Modelado de motor DC e implementación de un controlador tipo P

Entrega 2

Jaime Pérez Sánchez
Sergio Pérez Morillo
ETSIT – UPM – Sistemas electrónicos de control
Curso 2017-2018

Indice

Introducción

El objetivo de este trabajo es realizar el modelado experimental de un motor DC, para posteriormente diseñar un controlador tipo P que permita manejar la posición angular del mismo. Para ello utilizaremos un Arduino Due, diversas librerías (Serial, attachInterrupt y Due Timer) y software diseñado por el grupo de trabajo.

Seguiremos la siguiente estructura cronológica:

1. Desarrollar un software capaz de leer las señales proporcionadas por los encoders (entradas) y que utilice el módulo PWM del Arduino para controlar el motor (salida).
2. Realizar el modelado experimental del motor diseñando un banco de pruebas, que nos permita obtener la función de transferencia simplificada del mismo.
3. Finalmente, proceder al diseño e implementación del controlador tipo P de la posición angular. El cual estará sujeto a especificaciones de régimen permanente y transitorio
4. Conclusiones del trabajo realizado.

Los materiales utilizados son:

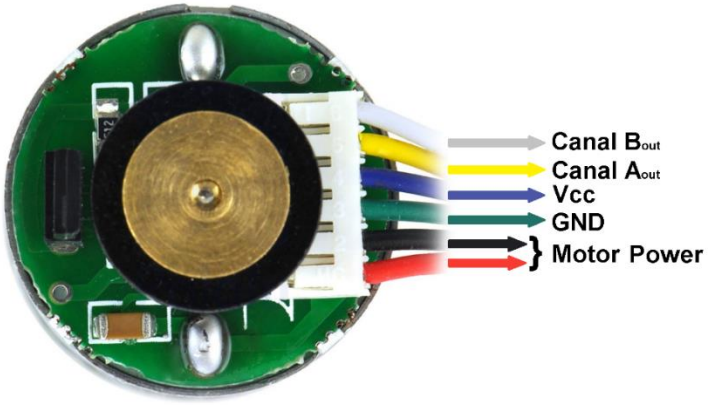
- Arduino Due 32-bit ARM (Atmel SAM3X8E ARM Cortex-M3 CPU)
- Etapa de potencia X-NUCLEO-IHM04A1
- Motor DC 12V
- Reductora 74,83:1
- Encoder magnético 48 CPR

1. Diseño e implementación de la arquitectura software en el hardware del RealLabo

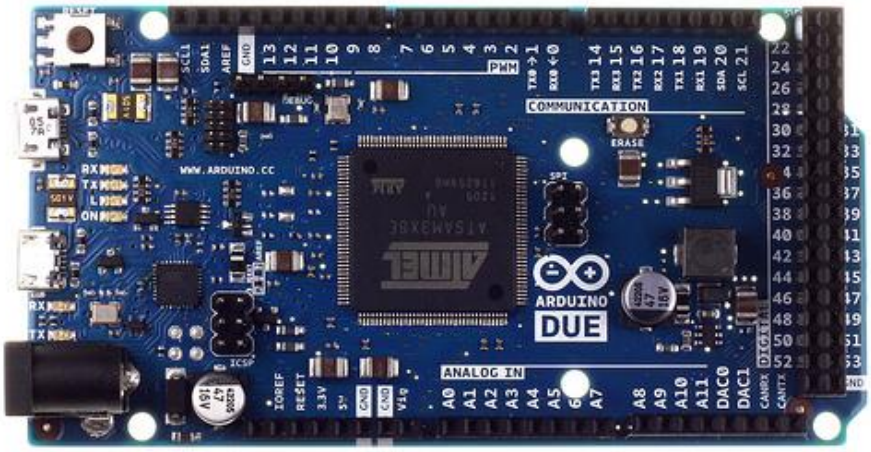
En este apartado implementaremos el software necesario para que, mediante el *Arduino Due*, seamos capaces de leer las señales de entrada que nos proporciona el encoder y generar las señales PWM que controlen el movimiento del motor.

