8 pts

psf_tp1_brignone

June 19, 2020

1 Trabajo práctico 1 - PSF

Autor: Matías Brignone

1.0.1 1. Demuestre si los siguientes sistemas son LTI

1.1. $y(t) = x(t) * cos(t)$

El sistema es lineal pero no es invariante en el tiempo, debido al término adicional $cos(t)$, que no depende de un desplazamiento en el tiempo de la entrada.

1.2. $y(t) = cos(x(t))$ El sistema es invariante en el tiempo pero no es lineal.

$$x(t) \rightarrow cos(x(t)) = y(t)$$

$$ax(t) \rightarrow cos(ax(t)) \neq acos(x(t)) = ay(t)$$

1.3. $y(t) = e^{x(t)}$

El sistema no es lineal y por lo tanto no es LTI.

$$x(t) \rightarrow e^{x(t)}$$

$$ax(t) \rightarrow e^{ax(t)} \neq ae^{x(t)} = ay(t)$$

1.4. $y(t) = \frac{1}{2}x(t)$

El sistema sí es LTI ya que satisface el principio de superposición (lineal) y la salida para una determinada entrada es la misma sin importar el tiempo en el cual se aplica la entrada (invariante en el tiempo).

1.0.2 2. Ruido de cuantización

2.1 Calcule la relación señal a ruido de cuantización teórica máxima de un sistema con un ADC de 24 / 16 / 10 / 8 / 2 bits. .

Para el caso del ruido de cuantización, la relación señal ruido (SNR) está dada por SNR $= 1,76 + 6,02 * N$, siendo N la cantidad de bits.

1. $N = 24 \rightarrow$ SNR $= 146.24$ dB

2. $N = 16 \rightarrow \text{SNR} = 98.08$ dB
3. $N = 10 \rightarrow \text{SNR} = 61.96$ dB
4. $N = 8 \rightarrow \text{SNR} = 49.92$ dB
5. $N = 2 \rightarrow \text{SNR} = 13.80$ dB

2.2 Dado un sistema con un ADC de 10 bits, ¿qué técnica le permitiría aumentar la SNR? ¿En qué consiste? .

Considerando que la potencia de ruido está uniformemente distribuida en todo el espectro, realizando un **sobremuestreo** es posible mejorar la relación señal ruido del sistema, ya que la misma potencia de ruido estaría distribuida a lo largo de un espectro mayor.

1.0.3 3. Filtro antialias y reconstrucción

3.1. Calcular el filtro antialias que utilizará para su práctica y/o trabajo final y justifique su decisión. .

El objetivo al utilizar un filtro antialias es eliminar, antes de hacer el muestreo (antes de ingresar al ADC), las frecuencias superiores a fs/2, siendo fs la frecuencia muestreo.

En el trabajo final se generarán señales de 250Hz como máximo, por lo tanto se debe usar una frecuencia de muestreo de al menos 500Hz (el doble del ancho de banda de la señal a muestrear). Para tener margen suficiente y teniendo en cuenta que la atenuación en un filtro de primer orden cae lentamente, se decide establecer una frecuencia de corte de 1KHz y una frecuecia de muestreo de 10KHz, que se encuentra lo suficientemente alejada del punto de corte en 1KHz como para que la señal ya se encuentre totalmente atenuada. esta bien pedo podrias haber puesto la Fc del filtro directamente en 250hz si de seguro no hy nada de mas de 250hz. te ganas 750hz de bajada

Entonces se diseña un filtro antialias RC de primer orden con una frecuencia de corte en ese valor ($f_{corte} = 1000Hz$) y fijando el valor $R = 1k\Omega$.

$$f_{corte} = \frac{1}{2\pi RC} \rightarrow C = \frac{1}{2\pi R f_{corte}} = 0.000000159F = 159nF \approx 140nF$$

Utilizando un capacitor de 140nF, la frecuencia de corte queda ubicada en aproximadamente 1.14KHz, lo cual es un valor acetapble para el trabajo.

1.0.4 4. Generación y simulación

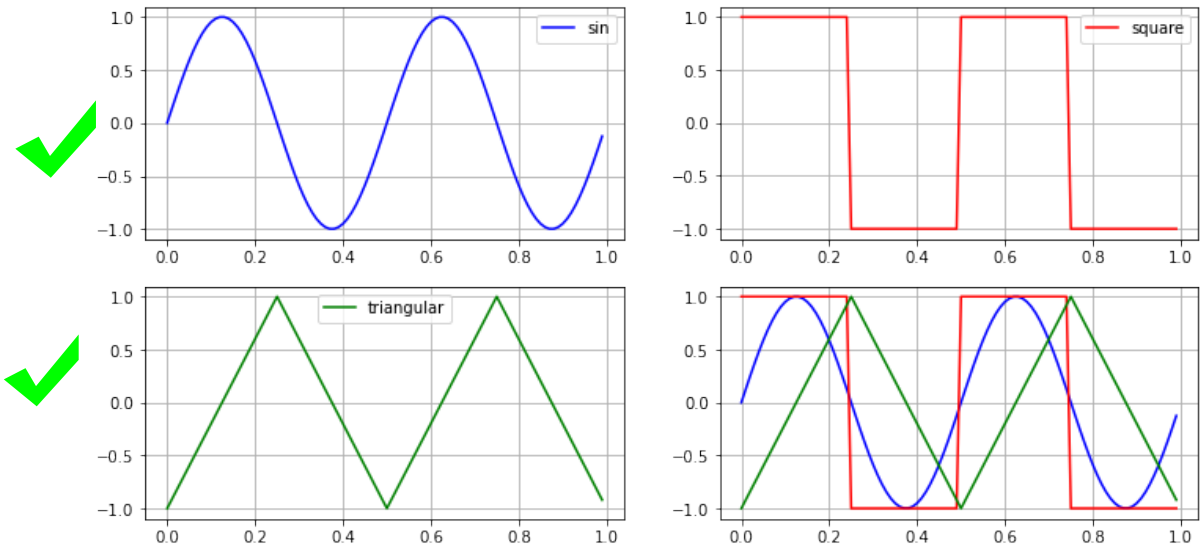
4.1. Genere un modulo o paquete con al menos las siguientes funciones: .

1. Senoidal (fs[Hz], f0[Hz], amp[0 a 1], muestras,fase [radianes])
2. Cuadrada (fs[Hz], f0[Hz], amp[0 a 1], muestras)
3. Triangular(fs[Hz], f0[Hz], amp[0 a 1], muestras)

[2]:
```
import numpy as np
import scipy.signal as sci
import matplotlib.pyplot as plt
```

```python
# onda senoidal
def sin_signal(fs, f0, amp, N, fase=0, n=None):
    """\
    fs:   frecuencia de sampleo [Hz]
    f0:   frecuencia de la senoidal [Hz]
    fase: fase de la señal [rad]
    amp:  amplitud de la señal [0 a 1]
    N:    cantidad de muestras
    n:    numero de muestra a retornar.
          Si es None devuelve todo el arreglo, caso contrario devuelve solamente
          el valor para el instante de tiempo correspondiente a esa muestra
    """
    if n is not None:
        return amp * np.sin(fase + 2 * np.pi * f0 * n * (1/fs))

    discrete_time = np.arange(0, N/fs, 1/fs)
    discrete_signal = amp * np.sin(fase + 2 * np.pi * f0 * discrete_time)

    return discrete_signal, discrete_time

# onda cuadrada
def square_signal(fs, f0, amp, N, fase=0, n=None):
    """\
    fs:   frecuencia de sampleo [Hz]
    f0:   frecuencia de la senoidal [Hz]
    fase: fase de la señal [rad]
    amp:  amplitud de la señal [0 a 1]
    N:    cantidad de muestras
    n:    numero de muestra a retornar.
          Si es None devuelve todo el arreglo, caso contrario devuelve solamente
          el valor para el instante de tiempo correspondiente a esa muestra
    """
    if n is not None:
        return amp * sci.square(fase + 2 * np.pi * f0 * n * (1/fs), duty=0.5)

    discrete_time = np.arange(0, N/fs, 1/fs)
    discrete_signal = amp * sci.square(fase + 2 * np.pi * f0 * discrete_time,␣
 ↪duty=0.5)

    return discrete_signal, discrete_time

# onda triangular
def tri_signal(fs, f0, amp, N, fase=0, n=None):
    """\
    fs:   frecuencia de sampleo [Hz]
    f0:   frecuencia de la senoidal [Hz]
```

```
    fase: fase de la señal [rad]
    amp:  amplitud de la señal [0 a 1]
    N:    cantidad de muestras
    n:    numero de muestra a retornar.
          Si es None devuelve todo el arreglo, caso contrario devuelve solamente
          el valor para el instante de tiempo correspondiente a esa muestra
    """
    if n is not None:
        return amp * sci.sawtooth(fase + 2 * np.pi * f0 * n * (1/fs), width=0.5)

    discrete_time = np.arange(0, N/fs, 1/fs)
    discrete_signal = amp * sci.sawtooth(fase + 2 * np.pi * f0 * discrete_time,␣
↪width=0.5)

    return discrete_signal, discrete_time


N    = 100
fs   = 100
amp  = 1
fase = 0
f0   = 2

sine_signal, sin_time = sin_signal(fs=fs, f0=f0, amp=amp, N=N, fase=fase)
sq_signal, sq_time = square_signal(fs=fs, f0=f0, amp=amp, N=N, fase=fase)
tri_signal, tri_time = tri_signal(fs=fs, f0=f0, amp=amp, N=N, fase=fase)

fig = plt.figure(figsize=(12,6))
fig.add_subplot(2, 2, 1)
plt.plot(sin_time, sine_signal, 'b', label="sin")
plt.grid()
plt.legend()
fig.add_subplot(2, 2, 2)
plt.plot(sq_time, sq_signal, 'r', label="square")
plt.grid()
plt.legend()
fig.add_subplot(2, 2, 3)
plt.plot(tri_time, tri_signal, 'g', label="triangular")
plt.grid()
plt.legend()
fig.add_subplot(2, 2, 4)
plt.plot(sin_time, sine_signal, 'b')
plt.plot(sq_time, sq_signal, 'r')
plt.plot(tri_time, tri_signal, 'g')
plt.grid()
plt.show()
```

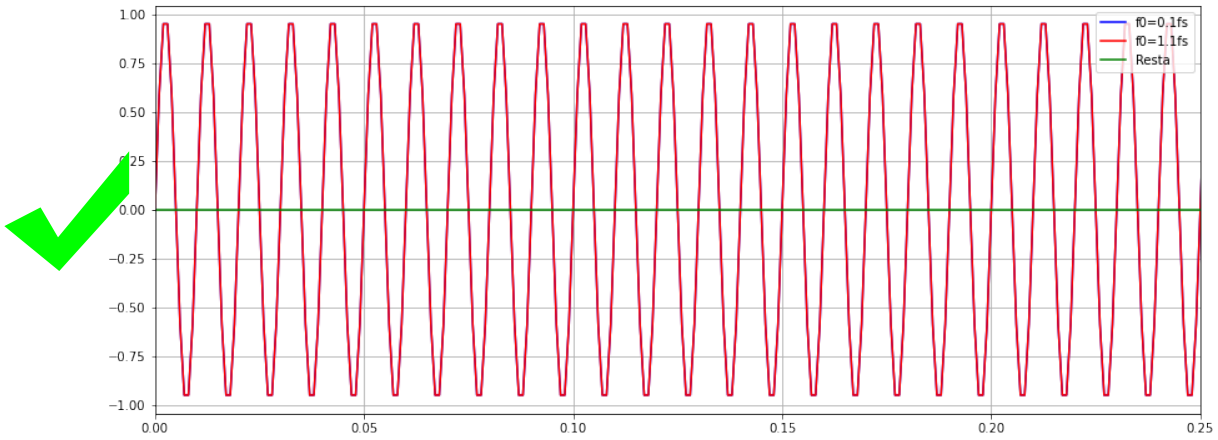


4.2. Usar fs = 1000, N = 1000, fase = 0 amp = 1 .

4.2.1. Con f0 = 0.1* fs y 1.1* fs ¿Cómo podría diferenciar las senoidales? .

En el caso de f0 = 1.1 * fs, la frecuencia de sampleo no es suficiente para poder recuperar la señal en su totalidad y por lo tanto se produce un efecto de aliasing, haciendo creer que la segunda señal es igual a la primera. Para lograr diferenciarlas, sería necesario samplear a una frecuencia mayor a los fines de cumplir con el teorema de Nyquist.

[3]:
```
N     = 1000
fs    = 1000
amp   = 1
fase = 0
f0_a = 0.1 * fs
f0_b = 1.1 * fs

sine_signal_a, sin_time_a = sin_signal(fs=fs, f0=f0_a, amp=amp, N=N, fase=fase)
sine_signal_b, sin_time_b = sin_signal(fs=fs, f0=f0_b, amp=amp, N=N, fase=fase)

fig = plt.figure(figsize=(15,6))
plt.plot(sin_time_a, sine_signal_a, 'b', label="f0=0.1fs")
plt.plot(sin_time_b, sine_signal_b, 'r', label="f0=1.1fs")
plt.plot(sin_time_b, sine_signal_b-sine_signal_a, 'g', label="Resta")
plt.xlim([0, 0.25])
plt.grid()
plt.legend()
plt.show()
```



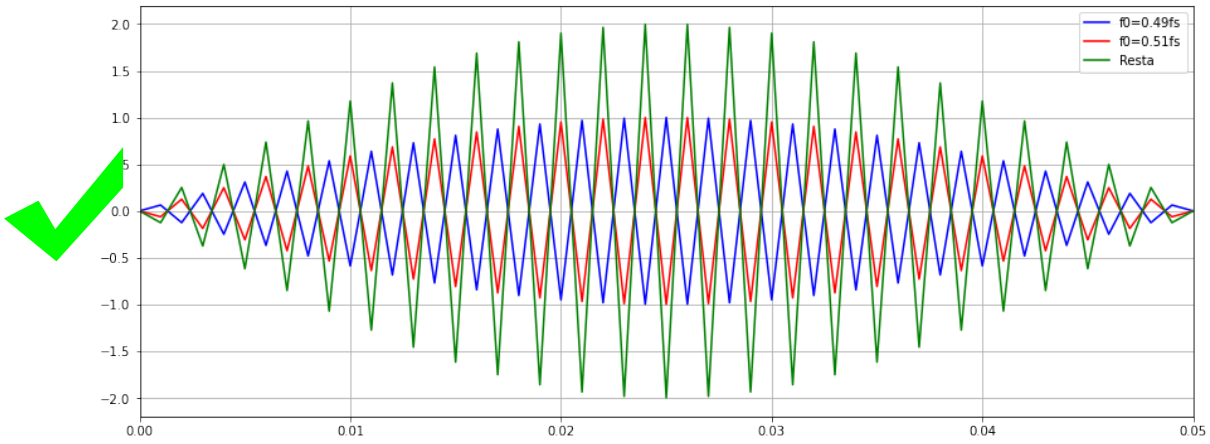
4.2.2. Con f0 = 0.49 * fs y 0.51 * fs ¿Cómo es la frecuencia y la fase entre ambas? .

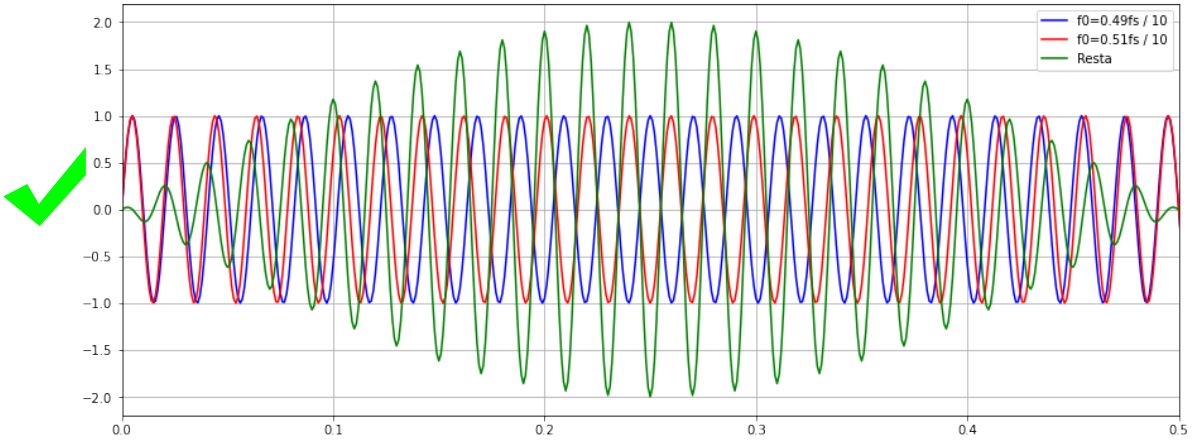
Al graficar las señales obtenidas, se ve que tienen la misma frecuencia pero fases opuestas, por lo que una toma siempre el valor negativo de la otra.

Sin embargo, al graficar las señales con una frecuencia de muestreo mayor, se ve que en realidad tienen frecuencias diferentes, produciéndose un desfasaje incremental entre ambas hasta llegar a tener fases opuestas, para luego comenzar a disminuir hasta volver al valor inicial, y así sucesivamente repitiendo el ciclo.

[4]:
```
N    = 1000
fs   = 1000
amp  = 1
fase = 0
f0_a = 0.49 * fs
f0_b = 0.51 * fs

sine_signal_a, sin_time_a = sin_signal(fs=fs, f0=f0_a, amp=amp, N=N, fase=fase)
sine_signal_b, sin_time_b = sin_signal(fs=fs, f0=f0_b, amp=amp, N=N, fase=fase)

fig = plt.figure(figsize=(15,6))
plt.plot(sin_time_a, sine_signal_a, 'b', label="f0=0.49fs")
plt.plot(sin_time_b, sine_signal_b, 'r', label="f0=0.51fs")
plt.plot(sin_time_b, sine_signal_b-sine_signal_a, 'g', label="Resta")
plt.grid()
plt.legend()
plt.xlim([0, 0.05])
plt.show()

f0_a = 0.49 * fs / 10
f0_b = 0.51 * fs / 10
```

```
sine_signal_a, sin_time_a = sin_signal(fs=fs, f0=f0_a, amp=amp, N=N, fase=fase)
sine_signal_b, sin_time_b = sin_signal(fs=fs, f0=f0_b, amp=amp, N=N, fase=fase)

fig = plt.figure(figsize=(15,6))
plt.plot(sin_time_a, sine_signal_a, 'b', label="f0=0.49fs / 10")
plt.plot(sin_time_b, sine_signal_b, 'r', label="f0=0.51fs / 10")
plt.plot(sin_time_b, sine_signal_b-sine_signal_a, 'g', label="Resta")
plt.grid()
plt.legend()
plt.xlim([0, 0.50])
plt.show()
```





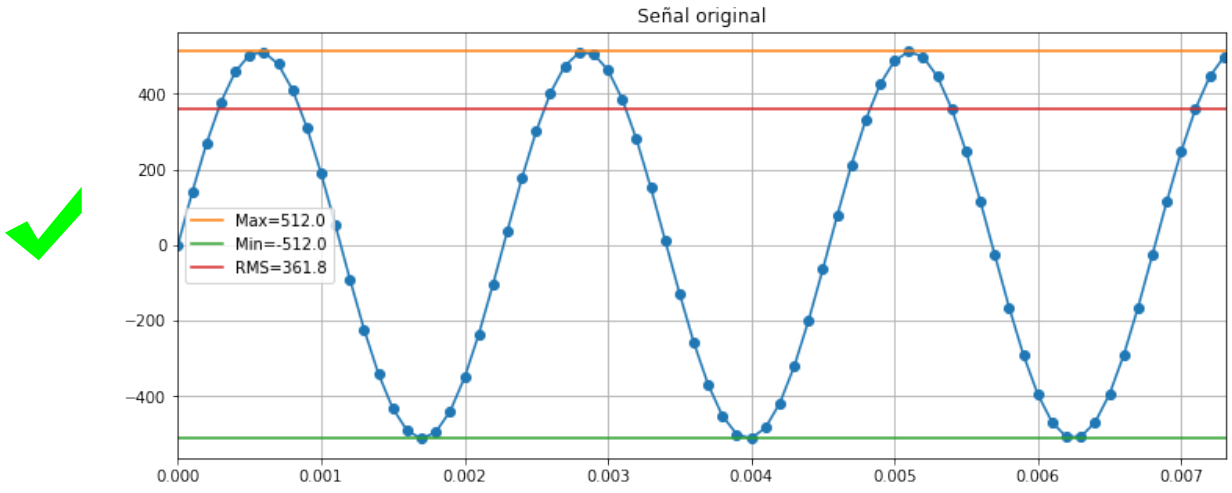
1.0.5 5. Adquisición y reconstrucción con la CIAA

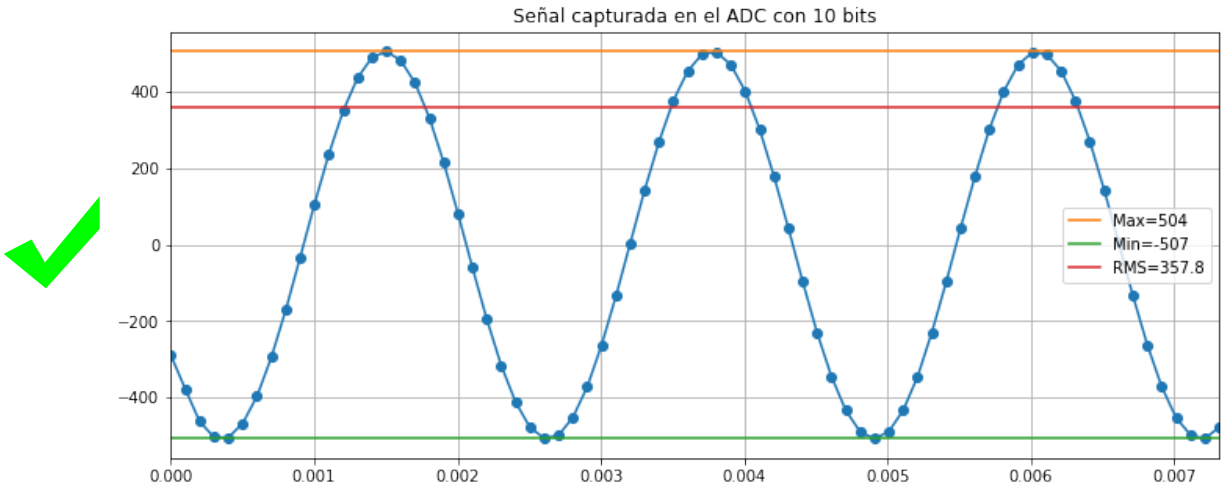
.

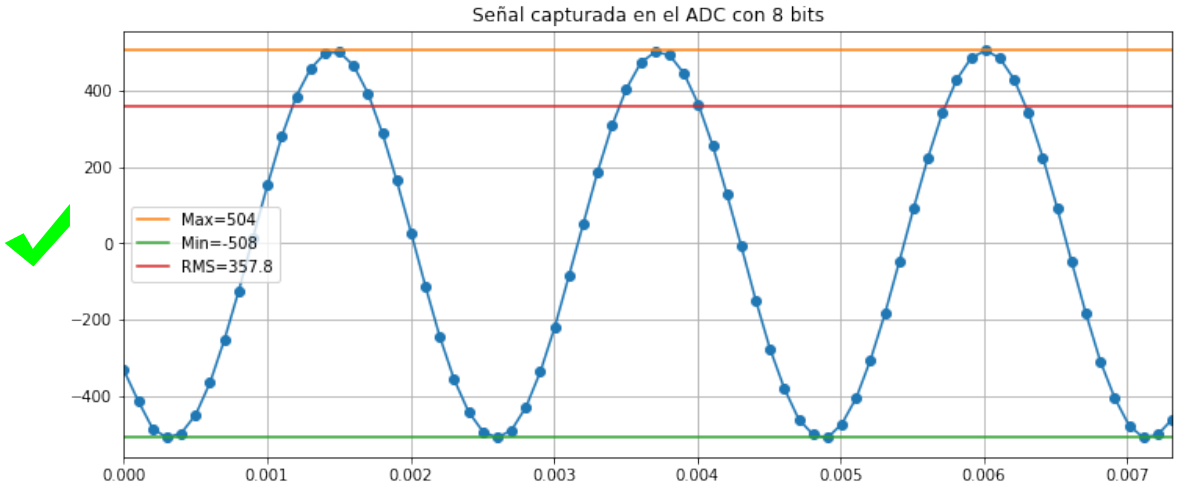
Se puede observar cómo a medida que se reduce la cantidad de bits utilizados, la señal obtenida comienza a saturarse en los valores mínimos y máximos.

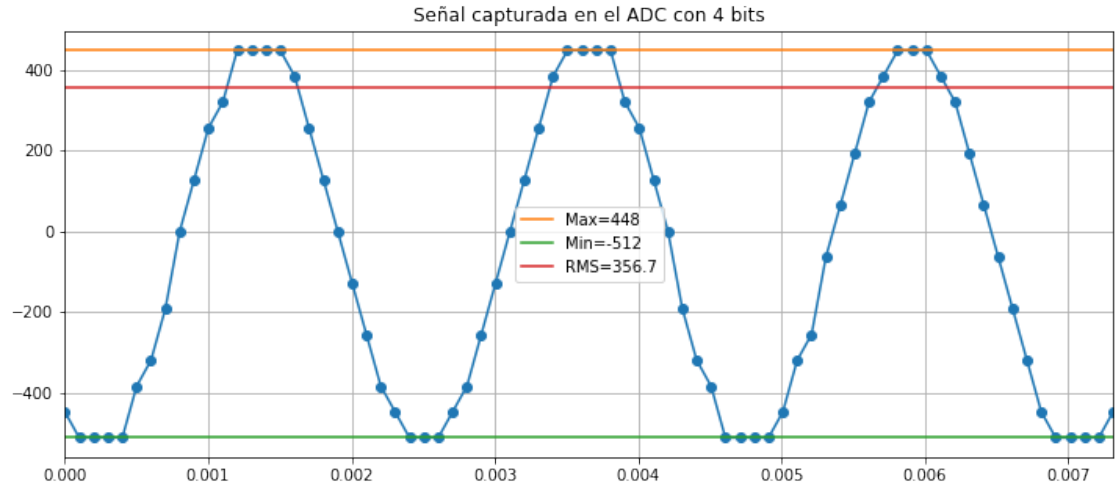
[49]:
```
header = b'header'

def findHeader(f):
    index = 0
    sync  = False
    while sync==False:
        data=b''
        while len(data) <1:
            data = f.read(1)
        if data[0]==header[index]:
            index+=1
            if index>=len(header):
                sync=True
                # print(sync)
        else:
            index=0
            # print(sync)

def readInt4File(f):
    raw=b''
    while len(raw)<2:
        raw += f.read(1)
    return (int.from_bytes(raw[0:2],"little",signed=True))



length = 512
fs     = 10000
f0     = 440
amp    = 512

original_signal, original_time = sin_signal(fs=fs, f0=f0, amp=amp, N=length,␣
 ↪fase=0)
max_value = np.full(len(original_signal), max(original_signal))
min_value = np.full(len(original_signal), min(original_signal))
rms_value = np.full(len(original_signal), np.sqrt(np.mean(np.
 ↪array(original_signal)**2)))

fig = plt.figure(figsize=(12,5))
max_index = 0
plt.plot(original_time[max_index:], original_signal[max_index:], "-o")
plt.plot(time, max_value, label="Max={}".format(round(max_value[0], 1)))
plt.plot(time, min_value, label="Min={}".format(round(min_value[0], 1)))
plt.plot(time, rms_value, label="RMS={}".format(round(rms_value[0], 1)))
```

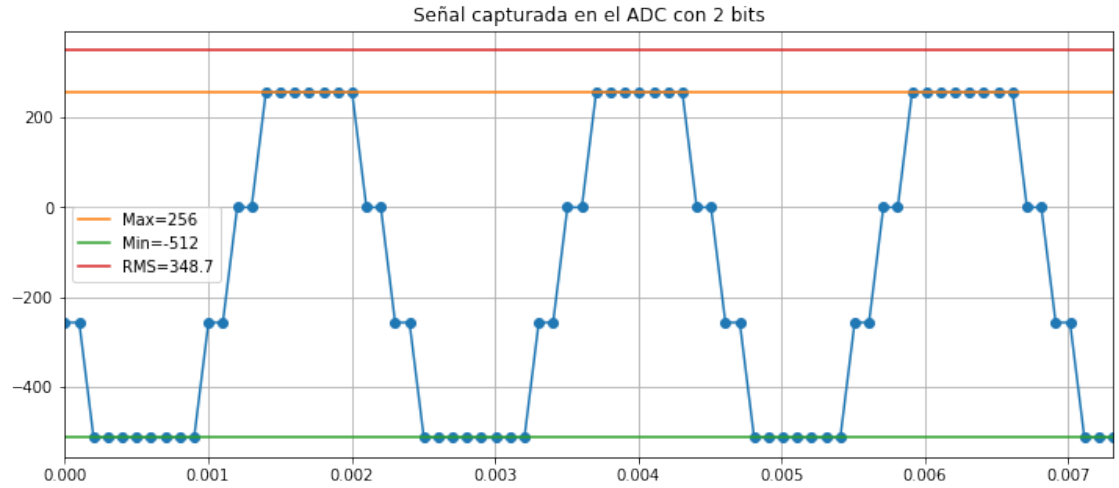
```
plt.xlim (0, length/fs / 7)
plt.grid()
plt.legend(loc="best")
plt.title("Señal original")
plt.show()



# datos obtenidos desde la CIAA
for bits in [10, 8, 4, 2]:
    logFile = open("log_440_{}bits.bin".format(bits), "r+b")
    adc_data = []
    for group in range(1):
        findHeader(logFile)
        for i in range(length):
            adc_data.append(readInt4File(logFile))


    max_value = np.full(len(adc_data), max(adc_data))
    min_value = np.full(len(adc_data), min(adc_data))
    rms_value = np.full(len(adc_data), np.sqrt(np.mean(np.array(adc_data)**2)))


    fig    = plt.figure(figsize=(12,5))
    adcAxe = fig.add_subplot (1, 1, 1)
    time   = np.linspace(0,length/fs,length)
    adcAxe.grid ()
    adcAxe.set_xlim ( 0 ,length/fs / 7)
    plt.plot(time, adc_data, "-o")
    plt.plot(time, max_value, label="Max={}".format(max_value[0]))
    plt.plot(time, min_value, label="Min={}".format(min_value[0]))
    plt.plot(time, rms_value, label="RMS={}".format(round(rms_value[0], 1)))
    plt.legend(loc="best")
    plt.title("Señal capturada en el ADC con {} bits".format(bits))
    plt.show()
```


Señal original
Max=512.0
Min=-512.0
RMS=361.8
400
200
0
−200
−400
0.000
0.001
0.002
0.003
0.004
0.005
0.006
0.007


Señal capturada en el ADC con 10 bits
Max=504
Min=-507
RMS=357.8
400
200
0
−200
−400
0.000
0.001
0.002
0.003
0.004
0.005
0.006
0.007


Señal capturada en el ADC con 8 bits
Max=504
Min=-508
RMS=357.8
400
200
0
−200
−400
0.000
0.001
0.002
0.003
0.004
0.005
0.006
0.007





Esta muy bien, sin embargo te quedo un poco incompleto, te falto la triangular y la cuadrada pero principalmente el histograma de error.
Por otro lado te quedara mejor si superpones la original con la capturada, asi se ve claramente las diferencias

1.0.6 6. Sistema de números

6.1. Explique brevemente algunas de las diferencias entre la representación flotante de simple precision (32b) y el sistema de punto fijo Qn.m. .

La representación en punto flotante permite trabajar con una mayor precisión y un mayor rango dinámico, simplificando además el desarrollo ya que no es necesario realizar consideraciones adicionales al operar con ellas.

La representación en punto fijo reduce significativamente el rango dinámico, y requiere interpretar correctamente los resultados de las operaciones, realizando los corrimientos de coma que sean

necesarios. Pero a cambio, operar con ellos resulta mucho menos costoso desde un punto de vista computacional, ya que para el procesador son simplemente números enteros, más allá de la interpretación decimal/fraccionaria que luego se le dé.

6.2. Escriba los bits de los siguientes números decimales (o el mas cercano) en float, Q1.15, Q2.14.

1. 0.5
2. -0.5
3. -1.25
4. 0.001
5. -2.001
6. 204000000

Te falto la representacion float y tenes algun problema en el codigo en algunos casos parece desfase de bits, pero en otros casos pone valores fuera de los limites.. raro.

[6]:
```python
def twos_comp(value, total_bits):
    if (value & (1 << (total_bits - 1))) != 0:    # chequeo de signo
        value = value - (1 << total_bits)         # negativo
    return value & ((2 ** total_bits) - 1)


def to_fixed_point(number, decimal_bits, fractional_bits):
    """\
    Nota: el metodo devuelve el numero mas cercano menor o igual al
    numero deseado, aun cuando el proximo numero mas grande pueda
    encontrarse en realidad mas cerca del valor deseado.
    """
    target_number = 0
    integer_number = 0
    bit_number = decimal_bits + fractional_bits

    original_number = number
    if number < 0:
        number = -number

    # decimal part
    for i in range(decimal_bits-1, -1, -1):
        # print(2**i)
        if target_number + 2**i <= number:
            target_number = target_number + 2**i
            integer_number = integer_number | (1 << bit_number)
        bit_number = bit_number - 1

    # fractional part
    for i in range(0, fractional_bits):
        #print(1 / (2**(i+1)))
        if target_number + 1 / (2**(i+1)) <= number:
            target_number = target_number + 1 / (2**(i+1))
            integer_number = integer_number | (1 << bit_number)
```

```
        bit_number = bit_number - 1

    if original_number < 0:
        target_number = -target_number
        integer_number = (~integer_number) + 1

    return target_number, integer_number


# number = 0.5
# decimal_bits = 1
# fractional_bits = 2
# total_bits = decimal_bits+fractional_bits
# fixed_point_number, int_fixed_point_number = to_fixed_point(number,␣
 ↪decimal_bits, fractional_bits)
# print(fixed_point_number)
# print(int_fixed_point_number, bin(twos_comp(int_fixed_point_number,␣
 ↪total_bits)))

numbers_to_convert = [0.5, -0.5, -1.25, 0.001, -2.001, 204000000]
numbers_q_1_15 = {"fixed_float": [0]*len(numbers_to_convert), "fixed_int":␣
 ↪[0]*len(numbers_to_convert)}
numbers_q_2_14 = {"fixed_float": [0]*len(numbers_to_convert), "fixed_int":␣
 ↪[0]*len(numbers_to_convert)}

for index, number in enumerate(numbers_to_convert):
    numbers_q_1_15["fixed_float"][index], numbers_q_1_15["fixed_int"][index] =␣
 ↪to_fixed_point(number=number,
                                                                        ␣
 ↪decimal_bits=1, fractional_bits=15)
    numbers_q_2_14["fixed_float"][index], numbers_q_2_14["fixed_int"][index] =␣
 ↪to_fixed_point(number=number,
                                                                        ␣
 ↪decimal_bits=2, fractional_bits=14)


for i in range(len(numbers_to_convert)):
    print("Number = {:9} | Q1.15 = {:16b} {:20} | Q2.14 = {:16b} ({})".
 ↪format(numbers_to_convert[i],
        twos_comp(numbers_q_1_15["fixed_int"][i], 1+15),␣
 ↪"("+str(numbers_q_1_15["fixed_float"][i])+")",
        twos_comp(numbers_q_2_14["fixed_int"][i], 2+14),␣
 ↪numbers_q_2_14["fixed_float"][i]))
```

```
Number =       0.5 | Q1.15 = 1000000000000000 (0.5)                | Q2.14 =
```

da 0x40 00 esto seria -1

```
1000000000000000 (0.5)
Number =      -0.5 | Q1.15 = 1000000000000000 (-0.5)      | Q2.14 =
1100000000000000 (-0.5)
Number =     -1.25 | Q1.15 = 1100000000000000 (-1.25)     | Q2.14 =
1100000000000000 (-1.25)
Number =     0.001 | Q1.15 =          1000000 (0.0009765625)       | Q2.14 =
100000 (0.0009765625)
Number =    -2.001 | Q1.15 =               10 (-1.999969482421875) | Q2.14 =
1111111111100000 (-2.0009765625)
Number = 204000000 | Q1.15 = 1111111111111110 (1.999969482421875)  | Q2.14 =
1111111111111110 (3.99993896484375)
```

da 0x20 00, tenes un desfase

0xC000 te dio igual que 0.5

0xE0 00

0xb0 00

no se puede

0x00 10

0x00 21 (desfasado)

no se puede

no se puede

no, seria 0x7F FF

seria 0x7F FF

[ ]: