Digital_Communication_Lab_Assignment

Name: Ayush Sharma

Roll No.: 18T5014

Branch: Electronics and Telecommunication

****Contents**--**

****Getting Started with Python and Data Visualization****

- Importing libraries
- Plotting various graph
- Sharing the finalized document

****Fourier Synthesis (of)****

- Square Wave
- Triangular Wave
- AM with Carrier Wave

****Pulse Code Modulation****

- Sampling of input signal
- Quantization of input signal
- Encoding of quatized signal

****Line Coding****

Return to Zero (RZ)

- Polar Line Coding
- Unipolar Line Coding
- Bipolar Line Coding

****Non-Return to Zero****

- Polar Line Coding
- Unipolar Line Coding
- Bipolar Line Coding
- PSD for various line codes

In [ ]:

Importing required libraries

`NumPy - for efficient matrix & vector operations.`

`PyPlot - for plotting the graphs.`

`SciPy - for efficient scientific operation & computation.`

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

Pyplot is a Matplotlib module which provides a MATLAB-like interface.

Matplotlib is designed to be as usable as MATLAB, with the ability to use Python and the advantage of being free and open-source.

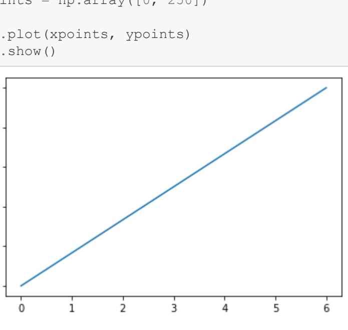
SciPy is a free and open-source Python library used for scientific computing and technical computing. SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering.

For import library/module we use `import` keyword with the name of the library/module.

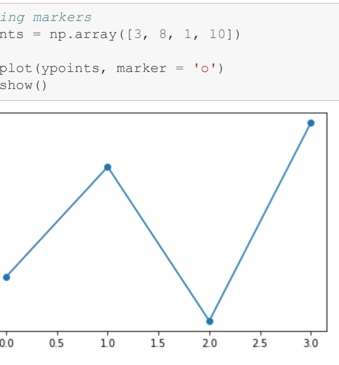
In [52]:
```
import numpy
import scipy
# etc
```

Plotting various graphs

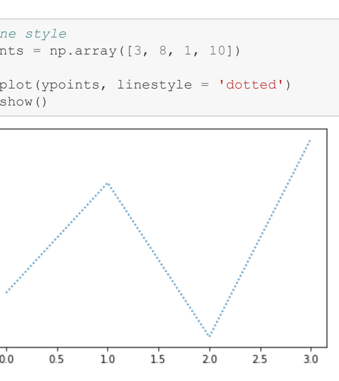
In [53]:
```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([0, 6])
ypoints = np.array([0, 250])

plt.plot(xpoints, ypoints)
plt.show()
```



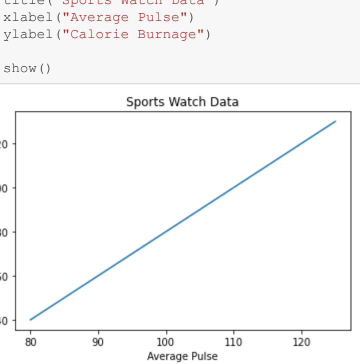
In [55]:
```
# Using markers
ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o')
plt.show()
```



In [56]:
```
# line style
ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, linestyle = 'dotted')
plt.show()
```

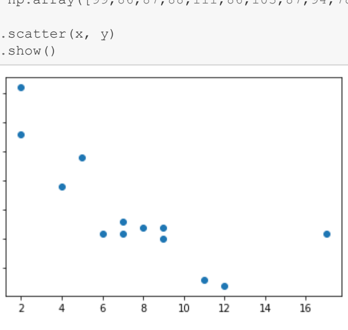


In [57]:
```
x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.plot(x, y)

plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.show()
```

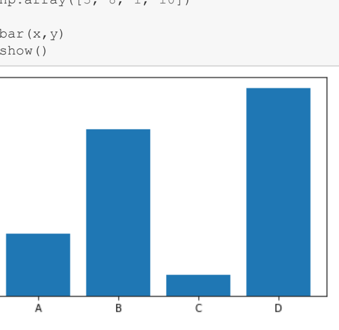


In [ ]:

In [58]:
```
# Scatter Plot
x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])

plt.scatter(x, y)
plt.show()
```



In [60]:
```
# Bar graphs
x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x,y)
plt.show()
```



In [ ]:

In [ ]:

For sharing the finalised file:

- You can go to the `file` option on the top left and choose `Download as` option.
- Then select the format you wisht to use for downloading the file, usually preferred is PDF(.PDF) or Jupyter Notebook(.ipynb)

Importing required libraries

`NumPy - for efficient matrix & vector operations.`

`PyPlot - for plotting the graphs.`

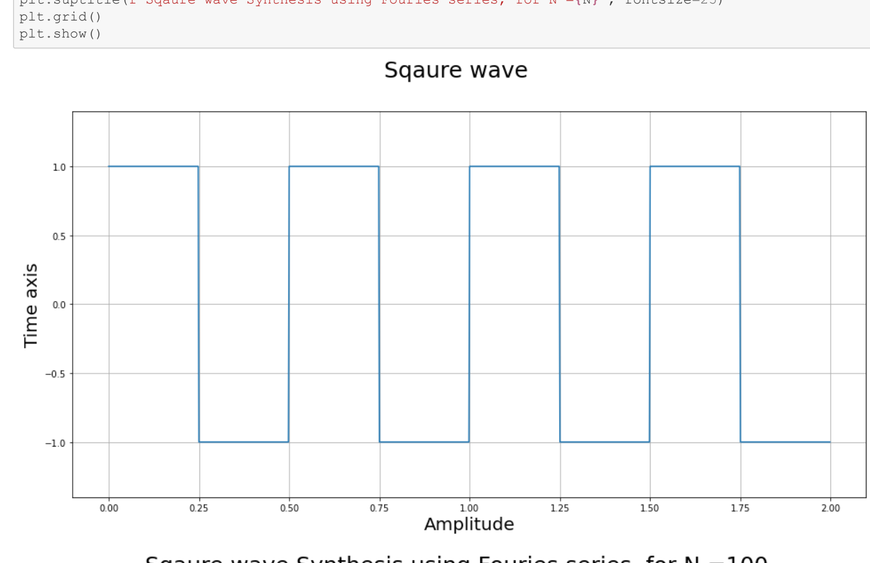
`SciPy - for efficient scientific operation & computation.`

