EE495 HW 2

Kevin Morales

Problem 02-01

In problem 01, we're implementing one functionality of an ADC. Deriving the quantized values xQ, without the bit code. By defining a function capable of handling arrays of values, we can eventually sample audio files in a range of bit quantizers.

Part A

In Part A, we're tasked with defining the function quantize395(x,B,r). This function is supposed to quantize to B bits by rounding. The scale of the ADC is defined as the min and max variables in vector r. By defining different boundaries in r, we can set decide if we're using offset or standard binary.

In [207…

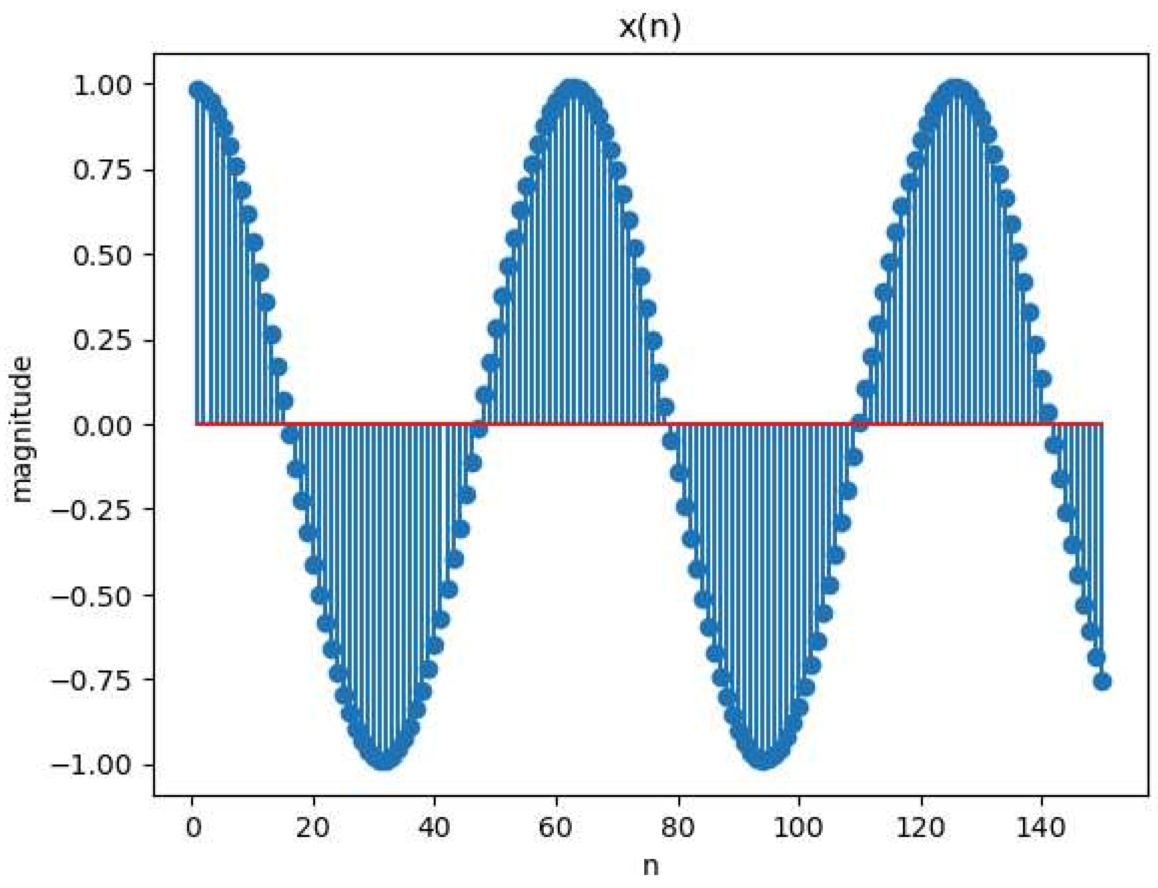
```
import matplotlib.pyplot as plt
import numpy as np
import math
def quantize395(x,B,r):
    x = np.array(x,dtype = float)
    [mn, mx] = r
    q = (mx-mn) / (2 **B)
    m = np.round(x / q)
    xq = m * q

    xq = np.where(xq >= mx, mx, xq)
    if mn < 0 and mx > 0:
        xq = np.where(xq == mx, xq-q, xq)
    xq = np.where(xq <mn, mn, xq)
    return xq

B = 4
R = [0,6]
x = [1.75, -2.3, 4.9, -2.215]
print("Natural Binary:\n")
xq = quantize395(x,B,R)
for original, quantized in zip(np.atleast_1d(x), np.atleast_1d(xq)):
    print(f"The input was x = {original}.  The quantized value is xQ = {quantized}.

R = [-3,3]
x = [1.75, -2.3, 4.9, -2.215]
print("\nOffset Binary:\n")
xq = quantize395(x,B,R)
for original, quantized in zip(np.atleast_1d(x), np.atleast_1d(xq)):
    print(f"The input was x = {original}.  The quantized value is xQ = {quantized}.
Natural Binary:

The input was x = 1.75.  The quantized value is xQ = 1.875.
The input was x = -2.3.  The quantized value is xQ = 0.0.
The input was x = 4.9.  The quantized value is xQ = 4.875.
The input was x = -2.215.  The quantized value is xQ = 0.0.

Offset Binary:

The input was x = 1.75.  The quantized value is xQ = 1.875.
The input was x = -2.3.  The quantized value is xQ = -2.25.
The input was x = 4.9.  The quantized value is xQ = 2.625.
The input was x = -2.215.  The quantized value is xQ = -2.25.
```

