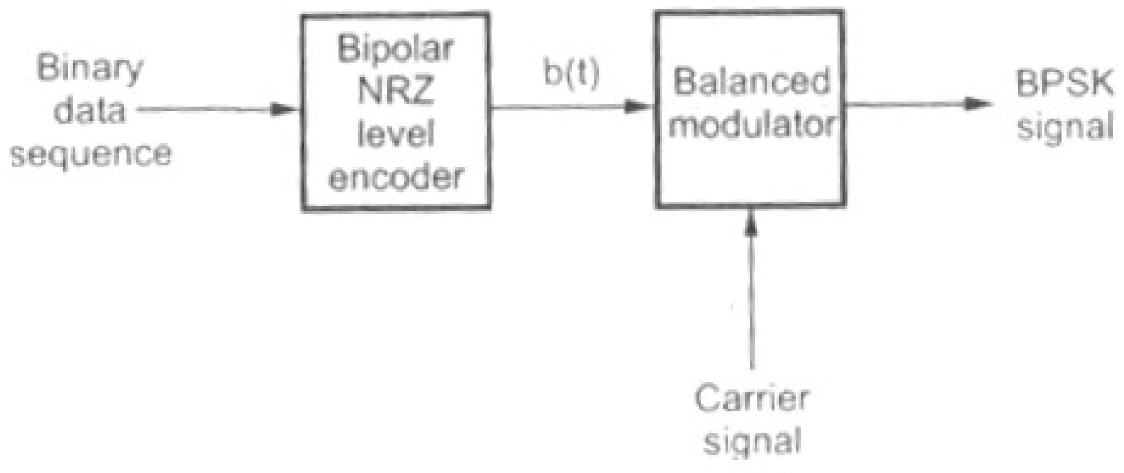
```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.special as sp
```

BPSK

Phase Shift Keying is the digital modulation technique in which the phase of the carrier signal is changed by varying the sine and cosine inputs at a particular time. PSK is widely used for wireless LANs, bio-metric, contactless operations, along with RFID and Bluetooth communications.

Binary Phase Shift Keying (**BPSK**) is also called as 2-phase PSK or Phase Reversal Keying. In this technique, the sine wave carrier takes two phase reversals such as 0° and 180°. BPSK is basically a Double Side Band Suppressed Carrier (DSBSC) modulation scheme, for message being the digital information.

BPSK generator :-



Algorithm to generate BPSK signal :-

1. Generate a random binary signal
2. Generate a carrier signal
3. Modulate the carrier with binary signal
4. Display the output BPSK waveforms

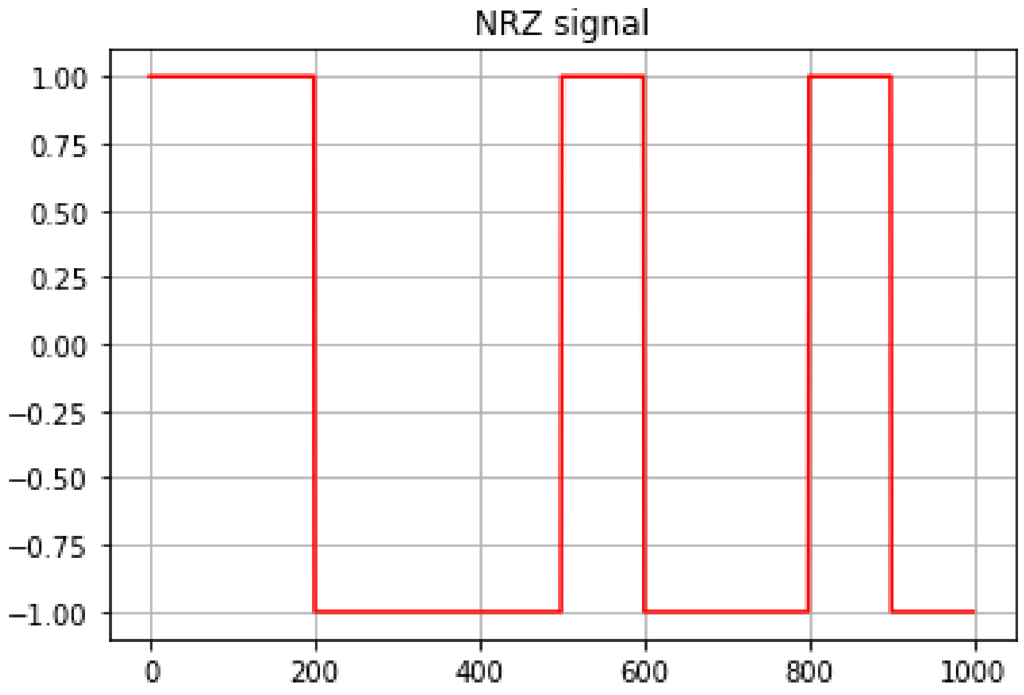
```python
#Generating random binary signal
N = 10 #length
binsig = np.random.randint(2, size = N) #binary signal of length N
print(binsig)
```

```
[1 1 0 0 0 1 0 0 1 0]
```

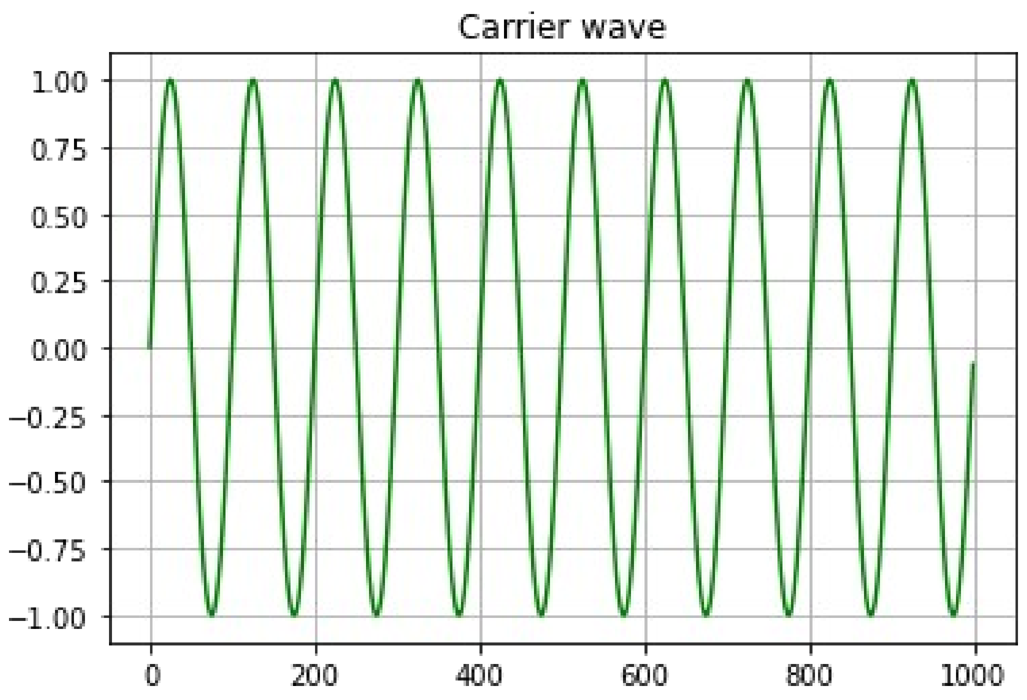
```python
binsig[binsig == 0] = -1 #generating NRZ sequence
print(binsig)
```

```
[ 1  1 -1 -1 -1  1 -1 -1  1 -1]
```

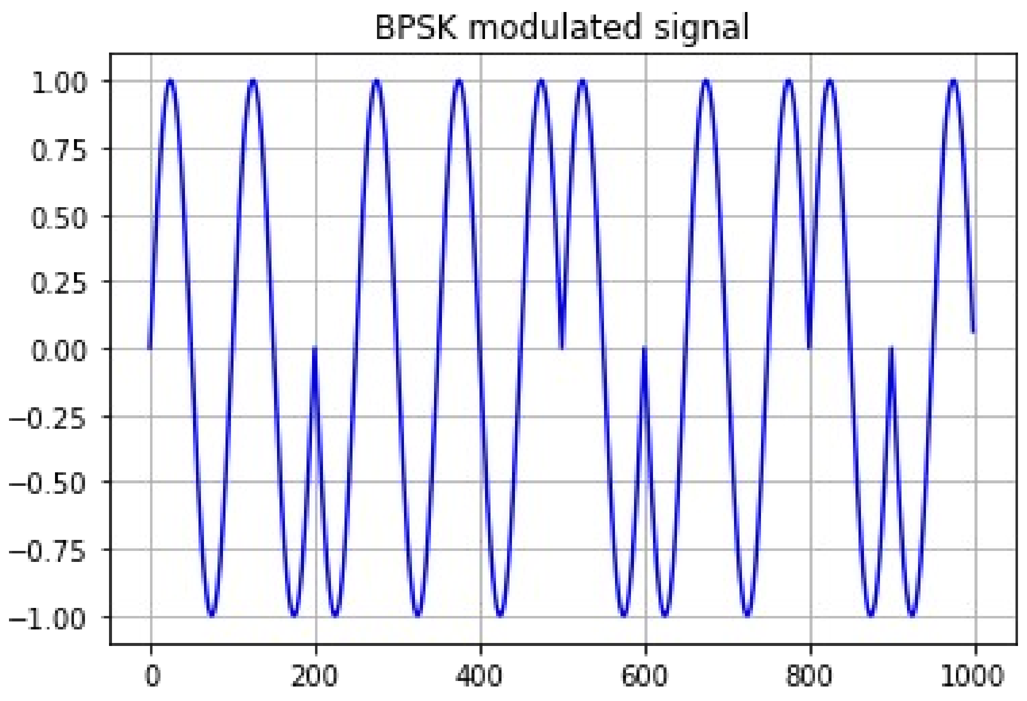
```
binsig = np.repeat(binsig, 100) #repeating each bit 100 times
plt.plot(binsig, 'r')
plt.title("NRZ signal")
plt.grid(1)
```



```
#Generating carrier wave
f = 0.01 #carrier frequency
t = np.arange(0,10/f) #time vector
carwav = np.sin(2*np.pi*f*t) #carrier wave
plt.plot(t, carwav, 'g')
plt.title("Carrier wave")
plt.grid(1)
```



```
#Modulating the carrier with NRZ signal
modsig = carwav * binsig #multiplying carrier wave and NRZ signal to
generate modulated signal
plt.plot(t, modsig, 'b')
plt.title("BPSK modulated signal")
plt.grid(1)
```



Bit error rate (**BER**) of a communication system is defined as the ratio of number of error bits and total number of bits transmitted during a specific period. It is the likelihood that a single error bit will occur within received bits, independent of rate of transmission.

For any given modulation, the BER is normally expressed in terms of signal to noise ratio (SNR). The bit error probability is given by :-

Pb = (1/2) * ERFC (√(Eb/N0))

A randomly generated bit stream is generated and converted to BPSK waveforms. BPSK waveforms are transmitted through an AWGN channel with a fixed SNR . The received symbols are converted again to bits. The received bits are compared with transmitted bits and error is calculated. The signal to noise ratio is then varied and the process is repeated. Monte Carlo simulation is used to calculate bit error probability.

Monte Carlo simulation describes a simulation in which a parameter of a system, such as the bit error rate (BER), is estimated using Monte Carlo techniques. Monte Carlo estimation is the process of estimating the value of a parameter by performing an underlying stochastic, or random, experiment.

