Trabajos Prácticos N° 1, 2 y 3:

Técnicas Básicas de Control Digital de Servomecanismos

5 de marzo de 2021

Alumnos :

Julián Eduardo	AGUIRRE	julian.aguirre@ib.edu.ar
Tomás	LIENDRO	tomas.liendro@ib.edu.ar
Gastón	OROZCO	gaston.orozco@ib.edu.ar

Docentes :

Felix	MACIEL
Fabián	LEMA
Norberto	ABAURRE
Santiago	PINCÍN

San Carlos de Bariloche, Argentina

Índice

1. Trabajo Práctico N° 1: Manejo del Tiempo

1.1. Ejercicio 1: Medición de Tiempos y Esperas

Listing 1: Medición de tiempo de ejecución de *trash operations*

```
/*Experimente con las funciones vistas en clase para medir intervalos de
tiempo y suspender la ejecución de un thread.*/

#include <iostream>
#include <chrono>
#include <thread>
#include <cmath>

using duration = std::chrono::duration<double>;
using time_point = std::chrono::steady_clock::time_point;

int main(){
    int n = 1000; // N. iter
    time_point t_start;  // Starting time
    for (int i=0; i<n; ++i){
        t_start=std::chrono::steady_clock::now();  // Get current time
        double aux = log(sqrt(i) * pow(i, 2))/(i + 1 + pow(i, i));  // Some
        trash operations
        printf("Time␣to␣calculate␣the␣\"aux\"␣variable␣and␣to␣print␣this␣message␣is:␣");
        }
        time_point t_end = std::chrono::steady_clock::now();  // Get current time
        // Get thread duration
        duration time_span =https://www.overleaf.com/project/6038ed72c6868c2ff4e6a1ba std:
        std::cout << time_span.count() << "s" << std::endl;  // Print duration

        // If we want to stop a thread:
        for (int i=0; i<10; ++i){  // same FOR loop ranging i from 0 to 9
        // Some trash operations
        double aux = log(sqrt(i) * pow(i, 2))/(i + 1 + pow(i, i));
        std::this_thread::sleep_for(1);  // Suspend execution 1 second
        }
    return 0;
}
```

1.2. Ejercicio 2: Temporización con intervalos discretos

El tiempo de muestreo es un parámetro fundamental para los controladores y plantas discretas. Es por eso que resulta importante la temporización del programa que se va a ejecutar con intervalos discretos.

A continuación se muestra la función que permitirá calcular el tiempo que debe esperar el programa para cumplir con el tiempo de muestreo Dt especificado. En la función *espera_siguiente_intervalo*, **t** es el tiempo medido antes de iniciar el bucle *for* de control, **n** es el número de iteración del bucle y **Dt** es el periodo de muestreo.

Listing 2: Función para calcular el tiempo de espera para un periodo de muestreo esperada de Dt

```
// funcion de espera
duration espera_siguiente_intervalo(const time_point &t, int n, const duration &Dt){
    duration tiempo = n * Dt - (std::chrono::steady_clock::now() - t);
    if (tiempo.count() > 0)
        std::this_thread::sleep_for(tiempo);
    return tiempo;
}
```

1.3. Ejercicio 3: Programa con control de tiempo

Utilizando la función *espera_siguiente_intervalo* descripta en la sección anterior, se implementó un código para tomar ciertos parámetros de entrada (como por ejemplo el número de muestras n, el periodo de ejecución Dt y el nombre del archivo) y poder ejecutar el programa temporizado. El programa que se muestra a continuación servirá de base para la implementación de otros programas que ejecuten otras funciones como se verá más adelante.

La salida de este programa resulta ser un archivo de texto con dos columnas, donde la primera contiene el tiempo de ejecución del programa y la segunda es el tiempo de espera del programa.

