Práctica 1: Acondicionamiento de Señales y GNU Radio

Juan David Camacho Gonzalez *Código: 2210428*
Jordy Pabon Carrillo *Código: 2210397*
Valentina Arguellez Angulo *Código: 2215670*

Universidad Industrial de Santander, Bucaramanga, Colombia

Repositorio de GitHub: https://github.com/singeku/ComII_G2_equipo2

Declaración de Originalidad y Responsabilidad

Los autores de este informe certifican que el contenido aquí presentado es original y ha sido elaborado de manera independiente. Se han utilizado fuentes externas únicamente como referencia y han sido debidamente citadas. Asimismo, los autores asumen plena responsabilidad por la información contenida en este documento.

Uso de IA: Se utilizó el modelo de lenguaje Gemini como asistencia técnica para la estructuración del código en LaTeX, resolución de errores de compilación en Ubuntu/WSL, sintaxis de comandos de Git y revisión gramatical de la justificación técnica. El diseño del filtro, la implementación en GNU Radio y el análisis de resultados fueron desarrollados íntegramente por los autores.

***Abstract*—This report shows the development and implementation of custom blocks for digital signal processing in GNU Radio using Python. Modules such as accumulators and differentiators were designed to analyze data behavior. Signal contamination by Gaussian noise was handled using a discrete moving average statistical filter. The effectiveness of this conditioning was validated by proposing its use in the stabilization of biopotentials, a critical stage in the processing of myoelectric signals for force control systems in the reproduction of hand movements. The results demonstrate that statistical averaging successfully attenuates noise fluctuations, guaranteeing a stable signal and maintaining an adequate latency for real-time applications.**

***Index Terms*—GNU Radio, Moving Average Filter, Signal Processing, Python.**

I. Introducción

El procesamiento digital de señales ha experimentado una evolución significativa con la consolidación de la Radio Definida por Software (SDR). Plataformas de código abierto como GNU Radio han transformado el análisis y la transmisión de datos, permitiendo a los desarrolladores trascender el uso de herramientas predefinidas [2], [3] para programar y compilar algoritmos personalizados en lenguajes como Python. Esta flexibilidad es crucial para adaptar el procesamiento de la información a las exigencias matemáticas y físicas de cada entorno.

En el desarrollo de la práctica, se exploran los fundamentos de la creación de bloques de procesamiento en tiempo real. En una primera etapa, se hace la implementación de operaciones matemáticas (acumulación y la diferenciación). Estos módulos conforman la base para el análisis del comportamiento dinámico de los sistemas y la evaluación de tasas de cambio en las señales a lo largo del tiempo.

Por otro lado, un desafío crítico en la instrumentación y adquisición de señales físicas es la presencia de ruido. El acondicionamiento estadístico se vuelve obligatorio en implementaciones de alta precisión; por ejemplo, durante el diseño y validación de un sistema de control de fuerza en la reproducción de movimientos básicos de la mano, donde la presencia de componentes ruidosas en las señales mioeléctricas puede desestabilizar la etapa de inicial. Para soluionar este problema, se propone la implementación de técnicas estadísticas de suavizado.

Finalmente, el documento detalla los resultados obtenidos tras la integración de los módulos de acumulación, diferenciación y filtrado estadístico, en la Sección III se encuentran las conclusiones derivadas del análisis del sistema.

II. Metodología

II-A. Acondicionamiento Estadístico: Filtro de Media Móvil

La presencia de ruido Gaussiano es un desafío inherente en la adquisición de señales físicas. Como mejora a los algoritmos base solicitados, y con el objetivo de mitigar las fluctuaciones aleatorias para aumentar la relación señal a ruido (SNR), se diseñó e implementó un bloque de procesamiento estadístico original. La estrategia seleccionada corresponde a un filtro discreto de media móvil, regido por la siguiente ecuación de diferencias [1]:

$$y[n] = \frac{1}{N} \sum_{i=0}^{N-1} x[n-i] \tag{1}$$

Donde $N = 10$ representa el tamaño de la ventana de promediado, $x[n]$ es la señal de entrada contaminada y $y[n]$ es la salida estabilizada.