Algorithm to evaluate error performance of BPSK waveform :-

1. Generate a string of message bits.
2. Encode using BPSK with energy per bit as Eb and represent it using points in a signal-space.
3. Simulate transmission of the BPSK modulated signal via an AWGN channel with variance N0/2.
4. Detect using an ML decoder and plot the probability of error as a function of SNR per bit Eb/N0.

```
EbN0dB = np.arange(-10,20,1) #EbN0 in dB
print(EbN0dB)
```

```
[-10  -9  -8  -7  -6  -5  -4  -3  -2  -1   0   1   2   3   4   5   6
7
   8   9  10  11  12  13  14  15  16  17  18  19]
```

```
EbN0log = 10 ** (EbN0dB/10) #EbN0 in log
print(EbN0log)
```

```
[ 0.1         0.12589254  0.15848932  0.19952623  0.25118864
0.31622777
  0.39810717  0.50118723  0.63095734  0.79432823  1.
1.25892541
  1.58489319  1.99526231  2.51188643  3.16227766  3.98107171
5.01187234
  6.30957344  7.94328235 10.         12.58925412 15.84893192
19.95262315
 25.11886432 31.6227766  39.81071706 50.11872336 63.09573445
79.43282347]
```

```
N = 1000000 #bit length
Pb = np.zeros(len(EbN0log)) #bit error probability
BER = np.zeros(len(EbN0log)) #bit error rate

for i in range(0, len(EbN0log)):
  ebn0 = EbN0log[i]
  Pb[i] = 0.5 * sp.erfc(np.sqrt(ebn0)) #calculating bit error
probability
  txbits = 2 * (np.random.rand(N) >= 0.5) - 1 #transmitted bit
sequence
  noise = 1 / np.sqrt(2 * ebn0) #generating noise
  noisybits = txbits + noise * np.random.randn(N) #generating noisy
bits
  rxbits = 2 * (noisybits >= 0) - 1 #received bit sequence
  errors = (txbits != rxbits).sum() #number of errors
  BER[i] = errors/N #BER is errors divided by bit length, assuming Eb
= 1
  print("**********")
  print("EbN0log = ", ebn0)
  print("Bit error probability (Pb) = ", Pb[i])
  print("Errors = ", errors)
  print("Bit error rate (BER) = ", BER[i])
  print("**********")
```

```
**********
EbN0log =  0.1
Bit error probability (Pb) =  0.32736042300928847
Errors =  327693
Bit error rate (BER) =  0.327693
**********
**********
EbN0log =  0.12589254117941673
Bit error probability (Pb) =  0.30791047071507943
Errors =  307793
Bit error rate (BER) =  0.307793
**********
**********
EbN0log =  0.15848931924611134
Bit error probability (Pb) =  0.2867145275814431
Errors =  286821
Bit error rate (BER) =  0.286821
**********
**********
EbN0log =  0.19952623149688797
Bit error probability (Pb) =  0.263789505256266
Errors =  264214
Bit error rate (BER) =  0.264214
**********
**********
EbN0log =  0.251188643150958
```

