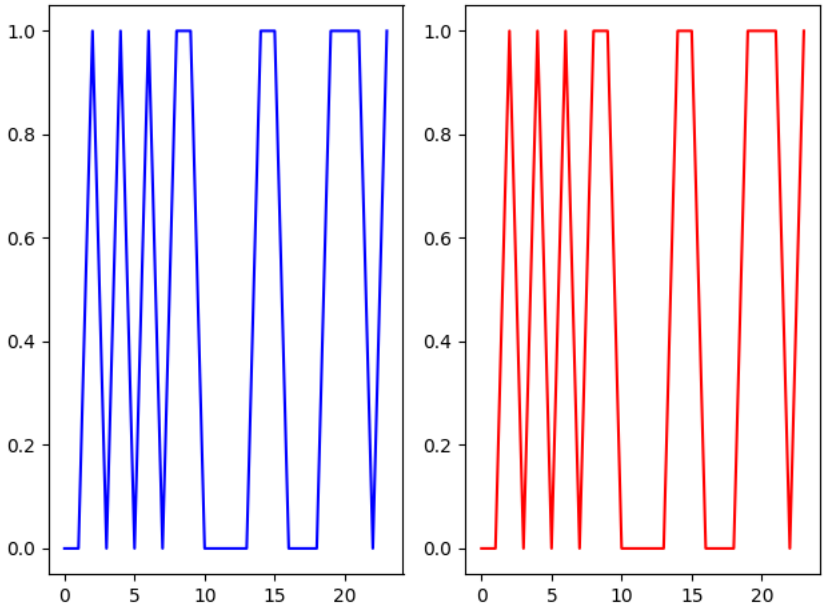




IV. Latency

We can observe a delay in the output data when compared to the input data. This is because the digital filter comprises delays within each tap, and the filter designs resulting from signal.firwin normally has an overall delay of numtaps//2. So we can reduce the delay by decreasing the numtaps, the following test is the corresponding waveform graph and the result when numtaps is equal to 4.


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
25
50
75
100
125
150
175
200


1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20