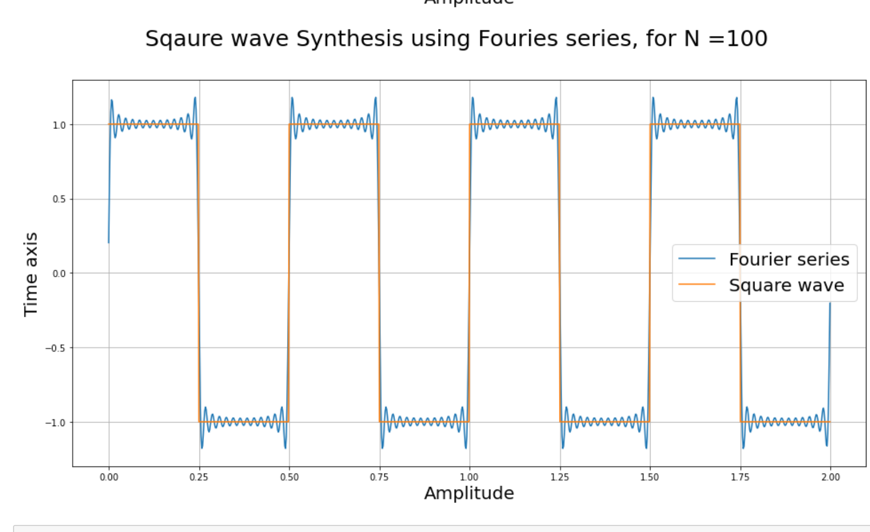
In [40]:
```
import numpy as np        # NumPy - for efficient matrices & vector operations.
from matplotlib import pyplot as plt # PyPlot - for plotting the graph.
from scipy import signal # SciPy - for efficient scientific operation & computation.

from numpy import pi, sin, power, sinc # pi is the irrational constant, sin = sine function,
        # power function is used to put first array elements raised to powers from second array, element-wise.
            # sinc = sin(x)/x

from scipy.integrate import quad # The quad function returns the two values, in which the first
        # number is the value of integral and the second value is the estimate of the absolute error in the value of integral.
from scipy.signal import square,triang
from scipy.integrate import simps
# Note: Comments in python are given by using pound symbol(#).
```

Fourier Synthesis (Assignment-I)

`Square Wave Synthesis`

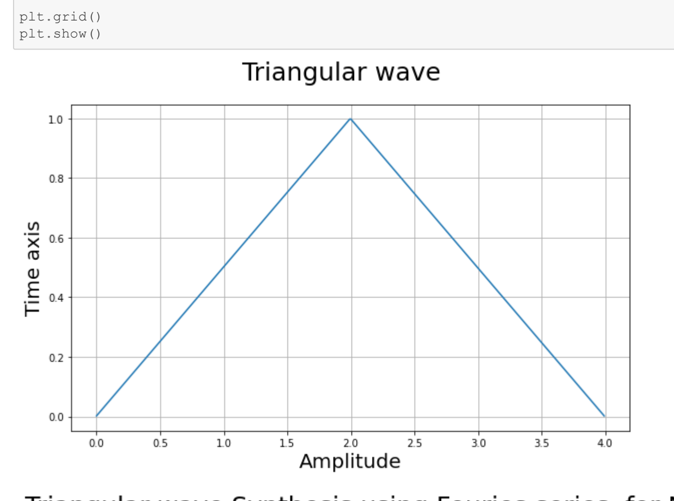
In [52]:
```
# Time period/Periodicity of the square wave function
L = 2

# No of waves in time period L
freq = 4
samples = 1000

# No. of terms
N = 100

# Generation of square wave
x = np.linspace(0,L,samples,endpoint=False)
y = square(2.0*np.pi*x*freq/L)

plt.figure(figsize=(15, 8))
plt.xlabel("Amplitude", fontsize=20)
plt.ylabel("Time axis", fontsize=20)
plt.suptitle("Square wave", fontsize=25)

plt.plot(x,y)
plt.grid()

x1,x2,y1,y2 = plt.axis()
plt.axis((x1,x2,-1.4, 1.4))

# Calculation of Fourier coefficients
a0 = 2./L*simps(y,x)
an = lambda n:2.0/L*simps(y*np.cos(2.*np.pi*n*x/L),x)
bn = lambda n:2.0/L*simps(y*np.sin(2.*np.pi*n*x/L),x)

s = (a0/2.)
for k in range(1,N+1):
    s = s + sum([an(k)*np.cos(2.*np.pi*k*x/L)+bn(k)*np.sin(2.*np.pi*k*x/L)])

# Plotting the Square Wave
plt.figure(figsize=(15, 8))
plt.plot(x,s)
plt.plot(x,y)
plt.xlabel("Amplitude", fontsize=20)
plt.ylabel("Time axis", fontsize=20)
plt.legend(["Fourier series", "Square wave"], loc='upper right', fontsize=20)
plt.suptitle(f"Square wave Synthesis using Fouries series, for N ={N}", fontsize=25)
plt.grid()
plt.show()
```