Listing 3: Programa de control de tiempo

```
// includes necesarios y prototipos de funciones
#include <chrono>
#include <thread>
#include <cmath>
#include <iostream>
#include <fstream>
#include <vector>

using duration = std::chrono::duration<double>;
using time_point = std::chrono::steady_clock::time_point;

duration espera_siguiente_intervalo(const time_point &t, int n, const duration &Dt);

int main ()
{
   // definicion de parametros del experimento como
   // numero de intervalos, tiempo de muestreo, etc
   int n;
   float Dt;
```

```
std::string fileName;
std::cout << "Ingresar cantidad de muestras n: ";
std::cin >> n;
std::cout << "Ingresar Dt: ";
std::cin >> Dt;
std::cout << "Ingresar nombre del archivo: ";
std::cin >> fileName;
fileName += ".txt";
duration Ddt = std::chrono::duration<double>(Dt);

// inicializaciones
time_point t0;
duration wait;
std::vector<duration> time_span;
std::vector<duration> t;
std::vector<duration> esp;
time_span.resize(n);
t.resize(n);
esp.resize(n);

// bucle controlado temporalmente
// primero establecemos el tiempo inicial t0
t0=std::chrono::steady_clock::now();  // Get current time
// e inmediatamente comenzamos con las iteraciones
for(int i=0; i<n;++i){
   // medimos tiempo de inicio de cada iteracion
   time_point t_start=std::chrono::steady_clock::now();  // Get current time
   // calculamos t[i] restandole t0
   t[i]= std::chrono::duration_cast<duration>(t_start-t0);

   // se ejecutan las acciones de control
   // aca no hay I/O innecesarios

      //...

   // esperamos hasta que sea el tiempo del
   // siguiente intervalo guardando la espera esp[i]
   esp[i] = espera_siguiente_intervalo(t0, i+1, Ddt);
}

// una vez terminado el experimento, se guardan los
// valores a archivos para su procesamiento
std::ofstream fout(fileName);

for (int i=0; i<n; ++i)
   fout << t[i].count() << "\t" << esp[i].count() << std::endl;
fout.close();
```

```
    return 0;
}

// funcion de espera

duration espera_siguiente_intervalo(const time_point &t, int n, const duration &Dt){
    duration tiempo = n * Dt - (std::chrono::steady_clock::now() - t);
    if (tiempo.count() > 0)
        std::this_thread::sleep_for(tiempo);
    return tiempo;
}
```

1.4. Ejercicio 4: Ensayar programa

Finalmente vamos a mostrar los resultados obtenidos tras la ejecución del programa del punto 3 para diversos valores de **Dt** (por ejemplo 0.1, 0.05, 0.02, 0.01, 0.005, 0.002, 0.001 segundos) en la Raspberry Pi asignada por la cátedra (Raspberry #3), a la que nos referiremos desde ahora como R3,

Compilamos el código del punto 3 con el comando:

```
g++ -Wall -std=c++11 -I/usr/local/include/libusb/ -o medicionTiempo
    medicionTiempo.cpp -lmccusb -lm -lhidapi-libusb -lusb-1.0
```

Esta línea de codigo permite compilar el código importando las librerías **-lmccusb -lm -lhidapi-libusb -lusb-1.0**. Luego, para ejecutar el código hacemos:

```
./medicionTiempo
```

Los resultados obtenidos para los tiempos de espera medios se resumen en la siguiente tabla:

Periodo de muestreo Dt (s)	Tiempo de ejecución medio (ms)	% de tiempo de ejecución
0.001	0.0931	9.31 %
0.002	0.0981	4.91 %
0.005	0.1181	2.36 %
0.01	0.1106	1.11 %
0.02	0.1148	0.57 %
0.05	0.1221	0.24 %
0.1	0.1212	0.12 %

Tal como se puede ver en la tabla, la velocidad de ejecución de un loop vacío en la Raspberry es suficientemente rápido para tener tiempos de espera positivos para periodos de muestro de hasta 0.001s.

2. Trabajo Práctico N° 2: Generador de funciones

2.1. Funciones

Se generaron 3 códigos con idea de poder generar *"funciones de onda"* para correrlas en las raspberry y poder visualizarlas en un osciloscopio. En la siguiente practica se podrán visualizar los resultados. En primer lugar se encuentra la función *Seno* que representaría un onda senoidal en el osciloscopio. Tiene la siguiente forma:

$$V(t) = V_0 * sin(\omega * t) + V_0$$

En segundo lugar, escribimos un código que representa la función conocida como *Chirp*. Esta función tiene la característica de que su frecuencia varia con el tiempo. En nuestro caso definimos a la Chirp de la siguiente manera:

$$V(t) * sin(\omega(t) * t) - cte$$

$$\omega(t) = \omega_0 + (t/t_2) * (t_2 - t_1)$$

con $V(t)$ definido como sigue:

$V_0 * t/t_0$	desde $t = 0$ hasta $t = t_0(10\,\%$ de $t_2)$
V_0	desde $t = t_0$ hasta $t = t_1$ $(80\,\%$ de $t_2)$
$V_0 * (t_2 - t)/(t_2 - t_1)$	desde $t = t_1$ hasta $t = t_2$ $(10\,\%$ de $t_2)$

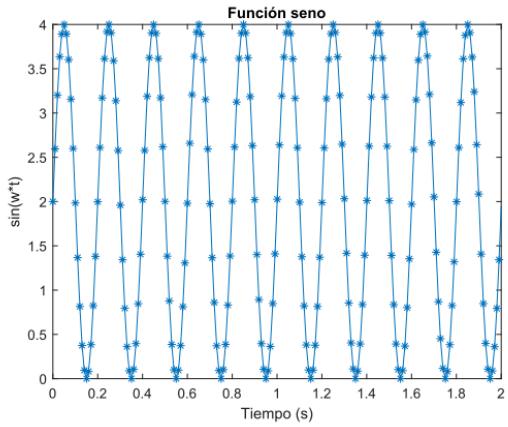
Tabla 1: Definición de la func. Chirp

Y por ultimo, agregamos la función *Escalones Aleatorios*, la cual genera escalones que van de 0 a V_0 de duración aleatoria, con una distribución plana entre T_1 y T_2. Cabe destacar que el código contiene una semilla de generación de números aleatorios fija, para poder reproducirse la misma secuencia múltiples ocasiones.