As you can see from the results above, our function is capable of handling arrays of values and quantizing them. In this specific example a 4 bits we're used to quantize the inputs. The inputs were quantized once, with natural binary, by defining our range as [0,6]; and again with binary offset, by defining our range as [-3,3]. On the natural binary outputs, we see no negative values. All negatives are made into zeros, and 4.9 is able to have minimal error as 4.875. When looking at binary offset, we can see that negative values can be represented, but 4.9 has a substantial error of 2.275. Our quantizer function worked as expected.

Part B

In part B, we defined and plotted signal x(n), a cosine function, with a magnitude ranging from .99 to -.99.

In [90]:
```python
n=np.arange(1,151)
x = 0.99*np.cos(n/10)
plt.stem(n,x)
plt.title("x(n)")
plt.ylabel("magnitude")
plt.xlabel("n")
plt.show()
```



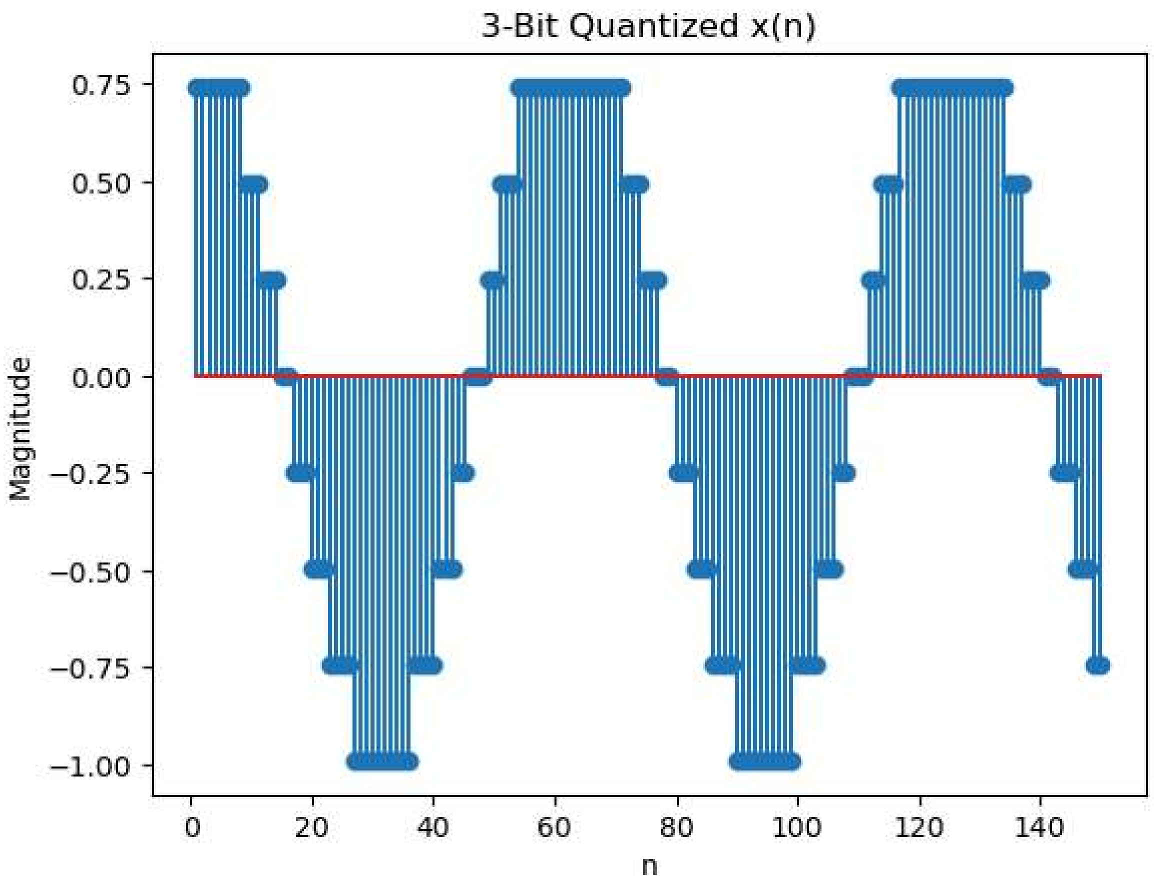
The signal x(n) can be seen above, as expected a cosine function clearly visible with enough samples not quantized yet.

Part C

In part C we're to run the cosine function x(n) defined in part b, through our quantize function with a 3 bit quantizer.

In [211…

```
B = 3
R = [-0.99, 0.99]
n=np.arange(1,151)
x = 0.99*np.cos(n/10)
xq3 = quantize395(x,B,R)

plt.stem(n,xq3)
plt.title("3-Bit Quantized x(n)")
plt.ylabel('Magnitude')
plt.xlabel('n')
plt.show()
```



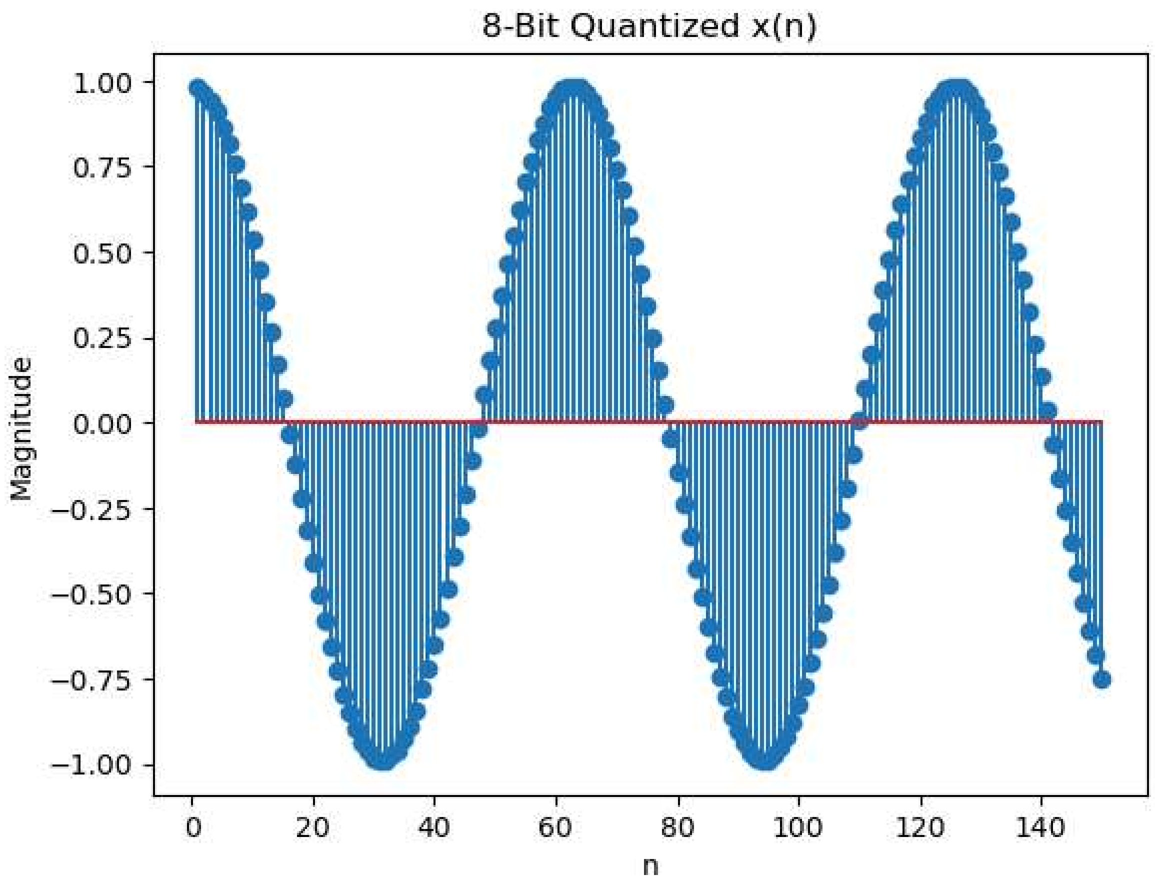
Seen above is the result of quantizing the cosine function with 3 bits. As expected the result only has 8 quantization levels. Three levels are above 0, one is zero, and the last four are below zero.

