Facultad de Ingeniería
Universidad de Buenos Aires

Laboratorio de control automático (86.22)

Trabajo Práctico N°2

Control de un motor de corriente continua

2^{do} Cuatrimestre, 2019

Bergler, Martín	98028	martin.bergler.es2a@gmail.com
Neumarkt Fernández, Leonardo	97471	leoneu928@gmail.com

Índice

1. Enunciado

Realizar el control de la velocidad de un motor de corriente continua utilizando un módulo Arduino. Para armar el banco de trabajo se debe implementar el siguiente hardware:

- Desarrollar un driver o actuador. Por ejemplo, para manejar el motor se puede utilizar un único transistor si el control de velocidad funcionará para un solo lado o un puente H para controlar la velocidad en ambos sentidos.
- Desarrollar un sensor para medir la variable a controlar. En el caso del motor, para medir velocidad, los sensores más habituales son encoders, ya sean comerciales o armados a partir de un dispositivo óptico (reflectivo o ranurado) y una rueda dentada acoplada al eje del motor. En otros casos puede ser necesario algún circuito acondicionador o amplificador de la variable de interés.

Para realizar el control en Arduino o similar se deben implementar en software las siguientes rutinas:

- Acondicionamiento de las mediciones del sensor. Para el caso del motor, sería el cálculo de la velocidad del motor a partir de la señal del encoder.
- Cálculo de la acción de control. El controlador a utilizar puede ser de cualquier tipo, siendo el más habitual el controlador PID que no requiere contar con un modelo del sistema a controlar.
- Generación y acondicionamiento de una señal de control que alimenta al actuador a partir de la salida de la rutina de control.
- Manejo de una comunicación serie con la PC.

Se debe realizar una interfase desde la PC con el Arduino de modo de poder:

- Ajustar las ganancias del controlador.
- Ajustar el valor de la señal de referencia.
- Visualizar las siguientes señales para verificar el correcto funcionamiento del controlador:
  - Variable a controlar.
  - Señal de referencia.
  - Señal de acción de control (por ejemplo, ciclo de trabajo en el caso de un actuador con PWM).

Al final del documento describimos la implementación requerida para esta comunicación. Se pide además:

- Ajustar las ganancias del controlador, justificando la elección de sus valores.
- Ensayar el sistema a lazo abierto y obtener al menos:
  - Un modelo simple para simular el comportamiento de la planta.
  - Evaluar por simulación el controlador a lazo cerrado que se va a implementar con el Arduino con el modelo identificado.
- Ensayar al sistema en lazo cerrado y obtener al menos:
  - La respuesta a escalones de distinta amplitud en la señal de referencia.
  - La respuesta a escalones de distinta amplitud y duración en alguna perturbación del sistema.

1.1. Especificaciones de la comunicación serie

La comunicación entra la PC y el *Arduino* debe ser serie punto a punto. Para esto se deben implementar las siguientes tramas de datos:

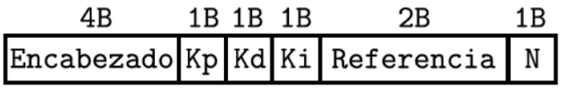

Figura 1: Trama de datos de la PC al Arduino.

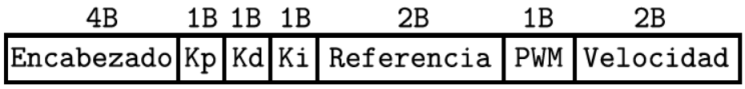

Figura 2: Trama de datos del Arduino a la PC.

Cada campo de las tramas de la Figura (1) y Figura (2) se explica en la Tabla (1). A estas tramas de datos se pueden agregar campos adicionales para *debug*.

Campo	Ancho	Descripción
Encabezado	4 B	Encabezado de la trama. Trama 1: abcd, Trama 2: efgh
Kp	1 B	Ganancia Proporcional
Kd	1 B	Ganancia derivativa
Ki	1 B	Ganancia integral
Referencia	2 B	Referencia de velocidad
N	1 B	Constate del filtro derivativo
PWM	1 B	Acción de control
Velocidad	2 B	Variable controlada

Tabla 1: Campos de la trama

2. Introducción

En esta sección se muestra una introducción teórica a los temas involucrados en la realización del Trabajo Práctico:

- *Motor de corriente continua*: convierte energía eléctrica en mecánica, provocando un movimiento rotatorio, gracias a la acción de un campo magnético. El principio de funcionamiento básico de un motor de CC se explica a partir del caso de una espira de material conductor inmersa en un campo magnético, a la cual se le aplica una diferencia de potencial entre sus extremos. Entonces, dado que cuando un conductor, por el que pasa una corriente eléctrica, se encuentra inmerso en un campo magnético, éste experimenta una fuerza según la Ley de Lorentz.

- *Encoder*: es un transductor rotativo, que mediante una señal eléctrica sirve para indicar la posición angular de un eje, velocidad y aceleración del rotor de un motor. En este caso, se compone de un disco ranurado radialmente conectado a un eje giratorio, que bloquean el paso de la luz emitida por la fuente de luz, emisores infrarrojos. A medida que el eje rota, el emisor infrarrojo emite luz que es recibida por el sensor óptico, generando los pulsos digitales a medida que la luz cruza a través del disco o es bloqueada en diferentes secciones de este. Esto produce una secuencia que puede ser usada para controlar la velocidad del motor.

- *Driver*: Si bien el motor funciona a 5V es necesario controlarlo, por lo tanto según la corriente que pase por la base, el transistor dejará pasar más o menos corriente del Colector al Emisor, de tal manera que el transistor actúa como un interruptor. En este caso se utilizó un transistor TIP122, el cual es un Darlington NPN. Se optó por este modelo ya que con una pequeña corriente de base se genera una corriente de colector mucho mayor que si se utilizara un único transistor discreto.

- *Arduino UNO*: es una placa de microcontrolador de código abierto basado en el microchip ATmega328P. La placa tiene 14 pines digitales, 6 pines analógicos y programables con el *Arduino IDE* a través de un cable USB tipo B. Se utilizó esta placa por la versatilidad que dispone y por las bibliotecas propias del lenguaje de *Arduino* como también el paquete de soporte que posee *Simulink*, por el que se puede conectar por los puertos del *Arduino* mediante bloques.

