```python
# Author: Ramesh Pai
# Affiliation: 201104047, TE-E&TC Engg, Sem.5, 2021-22, GCE.

# importing python modules
import matplotlib.pyplot as plt
import numpy as np
from scipy.fft import fft, fftshift, ifft, ifftshift

# vm: Amplitude of Message Signal
# fm: fequency(Hz) of Message Signal
# bits: number of bits per sample

def values(vm, fm, bits):

    fs = 600*fm #sampling frequency(fs >= 2*fm(message signal frequency))
    dt = 1/fs #sample time interval or time-steps for time-domain signal
    t = np.arange(0, 0.1, dt) #time indices for time-domain signal
    n = np.size(t) #number of samples
    df = fs/n #frquency interval or frequency-steps for frequency-spectrum
    f = np.arange(-fs/2, fs/2, df) #frequency indices for frquency-spectrum

    # plot1: Message Signal v/s Time
    v_m = vm*np.sin(2*np.pi*fm*t) #message signal
    plt.subplot(2, 2, 1)
    plt.plot(t, v_m)
    plt.title("Message Signal", loc='left')
    plt.xlabel("t(sec)", loc='right')
    plt.ylabel("v_m(Volts)")

    # plot2: Quantized Signal v/s Time
    v_m_shifted = v_m + vm #shifted message signal to make quantization easier

    # QUANTIZING:
    vH = max(v_m_shifted) #maximum voltage
    vL = min(v_m_shifted) #minimum voltage
    L = 2 ** bits #number of levels / intervals
    stepSize = (vH - vL)/L #stepsize

    qLevelList = [] #list for storing Quantization levels
    for x in range(L):
        x *= stepSize
        qLevelList.append(x)

    qSignal = np.zeros(len(v_m_shifted)) #storing quantized signal values in array of zeroes
    for x in range(len(v_m_shifted)):
        for y in qLevelList:
            if ((v_m_shifted[x] >= y) and (v_m_shifted[x] <= y + stepSize)):
                qSignal[x] = y

    plt.subplot(2, 2, 2)
    plt.plot(t, qSignal, label = 'Quantized Signal')
    plt.plot(t, v_m_shifted, linestyle = 'dotted', color = 'r', label = 'Original Signal')
    plt.title("Quantized Signal", loc='left')
    plt.xlabel("t(sec)", loc='right')
    plt.ylabel("qSignal(Volts)")
    plt.legend()
    plt.grid()
```

```python

    # plot3: Reconstructed Signal v/s Time

    # ENCODING:
    enCodedList = [] #list of encoded quantization values
    for x in qLevelList:
        codeNum = qLevelList.index(x) #assigning code numbers
        deciToBin = bin(codeNum)[2:]
        if len(deciToBin) < bits:
            deciToBin = "0"*(bits - len(deciToBin)) + deciToBin
        enCodedList.append(deciToBin)

    # DECODING:
    deCodedList = [] #list of decoded values
    for x in enCodedList:
        decNum = enCodedList.index(x)
        x = qLevelList[decNum]
        deCodedList.append(x)

    print(enCodedList, deCodedList) #printing encoded & decoded values

    # RECONSTUCTION:
    v_m_reconstructed  = qSignal - vm #reconstruction
    plt.subplot(2, 2, 3)
    plt.plot(t, v_m_reconstructed)
    plt.title("Reconstructed Signal", loc='left')
    plt.xlabel("t(sec)", loc='right')
    plt.ylabel("v_m_reconstructed(Volts)")

    # plot4: Recovered Signal v/s Time

    #FILTER DESIGN:
    spec_vm_rec = fftshift(fft(v_m_reconstructed)) #FFT of Reconstructed Signal(Complex in nature).

    filter = [] #List having array of 0's and 1's
    for z in f:
        if z < (fm + 10) and z > -(fm + 10):
            z = 1
            filter.append(z) #Assigning 1 to frequencies below Cutoff
        else:
            z = 0
            filter.append(z) #Assigning 0 to frequencies above Cutoff

    v_m_recovered = ifft(ifftshift(filter * spec_vm_rec)) #Inverse FFT to get Recovered Signal
    plt.subplot(2, 2, 4)
    plt.plot(t, v_m_recovered, label="Recovered Signal(post LPF)")
    plt.plot(t, v_m, linestyle="dotted", label="Original Signal")
    plt.title("Recovered Signal", loc='left')
    plt.xlabel("t(sec)", loc='right')
    plt.ylabel("v_m_recovered(Volts)")
    plt.legend()
    plt.show()

values(4, 20, 3) #assigning values to parameters
```