Part d

The final part of problem 3 is to repeat part c with an 8-bit quantizer instead.

In [213…

```
B = 8
xq4 = quantize395(x,B,R)
plt.stem(n,xq4)
plt.title("8-Bit Quantized x(n)")
plt.ylabel('Magnitude')
plt.xlabel('n')
plt.show()
```



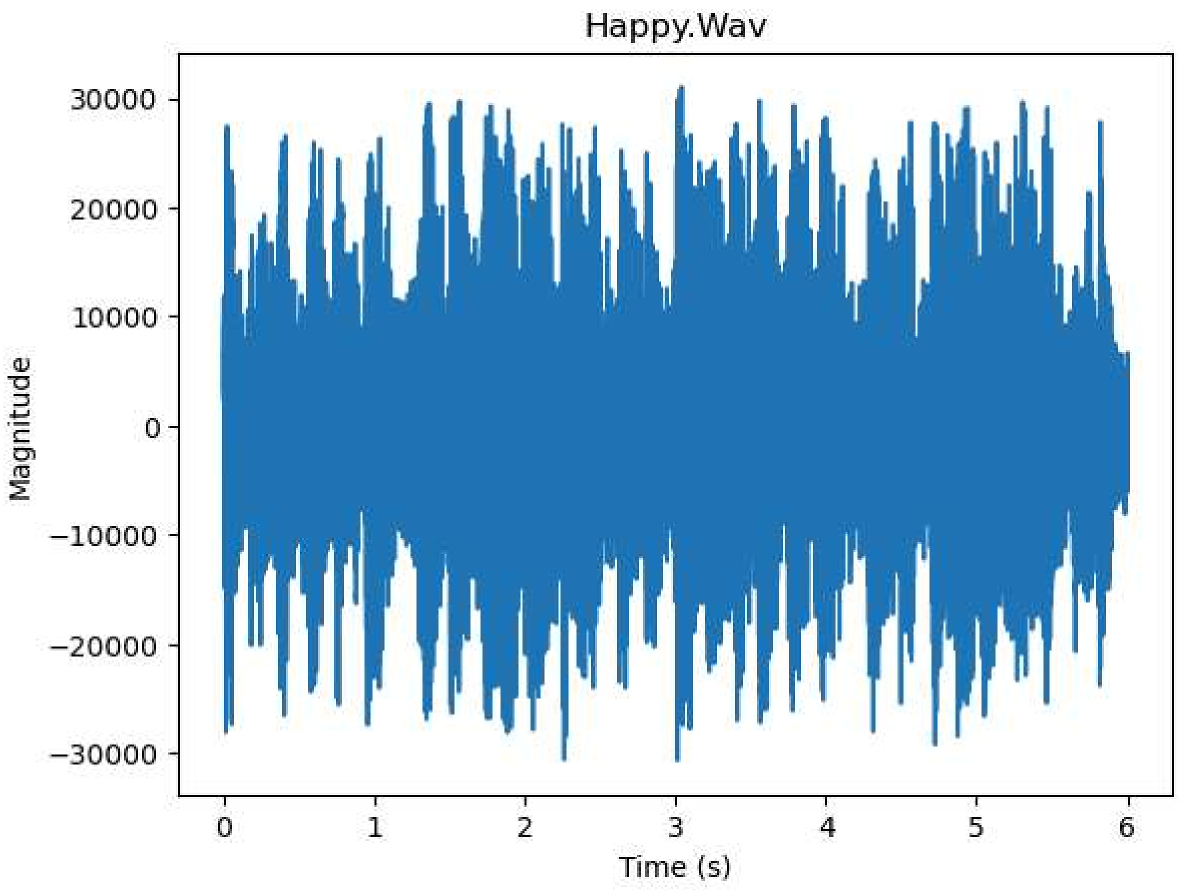
Seen above is the resulting stem plot. With 8 bits, there are 256 quantization levels, with this many levels, there are no visible distortions to the signals' magnitude. There are of course some distortions with the processing that took place, but not to the naked eye.