- *Solver*: Se llama *Solver* al método aplicado por *Simulink* para resolver un conjunto de ecuaciones diferenciales, las cuales representan un modelo. En el simulador se proveen varios *Solvers*. El tipo a utilizar dependerá de la simulación a realizar y se deberá tener en cuenta factores como el tipo de paso (fijo o variable), la dinámica del sistema, la mínima resolución que necesitamos para el problema en cuestión, la velocidad de cómputo, entre otros.

- *Control PID*: Un controlador PID (Proporcional, Integral, Derivativo) funciona mediante realimentación y mide y corrige la desviación a la salida de un sistema comparándola con una señal de referencia. El término proporcional sirve para que el error en estado estacionario tienda a cero. Esta constante controla los sobre picos que tiene el sistema frente a una excitación. A diferencia de los otros dos términos, este no tiene dependencia temporal.

  Luego se tiene el termino integral que busca eliminar el error en estado estacionario que no pudo ser corregido por el control proporcional. Lo que realiza el control integral es integrar el error para poder promediarlo, luego se multiplica por una constante Ki y finalmente se suma al proporcional. Por último tenemos el término derivativo, el cuál juega un rol importante cuando aparece una variación en el valor del error. En este caso se deriva la señal de error respecto del tiempo, luego se la multiplica por una constante Kd y finalmente se suma a los dos casos anteriores para formar el control propiamente dicho PID.

3. Implementación del motor

3.1. Hardware

3.1.1. Motor y driver

Para controlar correctamente el motor se implementó el siguiente circuito, en donde se utilizó un transistor darlington NPN (TIP122) como *driver* con un resistor de 500 Ω para establecer la corriente de base. A menor valor de resistencia, mayor valor de corriente de base, y en consecuencia mayor corriente de colector. Además se colocó un diodo 1n4148 para proveer un camino seguro para la corriente de acuerdo a los efectos inductivos que presenta el motor, si se desconecta de la alimentación, la energía acumulada por el inductor iría hacia el transistor y podría llegar al *Arduino*, por lo tanto el diodo se encarga que la corriente circule correctamente.

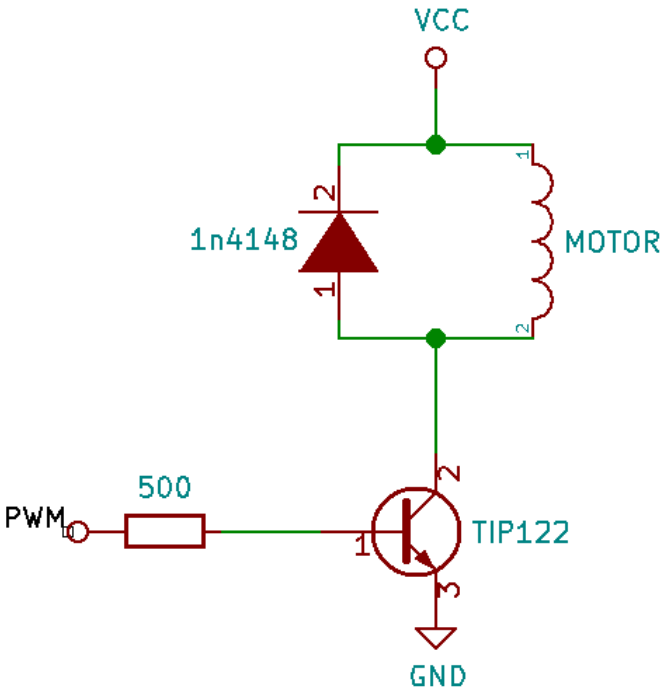

Figura 3: Circuito realizado.

3.1.2. Encoder

El *encoder* dispone de un sensor infrarrojo, el cual detecta las interrupciones de la señal óptica que establece el disco ranurado. En este proyecto se utilizó un disco con 32 ranuras. El *encoder* se alimenta con 5 V, los cuales son entregados por el mismo *Arduino*, además mediante el puerto 2 se lee la señal que detecta el sensor óptico. Entre la salida del sensor y la entrada del *Arduino* se colocó una resistencia de *pull down* de 1 kΩ.

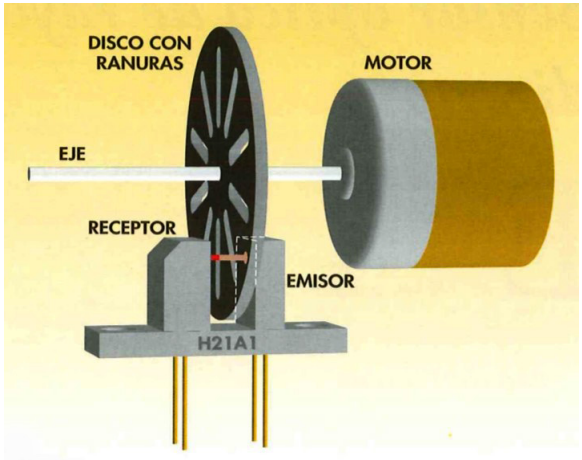

Figura 4: Esquemático de hardware del encoder.

Figura 5: Motor y encoder utilizados.

3.1.3. Comunicación serie

El *Arduino* UNO dispone de una unidad UART (*universally asynchronous receiver/transmitter*). La comunicación serie se realizó a través del puerto USB, sin embargo, si no se quisiera utilizar el puerto USB se puede realizar mediante los pines 0 (RX) y 1 (TX). El código empleado para la comunicación serie correspondiente se puede observar en el anexo.

En este proyecto se utilizaron los pines de 5 V y GND para alimentar tanto al motor como al *encoder*, el pin 2, útil para realizar interrupciones y el pin 5 para, el cuál es una salida de PWM.

Figura 6: Arduino UNO

3.2. Software

Para realizar el trabajo se utilizó el IDE (*Integrated Development Environment*) de *Arduino* como plataforma para programar el microcontrolador y *Matlab* para realizar la interfase de comunicación con el sistema y de esta manera ajustar valores y visualizar los datos. Cabe aclarar que las variables se definieron como globales, ya que son utilizadas por varias funciones.