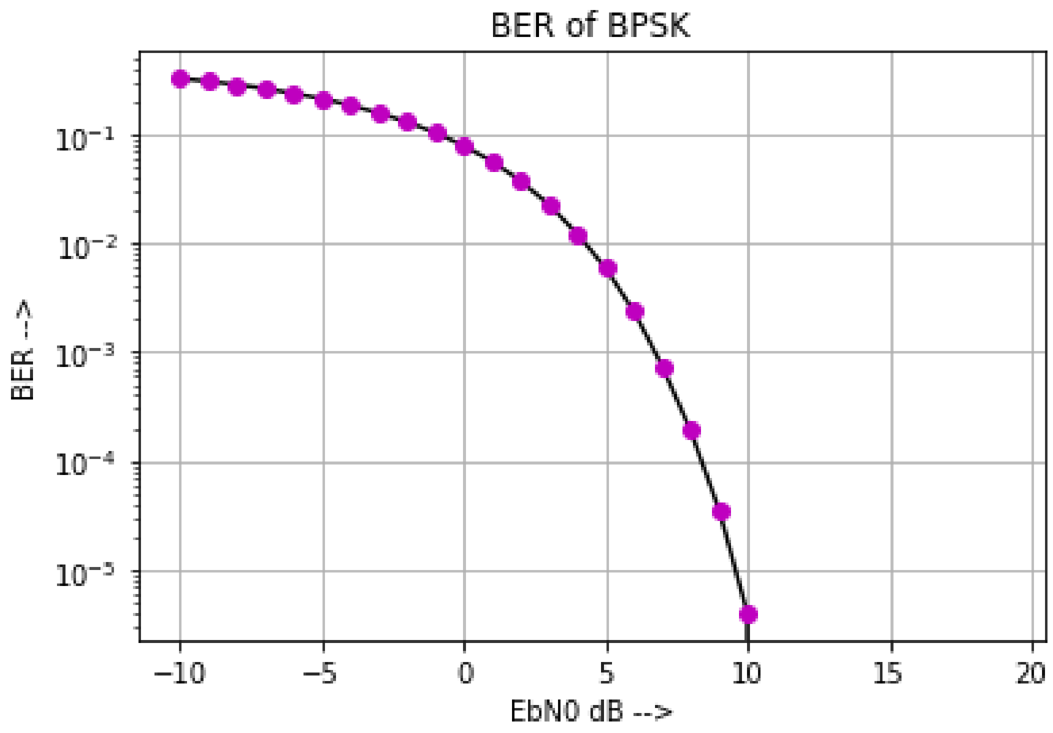
```
Bit error probability (Pb) =  0.23922871076767194
Errors =  239041
Bit error rate (BER) =  0.239041
**********
**********
EbN0log =  0.31622776601683794
Bit error probability (Pb) =  0.2132280183576204
Errors =  213109
Bit error rate (BER) =  0.213109
**********
**********
EbN0log =  0.3981071705534972
Bit error probability (Pb) =  0.186113817483389
Errors =  185669
Bit error rate (BER) =  0.185669
**********
**********
EbN0log =  0.5011872336272722
Bit error probability (Pb) =  0.15836831880959795
Errors =  157709
Bit error rate (BER) =  0.157709
**********
**********
EbN0log =  0.6309573444801932
Bit error probability (Pb) =  0.13064448852282917
Errors =  130509
Bit error rate (BER) =  0.130509
**********
**********
EbN0log =  0.7943282347242815
Bit error probability (Pb) =  0.10375909595340632
Errors =  104208
Bit error rate (BER) =  0.104208
**********
**********
EbN0log =  1.0
Bit error probability (Pb) =  0.07864960352514258
Errors =  78883
Bit error rate (BER) =  0.078883
**********
**********
EbN0log =  1.2589254117941673
Bit error probability (Pb) =  0.05628195197654147
Errors =  56716
Bit error rate (BER) =  0.056716
**********
**********
EbN0log =  1.5848931924611136
Bit error probability (Pb) =  0.03750612835892598
Errors =  37599
```

```
Bit error rate (BER) =  0.037599
**********
**********
EbN0log =  1.9952623149688795
Bit error probability (Pb) =  0.02287840756108532
Errors =  22855
Bit error rate (BER) =  0.022855
**********
**********
EbN0log =  2.51188643150958
Bit error probability (Pb) =  0.01250081804073755
Errors =  12264
Bit error rate (BER) =  0.012264
**********
**********
EbN0log =  3.1622776601683795
Bit error probability (Pb) =  0.005953867147778662
Errors =  5962
Bit error rate (BER) =  0.005962
**********
**********
EbN0log =  3.9810717055349722
Bit error probability (Pb) =  0.002388290780932807
Errors =  2380
Bit error rate (BER) =  0.00238
**********
**********
EbN0log =  5.011872336272722
Bit error probability (Pb) =  0.0007726748153784446
Errors =  747
Bit error rate (BER) =  0.000747
**********
**********
EbN0log =  6.309573444801933
Bit error probability (Pb) =  0.00019090777407599314
Errors =  190
Bit error rate (BER) =  0.00019
**********
**********
EbN0log =  7.943282347242816
Bit error probability (Pb) =  3.3627228419617505e-05
Errors =  36
Bit error rate (BER) =  3.6e-05
**********
**********
EbN0log =  10.0
Bit error probability (Pb) =  3.872108215522035e-06
Errors =  4
Bit error rate (BER) =  4e-06
**********
```

```
**********
EbN0log =  12.589254117941675
Bit error probability (Pb) =  2.613067953575199e-07
Errors =  0
Bit error rate (BER) =  0.0
**********
**********
EbN0log =  15.848931924611133
Bit error probability (Pb) =  9.006010350628754e-09
Errors =  0
Bit error rate (BER) =  0.0
**********
**********
EbN0log =  19.952623149688797
Bit error probability (Pb) =  1.3329310175300506e-10
Errors =  0
Bit error rate (BER) =  0.0
**********
**********
EbN0log =  25.118864315095795
Bit error probability (Pb) =  6.810189128780772e-13
Errors =  0
Bit error rate (BER) =  0.0
**********
**********
EbN0log =  31.622776601683793
Bit error probability (Pb) =  9.123957362628105e-16
Errors =  0
Bit error rate (BER) =  0.0
**********
**********
EbN0log =  39.810717055349734
Bit error probability (Pb) =  2.267395844454418e-19
Errors =  0
Bit error rate (BER) =  0.0
**********
**********
EbN0log =  50.11872336272722
Bit error probability (Pb) =  6.758969770654687e-24
Errors =  0
Bit error rate (BER) =  0.0
**********
**********
EbN0log =  63.09573444801933
Bit error probability (Pb) =  1.3960143109067526e-29
Errors =  0
Bit error rate (BER) =  0.0
**********
**********
EbN0log =  79.43282347242814
```

```
Bit error probability (Pb) =  1.0010739735708612e-36
Errors =  0
Bit error rate (BER) =  0.0
**********
```

```
plt.plot(EbN0dB, BER, 'k-', EbN0dB, BER, 'mo')
plt.xscale("linear")
plt.yscale("log")
plt.xlabel("EbN0 dB -->")
plt.ylabel("BER -->")
plt.title("BER of BPSK")
plt.grid(1)
```



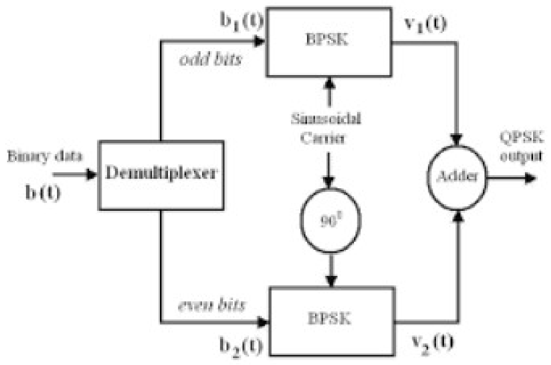
QPSK

Quadrature Phase Shift Keying (**QPSK**) is a variation of BPSK. If two or more bits are combined, then signal rate is reduced, the bandwidth required will be less. If two bits per symbol are transmitted, i.e. in each symbol we have 2 bits, and if phase shift keying is used, then it is Quadrature Phase shift keying.

In QPSK, the total 360 degree phase is divided into four phases.

The 4 phases that are used are :- 45°, 135°, 225°, 315°

QPSK generator :-



Algorithm to generate QPSK signal :-

1. Generate random bit streams
2. Start FOR loop
3. Generate corresponding message signal(bipolar form)
4. Multiply carrier 1 with odd bits of message signal and carrier 2 with even bits of message signal
5. Perform addition of odd and even modulated signals to get the QPSK modulated signal
6. Plot QPSK modulated signal.
7. End FOR loop.
8. Plot the binary data and carriers.
9. Add random noise to signal.
10. Based on the waveform decode the output.
11. Plot the input and output waveforms