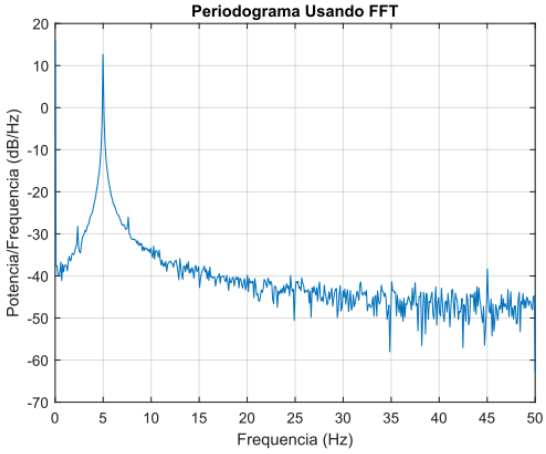
2.2. Ejercicio 4: Análisis espectral

2.2.1. Función senoidal

Se realizó la generación de una función senoidal con una frecuencia de 5 Hz, tomando 1024 muestras a intervalos de 0.01 segundos y amplitud $V0$ de 2V. Los resultados se presentan a continuación:



(a) Función seno

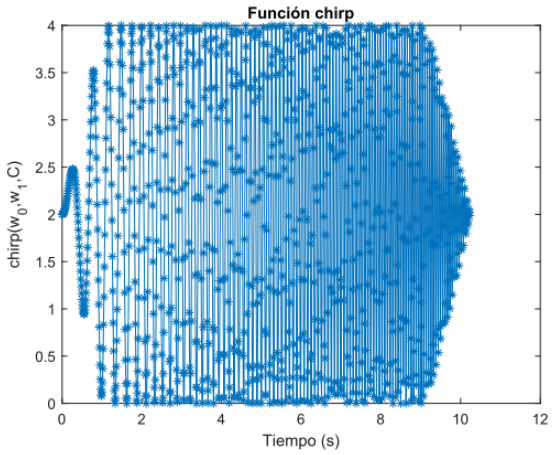


(b) Espectrograma de la función seno

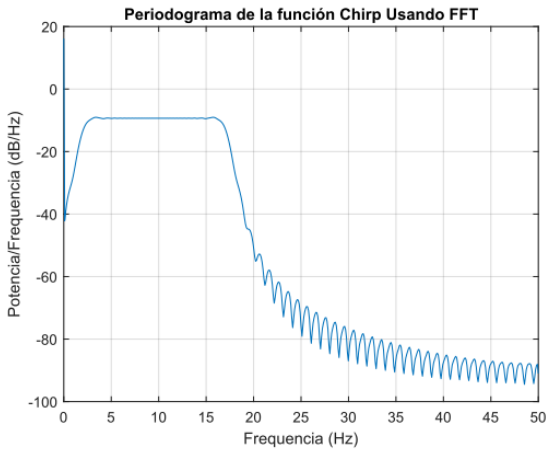
Tal como puede verse en la Figura 1b, el espectrograma de la función seno presenta un pico en 5Hz. La posición del pico se corresponde con la frecuencia de la señal seno generada en 1a.

2.2.2. Función chirp

En segundo término, se realizó la generación del archivo con los datos correspondientes a una función chirp, nuevamente tomando 1024 muestras a intervalos de 0.01 segundos. Las frecuencias mínima y máximas se fijaron en 1 y 10 Hz respectivamente con amplitud de $V0 = 2V$ y la constante C=-2. Las imágenes correspondientes se muestran a continuación:



(a) Función chirp



(b) Espectrograma de la función chirp

En el caso del espectrograma de la función chirp de la Figura 2b, las frecuencias excitadas van desde 1Hz hasta 20Hz aproximadamente. Esto corresponde al doble de la frecuencia máxima de la chirp (que va entre 1Hz y 10Hz). La explicación a esto es que el periodograma se arma en función de la frecuencia instantánea de la planta, que dicho de otra manera es la derivada de la fase de la planta. Entonces, si la función chirp

responde a la ecuación:

