Cómo crear tu propias señales DDS para FPGA con Python

Creador: David Rubio G.

Entrada: https://soceame.wordpress.com/2024/11/20/como-crear-tu-propias-senales-dds-para-fpga-con-python/

Blog: https://soceame.wordpress.com/

GitHub: https://github.com/DRubioG

Fecha última modificación: 23/02/2025

Esta entrada va a ser puramente práctica, en ella te voy a enseñar mediante dos ejemplos a crear tus propias DDS para FPGA utilizando Python.

Esta entrada van a ser dos ejemplos de DDS con explicación. El primero es el más recurrente, te voy a explicar cómo generar un seno en una FPGA, y en el segundo te voy a explicar cómo utilizar un DDS para hacer operaciones, en este ejemplo te explicaré como hacer una raíz cuadrada en una FPGA.

A partir de los ejemplos y de los conceptos anteriores se puede reconstruir cualquier señal en una FPGA gracias a Python.

Además, **esto es solo una herramienta, el buen ingeniero de FPGAs construye cosas utilizando sus conocimientos y las herramientas que tiene a su disposición, y Python es una de ellas y totalmente gratuita.**

Antes de empezar se requiere de tener instalado **Python**, cualquier versión vale, y las librerías **numpy** (para operaciones) *[instalación: pip install numpy]* y **matplotlib** (para representaciones) *[instalación: pip install matplotlib]*.

Ejemplo Seno

El primer paso es importar las librerías con las que se va a trabajar.

```
import numpy as np
import matplotlib.pyplot as plt
```

A partir de este punto es todo técnico.

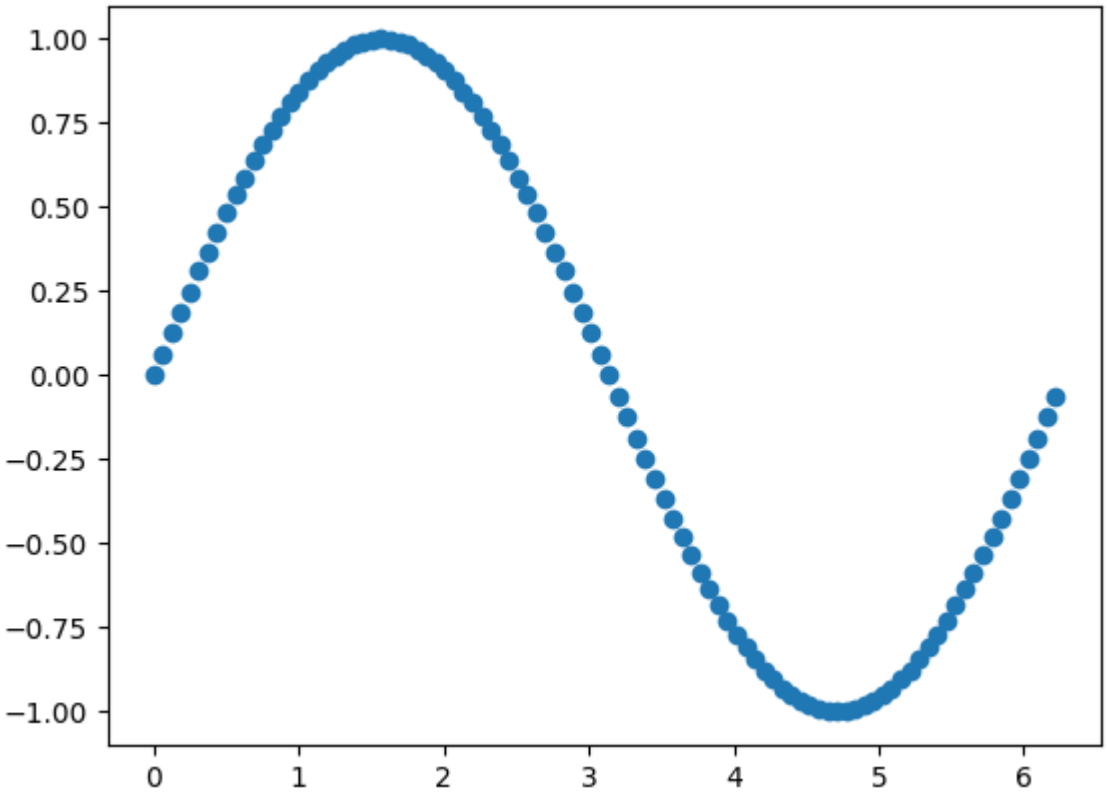
Y primer parámetro que hay que elegir es el número de muestras, y esto es muy importante, porque dependiendo del número de muestras elegidas se determina la frecuencia máxima de trabajo.

Pongo un ejemplo, si yo genero un seno con 100 muestras y la frecuencia de reloj que voy a utilizar es de 100MHz, eso significa que si cada ciclo de reloj emito una muestra, la frecuencia máxima del seno es del 1MHz. Sin embargo si genero un seno con 10 muestras, la frecuencia máxima del seno que voy a crear es de 10MHz.