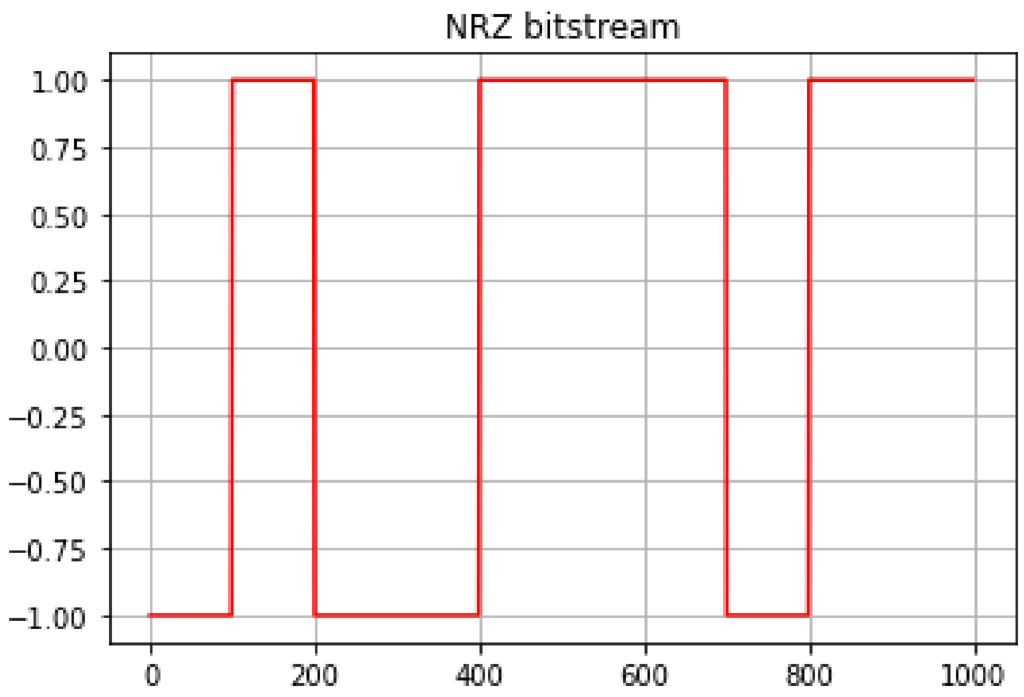
```
#Generating random bitstream
N = 10 #length
binsig = np.random.randint(2, size = N) #binary bitstream of length N
print(binsig)
```

```
[0 1 0 0 1 1 1 0 1 1]
```

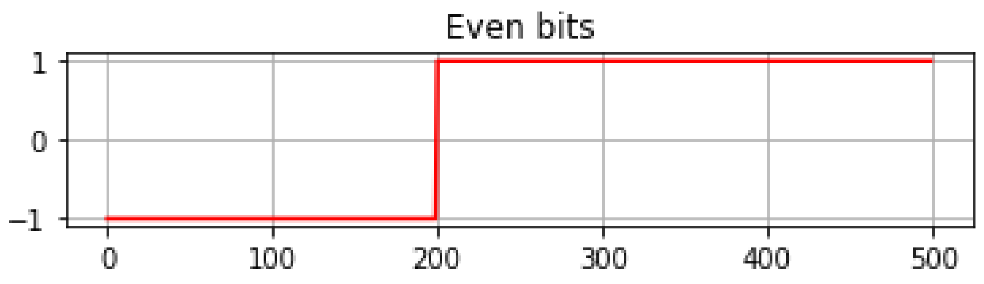
```
binsig[binsig == 0] = -1 #generating NRZ bitstream
print(binsig)
```

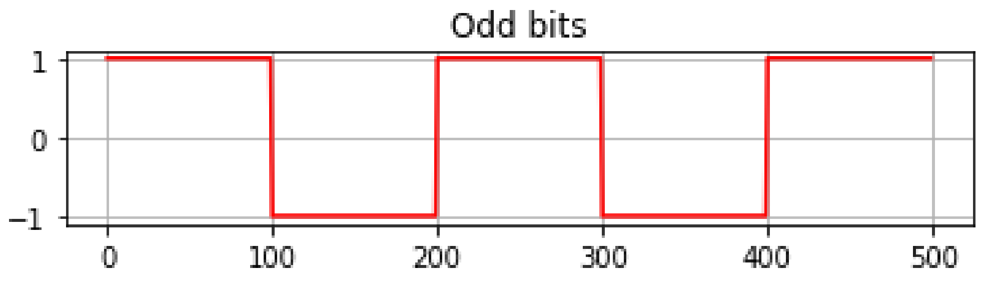
```
[-1  1 -1 -1  1  1  1 -1  1  1]
```

```
plt.plot(np.repeat(binsig,100) ,'r')
plt.title("NRZ bitstream")
plt.grid(1)
```

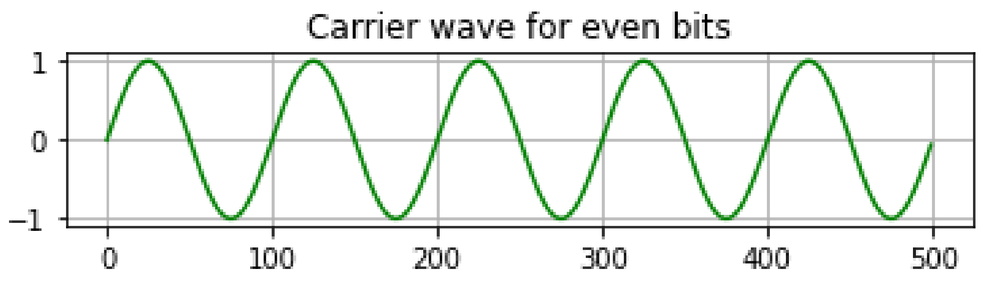


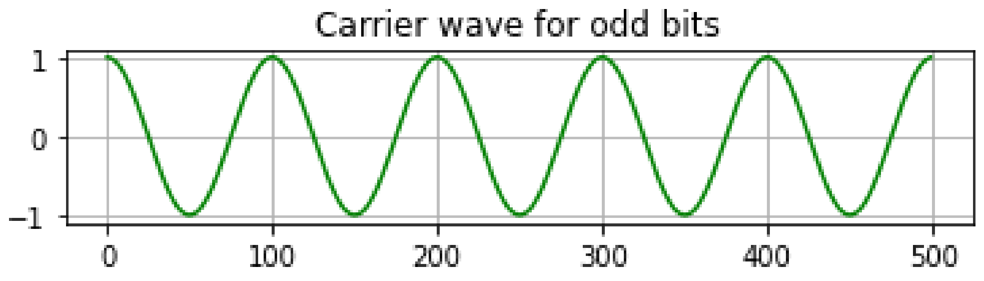
```
evensig = binsig[0::2] #even bits of bitstream
evensig = np.repeat(evensig, 100)
oddsig = binsig[1::2] #odd bits of bitstream
oddsig = np.repeat(oddsig, 100)
plt.subplot(3,1,1)
plt.plot(evensig, 'r')
plt.title("Even bits")
plt.grid(1)
plt.subplot(3,1,3)
plt.plot(oddsig, 'r')
plt.title("Odd bits")
plt.grid(1)
```