$$V = V0sin(w(t)t);$$

donde

$$w(t) = w_1 + w_2t;$$

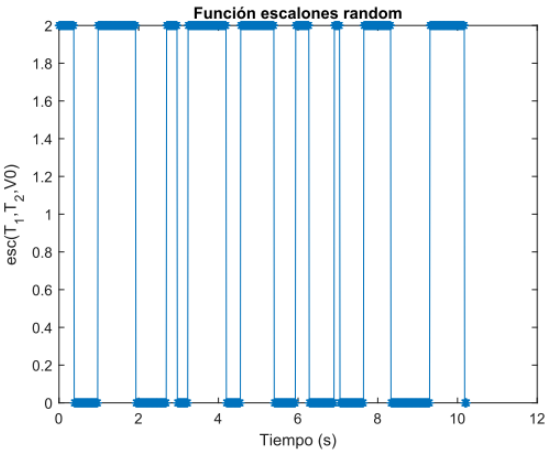
entonces, la derivada de la fase de la planta resulta:

$$\frac{d\phi}{dt} = \frac{dw(t)t}{dt} = \frac{d}{dt}(w_1 + w_2t)t = w_1 + 2tw_2$$

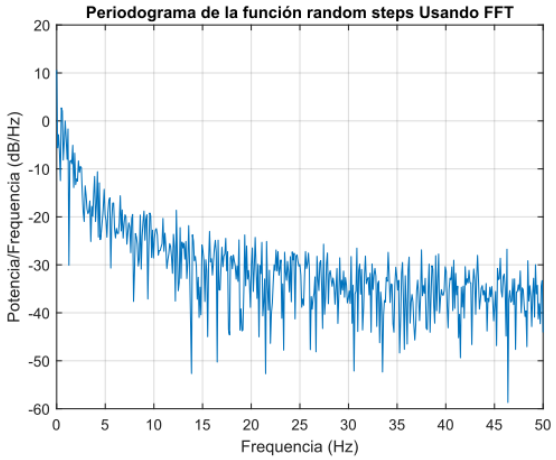
De esta ultima ecuación se ve que el rango de frecuencias activadas en el espectrograma va desde w_1 a $2w_2$, que en este ejemplo es de 1Hz a 20Hz.

2.2.3. Función escalones aleatorios

Por último, se generó la siguiente función escalón junto a su respectivo análisis espectral



(a) Función escalones aleatorios



(b) Espectrograma de la función escalones aleatorios

Como muestra la figura A, puede verse que la duración temporal de cada escalón es aleatoria en un rango de tiempo T1 y T2, y esto puede repetirse para distintos valores, según sea necesario el caso.

3. Trabajo Práctico N° 3: DAQ

3.1. Ejercicio 1: Conocimiento de la placa de adquisición

Mediante el uso del comando *"dmesg | greo MCC"* determinamos el modelo de las raspberry que utilizaremos: USB-1208FS.
Contamos con los manuales y la documentación de este modelo, y a modo de prueba ejecutamos un programa para comprobar el correcto funcionamiento de la misma. Para compilar dicho código utilizamos la siguiente sentencia:
g++ -Wall -std=c++11 -I/usr/local/include/libusb/ -o test-usb1208FS test-usb1208FS.cpp -lmccusb -lm -lhidapi-libusb -lusb-1.0
Con este programa pudimos realizar la calibración tanto de dos pines entrada y dos de salida de la placa adquisidora. A continuación se muestran los resultados:

Calibración de INPUT (PIN 1 - Channel 0 - Single Ended)

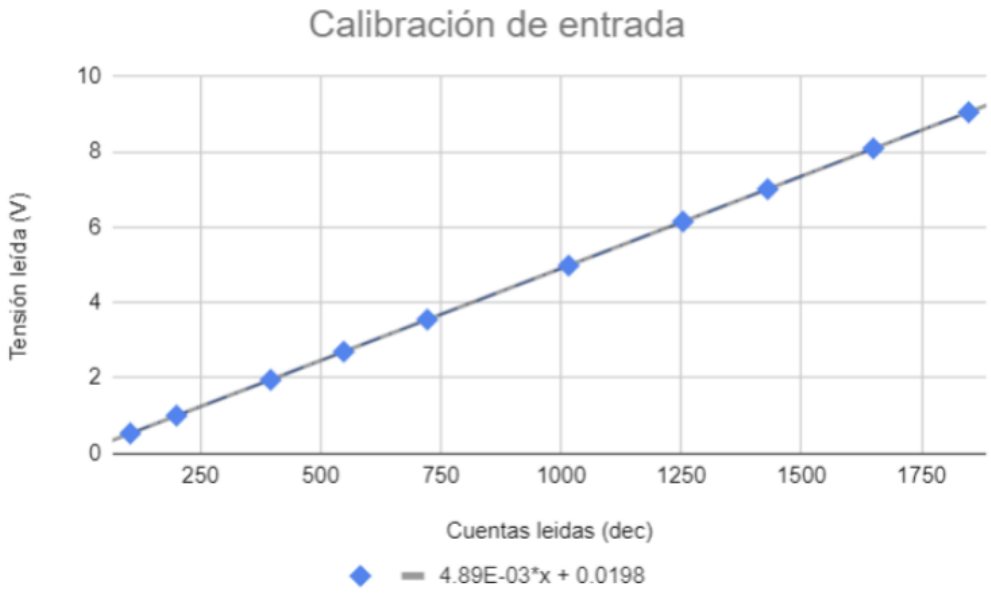


Figura 4: Calibración de Entrada

Con este gráfico podemos ver la siguiente ecuación de calibración:

$$Tension\_leida = 4{,}8910^{-3} Cuentas + 0{,}0198$$

Calibración de OUTPUT (PIN 13 - Channel 0)