3.2.1. Configuración y función principal

Inicialmente se tiene la función de configuración (*setup()*). En esta se configura la velocidad de comunicación en 9600 baudios y las interrupciones a utilizar. El pin 2 (PIN_INTERRUPT) se utiliza para que se active la interrupción que cuente cada vez que el *encoder* detecte una ranura de la rueda que pasa por este. Entonces se llamará a la función *contador_pulsos()*. Además se dispone de un *timer*, el cuál se activa luego de una ventana de tiempo *TW*, definida en 200 ms, la cuál llamará a la función *calcular_velocidad()*. A continuación se muestra la función descripta.

```
void setup()
{
  pinMode(PIN_INTERRUPT, INPUT_PULLUP);
  pinMode(PIN_PWM, OUTPUT);
  Serial.begin(9600);

  // Indico que cada vez que se detecte un flanco ascendente por el pin
   PIN_INTERRUPT,
  // entre a la función contador_pulsos(), la cuál suma un 1 a la variable cont:
  attachInterrupt(digitalPinToInterrupt(PIN_INTERRUPT), contador_pulsos, RISING)
   ;

  // Indico que cada TW*1000=200ms se realice una interrupcion que llame a la
  // función calcular_velocidad():
  tiempo.every(TW*1000,calcular_velocidad);
}
```

Como función principal se tiene *loop()*. En esta se recibe la trama de datos, si es posible, y además, cuando pasa el tiempo definido por la ventana de tiempo, se realiza el cálculo de la acción de control y se envía la trama con los datos actualizados. Finalmente se actualiza el valor de PWM en el pin correspondiente.

```
void loop()
{
  tiempo.update(); // Se llama a update para corroborrar si el intervalo de
   tiempo especificado termino
                   // y en caso afirmativo se llama a la función
   calcular_velocidad()
  if (Serial.available()>0)
  {
    recibir_trama();
  }

  if (interrupt_on==true)
  {
    interrupt_on = false;
    pid();
    enviar_trama();
  }

  analogWrite(PIN_PWM, pwm_); // Envio el valor del pwm al pin correspondiente
}
```

Por último se aclara que se tuvo que crear el tipo de dato *FLOATUNION_t*, el cuál se implementó, ya que no fue posible realizar una correcta comunicación con tipo de datos *float* o *int*, debido a que la función *Serial.write()* no enviaba correctamente este tipo de datos. A continuación se muestra el tipo *FLOATUNION_t* creado:

```
typedef union
{
  float number;
  byte bytes[4];
} FLOATUNION_t;
```

3.2.2. Calculo de velocidad a partir de señal del *encoder*

Inicialmente, para realizar el cálculo de la velocidad, es necesario conocer la cantidad de ranuras que el *encoder* pudo contar. Para esto se utiliza la función *contador_pulsos()*, la cuál se activa por medio de una interrupción.

```
void contador_pulsos()
{
  cont++;
}
```

Luego, una vez que transcurre la ventana de tiempo de 200 ms se calcula la velocidad, en RPM, mediante la función *calcular_velocidad()* y se inicializa el contador de ranuras en 0.

```
void calcular_velocidad()
{
  interrupt_on = true;
  vel = ((float)cont*60)/(CANT_RANURAS*TW);
  cont = 0;
}
```

3.2.3. Cálculo de la acción de control

Para el cálculo de la acción de control, se utilizaron las ecuaciones para realizar un PID discreto, como se describió en el trabajo práctico 1. Entonces se definió la salida del controlador por la ecuación 1.

$$u_k = P_k + I_k + D_k \tag{1}$$

Donde cada término se define a continuación:

$$P_k = K_p\,(r_k - y_k) \tag{2}$$

$$I_{k+1} = K_p K_i h \sum_{j=0}^{k} (r_k - y_k) = I_k + K_p K_i h\,(r_k - y_k) \tag{3}$$

$$D_k = \frac{\gamma}{\gamma + h} D_{k-1} - \frac{K_p K_d}{\gamma + h} (y_k - y_{k-1}) \tag{4}$$

Las constantes utilizadas en las ecuaciones (2), (3) y (4) son h, que es el período de muestreo, r_k es la señal de referencia, y_k es la variable medida, K_p la constante de proporcionalidad, K_i la constante de integración, K_d la constante de derivación y N el coeficiente del filtro.

Entonces se definió la siguiente función en *Arduino*:

```
void pid()
{
  float gamma = Kd/N;
  Pk = Kp*(ref - vel);
  Dk = (gamma/(gamma + TW))*Dk - (Kp*Kd/(gamma + TW))*(vel-vel_anterior);
  uk = Pk + Ik + Dk;
  if (uk>1)
    uk = 1;
  else if (uk<0)
    uk = 0;

  Ik = Kp*Ki*TW*(ref - vel) + Ik;
  vel_anterior = vel;
  pwm_ = 255*uk;
}
```

3.2.4. Comunicación Arduino-PC, PC-Arduino

Para realizar la comunicación se definieron dos funciones en *Arduino* y se utilizó *Simulink*.

Para enviar de la PC al *Arduino* se definió la función *recibir_trama()*. En esta se utilizan variables auxiliares de tipo *FLOATUNION_t* para poder enviarlas de forma correcta, como se explicó anteriormente. La lógica principal de la función consiste en que compara si se recibió una nueva trama, con el encabezado correspondiente y luego procede a leer los datos enviados para actualizar los valores de las constantes del controlador, así como también el valor de referencia. Además se definió que la referencia no sobrepase las 2400 RPM, ya que este fue el valor máximo que se adquirió al probar el motor y si se establece una referencia de valor mayor le llevará mayor tiempo al sistema hasta que vuelva a establecer su valor.

```
void recibir_trama() // PC al Arduino
{
  char header[4]="abcd";
  char header_recibido[4];
  FLOATUNION_t Kp_aux, Kd_aux, Ki_aux, ref_aux, N_aux;

  Serial.readBytes(header_recibido,4); // Leo el header para saber si comenzo
   una nueva trama
  // Comparo el header con los datos recibidos:
  if(header[0]==header_recibido[0]&&header[1]==header_recibido[1]&&header[2]==
   header_recibido[2]&&header[3]==header_recibido[3])
  {
    // Leo cada variable recibida
    Serial.readBytes(Kp_aux.bytes,4);
    Serial.readBytes(Kd_aux.bytes,4);
    Serial.readBytes(Ki_aux.bytes,4);
```

```
    Serial.readBytes(ref_aux.bytes,4);
    Serial.readBytes(N_aux.bytes,4);
  }

  // Reasigno los valores de las variables
  Kp=Kp_aux.number;
  Kd=Kd_aux.number;
  Ki=Ki_aux.number;
  ref=int(ref_aux.number);
  if(ref>2400) // Ajusto a este valor la referencia, ya que el motor no alcanza
   valores mayores
    ref=2400;
  else if(ref<0)
    ref==0;

  N=int(N_aux.bytes);
}
```