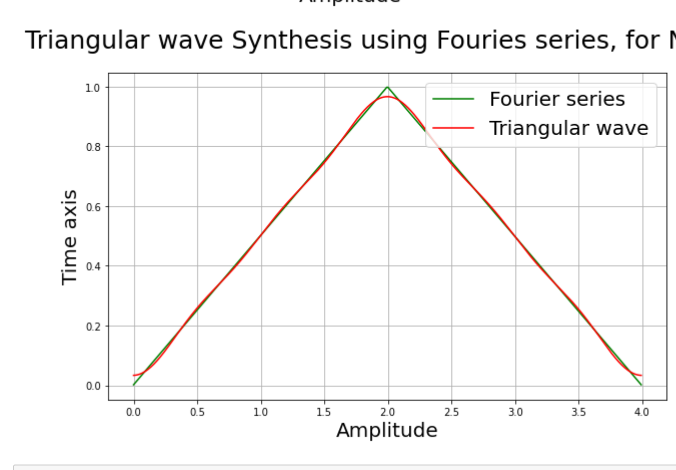




`Triangular Wave Synthesis`

```
plt.xlabel("Amplitude", fontsize=20)
plt.ylabel("Time axis", fontsize=20)
plt.legend(["Fourier series", "Triangular wave"], loc='upper right', fontsize=20)
plt.suptitle(f"Triangular wave Synthesis using Fouries series, for N ={N}", fontsize=25)

plt.grid()
plt.show()
```





In [ ]:

`Pulse Code Modulation`

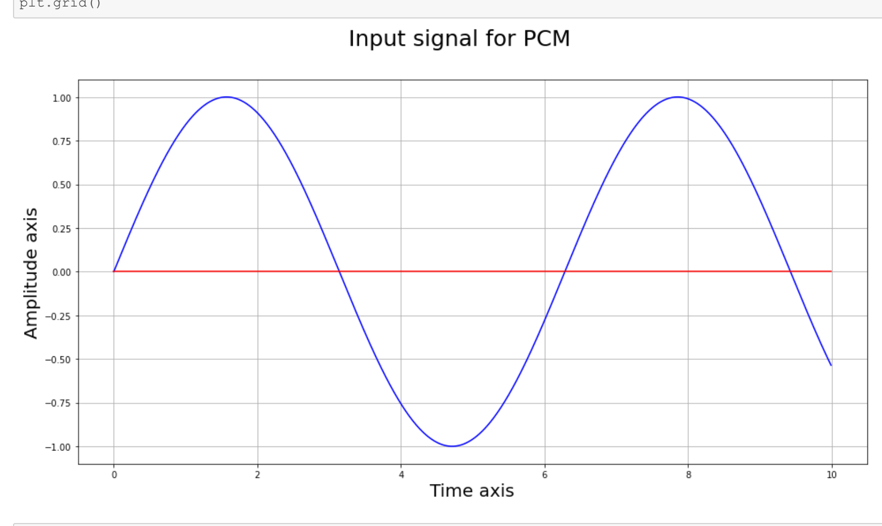
Sampling, Quantization and Encoding for a PCM system

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from numpy import cos, sin, pi, abs
```

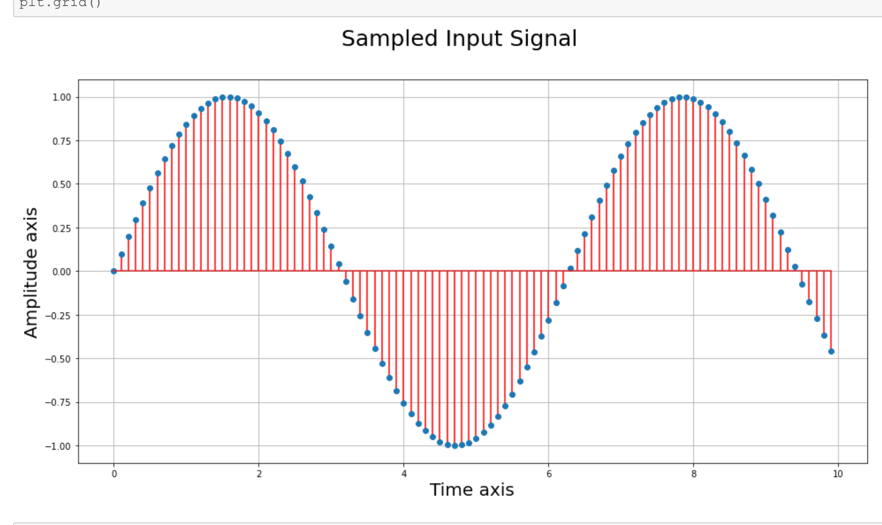
In [2]:
```
def decimalToBinary(n):
    return bin(n).replace("0b", "")
```

In [3]:
```
def uniformQuantization(signal, min_level=-1, max_level=+1, bits=3, val=0):
    '''
    signal: it is the input signal
    min_level: is the minimum amplitude of the input signal
    max_level: is the maximum amplitude of the input signal
    bits: bitrate
    quant_level: is the number of level of quantization.
    '''
    quant_level = (2**bits)
    delta = (max_level - min_level) / quant_level
    sig_norm = (signal - min_level) * (quant_level - 1) / (max_level - min_level)
    #print(sig_norm)
    sig_norm[sig_norm > (quant_level - 1)] = quant_level - 1
    sig_norm[sig_norm < 0] = 0
    sig_norm_quant = np.around(sig_norm)
    #print("normQuant", sig_norm_quant)
    sig_quant = (sig_norm_quant) * (max_level-min_level) / (quant_level - 1) + min_level
    #print("Quant", sig_quant)
    if(val==1):
        return sig_norm_quant
    else:
        return sig_quant
```

In [39]:
```
# Variable declarations
dt = 0.01 # delta for spacing between vector values
A = 1     # Amplitude of the input signal
bits = 3  # number of bits per sample
Fm = 1    # Frequency of the input signal