Problem 02-02

Part A

In [215…

```
import scipy
fs, happy = scipy.io.wavfile.read('Happy.wav')
Ts = 1 / fs
t = np.arange(0 , len(happy)) * Ts
plt.plot(t,happy)
plt.title("Happy.Wav")
plt.ylabel("Magnitude")
plt.xlabel("Time (s)")
plt.show()
```



The happy.wav file was successfully read and plotted against time, showing a highly complex waveform. Unlike the previous example of a single sinusoid, we can see many frequency components and amplitude variations.

Part B

In [221…

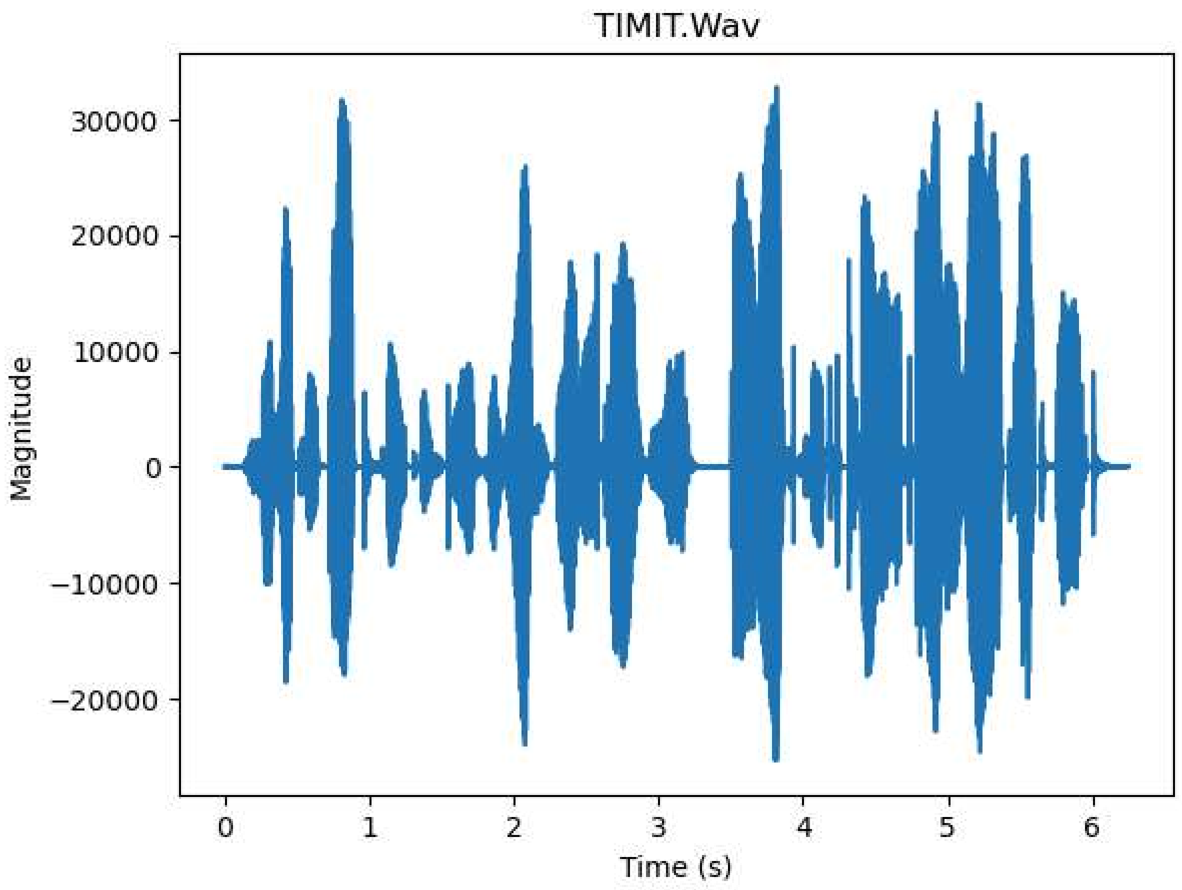
```
R = [-32768, 32767] # Range of 16 bit integer values
for B in range(1,17):
    happyq = quantize395(happy, B, R)
    #scipy.io.wavfile.write('Happy'+str(B)+'.wav', fs, happyq.astype('int16'))
```