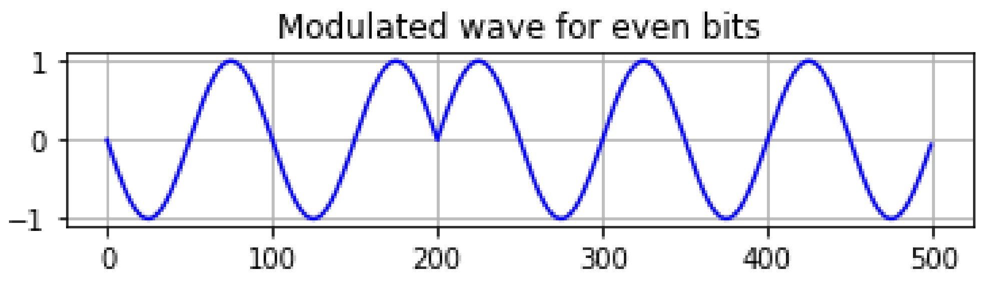


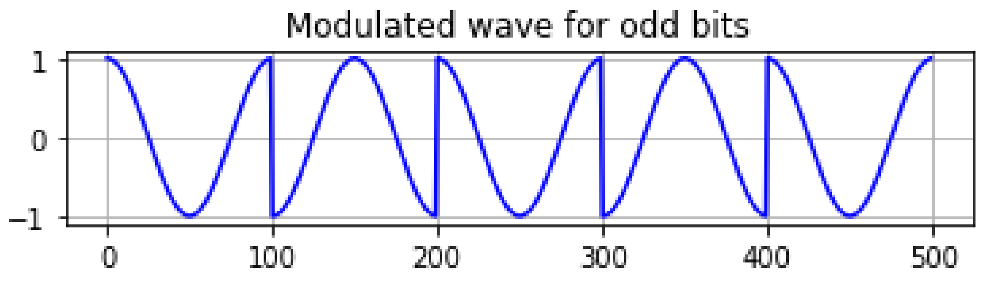
```
#Generating carrier waves
f = 0.01 #carrier frequency
t = np.arange(0,5/f) #time vector
carwaveven = np.sin(2*np.pi*f*t) #carrier for even signal
carwavodd = np.cos(2*np.pi*f*t) #carrier for odd signal
plt.subplot(3,1,1)
plt.plot(t, carwaveven, 'g')
plt.title("Carrier wave for even bits")
plt.grid(1)
plt.subplot(3,1,3)
plt.plot(t, carwavodd, 'g')
plt.title("Carrier wave for odd bits")
plt.grid(1)
```



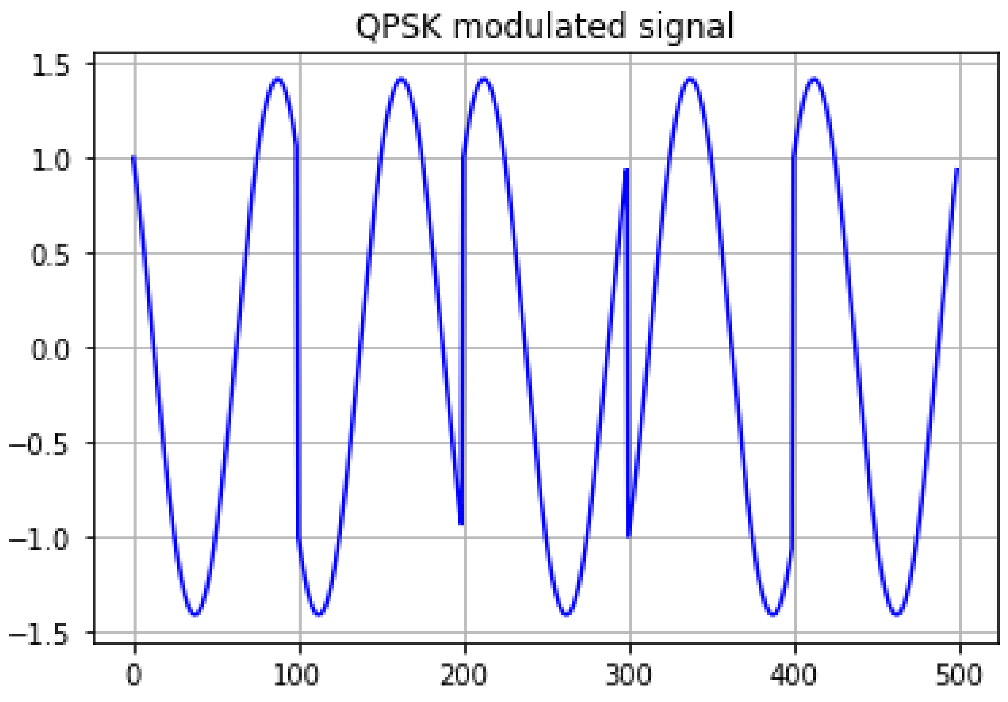


```
#Performing BPSK on even and odd bits seperately
modeven = evensig * carwaveven #modulated wave for even bits
mododd = oddsig * carwavodd #modulated wave for odd bits
plt.subplot(3,1,1)
plt.plot(t, modeven, 'b')
plt.title("Modulated wave for even bits")
plt.grid(1)
plt.subplot(3,1,3)
plt.plot(t, mododd, 'b')
plt.title("Modulated wave for odd bits")
plt.grid(1)
```





```
#Generating QPSK modulated signal
modsig = modeven + mododd
plt.plot(t, modsig, 'b')
plt.title("QPSK modulated signal")
plt.grid(1)
```