# Creating a vector for time
time = np.arange(0, 10, dt)

# Creating a vector for input signal against time vector
x_sin = np.sin(Fm * time)
#x_cos = np.cos(2*time)
x_axis = np.zeros_like(time)
sig = A * x_sin # Input signal

# Plotting the input signal
fig = plt.figure(figsize=(16, 8))
fig.suptitle('Input signal for PCM', fontsize=25)
plt.xlabel('Time axis', fontsize=20)
plt.ylabel('Amplitude axis', fontsize=20)
plt.plot(time, sig, 'b')
plt.plot(time, x_axis, 'r')
plt.grid()
```

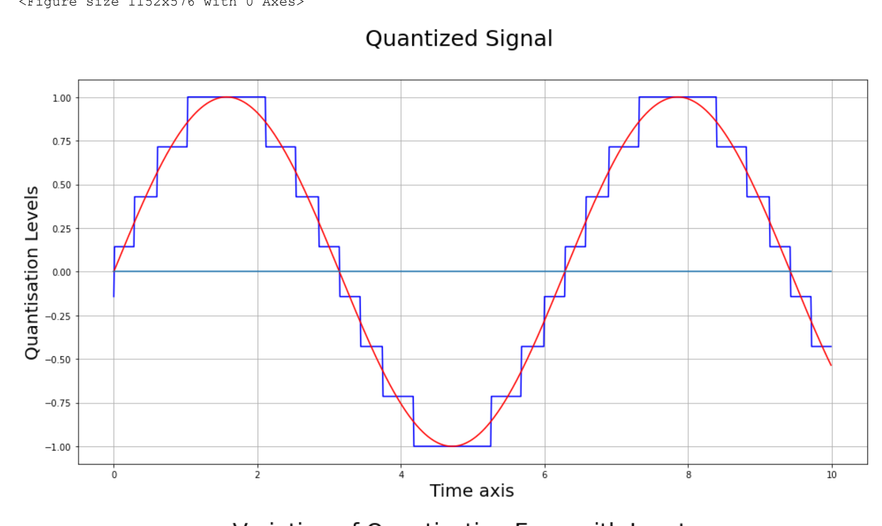


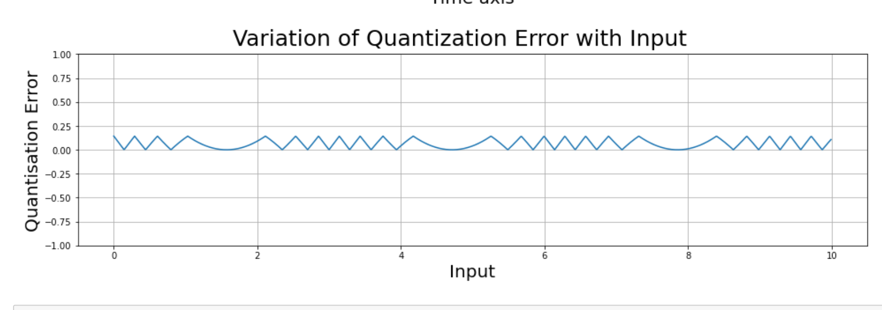
In [40]:
```
# Sampling of the input signal
samples = 100
Ts = 10
Fs = 1/Ts

samp_sig = sig[:: Ts]
time_ = time[::Ts]
fig = plt.figure(figsize=(16, 8))
fig.suptitle('Sampled Input Signal', fontsize=25)
plt.xlabel('Time axis', fontsize=20)
plt.ylabel('Amplitude axis', fontsize=20)
plt.stem(time_, samp_sig, 'r')
#plt.plot(time, sig, 'b')
plt.grid()
```



In [50]:
```
# Uniform Quantization of the input signal
num = uniformQuantization(sig, np.min(sig), np.max(sig), bits)

# Quantization Error vector
Q_error = abs(sig - num)

# Plotting the Quantized Signal with input signal for reference.
plt.figure(figsize=(16, 8))
fig = plt.figure(figsize=(16, 8))
fig.suptitle('Quantized Signal', fontsize=25)
plt.xlabel('Time axis', fontsize=20)
plt.ylabel('Quantisation Levels', fontsize=20)
plt.plot(time, num, 'b')
plt.plot(time, sig, 'r')
plt.plot(time, x_axis)
plt.grid()
plt.show()

# Plotting the Quantization Error for the input signal.
fig_ = plt.figure(1, figsize=(16, 4))
fig_.suptitle('Variation of Quantization Error with Input', fontsize=25)
plt.xlabel('Input', fontsize=20)
plt.ylabel('Quantisation Error', fontsize=20)
plt.plot(time, Q_error)
x1,x2,y1,y2 = plt.axis()
plt.axis((x1,x2,-1, 1))
plt.grid()
plt.show()
```

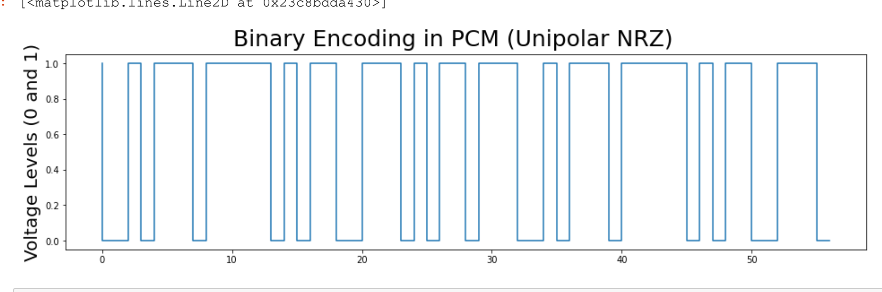
<Figure size 1152x576 with 0 Axes>





In [44]:
```
# Getting the quantisation levels for the input signal
sig_qnt = uniformQuantization(sig, val=1)  # xq , quantized signal

enc = ''
prev = ''
for i in sig_qnt:
    if not(i==prev):
        enc +=(decimalToBinary(int(i)))
    prev = i
#print(enc)

sig_encoded = list() # list to contain the encoded signal

for bin_ in enc:
    if (bin_ == '1'):
        sig_encoded.append(1)
    elif(bin_ == '0'):
        sig_encoded.append(0)