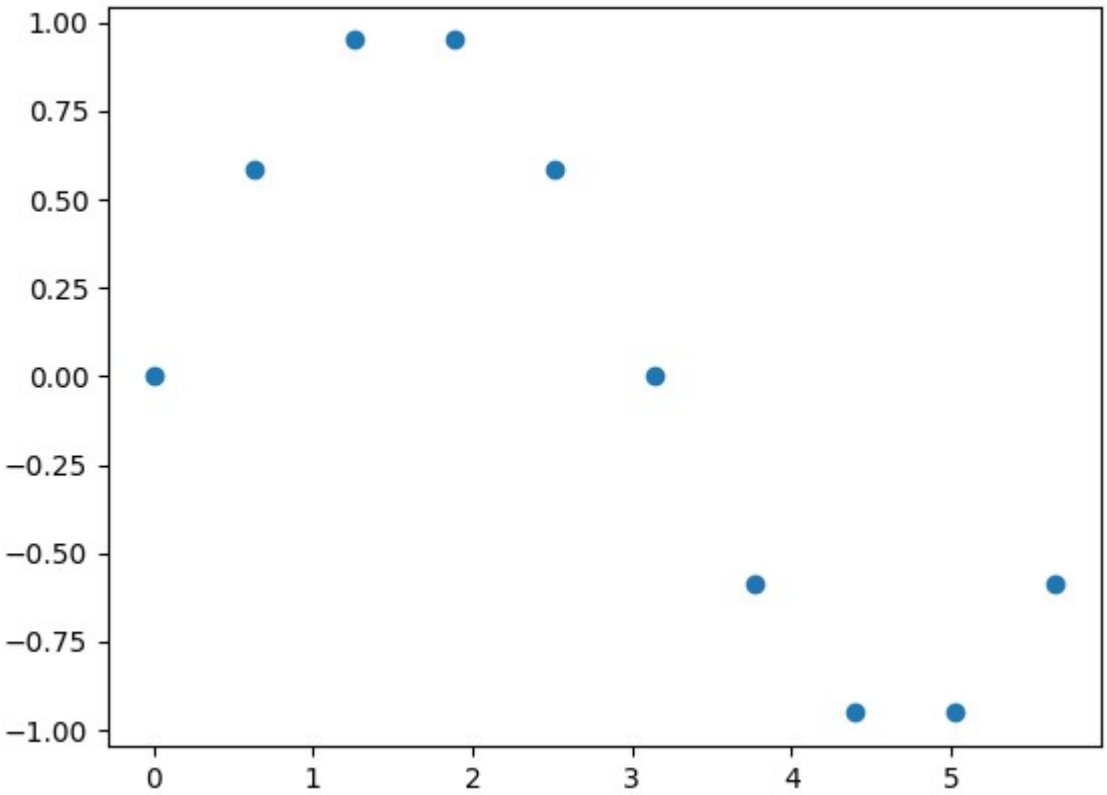
Para generar las muestras se utiliza la función *linspace* de Python, en la que le decimos desde que punto partimos, hasta donde vamos y el número de muestras. El punto del que partimos es el 0, el punto al que vamos es un periodo de una señal seno (2*pi) menos 1 y el número de muestras que queremos.

```
N = 100
t = np.linspace(0, 2*np.pi-(2*np.pi/N), N)
```

