INFORME DE LABORATORIO 2

Autores: *Daniel Santiago Arcila Gómez, Santiago Jose Vargas Higuera*

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

Resumen

En esta práctica se realizó la caracterización experimental de un motor DC utilizando una Raspberry Pi Pico. Se implementó un sistema que permite controlar el motor mediante señales PWM y medir su velocidad con un encoder óptico. Se desarrollaron versiones en Arduino (C++) y MicroPython, incluyendo un sistema de adquisición de datos y comunicación serial para registrar las curvas de respuesta del motor.

Palabras clave: Permutaciones gráciles, Backtracking, Branch and Bound, Lenguaje C, Medición de rendimiento, Optimización.

Diagrama de bloques

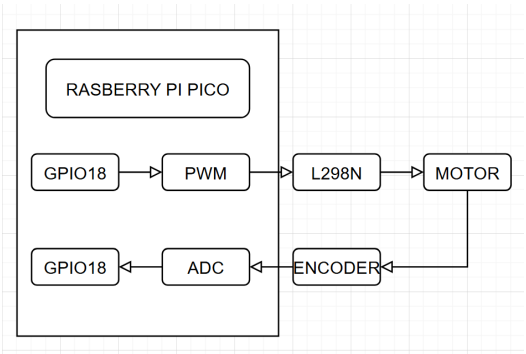

DIAGRAMA DE BLOQUES HARDWARE

Descripción del montaje

El motor se conectó al driver L298N, el cual fue controlado por la Raspberry Pi Pico a través de una señal PWM. El encoder óptico se fijó al eje del motor y sus señales se enviaron a un pin digital de la Pico para contar pulsos. Se diseñó un soporte para mantener fijo el conjunto.

1. Preparación del Hardware

Componentes utilizados

- Raspberry Pi Pico
- Motor DC pequeño sin reductor
- Driver de potencia (L298N)
- Encoder óptico con rueda dentada (20 ranuras)
- Fuente de alimentación de 5V
- Protoboard y cables

2. Medición de RPM

Se implementó una rutina para contar los pulsos del encoder y calcular las RPM cada 4 ms. La fórmula utilizada fue:

$$\text{RPM} = \frac{\text{Pulsos} \times 15000}{\text{Pulsos por revolución}}$$

Donde 15000 sale de (60 segundos para la conversion a rpm) dividido (0.0004 segundos que es el periodo de muestreo)

Pseudocódigo básico:

```
Cada 4 ms:
  Leer pulseCount
  Calcular RPM
```

```
Resetear pulseCount
```

3. Control del motor por PWM

Se implementó una salida PWM con resolución del 1 % desde 0 % hasta 100 %. Se verificó que los valores aplicados variaban adecuadamente la velocidad del motor y se encontró el mínimo PWM que permitía iniciar el giro (aprox. 20 %).

Pseudocódigo básico:

```
Si PWM esta entre 1 y 100:
    pwm_actual += 1
    aplicar pwm_actual al pin PWM
    calcular rpms
    si rpms > 0:
        pwm_minimo = pwm_actual
```

4. Captura de la Curva de Reacción

Se implementó una rutina que incrementa y luego decrementa el PWM en escalones definidos (por ejemplo 20 %). Cada paso se mantiene por 2 segundos, y se muestrea cada 4 ms para registrar RPM, PWM y timestamp.

Estructura de datos

```
struct DataPoint {
  timestamp: unsigned long
  rpm: float
  pwm: int
}
```

Pseudocódigo básico:

```
Si pwm_direction es ascendente:
    sumar paso del pwm
    Si PWM es mayor o igual a PWM maximo:
        saturamos a PWM maximo
        cambiamos la direccion a descendente
si pwm_direction es descendente:
    restamos paso del pwm
    Si PWM menor o igual a PWM minimo:
        saturamos a PWM minimo
        cambiamos la direccion a ascendente
```

5. Sistema de Caracterización

El sistema incluye una interfaz por comandos seriales:

- `START <valor>`: inicia la captura de datos con escalones de PWM
- `PWM <valor>`: aplica PWM manual y reporta continuamente RPM y PWM cada 500 ms

Flujo general del programa:

1. Esperar comandos por UART
2. Si `START`, iniciar captura con pasos de PWM
3. Si `PWM`, cambiar PWM manualmente
4. Medir RPM cada 4 ms
5. Si capturando, guardar en buffer
6. Al finalizar, enviar CSV por UART