# Plotting the Binary Encoded Signal for PCM
fig__ = plt.figure(figsize=(16, 4))
fig__.suptitle('Binary Encoding in PCM (Unipolar NRZ)', fontsize=25)
plt.xlabel('', fontsize=20)
plt.ylabel('Voltage Levels (0 and 1)', fontsize=20)
plt.plot(sig_encoded, drawstyle='steps')
```

Out[44]: [<matplotlib.lines.Line2D at 0x23c8bdda430>]



In [ ]:

In [ ]:

In [ ]:

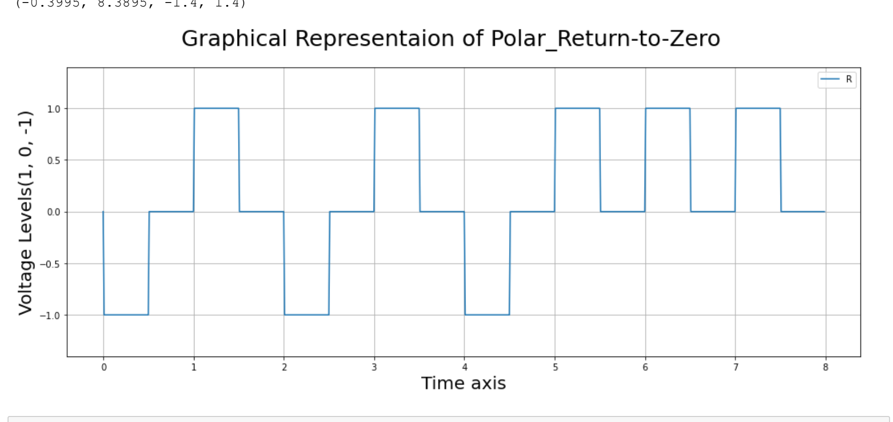
Line Coding (Assginment-II)

Polar RZ (Return to Zero)

In [3]:
```python
# Taking input from user of binary data to represent as Polar-Return-to-Zero.
binary_data = np.array(list(map(int, input("Enter the binary data to represent using Polar RZ: \n=> ").
split())))

# Declaring the variables.
bit_count = 1
node = 0
half = 0.5
dx = 0.01
time = np.arange(0, len(binary_data), dx)

# Converting the zeros in binary-data from input to -1.
for index, bit in enumerate(binary_data):
    if (bit == 0):
        binary_data[index] = -1
    else:
        binary_data[index] = 1

# Printing the converted data.
print("\nConverted the zeros in binary-data from input to -1.")
print(binary_data)

# Generating an empty vector of same shape as time.
y = np.empty_like(time)

# Converting binary to Return-to-Zero vector.
y[0] = 0
for index in range(1, len(time)):
    if (time[index]>= node and time[index] <= half):
        y[index] = binary_data[bit_count-1]
    elif(time[index] > half and time[index] < (half + 0.5)):
        y[index] = 0
    else:
        bit_count = bit_count + 1
        node = node + 1
        half = half + 1

y = np.where(y > 1, 0, y)

# Setting the labels for the Graph
fig = plt.figure(figsize=(16,6))
fig.suptitle('Graphical Representaion of Polar_Return-to-Zero', fontsize=25)
plt.xlabel('Time axis', fontsize=20)
plt.ylabel('Voltage Levels(1, 0, -1)', fontsize=20)

# Plotting the graph
plt.plot(time, y)
plt.legend('RZ')
plt.grid()

x1,x2,y1,y2 = plt.axis()
plt.axis((x1,x2,-1.4, 1.4))
```

```
Enter the binary data to represent using Polar RZ:
=>  0 1 0 1 0 1 1 1

Converted the zeros in binary-data from input to -1.
[-1  1 -1  1 -1  1  1  1]
```

Out[3]: (-0.3995, 8.3895, -1.4, 1.4)



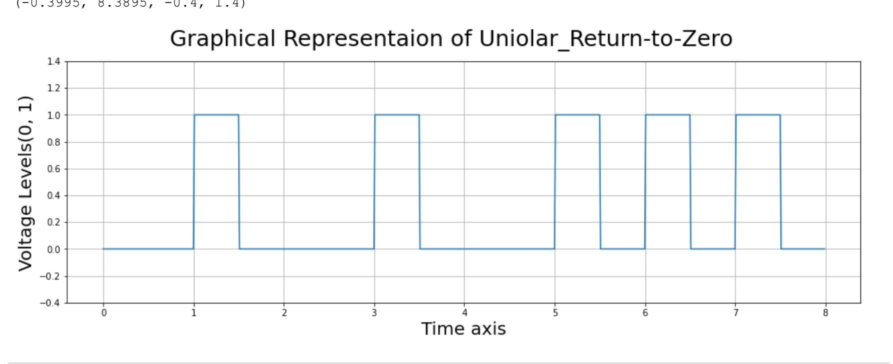
In [ ]:

Unipolar RZ (Return to Zero)

In [4]:
```python
# Taking input from user of binary data to represent as Unipolar-Return-to-Zero.
binary_data = np.array(list(map(int, input("Enter the binary data to represent using Unipolar RZ: \n=>
").split())))

# Declaring the variables.
bit_count = 1
node = 0
half = 0.5
dx = 0.01
time = np.arange(0, len(binary_data), dx)


# Printing the converted data.
print("\nData received from Input as Binary values: ")
print(binary_data)

# Generating an empty vector of same shape as time.
y = np.empty_like(time)

# Converting binary to Return-to-Zero vector.
y[0] = 0
for index in range(1, len(time)):
    if (time[index]>= node and time[index] <= half):
        y[index] = binary_data[bit_count-1]
    elif(time[index] > half and time[index] < (half + 0.5)):
        y[index] = 0
    else:
        bit_count = bit_count + 1
        node = node + 1
        half = half + 1

y = np.where(y > 1, 0, y)

# Setting the labels for the Graph
fig = plt.figure(figsize=(16,5))
fig.suptitle('Graphical Representaion of Uniolar_Return-to-Zero', fontsize=25)
plt.xlabel('Time axis', fontsize=20)
plt.ylabel('Voltage Levels(0, 1)', fontsize=20)

# Plotting the graph
plt.plot(time, y)
plt.grid()

x1,x2,y1,y2 = plt.axis()

plt.axis((x1,x2,-0.4, 1.4))
```

```
Enter the binary data to represent using Unipolar RZ:
=> 0 1 0 1 0 1 1 1

Data received from Input as Binary values:
[0 1 0 1 0 1 1 1]
```

Out[4]: (-0.3995, 8.3895, -0.4, 1.4)



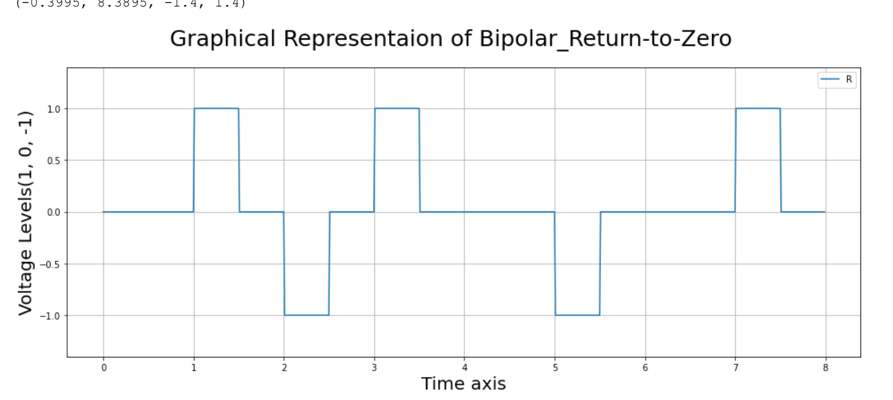
In [ ]:

Bipolar RZ (Return to Zero)

In [5]:
```python
# Taking input from user of binary data to represent as Bipolar-Return-to-Zero.
binary_data = np.array(list(map(int, input("Enter the binary data to represent using Bipolar RZ: \n=> "
).split())))

# Declaring the variables.
bit_count = 1
node = 0
half = 0.5
flag = 1
dx = 0.01
time = np.arange(0, len(binary_data), dx)

# Converting the binary-data from input to desired Bipolar values .
for index, bit in enumerate(binary_data):
    if (bit == 1): # if bin_data is 1
        if (flag == 1): # swap consecutive ones with one & negative one.
            binary_data[index] = 1
            flag = -1 # Flag set to -1(Negative one)
        else:
            binary_data[index] = -1
            flag = 1


# Printing the converted data.
print("\nConverted the zeros in binary-data from input to desired Bipolar values:")
print(binary_data)

# Generating an empty vector of same shape as time.
y = np.empty_like(time)

# Converting binary to Return-to-Zero vector.
y[0] = 0
for index in range(1, len(time)):
    if (time[index]>= node and time[index] <= half):
        y[index] = binary_data[bit_count-1]
    elif(time[index] > half and time[index] < (half + 0.5)):
        y[index] = 0
    else:
        bit_count = bit_count + 1
        node = node + 1
        half = half + 1

y = np.where(y > 1, 0, y)

# Setting the labels for the Graph
fig = plt.figure(figsize=(16,6))
fig.suptitle('Graphical Representaion of Bipolar_Return-to-Zero', fontsize=25)
plt.xlabel('Time axis', fontsize=20)
plt.ylabel('Voltage Levels(1, 0, -1)', fontsize=20)

# Plotting the graph
plt.plot(time, y)
plt.legend('RZ')
plt.grid()

x1,x2,y1,y2 = plt.axis()
plt.axis((x1,x2,-1.4, 1.4))
```

```
Enter the binary data to represent using Bipolar RZ:
=> 0 1 1 1 0 1 0 1

Converted the zeros in binary-data from input to desired Bipolar values:
[ 0  1 -1  1  0 -1  0  1]
```

Out[5]: (-0.3995, 8.3895, -1.4, 1.4)



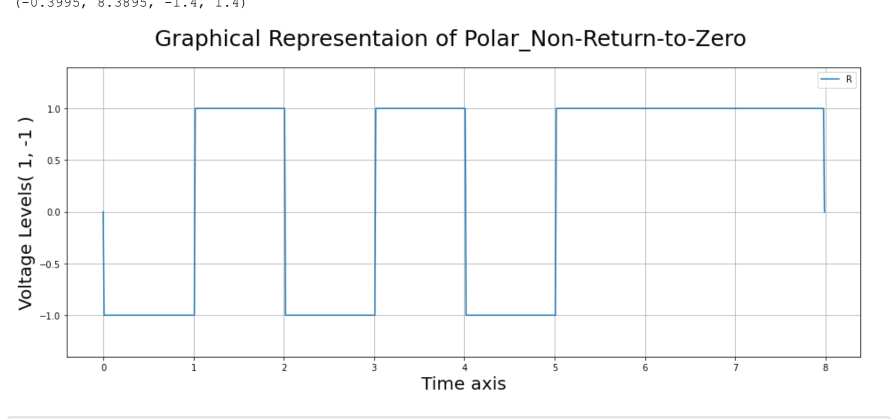
In [ ]:

Polar NRZ (Non-Return to Zero)

In [6]:
```python
# Taking input from user of binary data to represent as Polar-Non-Return-to-Zero.
binary_data = np.array(list(map(int, input("Enter the binary data to represent using Polar NRZ: \n=> ")
.split())))