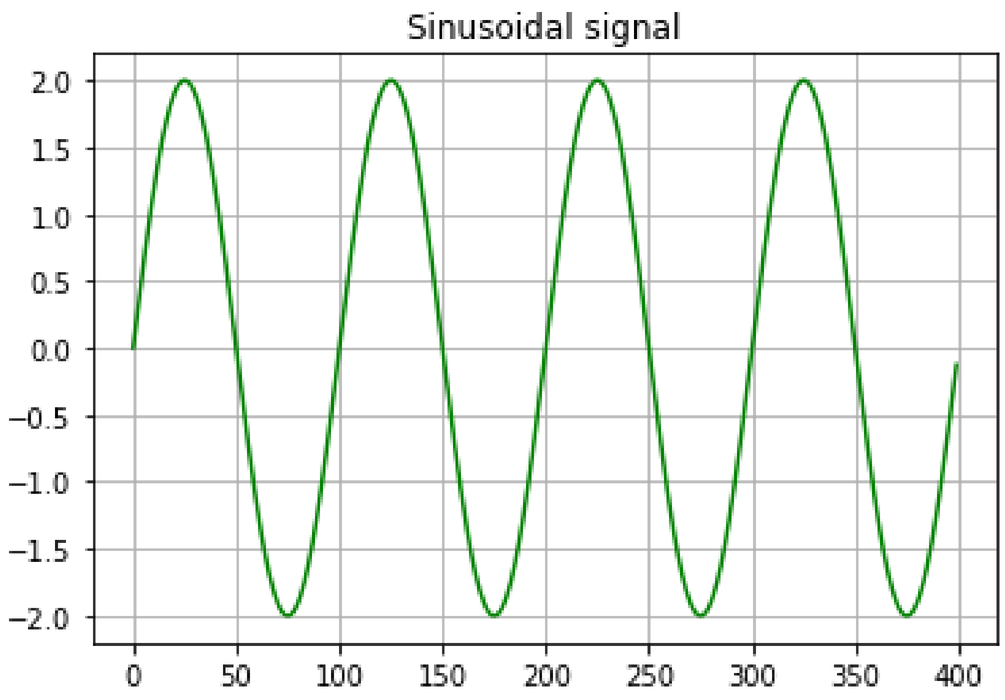


PCM

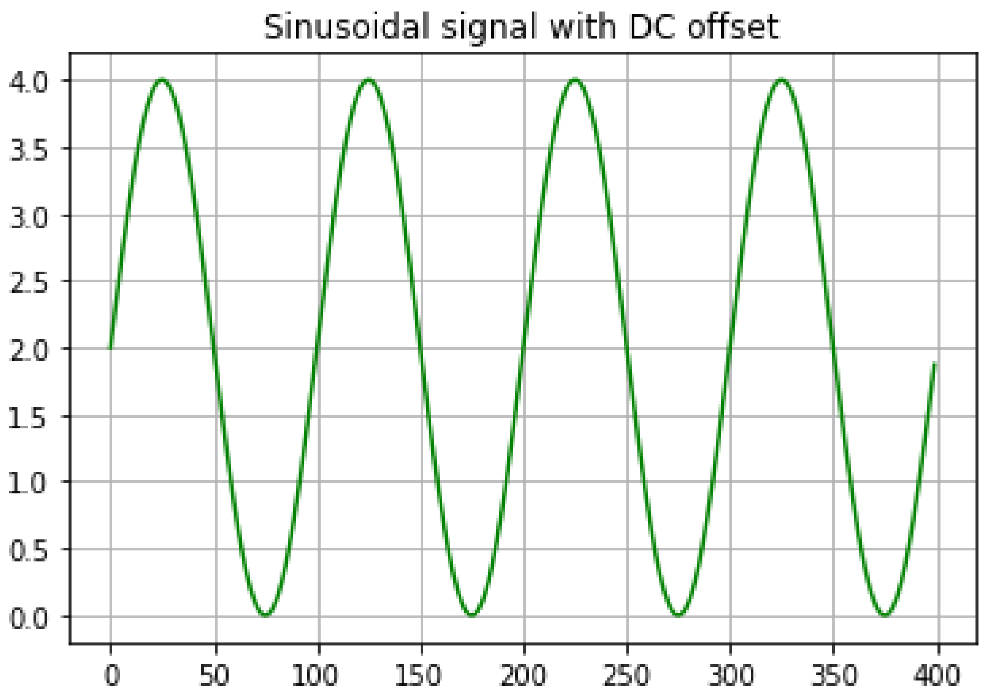
1. Generate a sinusoidal waveform with a DC offset so that it takes only positive amplitude value.
2. Sample and quantize the signal using an uniform quantizer with number of representation levels L. Vary L.Represent each value using decimal to binary encoder.

```
#Generating sinusoidal signal
L = int(input("L = "))
f = 0.01 #frequency
t = np.arange(0, 4/f) #time vector
sig = (L/2) * np.sin(2*np.pi*f*t) #sinusoidal signal
plt.plot(t, sig, 'g')
plt.title("Sinusoidal signal")
plt.grid(1)
```