Para saber qué funciones específicas tienen los pines y cables de nuestro material, debemos consultar la documentación proporcionada por los fabricantes:

- Motor y encoder **[1]**
  Hay 6 conexiones: 2 de lectura del encoder (Canales A y B), 2 de alimentación del motor, 1 de alimentación del encoder (5V) y otro a tierra (0V).

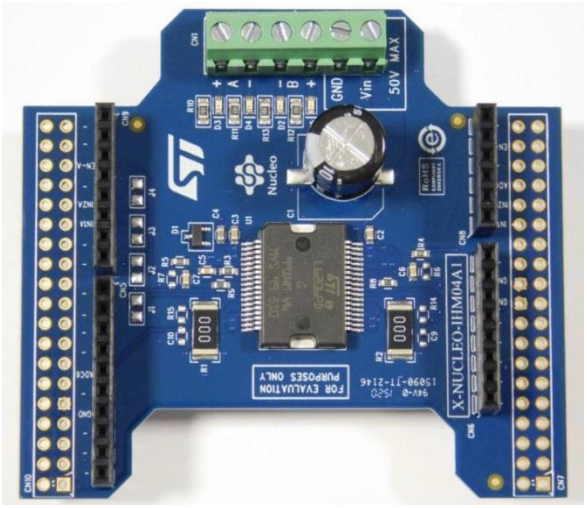


- *Arduino Due* **[2]**
  Utilizaremos los pines 3 y 7 para la lectura de datos del encoder, y los pines 39 y 41 para generar las señales PWM. Esto se explica más delante de forma detallada.

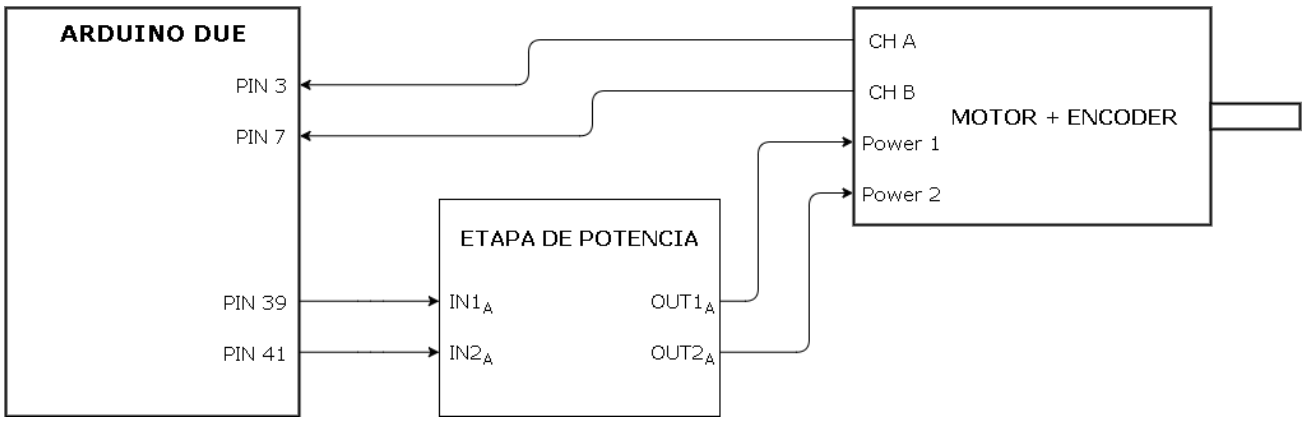


Due Pin Number	SAM3X Pin Name	Mapped Pin Name	Max Output Current (mA)	Max Current Sink (mA)
3	PC28	Digital Pin 3	15	9
7	PC23	Digital Pin 7	15	9
39	PC7	Digital Pin 39	15	9
41	PC9	Digital Pin 41	15	9

- Etapa de potencia **[3] [4]**
  Utilizaremos las entradas ($IN1_A$ e $IN2_A$) y salidas ($OUT1_A$ y $OUT2_A$) del puente A.

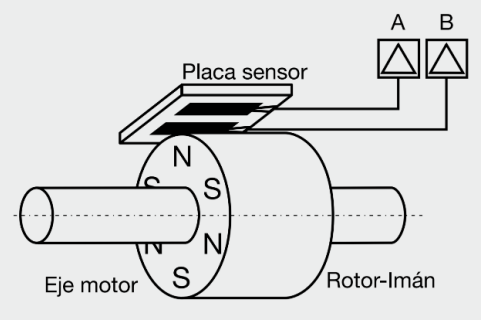


Finalmente, el esquema de conexiones del sistema es el siguiente:

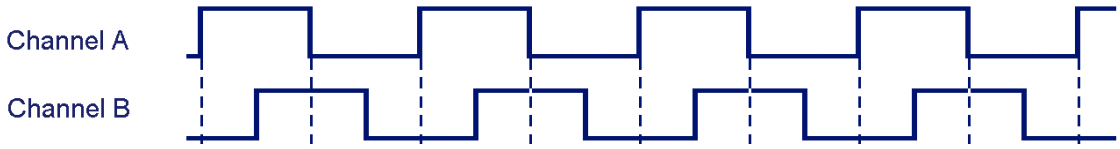


→ *Lectura de las señales del encoder*

El encoder es un sensor que se acopla al motor y proporciona información sobre el giro. Se basa en un disco magnético y 6 canales (2 de lectura: Canal A y Canal B).



Los canales de lectura proporcionan señales digitales cuadradas desfasadas $\frac{\pi}{2}$ entre sí. Como cada canal utiliza 12 pulsos por revolución, al contar los flancos de subida y bajada, obtenemos una resolución de 48 pasos por revolución del eje.



El software de manejo del encoder se basa una variable de giro que acumula el número de pasos, e interrupciones hardware de ambos flancos, es decir, en cualquier cambio. Estas interrupciones se activan en los pines 3 y 7 del Arduino Due mediante la función attachInterrupt() **[5]**.

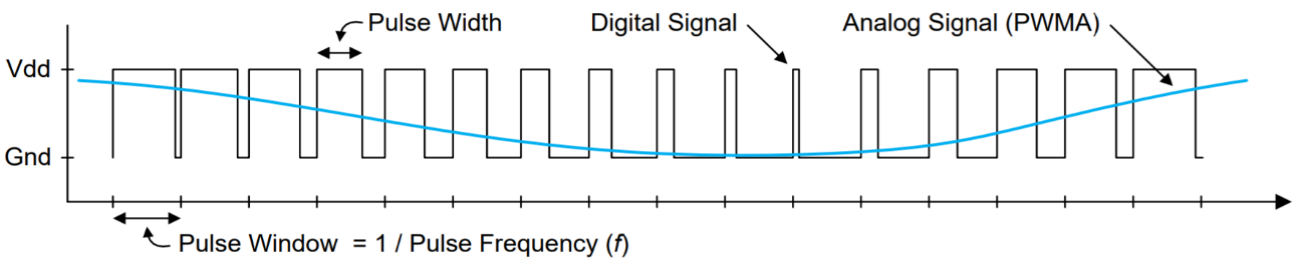
La función que atiende a estas interrupciones, llamada encoder() **[5]**, lee el valor de ambos canales y lo compara con el que tenía anteriormente. Y dependiendo del cambio realizado se suma o resta 1 al valor de giro.

Cabe destacar que al estar también acoplada al motor una reductora (74,83:1), un giro completo del eje exterior corresponde con 3591,84 pasos del encoder.

→ Manejo de las señales PWM para el control del motor

Para poder mover el motor hemos utilizado el puente A de la etapa de potencia. Las salidas OUT1A (A+) y OUT2B (A-) son las encargadas de enviar las señales necesarias al motor para que gire en un sentido concreto. Para ello debemos introducir una señal modulada por ancho de pulso (PWM) en las entradas del puente (IN1A e IN2A).

Inicialmente utilizamos la función analogWrite() **[5]** que proporciona un PWM, sin embargo esta señal es de 1KHz por defecto.



Para conseguir una frecuencia de 20KHz decidimos utilizar el módulo hardware PWM del SAM3X de Atmel [6] y consultamos en Internet las funciones necesarias para utilizarlo [7].