Part C

Part D

In [222…

```
fs, timit = scipy.io.wavfile.read('TIMIT.wav')
Ts = 1 / fs
t = np.arange(0 , len(timit)) * Ts
plt.plot(t,timit)
plt.title("TIMIT.Wav")
plt.ylabel("Magnitude")
plt.xlabel("Time (s)")
plt.show()
```



In [223…
```
for B in range(1,13):
    timitq = quantize395(timit, B, R)
    #scipy.io.wavfile.write('TIMIT'+str(B)+'.wav', fs, timitq.astype('int16'))
```

Problem 02-03

In the final problem, we're to remake the quantizing function. This time, we're going to define the function using successive approximation. By using this method, we're able to get the bit values associated with our quantized values. Using the bits, we're also able to define a DAC function to get an output Xq based on those bit values.

Part A

In the first part, we're to define the ADC function. This function should be able to handle truncating and rounding as requested, a range, and number of bits. By using successive approximation, we're able to individually set the bits of our output. Finally reversing the first bit, we're able to get the 2's complement.

In [224…
```
def ADC(x,R,B,Qtype): # Qtype is 'round' or 'truncate'
    orig = x
    num_bin = [0] *B
    Q = R / (2**B)
    mn = - R / 2
    mx = R / 2
    if(Qtype == 'round'):
        print("Rounding: ")
        x += Q * 0.5
    elif(Qtype == 'truncate'):
            print("Truncating: ")

    for i in range(0, B):
        print
        num_bin[i] = 1
        m = int(''.join(str(x) for x in num_bin), base = 2) - (2**B // 2)
        xQ = m * Q

        if(x < xQ):
            num_bin[i] = 0


    if num_bin[0] == 0:
        num_bin[0] = 1
    else:
        num_bin[0] = 0
    print(f"The input was: {orig}. The bit code is: {num_bin}")
    return num_bin

b1 = ADC(1.75, 6, 4, 'round')
print(b1)
b2 = ADC(-2.3, 6, 4, 'round')
b3 = ADC(6.5, 6, 4, 'round')
b4 = ADC(-4, 6, 4, 'round')
b5 = ADC(1.75, 6, 4, 'truncate')
b6 = ADC(-2.3, 6, 4, 'truncate')
b7 = ADC(6.5, 6, 4, 'truncate')
b8 = ADC(-4, 6, 4, 'truncate')
    #num_bin = np.asarray([int(x) for x in bin(num_dec)[2:]]
    #num_dec = int(''.join(str(x) for x in num_bin), base = 2)
```

```
Rounding:
The input was: 1.75. The bit code is: [0, 1, 0, 1]
[0, 1, 0, 1]
Rounding:
The input was: -2.3. The bit code is: [1, 0, 1, 0]
Rounding:
The input was: 6.5. The bit code is: [0, 1, 1, 1]
Rounding:
The input was: -4. The bit code is: [1, 0, 0, 0]
Truncating:
The input was: 1.75. The bit code is: [0, 1, 0, 0]
Truncating:
The input was: -2.3. The bit code is: [1, 0, 0, 1]
Truncating:
The input was: 6.5. The bit code is: [0, 1, 1, 1]
Truncating:
The input was: -4. The bit code is: [1, 0, 0, 0]
```