Un seno de 100 muestras se ve así

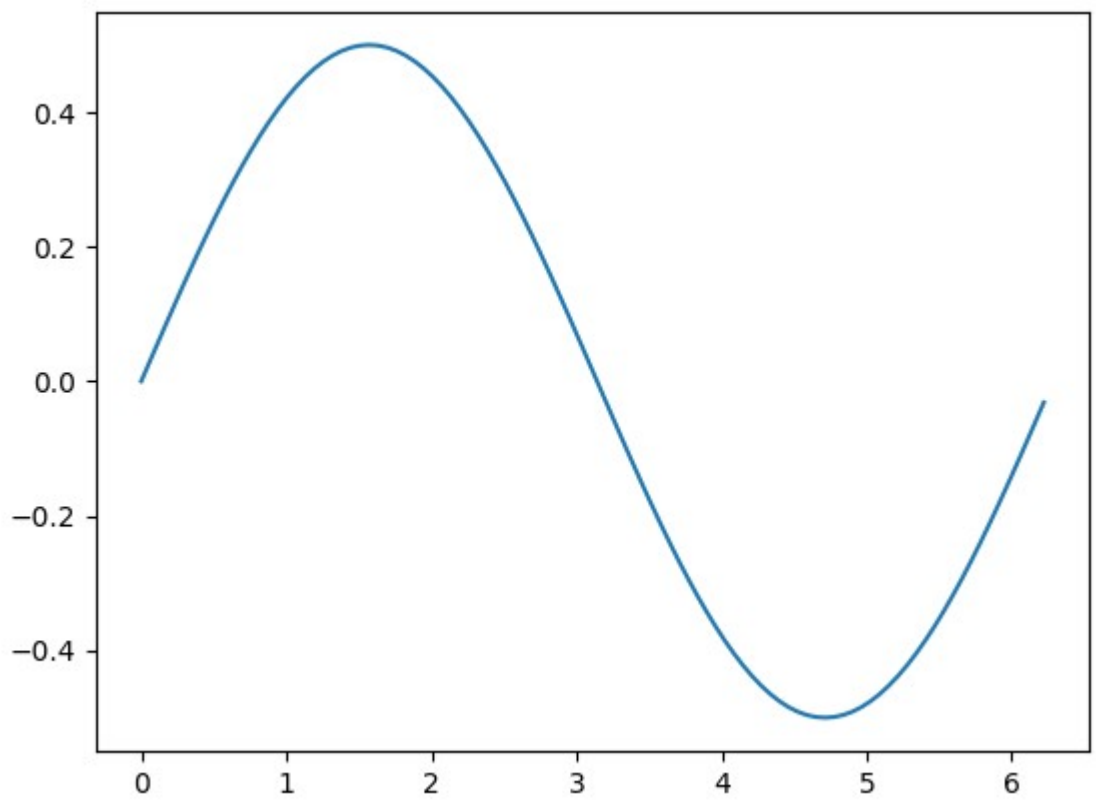


Y uno de 10 muestras así

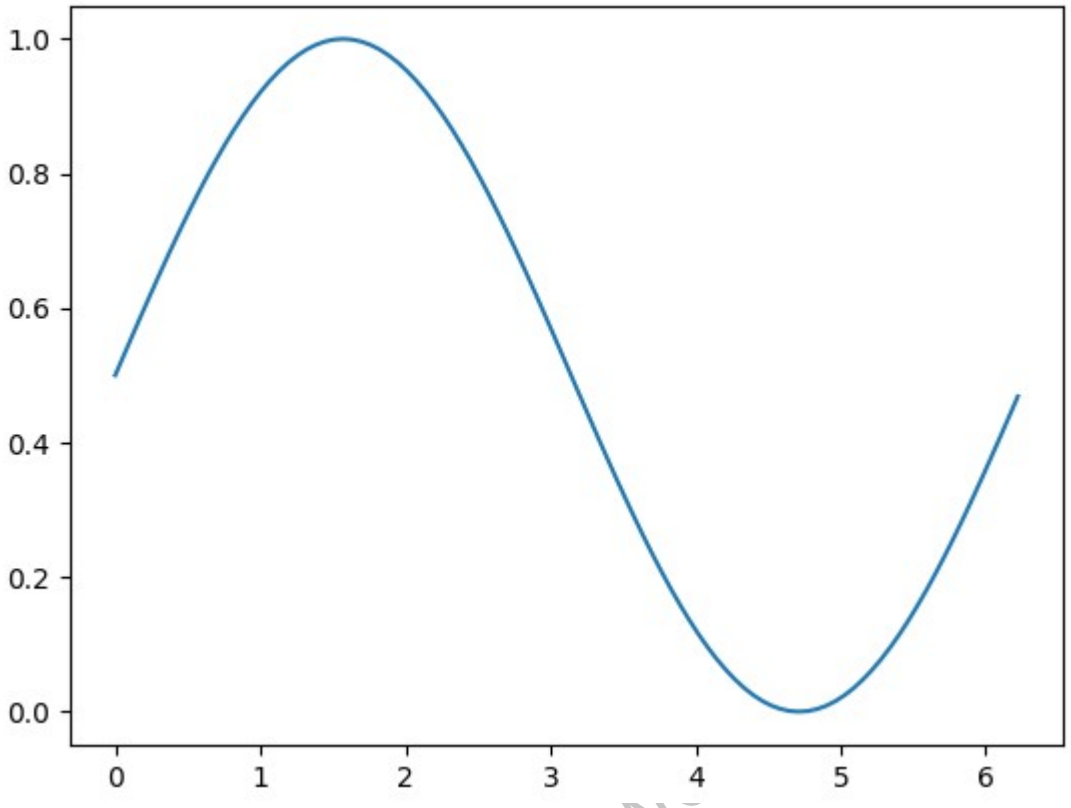


Una vez tenemos las muestras, generamos la señal seno. Para ello es importante tener en cuenta que la señal seno que genera la función *sin* de numpy es una señal de [+1, -1]. Por lo que a menos que queramos trabajar con datos negativos en complemento a dos. Lo mejor es modificar la señal.

Para ello tenemos que dividir la señal a la mitad, para que quede en un rango de [+0.5, -0.5].



Y sumarle la mitad, para que quede en un rango de [+1, 0]



El código necesario para generar esto es el siguiente

```
x = np.sin(t)*(1/2) + (1/2)
```

Entonces, ya tenemos la señal. Ahora toca cuantificarla, eso significa darle valores digitales a una señal continua.

Para ello tenemos que entender que el rango de valores sobre el que se va a cuantificar va de 0 a 2^N_bits -1.

De tal manera que si se quiere cuantificar la señal seno en 8 bits, el rango sobre el que se va a cuantificar es [0, 255].

Entonces, y aquí viene el truco, como la señal de referencia que estamos usando va de 0 a 1, solo sería necesario multiplicar la señal seno por 2^N_bits -1, y truncar/redondear para que todos los datos queden como enteros.

El truncamiento es necesario porque Python considera los valores como números decimales.

```
[127.5, 135.50579123998747, 143.4799872794488, 151.3911176096799, 159.2079606613519, 166.8996667828058, 174.43588046729644, 181.78685967454678, 188.9235934479687, 195.81791635982205, 202.4426196672
9033, 208.77155869295794, 214.7797560059078, 220.44349999622997, 225.74043845391313, 230.6496667828058, 235.15181050150693, 239.2291017055926, 242.8654491894175, 246.04650195075206, 248.7597058276321, 250.99435304390847, 252.74162446790783, 253.99462441759593, 254.74840787460462, 255.0, 254.74840787460462, 253.99462441759593, 252.74162446790783, 250.99435304390847, 248.7597058276321, 246.04650195075206, 242.8654491894175, 239.2291017055926, 235.15181050150693, 230.6496667828058, 225.74043845391313, 220.44349999622995, 214.77975600590779, 208.77155869295794, 202.44261966729036, 195.81791635982205, 188.9235934479687, 181.78685967454675, 174.43588046729644, 166.89966678280575, 159.2079606613519, 151.3911176096799, 143.47998727944878, 135.50579123998742, 127.49999999999996, 119.49420876001254, 111.52001272055121, 103.60888239032009, 95.79203938648098, 88.10033321719418, 80.56411953270351, 73.21314032545324, 66.07640655203129, ...]
```

Al truncar queda así.

```
[127, 135, 143, 151, 159, 166, 174, 181, 188, 195, 202, 208, 214, 220, 225, 230, 235, 239, 242, 246, 248, 250, 252, 253, 254, 255, 254, 253, 252, 250, 248, 246, 242, 239, 235, 230, 225, 220, 214, 208, 202, 195, 188, 181, 174, 166, 159, 151, 143, 135, 127, 119, 111, 103, 95, 88, 80, 73, 66, 59, 52, 46, 40, 34, 29, 24, 19, 15, 12, 8, 6, 4, 2, 1, 0, 0, 0, 1, 2, 4, 6, 8, 12, 15, 19, 24, 29, 34, 40, 46, 52, 59, 66, 73, 80, 88, 95, 103, 111, 119]
```

Para realizar esta operación, se utiliza el siguiente código

```
N_bits = 8
x_dig = [int(i*(2**N_bits-1)) for i in x]
```

Entonces, ahora tenemos un array de números enteros que conforman un seno, pero a VHDL o Verilog esto no le convence. Entonces, hay que convertir los números o a binario o a hexadecimal.

***NOTA**: si vas a trabajar en VHDL, utiliza código binario, porque la representación en código hexadecimal (x»..») solo está reserva a múltiplos de 4 bits, como 4, 8, 12, 16, etc. Entonces, si quieres un seno de 7 bits, no puedes representarlo en hexadecimal.*

Para convertir a hexadecimal solo se necesita utilizar la función *hex*.

```
N_bits = 8
x_dig = [hex(int(i*(2**N_bits-1))) for i in x]
```

Y genera un hexadecimal tal que así.

```
['0x7f', '0x87', '0x8f', '0x97', '0x9f', '0xa6', '0xae', '0xb5', '0xbc', '0xc3', '0xca', '0xd0', '0xd6', '0xdc', '0xe1', '0xe6', '0xeb', '0xef', '0xf2', '0xf6', '0xf8', '0xfa', '0xfc', '0xfd', '0xfe', '0xff', '0xfe', '0xfd', '0xfc', '0xfa', '0xf8', '0xf6', '0xf2', '0xef', '0xeb', '0xe6', '0xe1', '0xdc', '0xd6', '0xd0', '0xca', '0xc3', '0xbc', '0xb5', '0xae', '0xa6', '0x9f', '0x97', '0x8f', '0x87', '0x7f', '0x77', '0x6f', '0x67', '0x5f', '0x58', '0x50', '0x49', '0x42', '0x3b', '0x34', '0x2e', '0x28', '0x22', '0x1d', '0x18', '0x13', '0xf', '0xc', '0x8', '0x6', '0x4', '0x2', '0x1', '0x0', '0x0', '0x0', '0x1', '0x2', '0x4', '0x6', '0x8', '0xc', '0xf', '0x13', '0x18', '0x1d', '0x22', '0x28', '0x2e', '0x34', '0x3b', '0x42', '0x49', '0x50', '0x58', '0x5f', '0x67', '0x6f', '0x77']
```

Y para convertirlo a código binario, se utiliza la función *bin*.

```
N_bits = 8
x_dig = [bin(int(i*(2**N_bits-1))) for i in x]
```

Y genera un código binario así.

```
['0b1111111', '0b1000011', '0b1000111', '0b1001011', '0b1001111', '0b1010011', '0b1010110', '0b1011010', '0b1011100', '0b1011111', '0b1100001', '0b1100011', '0b1100110', '0b1101001', '0b1101011', '0b1101101', '0b1101110', '0b1110000', '0b1110010', '0b1110011', '0b1110100', '0b1110101', '0b1110110', '0b1110111', '0b1111000', '0b1111001', '0b1111010', '0b1111100', '0b1111101', '0b1111110', '0b1111111', '0b1111110', '0b1111101', '0b1111100', '0b1111010', '0b1111000', '0b1110110', '0b1110100', '0b1110011', '0b1110010', '0b1110000', '0b1101110', '0b1101101', '0b1101011', '0b1101001', '0b1100111', '0b1100101', '0b1100011', '0b1100001', '0b1011111', '0b1011101', '0b1011011', '0b1011001', '0b1010111', '0b1010101', '0b1010011', '0b1010001', '0b1001111', '0b1001101', '0b1001011', '0b1001001', '0b1000111', '0b1000101', '0b1000011', '0b1000001', '0b0111111', '0b0111101', '0b0111011', '0b0111001', '0b0110111', '0b0110101', '0b0110100', '0b0110010', '0b0110000', '0b0101110', '0b0101101', '0b0101011', '0b0101010', '0b0101000', '0b0100111', '0b0100101', '0b0100100', '0b0100010', '0b0100001', '0b0011111', '0b0011110', '0b0011100', '0b0011011', '0b0011010', '0b0011001', '0b0010111', '0b0010110', '0b0010101', '0b0010100', '0b0010011', '0b0010010', '0b0010001', '0b0010000', '0b0001111', '0b0001110', '0b0001101', '0b0001100', '0b0001011', '0b0001010', '0b0001010', '0b0001001', '0b0001000', '0b0000111', '0b0000111', '0b0000110', '0b0000101', '0b0000101', '0b0000100', '0b0000100', '0b0000011', '0b0000011', '0b0000010', '0b0000010', '0b0000010', '0b0000001', '0b0000001', '0b0000001', '0b0000001', '0b0000000', '0b0000000', '0b0000000', '0b0000000', '0b0000000', '0b0000000', '0b0000000', '0b0000000', '0b0000000', '0b0000001', '0b0000001', '0b0000001', '0b0000001', '0b0000010', '0b0000010', '0b0000010', '0b0000011', '0b0000011', '0b0000100', '0b0000100', '0b0000101', '0b0000101', '0b0000110', '0b0000111', '0b0000111', '0b0001000', '0b0001001', '0b0001010', '0b0001010', '0b0001011', '0b0001100', '0b0001101', '0b0001110', '0b0001111', '0b0010000', '0b0010001', '0b0010010', '0b0010011', '0b0010100', '0b0010101', '0b0010110', '0b0010111', '0b0011010', '0b0011011', '0b0011100', '0b0011110', '0b0011111', '0b0100001', '0b0100010', '0b0100100', '0b0100101', '0b0100111', '0b0101001', '0b0101011', '0b0101101', '0b0101110', '0b0110000', '0b0110010', '0b0110011', '0b0110101', '0b0110110', '0b0111000', '0b0111001', '0b0111011', '0b0111101', '0b0111110', '0b1000000', '0b1000010', '0b1000011', '0b1000101', '0b1000111', '0b1001001', '0b1001011', '0b1001101', '0b1001110', '0b1010000', '0b1010010', '0b1010011', '0b1010101', '0b1010110', '0b1011000', '0b1011001', '0b1011011', '0b1011100', '0b1011110', '0b1011111', '0b1100001', '0b1100011', '0b1100100', '0b1100110', '0b1100111', '0b1101001', '0b1101010', '0b1101100', '0b1101101', '0b1101110', '0b1110000', '0b1110001', '0b1110010', '0b1110011', '0b1110101', '0b1110110', '0b1110111']
```

Es importante darse cuenta que los valores tanto en binario como en hexadecimal no están completos, eso significa que el formato que entrega Python está sesgado porque hay eliminado los 0s de la izquierda. Por lo que antes de pasarle los datos al sintetizador de la FPGA, se requiere de adaptarlos.

Entonces, el dato antes de pasarse a VHDL, tiene que ser homogéneo a 8 bits.

```
"11110010"
"11110110"
"11111000"
"11111010"
"11111100"
"11111101"
"11111110"
"11111111"
"11111110"
"11111101"
"11111100"
"11111010"
"11111000"
"11110110"
"11110010"
"11101111"
"11101011"
"11100110"
"11100001"
"11011100"
"11010110"
"11010000"
"11001010"
"11000011"
"10111100"
"10110101"
"10101110"
"10100110"
"10011111"
"10010111"
"10001111"
"10000111"
"01111111"
"01110111"
```


Una posible solución es algo parecido a esto para arreglarlo, se puede hacer tan potente como uno quiera, además, se puede llegar a generar el fichero en VHDL o Verilog directamente desde Python, lo que facilita mucho la reimplementación o la actualización de los datos.

```
for dato in x_dig:
    dato_aux = dato[2:]
    dato_cero = ""
    for j in range(N_bits-len(dato_aux)):
        dato_cero += "0"
    dato_final = dato_cero + dato_aux
    print("\"" + dato_final + "\",")
```

El código completo:

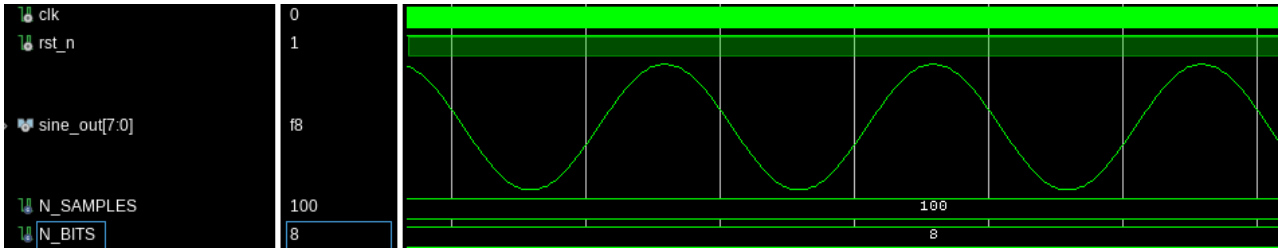
```
N = 100
t = np.linspace(0, 2*np.pi-(2*np.pi/N), N)
```

```
x = np.sin(t)*(1/2) + (1/2)

N_bits = 8
x_dig = [bin(int(i*(2**N_bits-1))) for i in x]

for dato in x_dig:
    dato_aux = dato[2:]
    dato_cero = ""
    for j in range(N_bits-len(dato_aux)):
        dato_cero += "0"
    dato_final = dato_cero + dato_aux
    print("\"" + dato_final + "\",")
```