# Declaring the variables.
bit_count = 1

dx = 0.01
time = np.arange(0, len(binary_data), dx)

# Converting the zeros in binary-data from input to desired Polar format.
for index, bit in enumerate(binary_data):
    if (bit == 0):
        binary_data[index] = -1
    else:
        binary_data[index] = 1

# Printing the converted data.
print("\nConverted the zeros in binary-data from input to -1(desired Polar format).")
print(binary_data)

# Generating an empty vector of same shape as time.
y = np.empty_like(time)

# Converting binary to Non-Return-to-Zero vector.
y[0] = 0
for index in range(1, len(time)):
    if (time[index] <= bit_count):
        y[index] = binary_data[bit_count-1]
    else:
        y[index] = binary_data[bit_count-1]
        bit_count = bit_count + 1
y[-1] = 0
y = np.where(y > 1, 0, y)

# Setting the labels for the Graph
fig = plt.figure(figsize=(16,6))
fig.suptitle('Graphical Representaion of Polar_Non-Return-to-Zero', fontsize=25)
plt.xlabel('Time axis', fontsize=20)
plt.ylabel('Voltage Levels( 1, -1 )', fontsize=20)

# Plotting the graph
plt.plot(time, y)
plt.legend('RZ')
plt.grid()

x1,x2,y1,y2 = plt.axis()
plt.axis((x1,x2,-1.4, 1.4))
```

```
Enter the binary data to represent using Polar NRZ:
=> 0 1 0 1 0 1 1 1

Converted the zeros in binary-data from input to -1(desired Polar format).
[-1  1 -1  1 -1  1  1  1]
```

Out[6]: (-0.3995, 8.3895, -1.4, 1.4)



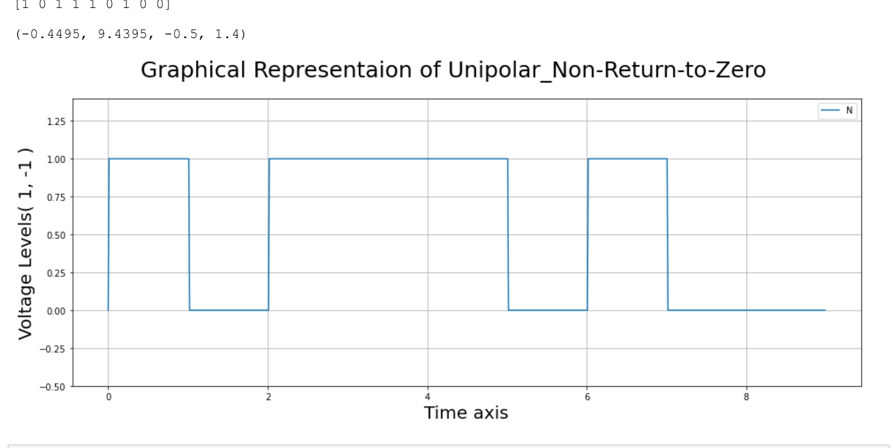
In [ ]:

Unipolar NRZ (Non-Return to Zero)

In [7]:
```python
# Taking input from user of binary data to represent as Unipolar-Non-Return-to-Zero.
binary_data = np.array(list(map(int, input("Enter the binary data to represent using Unipolar NRZ: \n=>
").split())))

# Declaring the variables.
bit_count = 1

dx = 0.01
time = np.arange(0, len(binary_data), dx)

# Converting the zeros in binary-data from input to desired Polar format.
for index, bit in enumerate(binary_data):
    if (bit == 1):
        binary_data[index] = 1
    else:
        binary_data[index] = 0

# Printing the converted data.
print("\nConverted the zeros in binary-data from input to desired Unipolar format.")
print(binary_data)

# Generating an empty vector of same shape as time.
y = np.empty_like(time)

# Converting binary to Non-Return-to-Zero vector.
y[0] = 0 # Setting first element zero to start the graph from x-axis.
for index in range(1, len(time)):
    if (time[index] <= bit_count):
        y[index] = binary_data[bit_count-1]
    else:
        y[index] = binary_data[bit_count-1]
        bit_count = bit_count + 1
y[-1] = 0 # Setting last element zero to end the graph on x-axis.
y = np.where(y > 1, 0, y)

# Setting the labels for the Graph
fig = plt.figure(figsize=(16,6))
fig.suptitle('Graphical Representaion of Unipolar_Non-Return-to-Zero', fontsize=25)
plt.xlabel('Time axis', fontsize=20)
plt.ylabel('Voltage Levels( 1, -1 )', fontsize=20)

# Plotting the graph
plt.plot(time, y)
plt.legend('NRZ')
plt.grid()

x1,x2,y1,y2 = plt.axis()
plt.axis((x1,x2,-0.5, 1.4))
```

```
Enter the binary data to represent using Unipolar NRZ:
=> 1 0 1 1 1 0 1 0 0

Converted the zeros in binary-data from input to -1(desired Polar format).
[1 0 1 1 1 0 1 0 0]
```

Out[7]: (-0.4495, 9.4395, -0.5, 1.4)



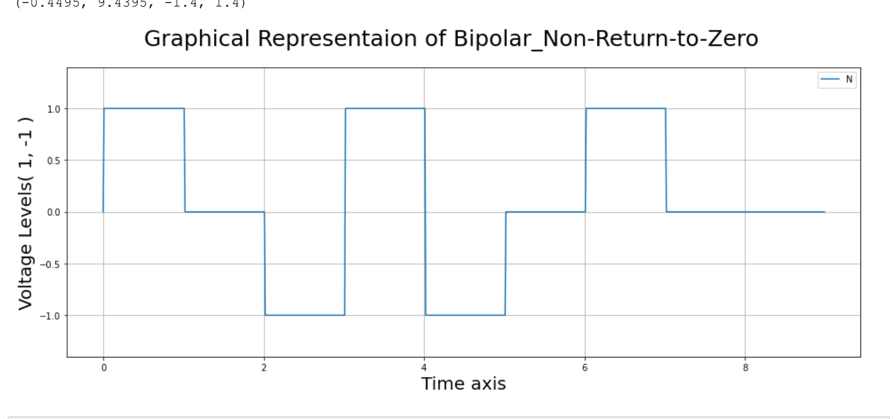
In [ ]:

Bipolar NRZ (Non-Return to Zero)

In [9]:
```python
# Taking input from user of binary data to represent as Bipolar-Non-Return-to-Zero.
binary_data = np.array(list(map(int, input("Enter the binary data to represent using Bipolar NRZ: \n=>
").split())))

# Declaring the variables.
bit_count = 1
flag = 1
dx = 0.01
time = np.arange(0, len(binary_data), dx)

# Converting the zeros in binary-data from input to desired Bipolar format.
for index, bit in enumerate(binary_data):
    if (bit == 1):
        if (flag == 1):
            binary_data[index] = 1
            flag = -1
        else:
            binary_data[index] = -1
            flag = 1
    else:
        binary_data[index] = 0

# Printing the converted data.
print("\nConverted the zeros in binary-data from input to desired Bipolar format:")
print(binary_data)

# Generating an empty vector of same shape as time.
y = np.empty_like(time)

# Converting binary to Non-Return-to-Zero vector.
y[0] = 0 # Setting first element zero to start the graph from x-axis.
for index in range(1, len(time)):
    if (time[index] <= bit_count):
        y[index] = binary_data[bit_count-1]
    else:
        y[index] = binary_data[bit_count-1]
        bit_count = bit_count + 1
y[-1] = 0 # Setting last element zero to end the graph on x-axis.
y = np.where(y > 1, 0, y)

# Setting the labels for the Graph
fig = plt.figure(figsize=(16,6))
fig.suptitle('Graphical Representaion of Bipolar_Non-Return-to-Zero', fontsize=25)
plt.xlabel('Time axis', fontsize=20)
plt.ylabel('Voltage Levels( 1, -1 )', fontsize=20)

# Plotting the graph
plt.plot(time, y)
plt.legend('NRZ')
plt.grid()

x1,x2,y1,y2 = plt.axis()
plt.axis((x1,x2,-1.4, 1.4))
```

```
Enter the binary data to represent using Bipolar NRZ:
=> 1 0 1 1 1 0 1 0 0

Converted the zeros in binary-data from input to desired Bipolar format:
[ 1  0 -1  1 -1  0  1  0  0]
```

Out[9]: (-0.4495, 9.4395, -1.4, 1.4)

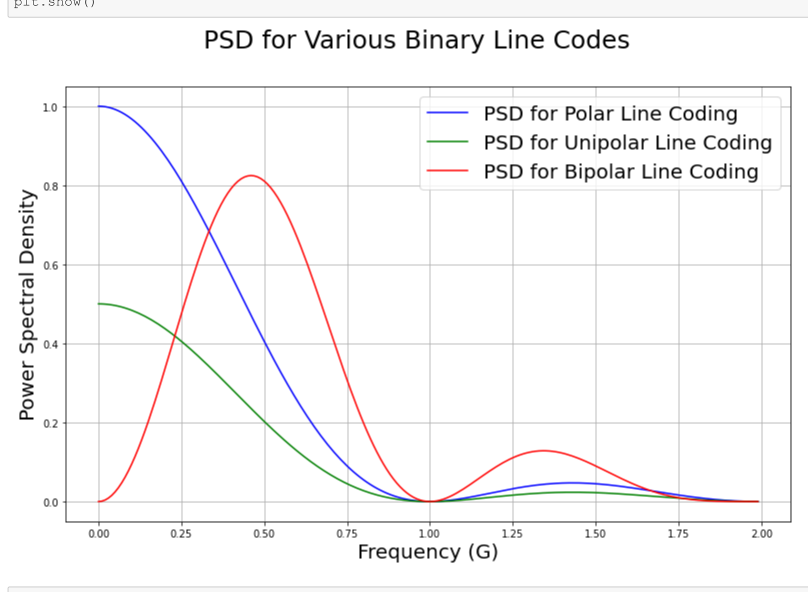


In [ ]:

`Power_Spectral_Density_of_Line_coding`

Comparison of Power Spectral Density of Polar, Unipolar, Bipolar

In [12]:
```python
# Initial Parameter
Rb = 1        # The data rate, Rb
Tb = 1/Rb     # The bit period, Tb
dx = 0.01     # dx is step size, which eventually is the factor
              # deciding the smoothness of curve.

G = np.arange(0, 2*Rb, dx*Rb) # Frequency
t = G * Tb

## PSD of Polar NRZ line coded signal
PSD_1 =  (Tb * sinc(t) * sinc(t))

## PSD of Unipolar NRZ line coded signal
PSD_2 = (Tb * sinc(t) * sinc(t))/2

## PSD of Bipolar NRZ line coded signal
PSD_3 = (Tb*((sinc(t/2))**2) * (sin(pi * t))**2)

## Labels for legend
psd1 = "PSD for Polar Line Coding"
psd2 = "PSD for Unipolar Line Coding"
psd3 = "PSD for Bipolar Line Coding"

# Labeling the plots
fig = plt.figure(figsize=(13,8))
fig.suptitle('PSD for Various Binary Line Codes', fontsize=25)
plt.xlabel('Frequency (G)', fontsize=20)
plt.ylabel('Power Spectral Density', fontsize=20)

# Plotting the PSDs of various line codes
plt.plot(G, PSD_1, 'b')
plt.plot(G, PSD_2, 'g')
plt.plot(G, PSD_3, 'r')
plt.legend([psd1, psd2, psd3], fontsize=20)

plt.grid()
plt.show()
```



In [ ]: