# INFORME DE LABORATORIO 1

**Autores:** *Angie Paola Jaramillo Ortega, Juan Manuel Rivera Florez*

*Laboratorio de Electrónica Digital 3*
*Departamento de Ingeniería Electrónica y de Telecomunicaciones*
*Universidad de Antioquia*

## Resumen

Esta práctica de laboratorio se centra en la caracterización experimental de un motor DC mediante control PWM y medición de velocidad con encoder óptico. Se implementó un sistema de adquisición de datos basado en Raspberry Pi Pico, programado en Arduino y MicroPython, para comparar ambos entornos. El estudio analizó la respuesta del motor a escalones de voltaje PWM, registrando la velocidad angular (RPM) mediante un encoder de 20 ranuras. El montaje hardware utilizó un driver L298N y técnicas de conteo por interrupciones para calcular las RPM. Los resultados revelaron desafíos en la adquisición estable de pulsos del encoder, especialmente en velocidades extremas debido a vibraciones mecánicas y limitaciones de muestreo. El análisis comparativo demostró que aunque Arduino y MicroPython agilizan la creación de prototipos, introducen imprecisiones en tareas de control en tiempo real.

**Palabras clave:** Motor DC, control PWM, encoder, Raspberry Pi Pico, Arduino, MicroPython

## Introducción

Los motores de corriente continua (DC) son componentes fundamentales en sistemas de control y automatización, gracias a su simplicidad, eficiencia y facilidad de control. En aplicaciones industriales, robótica y sistemas embebidos, es esencial caracterizar su comportamiento para diseñar estrategias de control efectivas. Esta práctica se enfoca en la caracterización experimental de un motor DC, mediante el análisis de su respuesta ante señales PWM y la medición precisa de su velocidad utilizando un encoder óptico.

El objetivo es obtener un modelo dinámico del motor que relacione el voltaje aplicado (vía PWM) con la velocidad angular (RPM). Para ello, se implementó un sistema de adquisición de datos basado en una Raspberry Pi Pico, programada tanto en Arduino como en MicroPython, lo que permitió comparar el rendimiento de ambos entornos. Se controló el motor mediante señales PWM ajustables y se midió su velocidad en RPM usando un encoder óptico, registrando la curva de reacción del sistema ante escalones de voltaje.

## Implementación

### Configuración del Sistema

Para el control del motor DC, se utilizó el driver L298N, un puente H dual ampliamente empleado en aplicaciones de control de motores por su capacidad de manejar altas corrientes y su compatibilidad con microcontroladores de bajo voltaje. Este componente resultó fundamental para la interfaz entre la Raspberry Pi Pico y el motor, permitiendo un control preciso de velocidad mediante señales PWM.

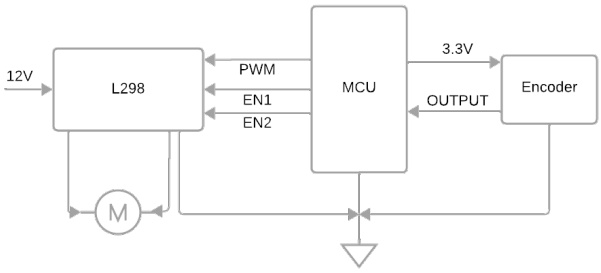


Figura **0-1**: Diagrama de bloques de Hardware

Para las conexiones eléctricas, la señal de control PWM proveniente del microcontrolador se conectó directamente a la entrada ENABLE del L298N, lo que permitió regular la velocidad del motor mediante modulación por ancho de pulsos. Las salidas del puente H del driver se conectaron a los terminales del motor, asegurando una correcta polarización para el giro en un sentido. La alimentación del motor y del driver se realizó mediante una fuente externa de 12V, mientras que el microcontrolador se alimentó mediente el puerto USB durante todas las pruebas.

El encoder óptico recibió alimentación de 3.3V directamente desde el microcontrolador, asegurando compatibilidad de niveles lógicos. Su señal de salida se conectó a una entrada digital para el conteo de los pulsos generados por el disco codificado de 20 ranuras.