Diagrama de Flujo

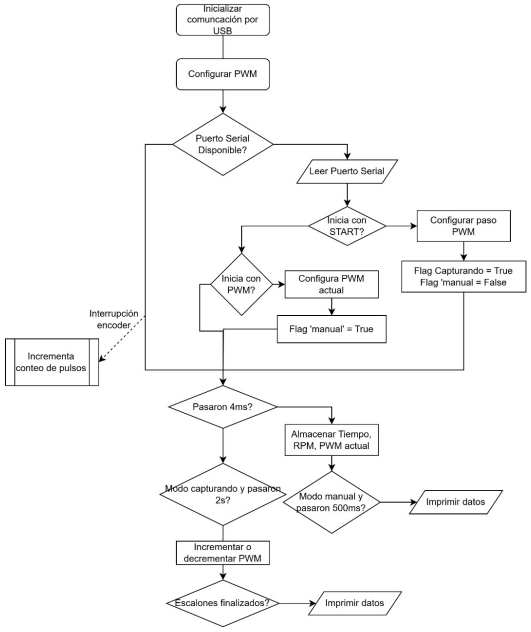

DIAGRAMA DE FLUJO

6. Modelo Analítico

El modelo propuesto para el motor es un sistema de primer orden con retardo:

$$G(s) = \frac{K}{\tau s + 1} e^{-Ts}$$

Donde se determinaron los parámetros K, τ y T ajustando curvas obtenidas experimentalmente mediante mínimos cuadrados.

Comparación Arduino vs MicroPython

Característica	Arduino (C++)	MicroPython
Precisión de lectura	Alta	Media
Resolución PWM	8-bit nativo	16-bit (ajustable)
Velocidad ejecución	Rápida	Moderada
Facilidad desarrollo	Media	Alta

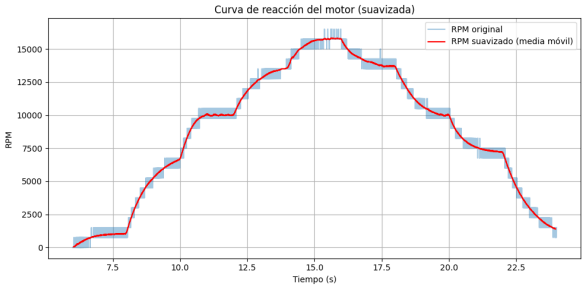

Figura **0-1**: Datos tomados con Arduino IDE

Por cuestiones de límite en el buffer se tomó una cantidad limitada de datos en micropython, y no se puede almacenar una gran cantidad de datos en su buffer como se permite con Arduno IDE.

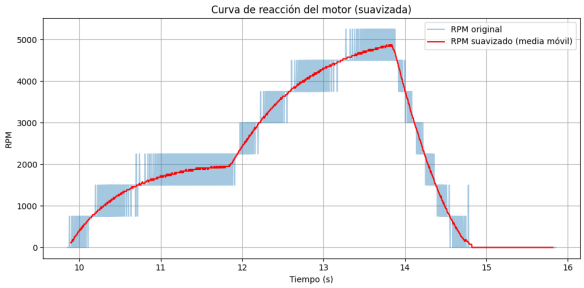

Figura **0-2**: Datos tomados con MicroPython

Problemas encontrados

- En MicroPython, algunos pulsos se perdían a altas velocidades: causado por la latencia del intérprete.
- El encoder óptico generaba ruido: se mitigó con un filtro por software ademas de un capacitor en hardware.
- La resolución PWM de 1 % requería configuración precisa del timer. Fue dificil manejar 2 timer distintos al mismo tiempo, uno para el buffer y los datos cada 4 ms y otro para el PWM.
- Buffer se llenaba para casos particulares. En pruebas largas (mayor a 16000 muestras), el buffer de datos en MicroPython puede agotar memoria. Esto nos afecta para casos donde el paso del pwm era menor a 7 % ya que necesitabamos mas memoria para poder almacenar todos los datos.
- Para casos donde las rpms eran menores a 750 no era posible tomar datos cada 4 ms pues en este tiempo el encoder aun no daba ni una vuelta.

Conclusión

La comparación entre plataformas mostró que Arduino ofrece mejor rendimiento en tareas de tiempo crítico, mientras que MicroPython facilita el desarrollo y depuración. En MicroPython no es posible utilizar la memoria de manera eficiente como se puede hacer en el entorno de Arduino, lo cual puede mejorar el desempeño a la hora de almacenar valores en el buffer.

Para la implementación de un sistema que permita caracterizar un motor, es importante adecuar el montaje para tener las mediciones adecuadas. Además, es importante tener en cuenta factores como el tiempo de muestreo, ya que si se toman muestras en un período demasiado corto, puede suceder que no se alcancen a registrar las rpms; por otro lado, si se tiene una frecuencia de muestreo muy pequeña, no se puede garantizar la capturar toda la forma de la señal y por consiguiente se pierde información necesaria para su caracterización, dando paso a fenómenos como aliasing.