Based on the output of our code block, where we ran the function with 8 different combinations, we can confidently deduce that the ADC function is able to quantize a variety of values into their 2's complement bit representation.

Part B

In the final part of problem 2, we're to reverse the previous process, and design DAC. the inputs are simply the bits previously derived, and R the range of our ADC.

In [225…

```python
def DAC(b, R):
    num_bin = b
    B = len(b)
    q = R / (2**B)

    if(num_bin[0] == 0):
        num_bin[0] = 1
    else:
        num_bin[0] = 0

    m = int(''.join(str(x) for x in num_bin), base = 2) - (2**B // 2)
    xQ = m * q
    return xQ
print("Rounding:")
print(f"The input was b = {b1}.")
xQ1 = DAC(b1, 6)
print(f"The sample value is xQ = {xQ1}. The original value was x = 1.75")
print(f"The input was b = {b2}.")
xQ2 = DAC(b2, 6)
print(f"The sample value is xQ = {xQ2}. The original value was x = -2.3")
print(f"The input was b = {b3}.")
xQ3 = DAC(b3, 6)
print(f"The sample value is xQ = {xQ3}. The original value was x = 6.5")
print(f"The input was b = {b4}.")
xQ4 = DAC(b4, 6)
print(f"The sample value is xQ = {xQ4}. The original value was x = -4.0")

print("Truncating:")
print(f"The input was b = {b5}.")
xQ5 = DAC(b5, 6)
print(f"The sample value is xQ = {xQ5}. The original value was x = 1.75")

print(f"The input was b = {b6}.")
xQ6 = DAC(b6, 6)
print(f"The sample value is xQ = {xQ6}. The original value was x = -2.3")

print(f"The input was b = {b7}.")
xQ7 = DAC(b7, 6)
print(f"The sample value is xQ = {xQ7}. The original value was x = 6.5")

print(f"The input was b = {b8}.")
xQ8 = DAC(b8, 6)
print(f"The sample value is xQ = {xQ8}. The original value was x = -4.0")
```

```
Rounding:
The input was b = [0, 1, 0, 1].
The sample value is xQ = 1.875. The original value was x = 1.75
The input was b = [1, 0, 1, 0].
The sample value is xQ = -2.25. The original value was x = -2.3
The input was b = [0, 1, 1, 1].
The sample value is xQ = 2.625. The original value was x = 6.5
The input was b = [1, 0, 0, 0].
The sample value is xQ = -3.0. The original value was x = -4.0
Truncating:
The input was b = [0, 1, 0, 0].
The sample value is xQ = 1.5. The original value was x = 1.75
The input was b = [1, 0, 0, 1].
The sample value is xQ = -2.625. The original value was x = -2.3
The input was b = [0, 1, 1, 1].
The sample value is xQ = 2.625. The original value was x = 6.5
The input was b = [1, 0, 0, 0].
The sample value is xQ = -3.0. The original value was x = -4.0
```

The results of the ADC then DAC running our values match the tables from ICE 2.3 and 2.4. The DAC is capable of converting 2's complement bit values of any range and converting them to appropriate analog values in the range of our quantizer. The 8 combinations show the capability of the DAC processing the ADC outputs.

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```