Durante la implementación, uno de los principales problemas fue la variabilidad en las lecturas del encoder debido a vibraciones y movimientos no deseados. Para solucionarlo, se diseñó un soporte rígido que mantenía fijos tanto el motor como el sensor óptico para asegurar una distancia constante entre el disco del encoder y el fotosensor durante toda la práctica.

## 0.0.1. Arduino

### Sistema de medición de RPM

**Algorithm 1:** Cálculo de RPM desde encoder

**Data:** Constantes: PulsosPorVuelta = 20, ENCONDER_PIN = 16, bufferSize = 5
**Result:** Envía RPM por serial cada 5 segundos

```
1  Función contarPulsos()
2      contadorPulsos ← contadorPulsos + 1;
3  Procedimiento setup()
4      Inicializar Serial a 115200 bauds;
5      Configurar ENCONDER_PIN como
         INPUT_PULLUP;
6      Habilitar interrupción en ENCONDER_PIN
         (RISING);
7      tiempoAnterior ← millis();
8  Procedimiento loop()
9      while verdadero do
10         tiempoActual ← millis();
11         if tiempoActual - tiempoAnterior ≥
             1000 then
12             Deshabilitar interrupciones;
13             count ← contadorPulsos;
14             contadorPulsos ← 0;
15             Habilitar interrupciones;
16             dataBuffer[bufferIndex] ← count;
17             bufferIndex ← bufferIndex + 1;
18             tiempoAnterior ← tiempoActual;
19         end
20         if bufferIndex ≥ bufferSize then
21             sum ← 0;
22             for i ← 0 to bufferSize - 1 do
23                 sum ← sum + dataBuffer[i];
24             end
25             rpm ← ((sum / bufferSize) /
                 PulsosPorVuelta) × 60;
26             Enviar rpm por Serial;
27             bufferIndex ← 0;
28         end
29     end
```

El código Arduino presentado implementa un sistema básico para medir la velocidad de rotación (RPM) de un motor DC utilizando un encoder óptico.El núcleo del funcionamiento reside en el bucle principal, donde cada segundo se lee el valor acumulado de pulsos. Los valores de pulsos por segundo se almacenan en un buffer de 5 posiciones, equivalente a 5 segundos de datos.

Cuando el buffer se llena, el programa calcula el RPM promedio mediante una fórmula que considera los pulsos

acumulados durante el período de muestreo. La conversión a revoluciones por minuto se realiza dividiendo el promedio de pulsos por segundo entre el número de ranuras del disco encoder y multiplicando por 60 para escalar a minutos. Este valor final proporcionan una medición estable al promediar múltiples muestras, lo que reduce el impacto de variaciones momentáneas en la velocidad del motor.

El planteamiento inicial del código medía los pulsos del encoder cada segundo y calculaba las RPM de forma instantánea, pero esta aproximación presentaba dos problemas principales. En primer lugar, las lecturas variaban significativamente entre cada medición debido a pequeñas irregularidades mecánicas en el motor, vibraciones o imperfecciones en el disco del encoder.

Para resolver estos problemas, se implementó un sistema de promediado que almacena cinco mediciones consecutivas (equivalentes a cinco segundos de datos) antes de calcular las RPM. Esta solución surgió tras analizar experimentalmente el comportamiento del sistema: se observó que al tomar múltiples muestras y promediarlas, las fluctuaciones aleatorias se compensaban entre sí, dando como resultado valores más estables y representativos de la velocidad real del motor.

**Algorithm 2:** Generación de señal PWM

**Data:** Constantes:
PWM_PIN = 15,
PWM_FREQ = 250 Hz,
PWM_RESOLUTION = 8 bits
**Result:** Variación de señal PWM cada 3 segundos

```
1  Procedimiento setup()
2      Inicializar Serial a 115200 bauds;
3      Configurar PWM_PIN como salida;
4      Establecer frecuencia PWM a
         PWM_FREQ;
5      Establecer resolución PWM a
         PWM_RESOLUTION bits;
6  Procedimiento loop()
7      while verdadero do
8          for ciclo ← 0 to 100 step 10 do
9              valorPWM ← (ciclo × 255) / 100;
10             EscribirvalorPWMenPWM_PIN;
11             ImprimirvalorPWMporSerial;
12             Esperar3000ms;
13         end
14         Escribir 0 en PWM_PIN;
15         Esperar2000ms;
16     end
```

## Sistema de control en lazo abierto del motor a través de señal PWM

Este segundo código Arduino implementa un sistema de control PWM para un motor DC, diseñado específicamente para pruebas de caracterización. En el bucle principal, el programa ejecuta una secuencia automática que varía el ciclo de trabajo en incrementos del 10 %, desde 0 % hasta 100 %. Cada nivel de PWM se mantiene durante 3 segundos (3000 ms), permitiendo que el motor alcance su velocidad estable correspondiente antes de pasar al siguiente valor. La conversión del porcentaje de ciclo de trabajo a valor PWM se realiza mediante la función map(), que escala linealmente el rango 0-100 % a 0-255 (8 bits). Durante este proceso, el código envía por serial el valor PWM actual (0-255).

## Sistema de captura de la curva de reacción

Este tercer código Arduino implementa un sistema completo de caracterización dinámica para un motor DC, combinando control PWM y medición de velocidad mediante encoder. El núcleo del programa realiza una prueba automatizada que aplica escalones de PWM del 20 % (0 % a 100 % y vuelta a 0 %), manteniendo cada nivel durante 2 segundos. Durante este proceso, captura datos cada 4ms almacenando tres variables clave: tiempo transcurrido, valor PWM aplicado (0-255) y RPM calculados. La estructura de datos emplea un buffer circular capaz de almacenar hasta 5,500 muestras, suficiente para la prueba completa. Cada cambio de escalón actualiza el valor PWM, mientras que la rutina de interrupción cuenta los pulsos del encoder sin afectar el timing principal. Al finalizar la secuencia, el sistema envía todos los datos por serial en formato CSV y se detiene, proporcionando información lista para analizar la curva de respuesta del motor.

**Algorithm 3:** Control de Motor con PWM y Encoder

**Data:**

**Constantes:**

| | |
|---|---|
| PIN_ENCODER | = 16 |
| PIN_PWM | = 15 |
| PULSOS_POR_VUELTA | = 20 |
| FRECUENCIA_PWM | = 250 Hz |
| RESOLUCION_PWM | = 8 bits |
| ESCALON_PORCENTAJE | = 20 % |
| NUM_ESCALONES_SUBIDA | = 6 |
| NUM_ESCALONES_BAJADA | = 5 |
| MUESTRAS_POR_ESCALON | = 500 |

**Result:** Datos CSV de tiempo, PWM y RPM enviados por Serial

```
1  Función contarPulsos()
2      conteo_pulsos ← conteo_pulsos + 1;
3  Procedimiento setup()
4      Configuración inicial de pines, resolución y
         frecuencia de pwm.
5  Procedimiento loop()
6      while verdadero do
7          tiempo_actual ← millis();
8          if tiempo_actual -
             tiempo_ultima_muestra ≥ 4 ms then
9              Leer pulsos del encoder;
10             Calcular RPM
11             if escalon_actual <
                 NUM_ESCALONES_SUBIDA
                 then
12                 porcentaje_pwm ←
                     escalon_actual ×
                     ESCALON_PORCENTAJE;
13             else
14                 porcentaje_pwm ←
                     (TOTAL_ESCALONES -
                     escalon_actual) ×
                     ESCALON_PORCENTAJE;
15             end
16             valor_pwm ←
                 mapear(porcentaje_pwm, 0, 100,
                 0, 255);
17             Escribir valor_pwm en PIN_PWM;
18             Almacenar {tiempo, valor_pwm,
                 RPM} en buffer;
19         end
20         if tiempo_actual - tiempo_inicio_escalon
             ≥ 2000 ms then
21             Incrementar escalon_actual;
22             if escalon_actual ≥
                 TOTAL_ESCALONES then
23                 Enviar todos los datos por
                     Serial en formato CSV;
24                 Detener programa;
25             end
26         end
27     end
```

El principal desafío fue obtener mediciones consistentes del encoder en todo el rango de velocidades. A altas revoluciones, observamos que el sistema perdía pulsos, registrando menos vueltas de las reales. Esto ocurría porque las ranuras del disco pasaban demasiado rápido para que el sensor óptico las detectara con precisión, especialmente cuando superábamos cierta velocidad crítica. Las vibraciones mecánicas del motor y pequeños desajustes en la alineación del disco agravaban este problema.

En el extremo opuesto, a velocidades muy bajas enfrentamos el problema contrario: el motor podía estar girando, pero a veces el sistema marcaba 0 RPM. Esto sucedía porque el intervalo entre pulsos era tan largo que, si la medición coincidía justo en ese periodo de espera, no detectaba movimiento.

## Sistema de caracterización de un motor DC

Este código representa la implementación final del sistema integrado para caracterizar motores DC. Combina todas las funcionalidades desarrolladas previamente en un único programa robusto con tres modos de operación: espera (WAIT), control manual (MANUAL) y captura automática (CAPTURA).

El sistema utiliza un pin para generar señales PWM (configurado a 250Hz con resolución de 8 bits) que controlan el driver L298N, mientras que otro pin recibe los pulsos del encoder óptico mediante una interrupción hardware. La estructura principal emplea una máquina de estados que responde a comandos seriales: "PWM X"para control manual (donde X es el porcentaje de ciclo de trabajo) y "START X"para iniciar la captura automática con incrementos del X % especificado.

En modo captura, el código ejecuta una prueba automatizada que aplica escalones de voltaje al motor, primero ascendentes y luego descendentes, manteniendo cada nivel durante 2 segundos. Durante este proceso, captura datos cada 4ms almacenando el tiempo transcurrido, el valor PWM aplicado (0-255) y las RPM calculadas. Para calcular las RPM, cuenta los pulsos del encoder en cada intervalo y aplica la fórmula que considera las 20 ranuras del disco.

El buffer puede almacenar hasta 20,000 muestras, suficiente para pruebas prolongadas sin llenar la memoria del microcontrolador. Cuando finaliza la secuencia de escalones, el sistema envía todos los datos por el puerto serial en formato CSV. En modo MANUAL, reporta cada

500ms el valor PWM actual y las RPM medidas, cumpliendo con el requisito de monitoreo en tiempo real.

Este código, aunque funcional, presenta varias limitaciones técnicas que afectan su precisión y robustez en ciertos escenarios. Uno de los principales problemas radica en el cálculo de las RPM, que utiliza un método basado en conteo de pulsos por intervalos fijos. Este enfoque genera imprecisiones a velocidades muy bajas, donde el motor puede estar girando, pero el sistema registra 0 RPM porque no detecta pulsos en la ventana de muestreo de 4 ms. A altas velocidades, ocurre el efecto contrario: el encoder puede perder pulsos al no poder responder lo suficientemente rápido, subestimando la velocidad real del motor.

Otra limitación importante es el manejo de memoria. Aunque el buffer de 20,000 muestras parece suficiente, en pruebas prolongadas o con motores de respuesta lenta puede llenarse antes de completar la caracterización, lo que fuerza un reinicio del proceso. Además, la estructura de datos fija consume una parte significativa de la RAM disponible en la Raspberry Pi Pico, limitando la capacidad de añadir más funcionalidades sin afectar el rendimiento.

En cuanto a la temporización, aunque el código evita el uso de delay(), la dependencia de millis() para gestionar los intervalos de muestreo y los escalones PWM introduce un pequeño margen de error acumulativo. En pruebas de varios minutos, estos micro-retrasos pueden desincronizar ligeramente las mediciones, afectando la precisión de la curva de respuesta obtenida.

Estas limitaciones no invalidan el funcionamiento básico del sistema, pero sí resalta áreas de mejora para una próxima iteración, donde se podría implementar filtrado digital de señales o técnicas de corrección de errores para aumentar la confiabilidad de las mediciones.

**Algorithm 4:** Control de Motor con Modos MANUAL/CAPTURA

**Constantes:**
PIN_ENCODER = 16
PULSOS_POR_VUELTA = 20
**Data:** PIN_PWM = 15
FRECUENCIA_PWM = 250 Hz
RESOLUCION_PWM = 8 bits
MAX_MUESTRAS = 20000
**Result:** Salida CSV con datos tiempo, PWM y RPM

```
Función contarPulsos()
    Incrementar conteo_pulsos;
Procedimiento setup()
    Inicializar Serial y hardware (encoder + PWM);
    Imprimir mensaje inicial;
Procedimiento loop()
    while verdadero do
        Leer tiempo actual;
        if comando serial recibido then
            if comando es "START" then
                Inicializar parámetros de captura;
                Cambiar estado a CAPTURA;
                Imprimir encabezado CSV;
            else
                Establecer valor PWM manual;
                Cambiar estado a MANUAL;
            end
        switch estado do
            case MANUAL do
                Reportar PWM y RPM cada 500ms;
            end
            case CAPTURA do
                Muestrear RPM cada 4ms y almacenar;
                if cambio de escalón (cada 2000ms) then
                    Actualizar valor PWM según secuencia;
                    if secuencia completada then
                        Exportar datos y volver a WAIT;
                    end
                end
            end
        end
    end
```

## 0.0.2. Micropython

### Sistema de control en lazo abierto del motor a través de señal PWM

El código de MicroPython presentado implementa un sistema para medir la velocidad de rotación (RPM) de un motor DC y controlarlo mediante modulación por ancho de pulsos (PWM), diseñado para la caracterización del motor.

El núcleo del sistema de medición de RPM reside en la función encoder_handler() y su interacción con la función Start(). La función encoder_handler() es una rutina de interrupción que se ejecuta cada vez que el encoder óptico genera un pulso, incrementando un contador global (paso). En la función Start(), cada 500 milisegundos, se calcula el valor de RPM utilizando el incremento del contador paso y la relación conocida entre pulsos del encoder y revoluciones del motor, asumiendo 20 pulsos por revolución. Este cálculo se realiza con la fórmula rpm = paso * 60 / 20. Los valores de tiempo, referencia de PWM y RPM se imprimen en la salida estándar.

El sistema de control PWM se implementa a través de las funciones reftoPWM() y Move(). La función reftoPWM() convierte un valor de referencia de porcentaje (0-100 %) a un valor de ciclo de trabajo PWM (0-65535) que es compatible con la función duty_u16() de MicroPython. La función Move() aplica este valor de ciclo de trabajo a los pines de salida PWM (16 y 17) de la Raspberry Pi Pico, controlando así la potencia suministrada al motor. La dirección del motor se determina también en esta función, aunque el código proporcionado solo parece estar configurado para una dirección.

El planteamiento inicial del código se centra en la captura y presentación de datos de velocidad en intervalos de 500ms y el control básico de la velocidad del motor. Sin embargo, este enfoque carece de la capacidad de almacenar los datos capturados en un buffer para su posterior análisis y no implementa un control manual flexible del motor a través de la interfaz serial, como se especificó en los requerimientos.

**Algorithm 5:** Control de Motor con Encoder

**Data:** Variables globales: $paso \leftarrow 0$ Periféricos:
$r\_pwm \leftarrow PWM(Pin(16), 20000, duty\_u16 = 0)$
$l\_pwm \leftarrow PWM(Pin(17), 20000, duty\_u16 = 0)$
$enable \leftarrow Pin(18, OUT)$, $enable.on()$
$encoder \leftarrow Pin(19, IN)$
**Result:** Controla la velocidad del motor y muestra RPM cada 500 ms