Ahora lo que hacemos es coger las muestras generadas en Python y se las metemos en un fichero VHDL. Para hacerlo se tiene que utilizar un *array*, para ello se utiliza el *type*.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


entity test_seno is
    generic(
        N_SAMPLES : integer := 100;
        N_BITS : integer := 8
    );
    Port (
        clk : in std_logic;
        rst_n : in std_logic;
        sine_out : out std_logic_vector(N_BITS-1 downto 0)
    );
end test_seno;

architecture Behavioral of test_seno is
type seno_type is array (0 to N_SAMPLES-1) of std_logic_vector(N_BITS-1 downto 0);

constant seno : seno_type := ("01111111", "10000111", "10001111", "10010111", "10011111", "10100110", "10101110",
                              "10110101", "10111100", "11000011", "11001010", "11010000", "11010110", "11011100",
                              "11100001", "11100110", "11101011", "11101111", "11110010", "11110110", "11111000",
                              "11111010", "11111100", "11111101", "11111110", "11111111", "11111110", "11111101",
                              "11111100", "11111010", "11111000", "11110110", "11110010", "11101111", "11101011",
                              "11100110", "11100001", "11011100", "11010110", "11010000", "11001010", "11000011",
                              "10111100", "10110101", "10101110", "10100110", "10011111", "10010111", "10001111",
                              "10000111", "01111111", "01110111", "01101111", "01100111", "01011111", "01011000",
                              "01010000", "01001001", "01000010", "00111011", "00110100", "00101110", "00101000",
                              "00100010", "00011101", "00011000", "00010011", "00001111", "00001100", "00001000",
                              "00000110", "00000100", "00000010", "00000001", "00000000", "00000000", "00000000",
                              "00000001", "00000010", "00000100", "00000110", "00001000", "00001100", "00001111",
                              "00010011", "00011000", "00011101", "00100010", "00101000", "00101110", "00110100",
                              "00111011", "01000010", "01001001", "01010000", "01011000", "01011111", "01100111",
                              "01101111", "01110111");


signal cont : integer range 0 to N_SAMPLES-1;

begin

    sine_out <= seno(cont);

    process(clk, rst_n)
    begin
        if rst_n = '0' then
            cont <= 0;
        elsif rising_edge(clk) then
            if cont = N_SAMPLES-1 then
                cont <= 0;
            else
                cont <= cont+1;
            end if;
        end if;
    end process;


end Behavioral;
```

Lo único que se necesitaría es de un reloj que incremente las cuentas del seno. Si se simula, se puede ver que la señal generada es la señal seno que se busca.



Si lo que se quiere es que la señal seno se pueda modificar por la fase, lo único que se tiene es que atacar el índice que genera el seno.

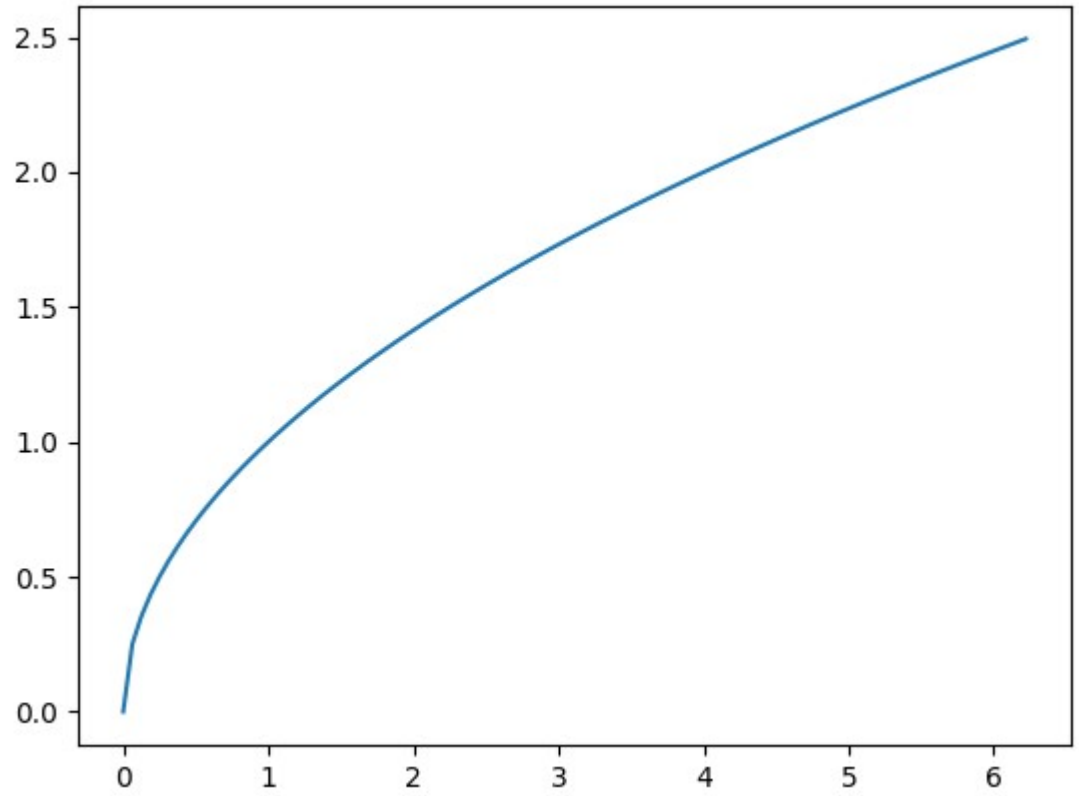
Bien, una forma de atacar este seno es generar un seno múltiplo de 360 muestras, lo que hace que cada ciclo de reloj se incremente 1º o multiplo de 1º, por lo que se puede manejar mejor la fase.

A raíz de cómo se genera esta señal se puede generar cualquier otra señal periódica *(o no periódica, como explico en el siguiente ejemplo)*.

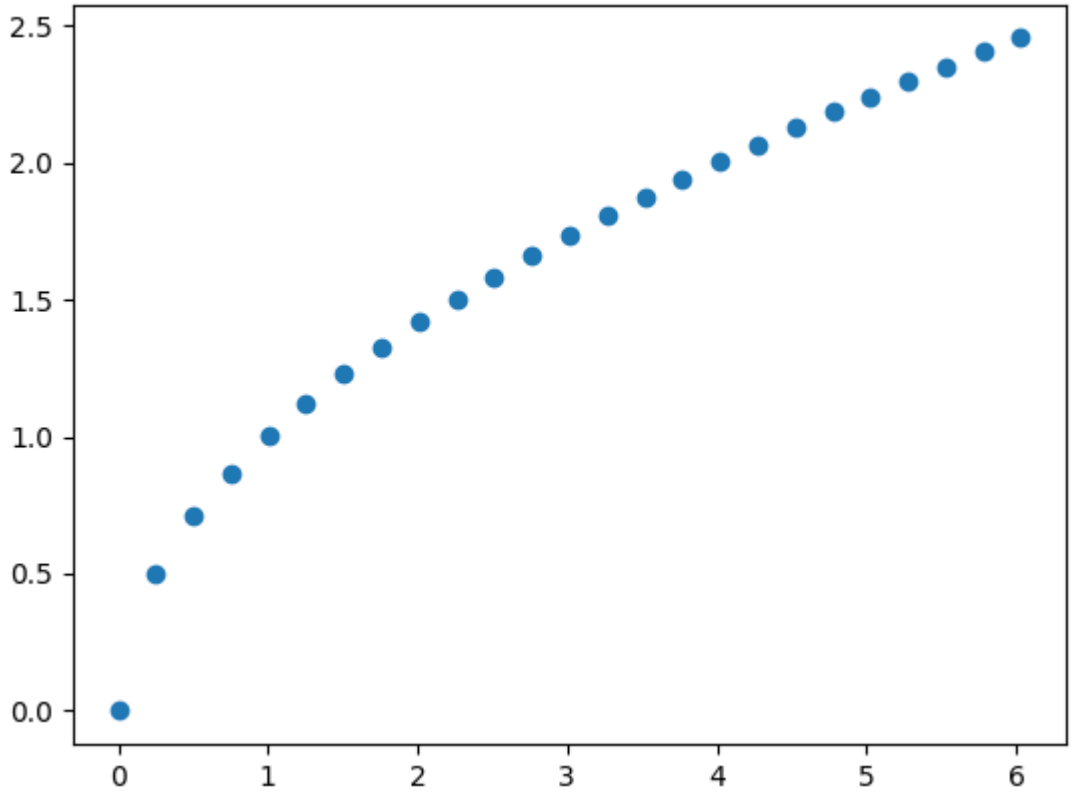
Ejemplo Raíz Cuadrada

En el ejemplo anterior se ha generado una señal seno, pues generar una raíz cuadrada es exactamente igual, solo hay que atacar el índice de la raíz cuadrada.

Para entenderlo más fácil, esta es la curva de una raíz cuadrada para 100 valores.



Bien, pues lo único que habría que hacer es digitalizar esta señal y hacer como en la señal seno del ejemplo anterior.



Es importante tener cuenta en este ejemplo que hay que acotar las muestras al número de bits deseado, debido a que el número de bits puede resultar menor que el del valor que se quiere calcular.

Para ello hay que ajustar las muestras a los valores deseados. Para ello se divide la lista entre el valor más elevado.

```
x = np.sqrt(t)/max(np.sqrt(t))
```

Con esto se genera una array de valores que van del 0 hasta el 1

```
array([0.        , 0.10050378, 0.14213381, 0.17407766, 0.20100756,
       0.22473329, 0.24618298, 0.26590801, 0.28426762, 0.30151134,
       0.31782086, 0.33333333, 0.34815531, 0.36237154, 0.37605072,
       0.38924947, 0.40201513, 0.41438771, 0.42640143, 0.43808583,
       0.44946657, 0.46056619, 0.47140452, 0.4819992 , 0.49236596,
       0.50251891, 0.51247074, 0.52223297, 0.53181602, 0.54122943,
       0.55048188, 0.55958137, 0.56853524, 0.57735027, 0.58603272,
       0.59458839, 0.60302269, 0.61134064, 0.61954692, 0.62764591,
       0.63564173, 0.6435382 , 0.65133895, 0.65904737, 0.66666667,
       0.67419986, 0.68164981, 0.68901921, 0.69631062, 0.70352647,
       0.71066905, 0.71774056, 0.72474308, 0.73167857, 0.73854895,
       0.74535599, 0.75210143, 0.75878691, 0.765414  , 0.77198419,
       0.77849894, 0.78495963, 0.79136757, 0.79772404, 0.80403025,
       0.81028739, 0.81649658, 0.82265891, 0.82877541, 0.83484711,
       0.84087497, 0.84685992, 0.85280287, 0.85870469, 0.86456622,
       0.87038828, 0.87617165, 0.8819171 , 0.88762536, 0.89329715,
       0.89893315, 0.90453403, 0.91010045, 0.91563303, 0.92113237,
       0.92659908, 0.93203373, 0.93743687, 0.94280904, 0.94815078,
       0.95346259, 0.95874497, 0.96399841, 0.96922337, 0.97442031,
       0.97958969, 0.98473193, 0.98984745, 0.99493668, 1.        ])
```

Ahora estos valores se digitalizan como se hizo con el seno.

```
['0b0', '0b11001', '0b100100', '0b101100', '0b110011', '0b111001', '0b111110', '0b1000011', '0b1001000', '0b1001100', '0b1010001', '0b1010101', '0b1011000', '0b1011100', '0b1011111', '0b1100011',
'0b1100110', '0b1101001', '0b1101100', '0b1101111', '0b1110010', '0b1110101', '0b1111000', '0b1111010', '0b1111101', '0b10000000', '0b10000010', '0b10000101', '0b10000111', '0b10001010', '0b100011
00', '0b10001110', '0b10010000', '0b10010011', '0b10010101', '0b10010111', '0b10011001', '0b10011011', '0b10011101', '0b10100000', '0b10100010', '0b10100100', '0b10100110', '0b10101000', '0b101010
10', '0b10101011', '0b10101101', '0b10101111', '0b10110001', '0b10110011', '0b10110101', '0b10110111', '0b10111000', '0b10111010', '0b10111100', '0b10111110', '0b10111111', '0b11000001', '0b110000
11', ...]
```

El código utilizado es el siguiente:

```
N = 100
t = np.linspace(0, 99, N)

x = np.sqrt(t)/max(np.sqrt(t))

N_bits = 8
x_dig = [bin(int(i*(2**N_bits-1))) for i in x]

for dato in x_dig:
    dato_aux = dato[2:]
    dato_cero = ""
    for j in range(N_bits-len(dato_aux)):
        dato_cero += "0"
    dato_final = dato_cero + dato_aux
    print("\"" + dato_final + "\",")
```

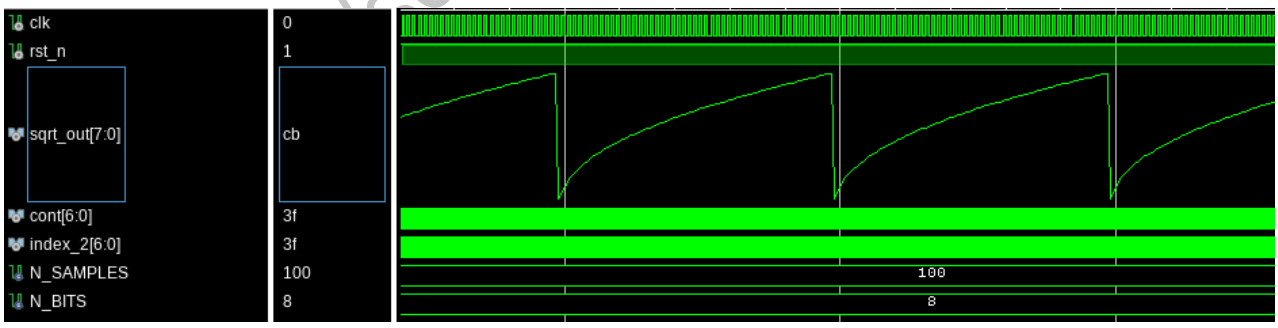
Estas muestras se transforman a código binario y se introducen en un fichero VHDL como en el ejemplo del seno.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


entity test_sqrt is
    generic(
        N_SAMPLES : integer := 100;
        N_BITS : integer := 8
    );
    Port (
        clk : in std_logic;
        rst_n : in std_logic;
        index : in std_logic_vector(6 downto 0);
        sqrt_out : out std_logic_vector(N_BITS-1 downto 0)
    );
end test_sqrt;

architecture Behavioral of test_sqrt is
type sqrt_type is array (0 to N_SAMPLES-1) of std_logic_vector(N_BITS-1 downto 0);

constant sqrt : sqrt_type := ("00000000","00011001","00100100","00101100","00110011","00111001","00111110","01000011","01001000","01001100",
                              "01010001","01010101","01011000","01011100","01011111","01100011","01100110","01101001","01101100","01101111",
                              "01110010","01110101","01111000","01111010","01111101","10000000","10000010","10000101","10000111","10001010",
                              "10001100","10001110","10010000","10010011","10010101","10010111","10011001","10011011","10011101","10100000",
                              "10100010","10100100","10100110","10101000","10101010","10101011","10101101","10101111","10110001","10110011",
                              "10110101","10110111","10111000","10111010","10111100","10111110","10111111","11000001","11000011","11000100",
                              "11000110","11001000","11001001","11001011","11001101","11001110","11010000","11010001","11010011","11010100",
                              "11010110","11010111","11011001","11011010","11011100","11011101","11011111","11100000","11100010","11100011",
                              "11100101","11100110","11101000","11101001","11101010","11101100","11101101","11101111","11110000","11110001",
                              "11110011","11110100","11110101","11110111","11111000","11111001","11111011","11111100","11111101","11111111");

signal index_aux : std_logic_vector(index'range);

begin

    index_aux <= index when to_integer(unsigned(index)) < N_SAMPLES-1 else
                 (others=>'0');


    sqrt_out <= sqrt(to_integer(unsigned(index_aux)));

end Behavioral;
```

Y lo único que habría que hacer, es en vez de generar un contador interno, meter un puerto de entrada que sea el valor que se desea calcular la raíz cuadrada, y ya se tendría el valor.

NOTA: Es importante tener en cuenta que es recomendable meter un sistema que garantice que el índice no exceda el límite del número de muestras.

Si decidimos representar la señal de la raíz cuadrada vemos que tiene la misma curva que la señal generada en Python.



Al igual que se ha generado una raíz cuadrada se puede generar un logaritmo, una exponencial o incluso una derivada. Y lo mejor es que se pueden empalmar operaciones para hacer cálculos más complejos.

Nota Final

Como se ha podido ver en toda esta entrada, generar una señal DDS propia no es difícil y que además tiene multitud de usos si la sabes generar.

Bien, pues si a todo esto le sumas la utilización de un sistema de coma fija (que lo único que hay que hacer es multiplicar el número entero cuantificación) puedes disparar la capacidad de generación de señales hasta límites insospechados.

Si la puedes generar en Python, se puede generar en una FPGA.