Figura 5: Calibración de Salida

El siguiente grafico muestra la ecuacion de calibracion de salida:

$$Cuentas = 998{,}5 * Tension\_salida + 1{,}665$$

3.2. Ejercicio 2: Funciones de adquisición

Mediante el uso de la librerias **libmccusb** y otras necesarias, junto con el ejemplo del código de testeo de la placa se implementaron las 4 funciones necesarias para el manejo de la placa de adquisición.

Función A: **Inicialización de la placa:** *libusb_ device_ handle *placa_init()*

Función B: **Cerrado de la placa:** *int placa_end(libusb_device_handle *udev)*

Función C: **Escritura de Voltaje:** *void write (libusb_device_handle *udev, int ch, float,V)*

Función D: **Lectura de Voltaje:** *float read(int ch, libusb_device_handle *udev)*

3.3. Ejercicio 3: Medición de tiempos en lectura y escritura

Se realizaron múltiples experimentos de escritura y lectura en la placa adquisidora, a diferentes periodos de muestreo Dt y se obtuvieron los tiempos de espera(s) correspondientes. Los resultados se puede observar en los siguientes gráficos donde muestra como es la variación de los tiempos de espera frente cuando el programa realiza iteraciones con y sin operaciones de lectura o escritura dentro del bucle de control.

3.3.1. Tiempos de escritura

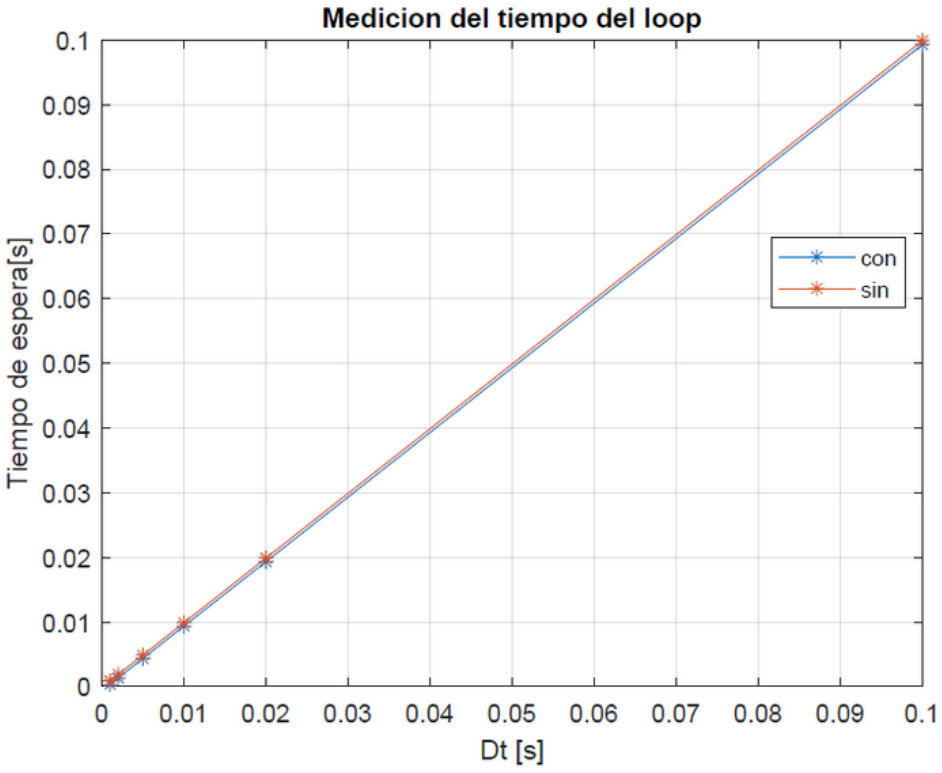


Figura 6: Medición de los tiempos de espera según el tiempo de muestreo

En la figura anterior se presentan los tiempos promedio de espera para diferentes períodos de muestreo, tanto para una operación de escritura por iteración y sin ella. La diferencia que se observa como una pequeña distancia entre las rectas es un buen estimador del tiempo que demora el hardware de la placa de adquisición en realizar la escritura en un canal de salida. Es esperable que estas dos rectas sean paralelas puesto que la demora introducida por la operación de escritura debe ser invariable, siendo esta una característica constructiva del dispositivo.

3.3.2. Tiempos de lectura

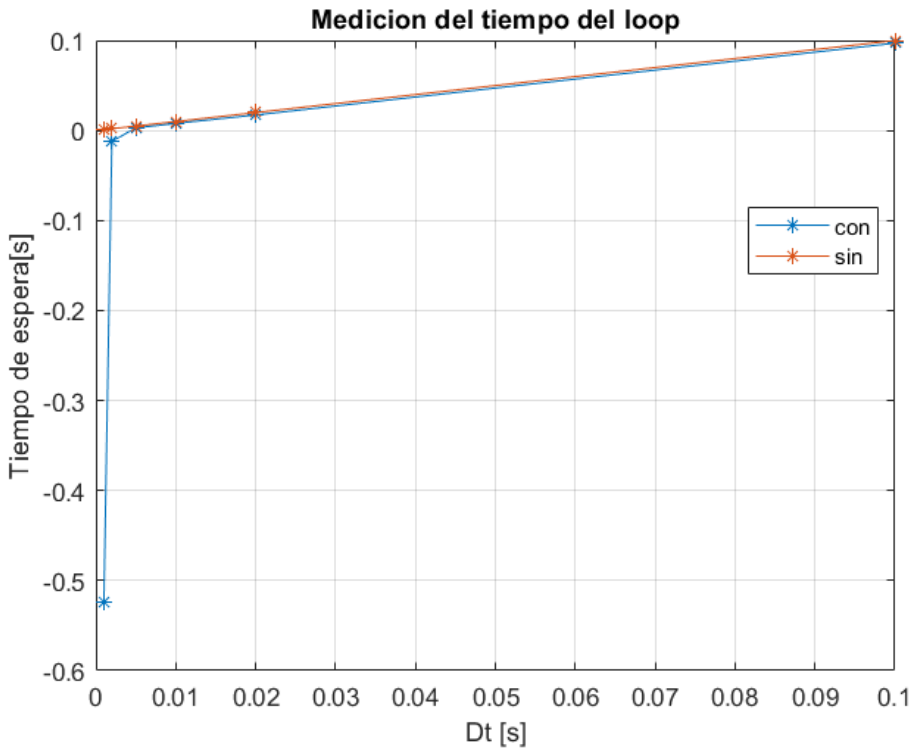


Figura 7: Tiempos de espera con y sin operación de lectura en el bucle de control

Del mismo modo que en la operación de escritura, ambas rectas se mantienen paralelas en concordancia a la disminución del tiempo de espera a medida que se reduce el periodo de muestreo. Sin embargo, en la figura siguiente se realiza un acercamiento a la región donde Dt esta en el orden de unos pocos milisegundos.