Para la comunicación del *Arduino* a la PC se creó la función *enviar_trama()*, la cuál escribe por el puerto serie la trama. Nuevamente se utilizaron variables auxiliares.

```
void enviar_trama() // Arduino a la PC
{
  // Convierto las variables pwm_ y ref a tipo float, para realizar
  // operaciones con otras variables de este tipo.
  float pwm_float=float(pwm_)*100/255;
  float ref_float=float(ref);

  FLOATUNION_t Kp_aux, Kd_aux, Ki_aux, ref_aux, pwm_aux, vel_aux;
  char header[4]="efgh";

  Kp_aux.number=Kp;
  Kd_aux.number=Kd;
  Ki_aux.number=Ki;
  ref_aux.number=ref_float;
  pwm_aux.number=pwm_float;
  vel_aux.number=vel;

  Serial.write(header,4);
  Serial.write(Kp_aux.bytes, 4);
  Serial.write(Kd_aux.bytes, 4);
  Serial.write(Ki_aux.bytes, 4);
  Serial.write(ref_aux.bytes, 4);
  Serial.write(pwm_aux.bytes, 4);
  Serial.write(vel_aux.bytes, 4);
  Serial.flush(); // Espera a que la transmision serial de datos se complete.
}
```

Para enviar las constantes del PID y la referencia se utilizó *Simulink*. Para enviar y recibir datos por el puerto serie se utilizaron los bloques *Serial Send* y *Serial Receive*, los cuales están configurados para tomar los datos del puerto correspondiente al que está conectado el *Arduino*. Además se colocaron *displays* para visualizar los datos de las tramas y *scopes* para ver la velocidad junto a la referencia y el PWM del motor. El diagrama correspondiente se muestra a continuación:

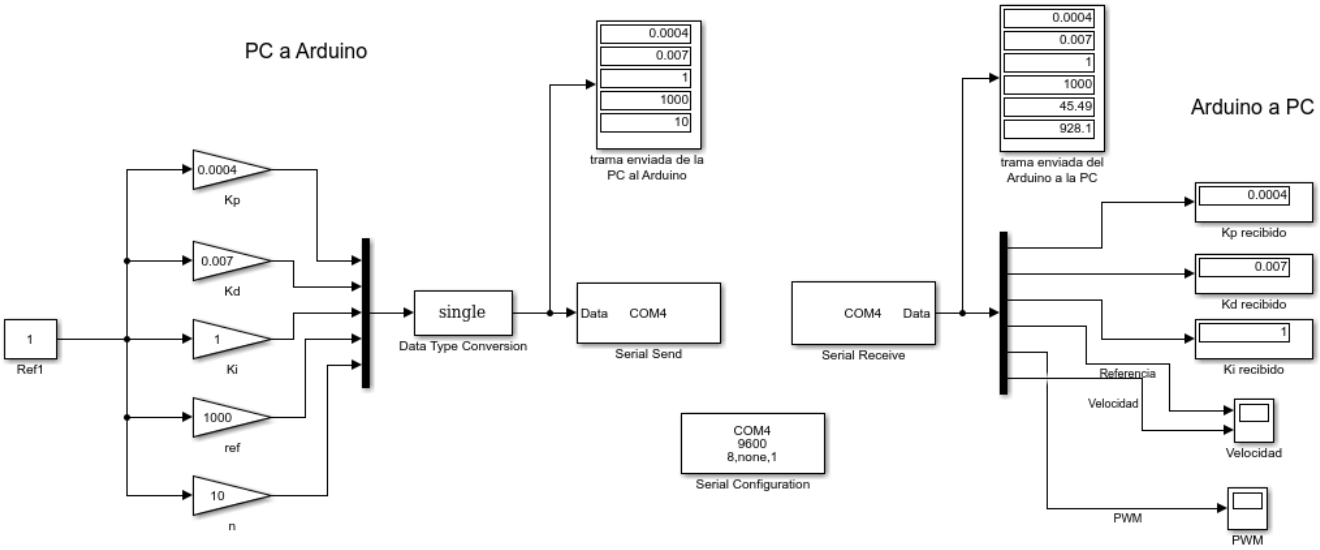

Figura 7: Comunicación mediante Simulink.

3.3. Ajustes del controlador

Para ajustar las ganancias K_p, K_i y K_d del controlador se probaron varios valores para estas y observando la respuesta del sistema se ajustaron para que se obtenga la respuesta esperada. Inicialmente se establecieron todas las ganancias en cero. Luego se incrementó K_p hasta conseguir una respuesta oscilatoria. Entonces se consiguió un valor de $K_p = 0,0004$. Luego se comenzó a aumentar el valor de K_d hasta obtener un valor de 0,007, el cuál redujo las oscilaciones. Finalmente se incrementó K_i, llevándolo a un valor de 1. Entonces con estos tres valores para las ganancias del controlador se obtuvieron los resultados esperados.

3.3.1. Ensayos a lazo cerrado

Con el PID ya configurado se realizaron diferentes ensayos a lazo cerrado.

Por un lado se realizó la respuesta al escalón, variando la referencia y observando el comportamiento del sistema. En la figura 8 se observa esta situación, en la que inicialmente se estableció un escalón de 500 rpm, luego se lo llevó a 1500 rpm, a continuación se pasó a 2200 rpm y finalmente termina en 1000 rpm.

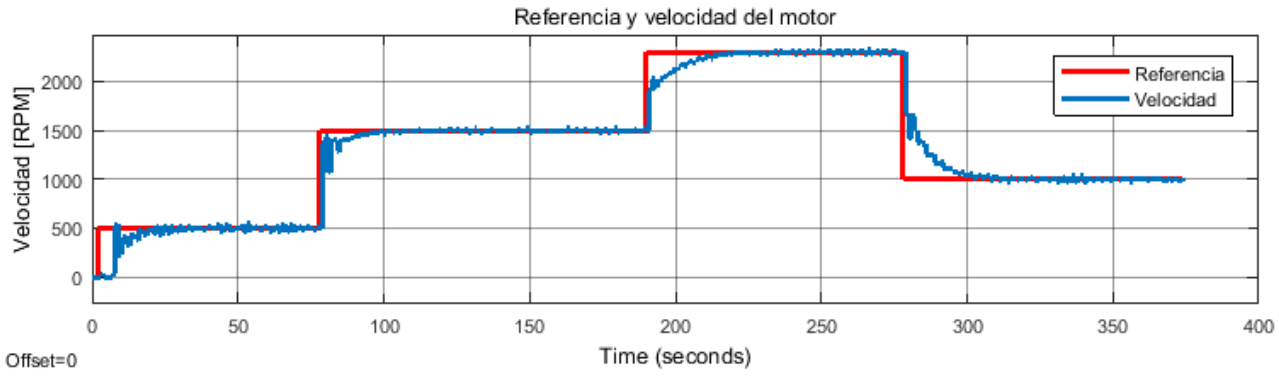

Figura 8: Respuesta al escalón a lazo cerrado.