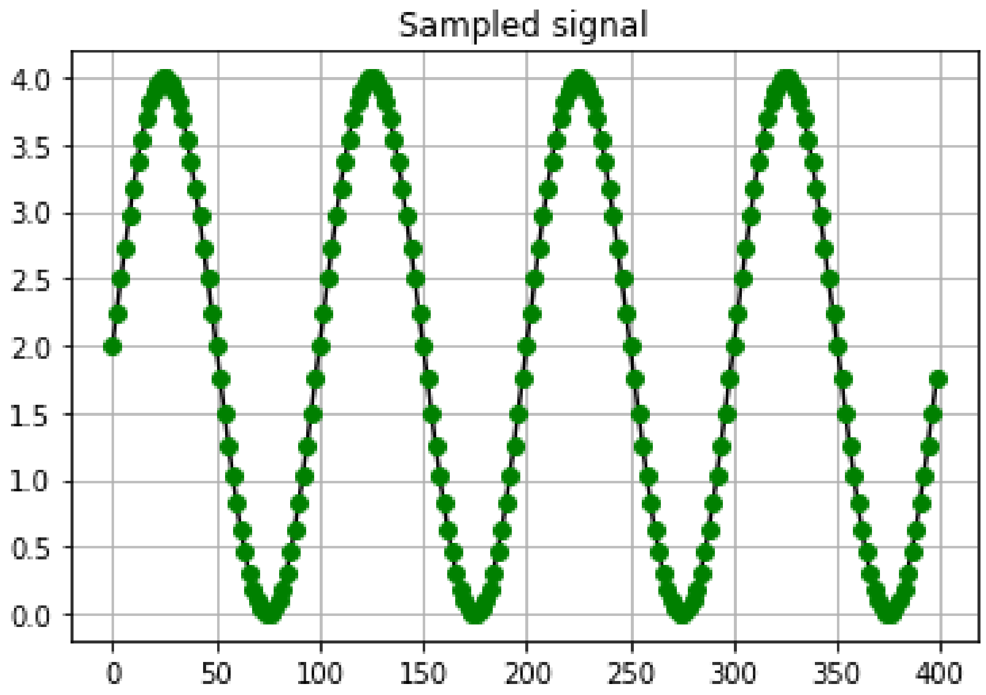
```
L = 4
```



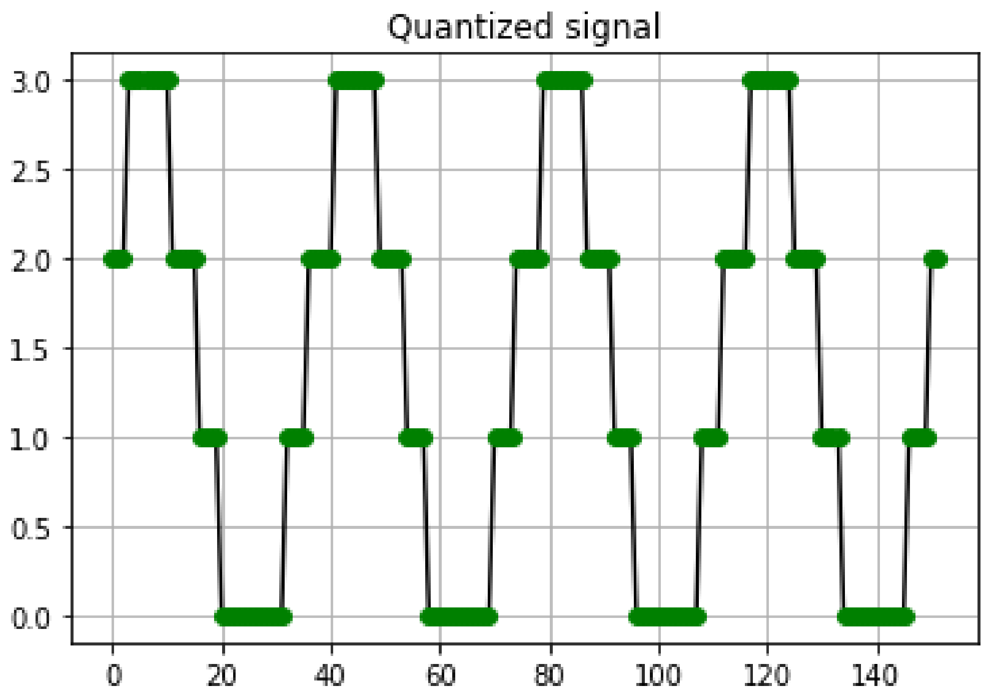
```
#Adding DC offset
dc = L/2 #DC offset
shiftedsig = sig + dc #sinusoidal signal with DC offset
plt.plot(t, shiftedsig, 'g')
plt.title("Sinusoidal signal with DC offset")
plt.grid(1)
```



```
#Sampling sinusoidal signal
fs = 50 * f #sampling frequency
ts = np.arange(0, 4/f, 1/fs) #sampling rate
sampledsig = (L/2) * np.sin(2*np.pi*f*ts) #sampled sinusoidal signal
sampledsig = sampledsig + dc
plt.plot(ts, sampledsig, 'k-', ts, sampledsig, 'go')
plt.title("Sampled signal")
plt.grid(1)
```



```
#Quantizing sinusoidal signal
quantlvls = np.linspace(0,L-1,L) #quantization levels
quantizedsig = []
for i in sampledsig:
  for j in quantlvls:
    if(((i <= j) and ((j-i) <= 0.5)) or ((i >= j) and ((i-j) <= 
0.5))):
      quantizedsig.append(j)
      break
plt.plot(quantizedsig, 'k-', quantizedsig, 'go')
plt.title("Quantized signal")
plt.grid(1)
```



```
#Encoding signal
encodedsig = []
bitlen = int(np.log2(L))
for i in quantizedsig:
    i = int(i)
    binary = bin(i).replace("0b","")
    z = "0" * (bitlen - len(binary))
    binary = z + binary
    encodedsig.append(binary)
print("Encoded signal : ", encodedsig)
```

```
Encoded signal :  ['10', '10', '10', '11', '11', '11', '11', '11',
'11', '11', '11', '10', '10', '10', '10', '10', '01', '01', '01',
'01', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00',
'00', '00', '01', '01', '01', '01', '10', '10', '10', '10', '10',
'11', '11', '11', '11', '11', '11', '11', '11', '10', '10', '10',
'10', '10', '01', '01', '01', '01', '00', '00', '00', '00', '00',
'00', '00', '00', '00', '00', '00', '00', '01', '01', '01', '01',
'10', '10', '10', '10', '10', '11', '11', '11', '11', '11', '11',
'11', '11', '10', '10', '10', '10', '10', '01', '01', '01', '01',
'00', '00', '00', '00', '00', '00', '00', '00', '00', '00', '00',
'00', '01', '01', '01', '01', '10', '10', '10', '10', '10', '11',
'11', '11', '11', '11', '11', '11', '11', '10', '10', '10', '10',
'10', '01', '01', '01', '01', '00', '00', '00', '00', '00', '00',
'00', '00', '00', '00', '00', '00', '01', '01', '01', '01', '10',
'10']
```