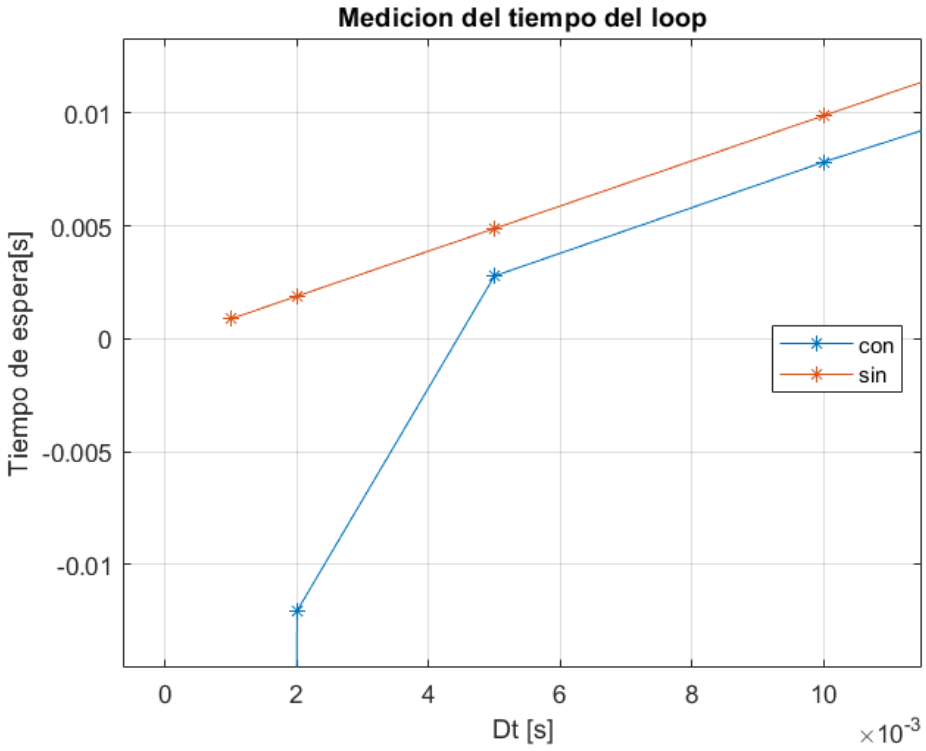


Figura 8: Tiempos de espera con y sin operación de lectura

En este caso, se aprecia como para los tiempos de muestreo de 0.001 y 0.002 segundos, el sistema es incapaz de realizar un proceso de lectura sin que con ello no se exceda del tiempo disponible antes de la siguiente iteración. Este problema se advierte de observar que los tiempos de espera correspondientes pasan a ser negativos en esta región. No se puede asegurar actualmente cuál es la causa que produce dicha demora en relevar un canal de entrada pero se estima que puede ser alguna operación dentro del código que se debe optimizar.

Como conclusión de los análisis anteriores, se puede confirmar que el sistema responde correctamente cuando se debe generar una señal en un canal de salida para todos los tiempos de muestreo consignados. En cambio, en operaciones de lectura en un canal de entrada analógico, es necesario tener en cuenta que los tiempos de este procesamiento están en el orden del milisegundo, imposibilitando usar períodos de muestreo cercanos a este valor.

3.4. Ejercicio 4: Determinación del Ruido

Para esta prueba, se realizaron mediciones de lectura en los canales analógicos 0 y 1, alterando los periodos de muestreo Dt y colocando cada canal a 2 voltios mediante una fuente externa o a 0 voltios conectando dicha entrada a GND. Luego, para cada prueba, se obtuvo un archivo de texto con los valores de tensión en cada instante de adquisición el cuál se proceso en una plantilla de cálculo para generar la siguiente tabla:

	Canal 0				Canal 1			
	0 V		*2 V*		*0 V*		*2 V*	
Dt [s]	Media	Desv.Est	Media	Desv.Est	Media	Desv.Est	Media	Desv.Est
0.1	-0,01951	0,00053	2,01297	0,00159	-0,01955	0,00031	2,01272	0,00043
0.01	-0,01952	0,00043	2,01289	0,00135	-0,01954	0,00000	2,01272	0,00075
0.001	-0,01954	0,00000	2,01283	0,00114	-0,01954	0,00000	2,01273	0,00053

Figura 9: Estadísticas de las lecturas para la discriminación del ruido

Como se utilizó la función de lectura implementada en el punto 2, los valores relevados se presentan en tensión y no en cuentas. De todos modos, el objetivo de este punto es determinar las fluctuaciones que puedan existir entre cada operación de lectura, lo que determinaría la presencia ruido u otras perturbaciones.
Analizando los resultados en la tabla, se observa que en todos los casos y en ambos canales, los valores de desviación estándar son un orden de magnitud menor que el mínimo valor apreciable por la placa adquisidora (aproximadamente 5 mV para la configuración con la que se hicieron las pruebas). A su vez, se puede advertir que el valor promedio de las lecturas es prácticamente constante, independientemente del valor de muestreo utilizado.
Teniendo en cuenta los resultados obtenidos, se concluye que el hardware utilizado para relevar señales eléctricas presenta una gran inmunidad al ruido eléctrico.

3.5. Ejercicio 5: Generador de Funciones

Se adaptaron las funciones creadas en la práctica 2, *Seno, Chirp, y Escalones Aleatorios* para implementarlas como una librería compilable dentro del programa de control de la placa adquisidora, con el fin de poder generar dichas señales en los canales de salida. Luego, para comprobar que la implementación fue correcta, se realizaron las pruebas necesarias mediante la configuración en el programa de control y el relevamiento del canal de salida utilizado mediante una punta de prueba y un osciloscopio. A continuación se muestran los resultados:

Función *Seno:*

Figura 10: Func. Generada SENO

Función *Chirp:*

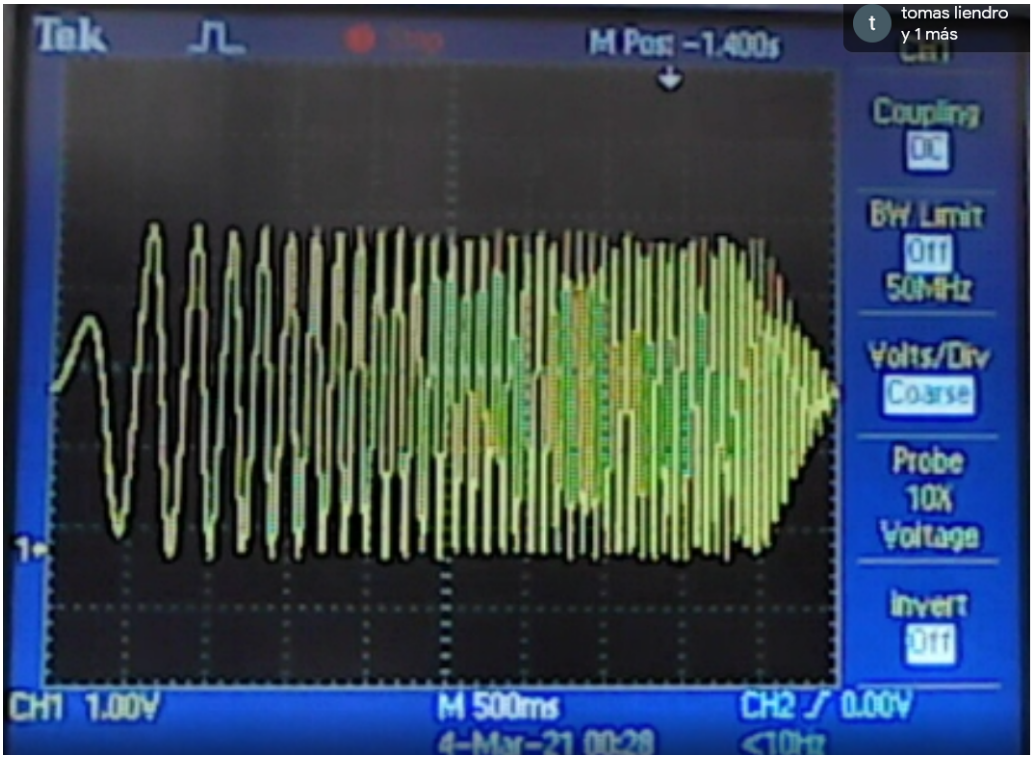


Figura 11: Func. generado CHIRP

Función *Escalón*

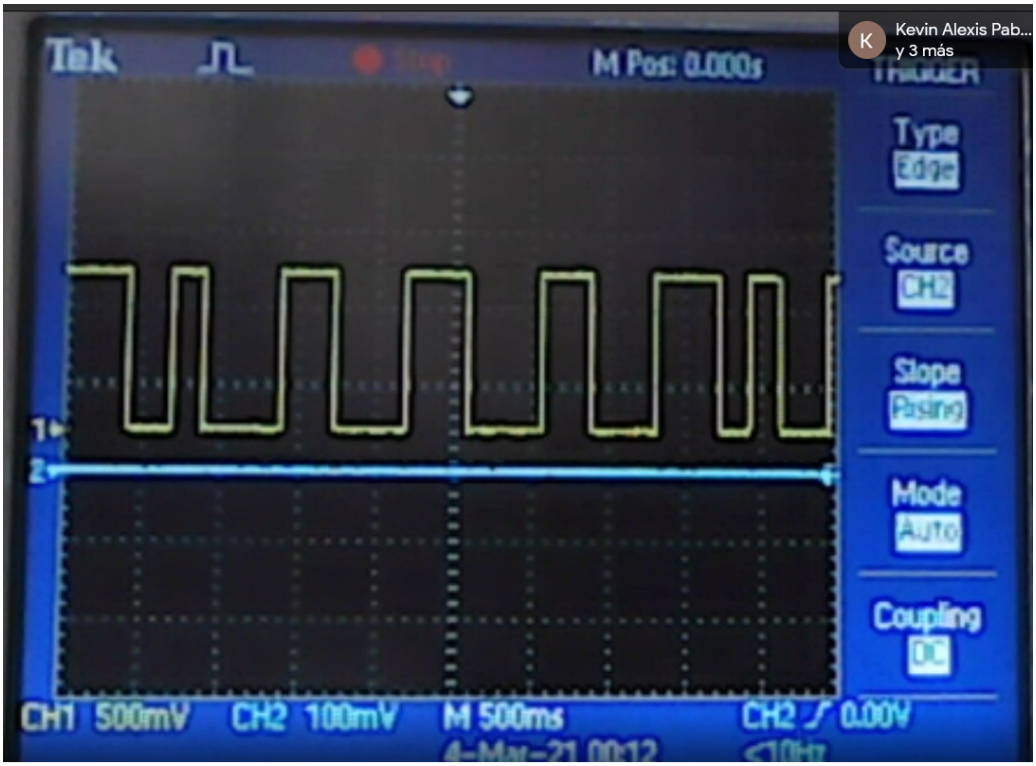


Figura 12: Func. Generada Escalon Aleatorio

Según se aprecia en las figuras anteriores, la implementación de las tres funciones presentadas en la consigna es correcta. Con las funciones generadas en clase, conseguimos obtener funciones que con los parámetros adecuados permiten visualizar en el osciloscopio distintos comportamientos interesantes para el estudio de futuros trabajos en la cátedra.

Referencias

[1] Códigos del TP1:
https://github.com/TomasLiendro/Deep_Learning/blob/master/TP1_Introduction%20to%20ML/