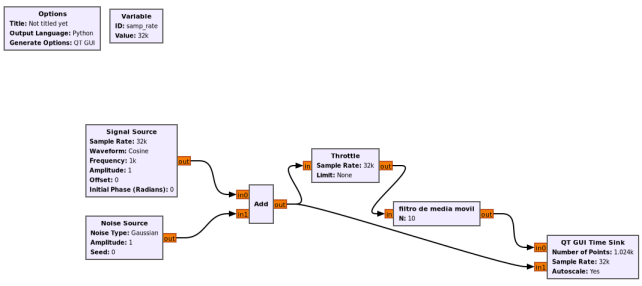
```
def __init__(self, N=10):
    # Inicializacion del buffer de tamano N
    self.N = N
    self.buffer = np.zeros(N)

def work(self, input_items, output_items):
```

```
    in_sig = input_items[0]
    out_sig = output_items[0]

    for i in range(len(in_sig)):
        # Desplazamiento circular para nueva muestra
        self.buffer = np.roll(self.buffer, -1)
        self.buffer[-1] = in_sig[i]

        # Calculo del promedio estadistico
        out_sig[i] = np.mean(self.buffer)

    return len(out_sig)
```

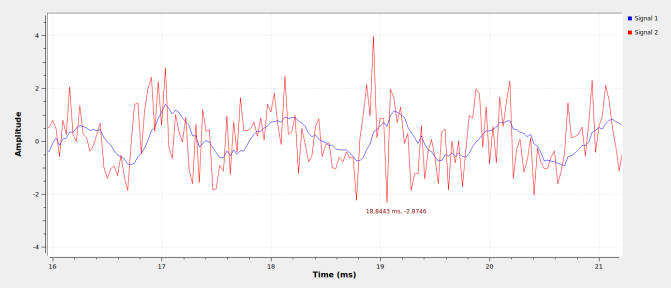
Listing 1: Implementación del bloque de media móvil con buffer circular.

Como se observa en el Código 1, la implementación se alejó de una estructura iterativa simple para adoptar un manejo de buffer circular mediante la función `np.roll`. Esta mejora permite que el algoritmo mantenga un historial exacto de las últimas N muestras con una carga computacional reducida, garantizando que el cálculo de `np.mean` se realice siempre sobre la ventana temporal correcta, evitando derivas en el promedio.

II-A1. Implementación y Resultados: La arquitectura del sistema se construyó utilizando el entorno de procesamiento por bloques. El filtro se programó desde cero mediante un *Embedded Python Block* y se integró como una mejora al diseño base. Como se evidencia en el flujograma de la Figura 1(a), la señal original se somete a una fuente de ruido antes de ingresar al bloque diseñado, permitiendo evaluar su desempeño dinámico.



(a) Flujograma con el módulo estadístico en Python.



(b) Comparación: señal ruidosa vs. señal estabilizada.

Figura 1: Diseño original y validación del filtro de media móvil para el acondicionamiento de la señal.

La capacidad de programar e integrar diseños algorítmicos originales es un requerimiento crítico en aplicaciones de alta complejidad, como el diseño y validación de un sistema de control de fuerza para una prótesis mioeléctrica con enfoque en la reproducción de movimientos básicos de la mano [4]. En este contexto, las señales biopotenciales crudas presentan una alta varianza estocástica que puede desestabilizar los comandos de la etapa inicial. Aplicar este promedio estadístico se vuelve el paso previo obligatorio para garantizar referencias estables.

Al ejecutar el sistema, los resultados obtenidos en la Figura 1(b) demuestran la eficacia de la mejora implementada. La estadística aplicada atenúa significativamente los picos del ruido Gaussiano. Aunque persiste un leve rizado debido al uso de una ventana de integración pequeña ($N = 10$), este diseño garantiza una baja latencia computacional, un factor indispensable para cumplir con el tiempo de respuesta inmediato que exige el control en tiempo real de actuadores mecánicos.