**Función** *reftoPWM(ref)*
  $x \leftarrow (ref \times 65535)/100$;
  **return** $x$;
**Función** *encoder_handler(pin)*
  paso ← paso + 1;
**Función** *Move(k)*
  **if** $k > 65535$ **then**
    ...
  **else if** $k < -65535$ **then**
    ...
  **else if** $(k \geq 0)$ ***and*** $(k < 65535)$ **then**
    $l\_pwm.duty\_u16(0)$;
    $r\_pwm.duty\_u16(entero(k))$;
  **else if** $(k > -65535)$ ***and*** $(k < 0)$ **then**
    $r\_pwm.duty\_u16(0)$;
    $l\_pwm.duty\_u16(entero(k))$;
**Función** *Start(value)*
  paso ← 0;
  $ref \leftarrow 0$;
  $rpm \leftarrow 0$;
  Imprimir "Tiempo(ms), Duty($send\_data \leftarrow ticks\_ms()$;
  $timer\_start \leftarrow ticks\_ms()$;
  Llamar $Move(reftoPWM(ref))$;
  **while** *verdadero* **do**
    $timer\_elapsed \leftarrow ticks\_diff(ticks\_ms(), timer\_start)$;
    **if** $timer\_elapsed \geq 500$ **then**
      $rpm \leftarrow (paso \times 60)/20$;
      Imprimir ", , ".format($ticks\_diff(ticks\_ms(), send\_data)$, $ref$, $rpm$);
      $paso \leftarrow 0$;
      $timer\_start \leftarrow ticks\_ms()$;
**Procedimiento** *main()*
  $encoder.irq(trigger = Pin.IRQ\_RISING, handler = encoder\_handler)$;
  Llamar $Start(20)$;
Llamar $main()$;

## Sistema de captura de la curva de reacción

El código en MicroPython se diseñó para automatizar la captura de la curva de reacción de un motor DC, un proceso fundamental para caracterizar su comportamiento dinámico. La idea central es aplicar una serie de escalones en la señal de control del motor (PWM) y registrar la respuesta del motor en términos de velocidad (RPM) a lo largo del tiempo. Esta información permite obtener una representación gráfica de cómo el motor responde a cambios en la entrada, lo cual es crucial para el diseño de sistemas de control.

El código avanza sistemáticamente a través de un ciclo de incremento y decremento del ciclo de trabajo del PWM. Inicialmente, el ciclo de trabajo aumenta en pasos predefinidos hasta alcanzar el máximo, y luego disminuye hasta cero. En cada paso, se mide la velocidad del motor a intervalos regulares, y los datos se almacenan temporalmente en un buffer. Este enfoque automatizado permite obtener un conjunto completo de datos para la caracterización del motor de manera eficiente.

Uno de los problemas identificados en la implementación es la frecuencia de muestreo subóptima. Aunque el código realiza mediciones de velocidad a intervalos regulares, la frecuencia de muestreo resultante (10 Hz) es considerablemente menor que la frecuencia recomendada de 250 Hz para capturar con precisión la dinámica transitoria del motor. Esta discrepancia puede llevar a la pérdida de información valiosa sobre la respuesta del motor a los cambios en la señal de control, lo que afecta la precisión del modelo obtenido. La causa probable de esta limitación podría ser la necesidad de equilibrar los recursos de procesamiento de la Raspberry Pi Pico entre las diversas tareas, como la lectura del encoder, la generación de la señal PWM y la gestión del bucle de captura de datos.

Otro problema potencial es la resolución limitada en el control del PWM. El código incrementa el ciclo de trabajo en pasos del 20 %, lo que puede resultar en una curva de reacción con una cantidad relativamente pequeña de puntos de datos. Esto es particularmente problemático en el rango de bajo voltaje, donde la respuesta del motor puede ser no lineal y donde una mayor resolución en el control del PWM sería beneficiosa para una caracterización más precisa. La elección de un incremento del 20 % podría haber sido una decisión de diseño para simplificar la implementación o para reducir el tiempo total requerido para la prueba, aunque esto se hace a expensas de la granularidad de los datos obtenidos.

**Algorithm 6:** Control de Motor con Rampa de Velocidad y Registro

**Data:** Variables globales:... Periféricos:...
**Result:** Incrementa la velocidad del motor al máximo y luego la reduce a cero, registrando tiempo, duty cycle y RPM.

```
Función reftoPWM(ref)
Función encoder_handler(pin)
Función Move(k)
Función Start(value)
    paso ← 0;
    ref ← 0;
    rpm ← 0;
    direc ← 1;
    end ← verdadero;
    buff ← [];
    send_data ← ticks_ms();
    chang_ref ← ticks_ms();
    timer_start ← ticks_ms();
    Llamar Move(reftoPWM(ref));
    while end do
        timer_elapsed ←
          ticks_diff(ticks_ms(), timer_start);
        if timer_elapsed ≥ 100 then
            rpm ← (paso × 60)/20;
            Agregar a buff: ", ,
              ".format(ticks_diff(ticks_ms(), send_data),
              ref, rpm);
            paso ← 0;
            timer_start ← ticks_ms();
        if ticks_diff(ticks_ms(), chang_ref) ≥
          3000 then
            if ref ≥ 100 then
                direc ← 0;
            if direc == 0 and ref == 0 then
                end ← falso;
            if direc == 1 then
                ref ← ref + value;
            else if direc == 0 then
                ref ← ref − value;
            Llamar Move(reftoPWM(ref));
            chang_ref ← ticks_ms();
    Imprimir buff;
    Limpiar buff;
    buff ← ["Tiempo(ms), Duty(Llamar
      Move(0);
Procedimiento main()
    encoder.irq(trigger =
      Pin.IRQ_RISING, handler =
      encoder_handler);
    Llamar Start(20);
Llamar main();
```

Además, existe la posibilidad de pérdida de información transitoria importante. Dado que el muestreo de la velocidad se realiza cada 100 ms y el ciclo de trabajo del PWM cambia cada 3 segundos, es probable que se pierdan detalles significativos de la respuesta del motor durante los breves períodos de transición que siguen a cada cambio en el voltaje aplicado. Estos transitorios contienen información valiosa sobre la dinámica interna del motor, como el tiempo muerto y la constante de tiempo, que son fundamentales para un modelado preciso. Esta limitación es una consecuencia directa de las elecciones realizadas en la frecuencia de muestreo y la duración de los escalones de PWM.

**Algorithm 7:** Control de Motor con Comandos START y PWM

**Data:** Variables globales:... Periféricos:...
**Result:** Controla la velocidad del motor mediante comandos "START valor"(rampa hasta valor) y "PWM valor"(duty constante por 5s).

```
1  Función reftoPWM(ref)
2  Función encoder_handler(pin)
3  Función Move(k)
4  Función Start(value)
5      ...
6      while ref ≤ 100 do
7          ...
8      ...
9  Función PWM(value)
10     ... while ticks_diff(ticks_ms(), chang_ref) ≤ 5000 do
11         ...
12     Llamar Move(0);
13 Procedimiento main()
14     encoder.irq(trigger = Pin.IRQ_RISING, handler = encoder_handler);
15     while verdadero do
16         Imprimir Ïngrese un comando:";
17         Leer Comando desde la entrada estándar;
18         if Comando comienza con "START" then
19             Si el valor numérico en Comando está entre 1 y 100: Llamar Start(valor_numérico);
20             else
21                 Imprimir "Valor incorrecto.";
22         else if Comando comienza con "PWM" then
23             Si el valor numérico en Comando está entre 1 y 100: Llamar PWM(valor_numérico);
24             else
25                 Imprimir "Valor incorrecto.";
26         else
27             Imprimir Çomando incorrecto.";
28         Comando ← carácter nulo;
29 Llamar main();
```

Este código en MicroPython implementa un sistema para la caracterización de un motor DC, combinando la medición de velocidad (RPM) con el control de la señal PWM y la interacción con el usuario a través de comandos. La idea principal es proporcionar una herramienta flexible para analizar el comportamiento del motor bajo diferentes condiciones de funcionamiento.

El código se estructura en torno a varias funciones: encoder_handler para el manejo de los pulsos del encoder, reftoPWM y Move para la generación y aplicación de la señal PWM, Start para la captura de la curva de reacción del motor, y PWM para el control de velocidad constante. La función main actúa como el punto de entrada del programa y la interfaz con el usuario, interpretando los comandos "STARTz "PWM"para ejecutar las funciones correspondientes.

La función Start realiza una barrida automática del ciclo de trabajo del PWM, incrementándolo en pasos definidos por el usuario hasta alcanzar el máximo (100 %). Durante este proceso, se registran los datos de tiempo, ciclo de trabajo y velocidad del motor, permitiendo la generación de la curva de reacción. La función PWM, por otro lado, permite al usuario establecer un ciclo de trabajo fijo y medir la velocidad del motor bajo esa condición específica.

La interacción con el usuario a través de comandos permite una mayor flexibilidad en las pruebas del motor. El usuario puede elegir entre la captura automática de la curva de reacción o el control de velocidad constante, adaptando el sistema a sus necesidades de caracterización.

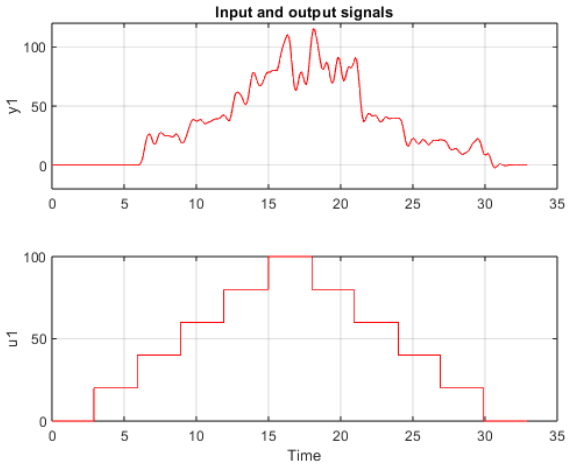


Figura **0-2**: Graficas de RPM(superior) y Duty(inferior) del motor.

$$G(s) = \frac{0{,}8061}{1 + 0{,}2355s} e^{-0{,}3856s} \qquad (0\text{-}1)$$

La ecuación que describe el comportamiento del motor (Función de transferencia), la cual tiene como entrada el Duty y la salida los RPM, esta tiene un retardo de 0.3856 segundos, con una constante proporcional de 0.8061 y un polo negativo. El comportamiento no es similar a la realidad, esto se debe a muchos factores, la medición de los RPM tiene mucho ruido, al llegar a un porcentaje de Duty, la señal del motor se satura, también se tiene en cuenta que es una FT de primer orden, esto hace de la respuesta una no cercana a lo experimentado.

# Conclusiones

- La gestión de interrupciones y temporizadores en MicroPython, aunque presente, puede no ofrecer la misma granularidad y predictibilidad que lenguajes de bajo nivel como C. Esto se traduce en variaciones en los tiempos de respuesta y dificultades para garantizar la ejecución precisa de tareas críticas en el tiempo, como la lectura de encoders y la generación de señales PWM con alta precisión. Además, el consumo de memoria y el sobrecosto de procesamiento de MicroPython pueden ser problemáticos en microcontroladores con recursos limitados, lo que restringe la complejidad de las tareas que se pueden realizar y la cantidad de datos que se pueden procesar y almacenar.
- A través de esta práctica, se evidenció que el uso de entornos como Arduino y MicroPython, si bien facilitan un rápido desarrollo de la solución, introducen limitaciones significativas en aplicaciones que requieren precisión temporal y manejo eficiente de recursos. La imposibilidad de acceder directamente al hardware y las abstracciones que ocultan detalles críticos del microcontrolador restringen el desempeño en tareas de adquisición de datos y control en tiempo real.En nuestro caso particular, esto se manifestó en dificultades para lograr mediciones completamente estables de RPM y un control PWM óptimo, especialmente en los rangos extremos de velocidad.
- Esta experiencia nos ha demostrado que la elección del entorno de desarrollo debe estar dictada por los requisitos específicos del proyecto. Para trabajos académicos y prototipos iniciales, Arduino y MicroPython pueden ser herramientas útiles. Sin embargo, cuando se requiere máximo

rendimiento, precisión y control sobre el hardware estos entornos de programación resultan ineficientes y difíciles de implementar en aplicaciones que comprometan requisitos funcionales computables.