Se puede observar que cuando cambia el valor de referencia, el sistema tarda un pequeño periodo de tiempo en acomodarse a la nueva referencia, pero luego se mantiene estable en este nuevo valor.

En la figura (9) se observa en ciclo de trabajo asociado a la situación anterior.

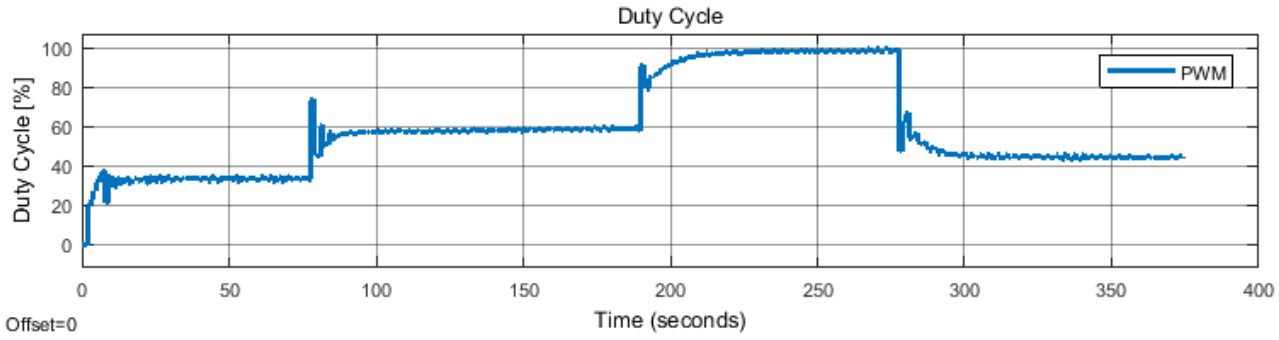

Figura 9: Duty Cycle de la respuesta al escalón a lazo cerrado.

Luego se configuró la referencia en un valor fijo de 1500 RPM y se perturbó el motor manualmente. En la figura 10 se observa esta situación.

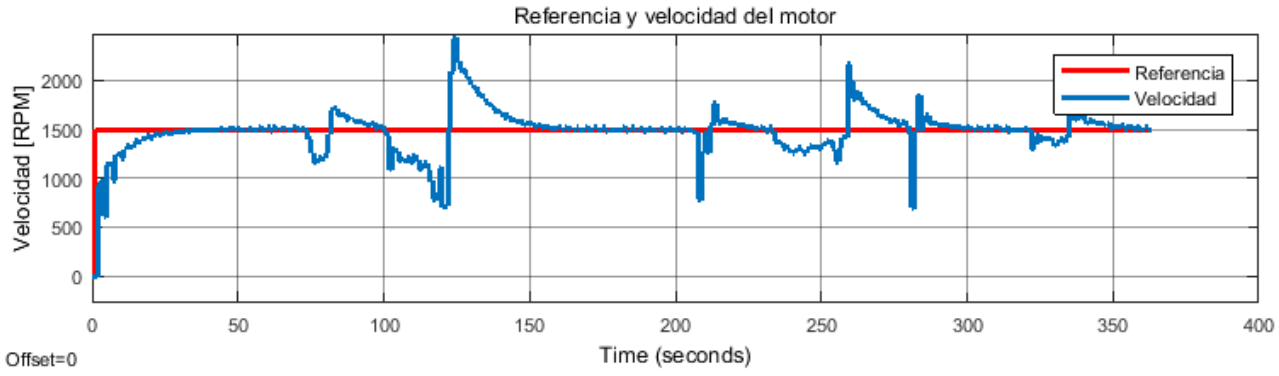

Figura 10: Respuesta a perturbaciones manuales.

Al perturbar el sistema se observa que la velocidad del motor disminuye, hasta que se lo deja de perturbar, lo cuál hace que la velocidad del aumente repentinamente. Entonces la velocidad consigue un valor mayor a la referencia, pero luego se alcanzará nuevamente en valor establecido.

En la figura 11 se muestra la variación del ciclo de trabajo correspondiente a las perturbaciones.

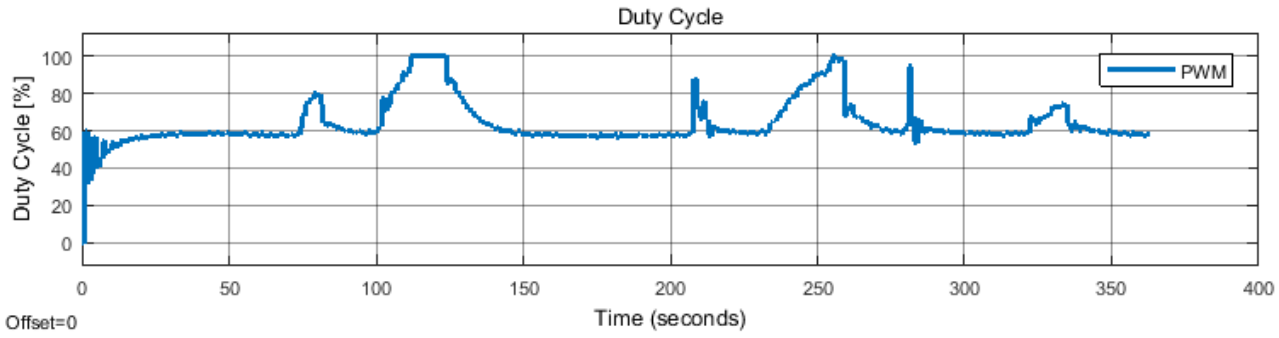

Figura 11: Duty Cycle de la respuesta a perturbaciones manuales.

4. Identificación del modelo

4.1. Ensayos a lazo abierto

Se escribió un programa en *Arduino*, que envíe por el pin de PWM el valor 255, es decir un ciclo de trabajo de 100 %, el cuál indica que el motor funciona a máxima velocidad. Entonces se midió la velocidad y se graficó con *Simulink*. El diagrama utilizado es el siguiente:

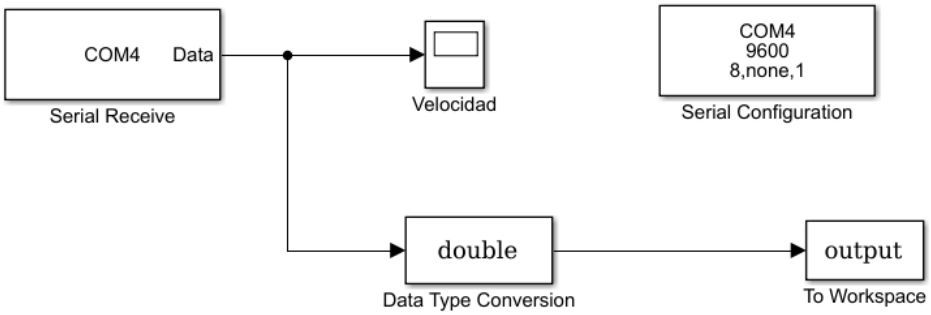

Figura 12: Diagrama para visualizar la velocidad del motor.

El resultado de la velocidad del motor, el cuál se obtiene por medio del *Scope* del diagrama de la figura 12, puede ser apreciado en la figura 13. La velocidad máxima se encuentra entre 2250 y 2438 rpm.

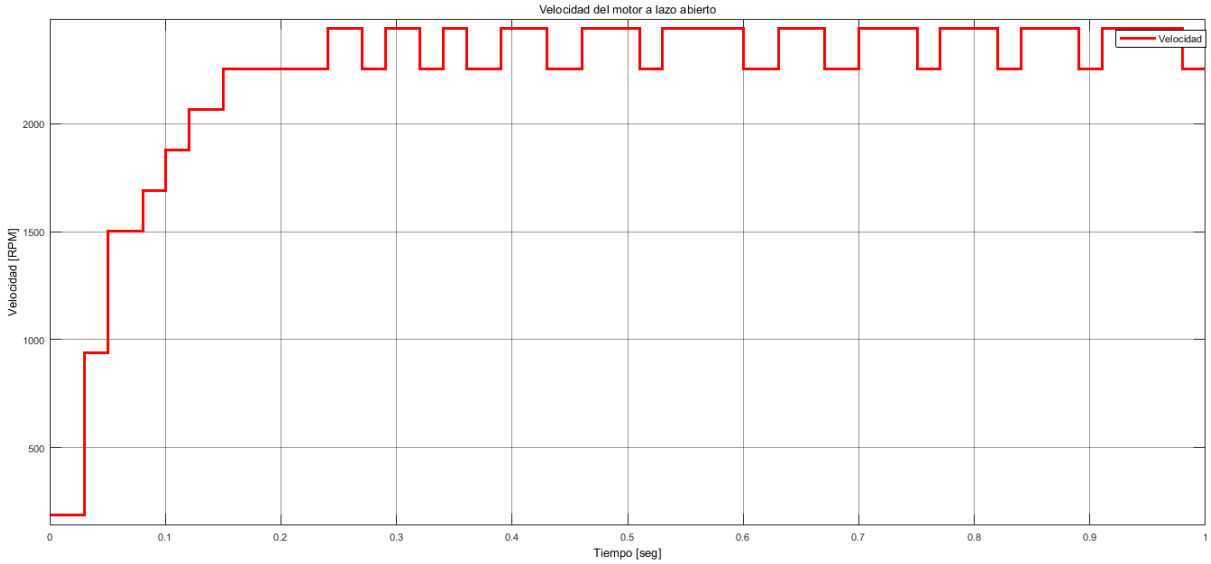

Figura 13: Velocidad del motor a lazo abierto.

Luego estos datos se exportaron al *workspace* de *Matlab* mediante el bloque *To Workspace* de *Simulink* y se buscó una transferencia mediante la función *arx()*, para poder identificar al sistema. La transferencia obtenida es la que se muestra en la ecuación (5).

$$H(z) = \frac{0{,}2129}{z^2 - 0{,}8184z} \tag{5}$$

En la siguiente figura se observa lo obtenido:

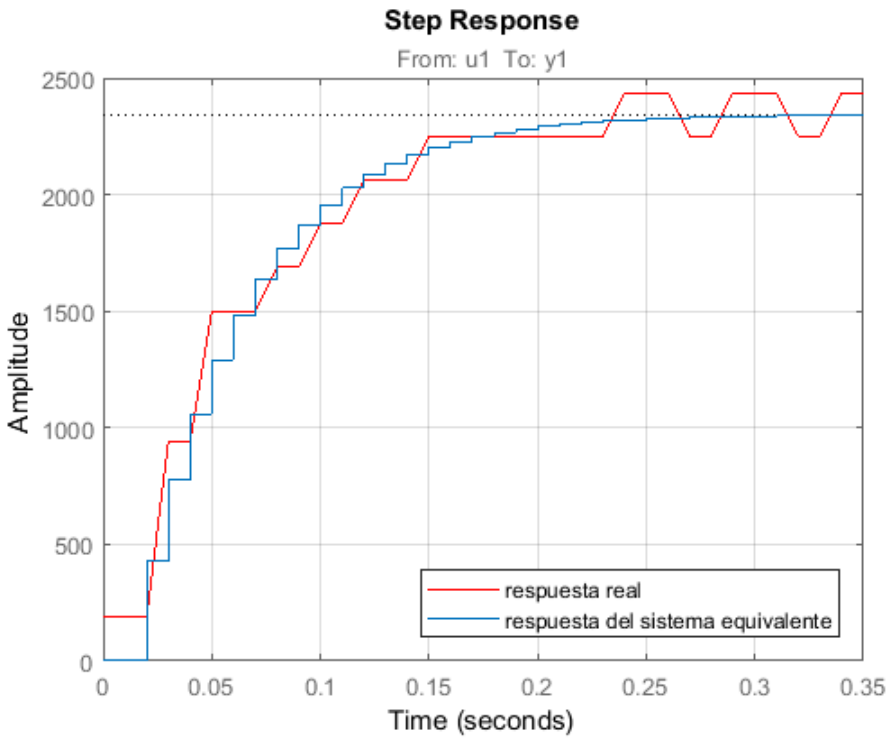

Figura 14: Identificación del motor.

4.2. Ensayo a lazo cerrado del modelo identificado

Una vez obtenida la transferencia se realizó la simulación del sistema a lazo cerrado, aplicando un bloque PID con las ganancias utilizadas al controlar el motor. El diagrama utilizado se muestra en la figura 15.

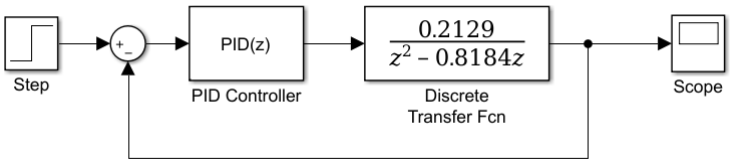

Figura 15: Diagrama de la simulación a lazo cerrado.

La respuesta del diagrama de la figura 15 se muestra a continuación.

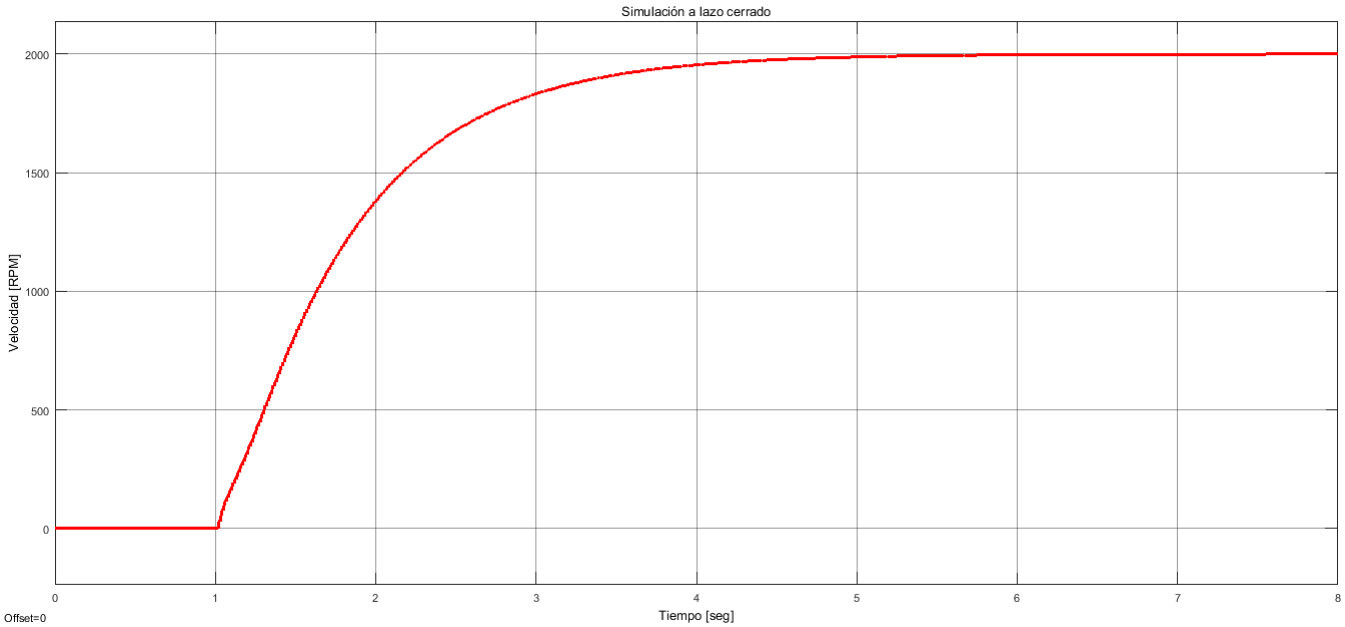

Figura 16: Simulación a lazo cerrado.

En la figura 16 se puede observar que se obtiene una respuesta esperada. Cabe aclarar que si se realiza un acercamiento a la curva, esta es discreta, como se espera.

5. Conclusiones

El propósito del presente trabajo práctico fue utilizar un PID discreto en una implementación práctica, la cual consistía en el control de velocidad de un motor de corriente continua mediante un *encoder* óptico. Mediante esta implementación se pudo comprobar el correcto uso del PID, observando como el sistema llega de manera correcta a una referencia dada. Si bien en un principio se establecieron los valores del PID mediante el método de Ziegler–Nichols, luego fueron ajustados mediante prueba y error para obtener finalmente la respuesta deseada. Esto se logó de manera sencilla observando el movimiento del motor y la velocidad mediante *Simulink*.

Se pudo comprobar también que cuando el sistema es sometido a perturbaciones externas, el mismo se estabiliza en poco tiempo a la referencia correspondiente.

Se logró establecer una correcta comunicación a través del puerto serie. En un principio hubieron dificultades correspondientes al tipo de los datos tanto enviados como recibidos, en consecuencia, se creó un nuevo tipo *FLOATUNION_t* para poder enviar correctamente las tramas.

A. Código

A.1. Arduino

A.1.1. Código de Arduino para controlar la velocidad del motor y enviar tramas.

```
#include "Timer.h"

#define PIN_PWM 5    // Pin de salida del PWM
#define PIN_INTERRUPT 2 //Tiene que ser el pin 2 o 3 porque es una interrupcion
#define CANT_RANURAS 32 //La cantidad de RANURAS del encoder
#define TW 0.2

Timer tiempo;

typedef union
{
  float number;
  byte bytes[4];
} FLOATUNION_t;

int cont=0;
float vel = 1, vel_anterior = 1;
float Kp=1, Kd=0, Ki=0;
int N=10;
float Pk=0, Dk=0, Ik=0, uk=0;
int pwm_=0;
volatile bool interrupt_on = false;
int ref=0; //Referencia del controlador


void setup()
{
  pinMode(PIN_INTERRUPT, INPUT_PULLUP);
  pinMode(PIN_PWM, OUTPUT);
  Serial.begin(9600);

  // Indico que cada vez que se detecte un flanco ascendente por el pin
   PIN_INTERRUPT,
  // entre a la función contador_pulsos(), la cuál suma un 1 a la variable cont:
  attachInterrupt(digitalPinToInterrupt(PIN_INTERRUPT), contador_pulsos, RISING)
   ;

  // Indico que cada TW*1000=200ms se realice una interrupcion que llame a la
  // función calcular_velocidad():
  tiempo.every(TW*1000,calcular_velocidad);
}


void loop()
{
  tiempo.update(); // Se llama a update para corroborrar si el intervalo de
   tiempo especificado termino
                   // y en caso afirmativo se llama a la función
   calcular_velocidad()
```

```
46    if (Serial.available()>0)
      {
48      recibir_trama();
      }
50
      if (interrupt_on==true)
52    {
        interrupt_on = false;
54      pid();
        enviar_trama();
56    }

58    analogWrite(PIN_PWM, pwm_); // Envio el valor del pwm al pin correspondiente
  }
60


62  void contador_pulsos()
  {
64    cont++;
  }
66


68  void calcular_velocidad()
  {
70    interrupt_on = true;
      vel = ((float)cont*60)/(CANT_RANURAS*TW);
72    cont = 0;
  }
74


76  void recibir_trama() // PC al Arduino
  {
78    char header[4]="abcd";
      char header_recibido[4];
80    FLOATUNION_t Kp_aux, Kd_aux, Ki_aux, ref_aux, N_aux;

82    Serial.readBytes(header_recibido,4); // Leo el header para saber si comenzo
       una nueva trama
      // Comparo el header con los datos recibidos:
84    if(header[0]==header_recibido[0]&&header[1]==header_recibido[1]&&header[2]==
       header_recibido[2]&&header[3]==header_recibido[3])
      {
86      // Leo cada variable recibida
        Serial.readBytes(Kp_aux.bytes,4);
88      Serial.readBytes(Kd_aux.bytes,4);
        Serial.readBytes(Ki_aux.bytes,4);
90      Serial.readBytes(ref_aux.bytes,4);
        Serial.readBytes(N_aux.bytes,4);
92    }

94    // Reasigno los valores de las variables
      Kp=Kp_aux.number;
96    Kd=Kd_aux.number;
      Ki=Ki_aux.number;
98    ref=int(ref_aux.number);
      if(ref>2400) // Ajusto a este valor la referencia, ya que el motor no alcanza
```

```
    valores mayores
100     ref=2400;
    else if(ref<0)
102     ref==0;

104   N=int(N_aux.bytes);
  }
106

108 void enviar_trama() // Arduino a la PC
  {
110   // Convierto las variables pwm_ y ref a tipo float, para realizar
    // operaciones con otras variables de este tipo.
112   float pwm_float=float(pwm_)*100/255;
    float ref_float=float(ref);
114
    FLOATUNION_t Kp_aux, Kd_aux, Ki_aux, ref_aux, pwm_aux, vel_aux;
116   char header[4]="efgh";

118   Kp_aux.number=Kp;
    Kd_aux.number=Kd;
120   Ki_aux.number=Ki;
    ref_aux.number=ref_float;
122   pwm_aux.number=pwm_float;
    vel_aux.number=vel;
124
    Serial.write(header,4);
126   Serial.write(Kp_aux.bytes, 4);
    Serial.write(Kd_aux.bytes, 4);
128   Serial.write(Ki_aux.bytes, 4);
    Serial.write(ref_aux.bytes, 4);
130   Serial.write(pwm_aux.bytes, 4);
    Serial.write(vel_aux.bytes, 4);
132   Serial.flush(); // Espera a que la transmisión serial de datos este se
      compelte.
  }
134

136 void pid()
  {
138   float gamma = Kd/N;
    Pk = Kp*(ref - vel);
140   Dk = (gamma/(gamma + TW))*Dk - (Kp*Kd/(gamma + TW))*(vel-vel_anterior);
    uk = Pk + Ik + Dk;
142   if (uk>1)
      uk = 1;
144   else if (uk<0)
      uk = 0;
146
    Ik = Kp*Ki*TW*(ref - vel) + Ik;
148   vel_anterior = vel;
    pwm_ = 255*uk;
150 }
```

A.1.2. Código de Arduino para medir la velocidad máxima del motor

```
#include <TimerOne.h> // Libreria para usar el objeto Timer1

#define PIN_PWM 5
#define PIN_INTERRUPT 2

// Defino variables globales:

int cont; // Contador de pulsos
float Tw=0.01; // Tiempo de ventana
float vel=0;
int ppv=32; // Pulsos por vuelta
bool interrupt_on=false;

typedef union
{
  float number;
  byte bytes[4];
} FLOATUNION_t;


void setup()
{
  Serial.begin(9600);
  pinMode(PIN_PWM,OUTPUT); // Seteo el pin 5 como salida PWM
  pinMode(PIN_INTERRUPT,INPUT); // Seteo el pin 2 para interrupciones
  attachInterrupt(digitalPinToInterrupt(PIN_INTERRUPT),contador_pulsos,RISING);
   // Configuro la interrupción del pin 2 para que cuente los pulsos
  Timer1.initialize(Tw*1e6); // Inicializo timer. Esta función toma us como
   argumento
  Timer1.attachInterrupt(calcular_velocidad); // Activo la interrupción y la
   asocio a calcular_velocidad
}

void loop()
{
  if(interrupt_on==true)
  {
    enviar_trama();
  }
  analogWrite(PIN_PWM,255); // Ajusta el valor del PWM
//  Serial.println(vel);
//  delay(Tw);
}

void contador_pulsos() // Función que cuenta los pulsos cuando salta la
   interrupción
{
  cont=cont+1;
}

void calcular_velocidad()
{
  vel=(60*cont)/(Tw*ppv); // Hago la conversión a RPM
```

```
50    cont=0; // Inicializo el contador
      interrupt_on=true;
52  }

54  void enviar_trama()
    {
56    FLOATUNION_t vel_aux;
      vel_aux.number=vel;
58
      Serial.write(vel_aux.bytes, 4);
60    Serial.flush(); // Espera a que la transmision serial de datos se complete.
    }
```

A.2. Matlab

```
Tw=0.01;
ref=2000;

 for i=1:length(in.signals.values);
     salida(i)=output.signals.values(:,:,i);
 end

order=[1,1,2];
data=iddata('salida',in.signals.values,Tw)
sys=arx(data,order)

figure()
opt=stepDataOptions('StepAmplitude',ref);
hold on
plot(output.time,salida,'r')
step (sys,opt) %Respuesta al escalon para comparar con el Simulink
legend('respuesta real','respuesta del sistema equivalente', 'location', 'southeast')
grid on
tf(sys) %Para ver la transferencia

[num,den]=tfdata(sys);
num=num{1};
den=den{1};
```

identificacion_motor.m