II-B. Acumulador discreto (con memoria)

Un acumulador es un sistema cuya salida corresponde a la suma acumulada de la señal de entrada. En tiempo discreto, si $x[n]$ es la señal de entrada, el acumulador ideal se define como:

$$y[n] = \sum_{k=0}^{n} x[k]. \tag{2}$$

De manera equivalente, puede escribirse en forma recursiva como:

$$y[n] = y[n-1] + x[n], \tag{3}$$

lo cual es especialmente útil para implementación en tiempo real, ya que la salida en el instante n se obtiene sumando la muestra actual a la salida anterior.

II-B1. Motivación de la memoria en GNU Radio (procesamiento por bloques): En GNU Radio, los bloques de tipo `sync_block` procesan la señal en vectores de N muestras durante cada llamada al método `work()`. En consecuencia, si se implementa un acumulador que calcule únicamente la suma acumulada *dentro del vector actual*, se obtiene una salida correcta *solo localmente*, pero el acumulado puede reiniciarse cada vez que `work()` se ejecute de nuevo con un nuevo bloque de datos.

Para reproducir el comportamiento del acumulador ideal $y[n] = y[n-1] + x[n]$ durante una ejecución continua, es necesario almacenar un **estado interno** que represente el valor acumulado al final del bloque anterior. Esta variable de estado (memoria) actúa como condición inicial para el siguiente bloque de muestras.

II-B2. Modelo matemático por bloques: Sea $\mathbf{x} = [x_0, x_1, \ldots, x_{N-1}]$ el vector de entrada en una llamada a `work()` y sea A el acumulado anterior (estado interno). La salida por bloques se define como:

$$\mathbf{y} = A + \mathrm{cumsum}(\mathbf{x}), \tag{4}$$

donde $\mathrm{cumsum}(\cdot)$ representa la suma acumulada elemento a elemento. El estado interno se actualiza con el último valor del acumulado del bloque:

$$A \leftarrow y_{N-1}. \tag{5}$$

De este modo, el acumulador mantiene continuidad temporal: el primer valor del bloque actual parte del acumulado total obtenido en el bloque anterior.

II-B3. Implementación del bloque (Embedded Python Block): La implementación se realizó mediante un bloque personalizado usando *Embedded Python Block*. El bloque se diseñó como `gr.sync_block` con una entrada y una salida de tipo `float32`. Para conservar continuidad entre bloques se añadió la variable de estado `acum_anterior`, inicializada en 0.

```
def __init__(self):
    # Memoria: permite continuidad entre llamadas a
    work()
    self.acum_anterior = 0.0

def work(self, input_items, output_items):
    x = input_items[0]      # entrada
    y0 = output_items[0]    # salida
    N = len(x)

    # Calculo vectorial usando la suma acumulada de
    NumPy
    acumulado = self.acum_anterior + np.cumsum(x)

    # Actualizacion de la memoria con el ultimo
    valor del bloque
    self.acum_anterior = acumulado[N - 1]

    # Escritura del vector de salida completo
    y0[:] = acumulado

    return len(x)
```

Listing 2: Implementacion del acumulador con persistencia de estado.

En cada llamada a `work()`, el bloque realiza los siguientes pasos:

- Lectura del vector de entrada: `x = input_items[0]`.
- Preparación del vector de salida: `y0 = output_items[0]`.
- Cálculo acumulado por bloques: `acumulado = acum_anterior + np.cumsum(x)`.
- Actualización de memoria: `acum_anterior = acumulado[N-1]`.
- Escritura de salida: `y0[:] = acumulado`.
- Retorno de N muestras procesadas: `return len(x)`.

II-B4. Flujograma de validación: Para validar el comportamiento del acumulador se construyó un flujo mínimo de prueba. El objetivo fue observar la correctitud de la suma acumulada y la continuidad de la salida en ejecución continua. El diagrama de bloques se muestra en la Figura 2.

Los parámetros generales de la prueba se resumen en la Tabla I.

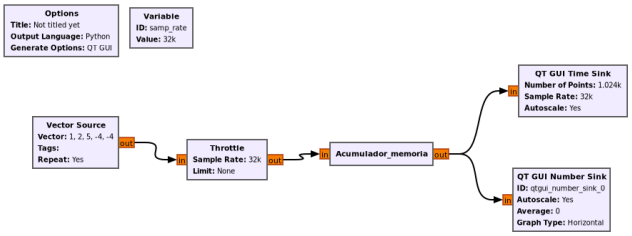


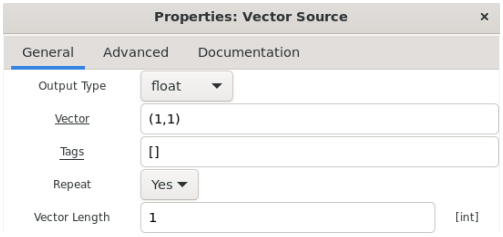
Figura 2: Flujograma de validación del acumulador con memoria.

Cuadro I: Parámetros generales del flujograma de validación.

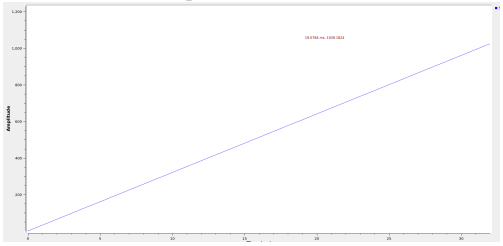
Bloque	Parámetro	Valor
Variable	`samp_rate`	32 kHz
Vector Source	Repeat	Yes
Throttle	Sample Rate	`samp_rate`
QT Time/Number Sink	Autoscale	Yes

II-B5. Validación y resultados (tres pruebas complementarias):

II-B5a. Prueba 1 (entrada constante, media $\neq 0$): Se configuró el *Vector Source* con un vector constante $x[n] = 1$. El acumulador ideal produce una rampa, lo cual evidencia la **deriva** inevitable cuando la señal de entrada posee componente DC (media no nula). Esto se comprueba en la Figura 3.



(a) Propiedades de entrada.

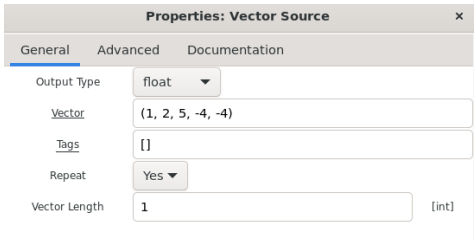


(b) Salida: Rampa creciente.

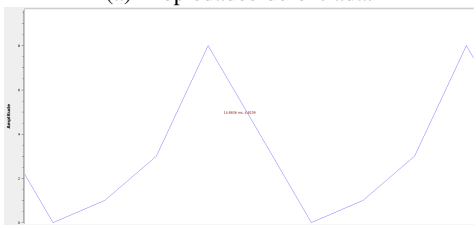
Figura 3: Prueba 1: Comportamiento ante entrada constante (media $\neq 0$).

II-B5b. Prueba 2 (secuencia de media cero): Para evitar el crecimiento indefinido, se empleó una secuencia repetitiva de media cero: $x[n] = [1,\ 2,\ 5,\ -4,\ -4,\ \ldots]$. La suma por periodo es cero, por lo que la salida permanece acotada (Figura 4).

II-B5c. Prueba 3 (onda cuadrada bipolar, media $= 0$): Se propuso una entrada bipolar ± 1. El acumulador incrementa



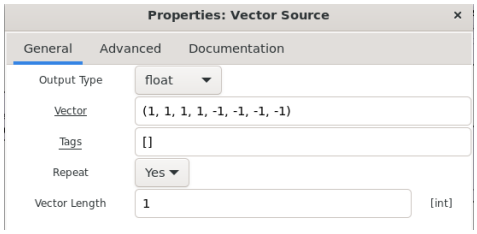
(a) Propiedades de entrada.



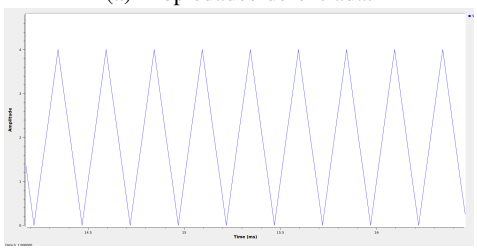
(b) Salida: Señal acotada.

Figura 4: Prueba 2: Comportamiento ante secuencia de media cero.

linealmente en el tramo positivo y decrementa en el negativo, generando una onda triangular (Figura 5), consistente con una integración discreta.



(a) Propiedades de entrada.



(b) Salida: Onda triangular.

Figura 5: Prueba 3: Comportamiento ante onda cuadrada bipolar.

II-B6. Aplicaciones: El acumulador es fundamental para la aproximación de integrales discretas, estimación de energía acumulada y construcción de métricas estadísticas.

II-C. Diferenciador Discreto

Un diferenciador es un sistema cuya salida corresponde a la derivada temporal de la señal de entrada. En tiempo continuo, si la señal de entrada es $x(t)$, el diferenciador ideal está definido como:

$$y(t) = \frac{d}{dt}x(t) \tag{6}$$

Este operador mide la tasa de cambio instantánea de la señal. Por ejemplo, si $x(t) = \sin(\omega t)$, entonces:

$$\frac{d}{dt}\sin(\omega t) = \omega\cos(\omega t) \tag{7}$$

lo cual implica que la derivada de una señal senoidal es otra señal senoidal desfasada $90°$ y con amplitud proporcional a la frecuencia angular. En sistemas discretos, la derivada se aproxima mediante diferencias finitas. La forma más sencilla del diferenciador discreto es:

$$y[n] = x[n] - x[n-1] \tag{8}$$

Esta expresión calcula la variación entre muestras consecutivas, permitiendo estimar la pendiente local de la señal. Si la señal permanece constante, la salida es cero; si la señal presenta un incremento o decremento, la salida toma valores positivos o negativos respectivamente.

II-C1. Implementación en GNU Radio: La implementación del diferenciador se realizó mediante un bloque personalizado en GNU Radio utilizando la opción *Embedded Python Block*. Este bloque fue programado para ejecutar la operación de diferencia discreta entre muestras consecutivas, de acuerdo con la expresión:

$$y[n] = x[n] - x[n-1] \tag{9}$$

El bloque fue diseñado como un `sync_block`, con una entrada y una salida de tipo `float32`, permitiendo procesar la señal muestra a muestra y mantener continuidad entre bloques mediante el almacenamiento del valor previo de la señal.

Para la validación del diferenciador se construyó un diagrama de bloques en GNU Radio compuesto por:

- **Signal Source:** Generador de señal, configurado con distintas formas de onda (senoidal, cuadrada) para evaluar el comportamiento del sistema.
- **Throttle:** Bloque utilizado para controlar la tasa de procesamiento y evitar el consumo excesivo de CPU en simulación.
- **Diferenciador (bloque Python):** Encargado de realizar la operación de diferencia discreta.
- **QT GUI Time Sink:** Utilizado para visualizar simultáneamente la señal original y la señal diferenciada en el dominio del tiempo.

```
def __init__(self):
    # Memoria: almacena la ultima muestra del bloque
     anterior
    self.x_prev = np.float32(0.0)

def work(self, input_items, output_items):
    x = input_items[0]
    y = output_items[0]

    # Diferenciacion del primer elemento usando el
    estado anterior
    y[0] = x[0] - self.x_prev
```

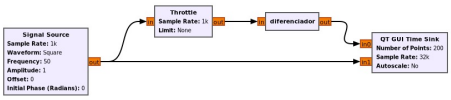
```
    # Calculo vectorial para el resto del bloque (
    Diferencia finita)
    y[1:] = x[1:] - x[:-1]

    # Actualizacion del estado para la proxima
    llamada a work()
    self.x_prev = x[-1]

    return len(y)
```

Listing 3: Implementacion del bloque diferenciador discreto mediante diferencias finitas.

Como se evidencia en el Código 3, la implementación del diferenciador optimiza el rendimiento mediante el uso de operaciones vectoriales. Al calcular el vector de salida a través de la resta de segmentos de memoria indexados (`x[1:] - x[:-1]`), se aprovecha la aceleración de hardware de NumPy, minimizando la latencia en el procesamiento de señales de alta frecuencia. Además, la gestión del estado `self.x_prev` asegura que el bloque detecte correctamente las transiciones en los límites de cada vector de datos, manteniendo la integridad de la derivada en ejecuciones de larga duración.

La Figura 6 presenta los flujogramas implementados para las distintas señales de excitación.

(a) Excitación con señal seno.



(b) Excitación con señal cuadrada.

Figura 6: Diagramas de bloques del diferenciador implementado en GNU Radio.

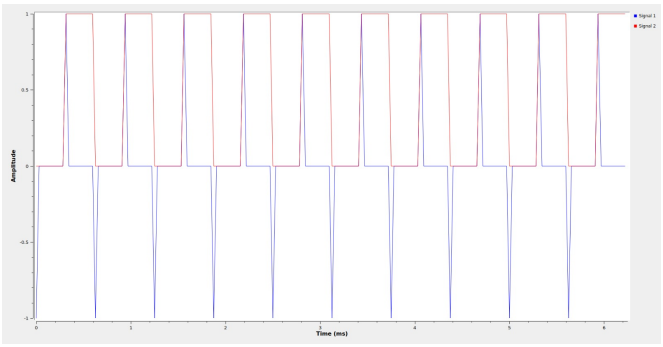
II-C2. Validación y Aplicación Práctica: El bloque diferenciador implementado fue evaluado utilizando dos tipos de señales de entrada: una señal senoidal y una señal cuadrada. El objetivo fue verificar experimentalmente el comportamiento del sistema frente a variaciones suaves y cambios abruptos en el tiempo, contrastando los resultados obtenidos con el modelo teórico de la derivada.

En el caso de la señal cuadrada (Figura 7(a)), se observó que la salida del diferenciador presentó impulsos de alta amplitud en los instantes donde la señal experimenta transiciones entre sus niveles máximo y mínimo. Durante los intervalos donde la señal permanece constante, la salida fue aproximadamente cero. Este comportamiento es coherente con la teoría, ya que la derivada de una función constante es nula, mientras que en los puntos de cambio brusco la pendiente es elevada. Por lo tanto, el diferenciador actúa como detector de flancos, generando picos positivos en las transiciones ascendentes y picos negativos en las descendentes.

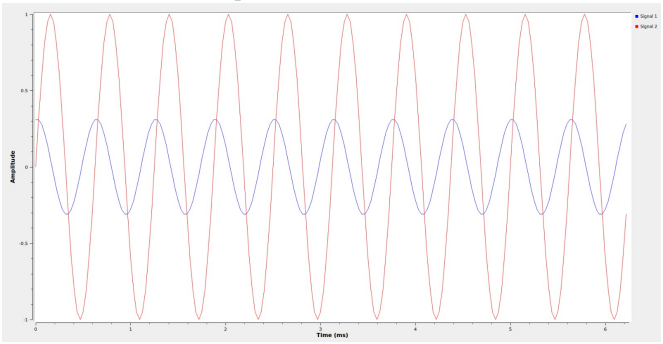
Por otro lado, al utilizar una señal senoidal como entrada (Figura 7(b)), se obtuvo una señal cosenoidal como salida, lo cual coincide con el resultado matemático esperado:

$$\frac{d}{dt}\sin(\omega t) = \omega\cos(\omega t) \tag{10}$$

Se evidenció además que la señal diferenciada presenta un desfase de $\pi/2$ radianes respecto a la señal original, lo cual es característico del proceso de derivación en funciones trigonométricas. Asimismo, la amplitud de la señal resultante depende de la frecuencia angular ω y de la frecuencia de muestreo utilizada en la implementación discreta.



(a) Respuesta ante onda cuadrada.



(b) Respuesta ante onda senoidal.

Figura 7: Resultados temporales de la implementación del diferenciador discreto.

Es importante mencionar que el diferenciador en tiempo discreto se implementa mediante una aproximación basada en diferencias entre muestras consecutivas. Debido a este proceso de discretización, pueden presentarse pequeñas variaciones en amplitud o ligeras irregularidades cuando la frecuencia de la señal de entrada aumenta o cuando la frecuencia de muestreo no es suficientemente alta.

III. Conclusiones

- El análisis del filtro estadístico de media móvil demostró que la selección del tamaño de la ventana de promediado establece un compromiso de diseño ineludible. Si bien el incremento en la cantidad de muestras atenúa de manera más eficiente la varianza del ruido Gaussiano, esto introduce proporcionalmente un retardo de fase en la salida, lo cual representa un parámetro restrictivo y limitante en sistemas de lazo cerrado que operan en tiempo real.
- A partir de los resultados, se concluye que el acumulador implementado reproduce matemáticamente el comportamiento de una integración discreta en ejecución continua. Se evidenció que ante entradas con componente en DC (media distinta de cero), el sistema presenta una deriva de crecimiento lineal; mientras que, con señales de media cero, la salida permanece acotada. Asimismo, la excitación con una señal cuadrada bipolar (± 1) generó un comportamiento triangular, consistente con la integración por tramos. Finalmente, se comprobó que la retención de estados pasados entre bloques de procesamiento es indispensable para evitar discontinuidades temporales en la señal integrada.
- El acondicionamiento digital se confirma como una etapa algorítmica crucial para el diseño y validación de sistemas de control de fuerza en prótesis electromecánicas. La estabilización de las señales mioeléctricas, mediante la reducción de ruido estadístico, mitiga la naturaleza estocástica del músculo. Esto garantiza que los actuadores reciban referencias de comando robustas, evitando oscilaciones o respuestas erráticas en la reproducción de movimientos básicos de la mano.
- La evaluación del diferenciador discreto validó empíricamente su respuesta como un sistema aproximador de derivadas. La prueba con señales cuadradas demostró su alta sensibilidad ante transiciones abruptas (actuando de forma homóloga a un filtro pasa altas), mientras que la respuesta ante señales senoidales corroboró el desfase teórico esperado en la derivación armónica. Esto reafirma el principio de que la diferenciación amplifica las componentes de alta frecuencia, haciendo obligatorio un filtrado previo en aplicaciones prácticas.

Referencias

[1] A. V. Oppenheim y R. W. Schafer, *Tratamiento de señales en tiempo discreto*, 3ra ed. Madrid, España: Pearson Educación, 2011.

[2] A. M. Wyglinski y D. Okin, *Software-Defined Radio for Engineers*. Norwood, MA, USA: Artech House, 2018.

[3] GNU Radio Project, "Embedded Python Block," *GNU Radio Wiki*, 2024. [En línea]. Disponible en: https://wiki.gnuradio.org/index.php/Embedded_Python_Block

[4] M. B. I. Reaz, M. S. Hussain, y F. Mohd-Yasin, "Techniques of EMG signal analysis: detection, processing, classification and applications," *Biological Procedures Online*, vol. 8, pp. 11-35, 2006.