Instance	Signal	I/O Line	Peripheral
PWM	PWMH2	PC7	B
PWM	PWMH3	PC9	B

Finalmente utilizamos los pines 39 (PWMH2 - PC7) y 41 (PWMH3 - PC9), e implementamos la función setPWMvolt() [5] que proporciona un ciclo de trabajo determinado a la señal PWM, dependiendo del voltaje que se desea introducir al motor.

Para que el sistema funcione debemos conectar mediante cables Arduino, los pines 39 y 41, con los pines de entrada al puente A.

2. Modelado experimental de un motor DC con la arquitectura hardware y software implementada

Para realizar el modelado experimental del motor nos hemos basado en el algoritmo propuesto en la asignatura [8], en este se necesita fijar una serie de parámetros:

-Escalón de entrada de duración 1,2 segundos. En el que la mitad del tiempo (0,6 s) es a una tensión elegida por nosotros y la otra mitad es 0 voltios.

-Un período (T) de muestreo de 1 milisegundo.

-Doce experimentos (Q) cada uno con una tensión distinta (de 1 a 12 voltios) en los que medimos el número de pasos del encoder del motor cada milisegundo.

-Diez pruebas (P) para cada experimento con las que luego se realiza una media aritmética de los valores obtenidos para obtener medidas más precisas.

Para la implementación software utilizamos las interrupciones periódicas (cada 1 ms) de la librería *DueTimer.*

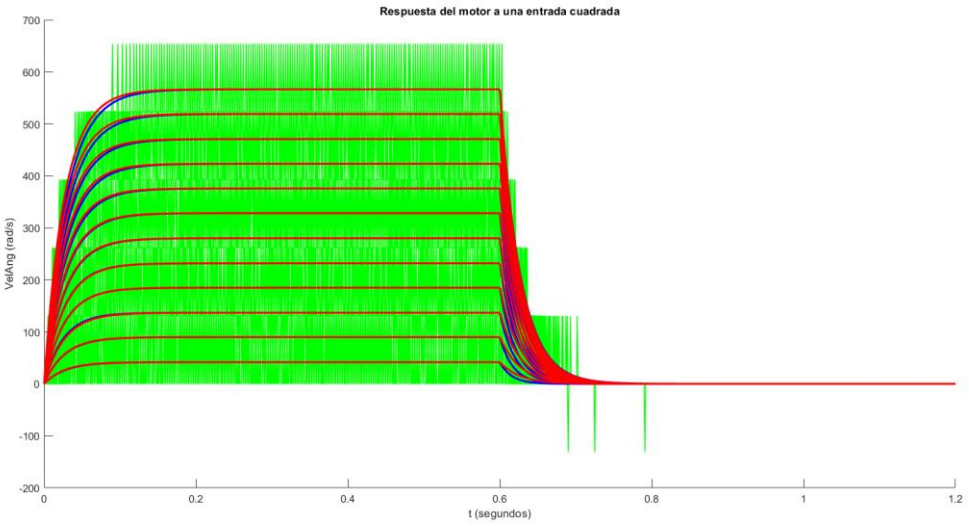
Lo primero será sacar los datos experimentales del motor, con lo que hacemos uso de Arduino. Para realizarlo con la mayor *brevedad* posible, haremos las diez pruebas de cada experimento seguidas, almacenando el valor de todas las muestras en un array (función *pulso_motor()* [5]). Una vez realizadas todas las pruebas hacemos la media, imprimimos los datos por el puerto serie (función *printDatos()* [5]) y pasamos al siguiente experimento.

Después de realizar todos los experimentos, los introducimos en el código de Matlab [9] (facilitado por los profesores) que calcula el polo s y la ganancia K que buscamos para caracterizar el motor mediante el algoritmo mencionado anteriormente.

El uso de Matlab es muy sencillo, el único problema es saber calcular las muestras que tiene cada archivo mediante los datos iniciales, en este caso tendremos 1201 muestras por archivo o experimento. Con los datos introducidos en los ficheros correspondiente, solo hace falta ejecutar la siguiente línea de código:

>> *ModeladoMotorDCVelAngPoloBucleB*

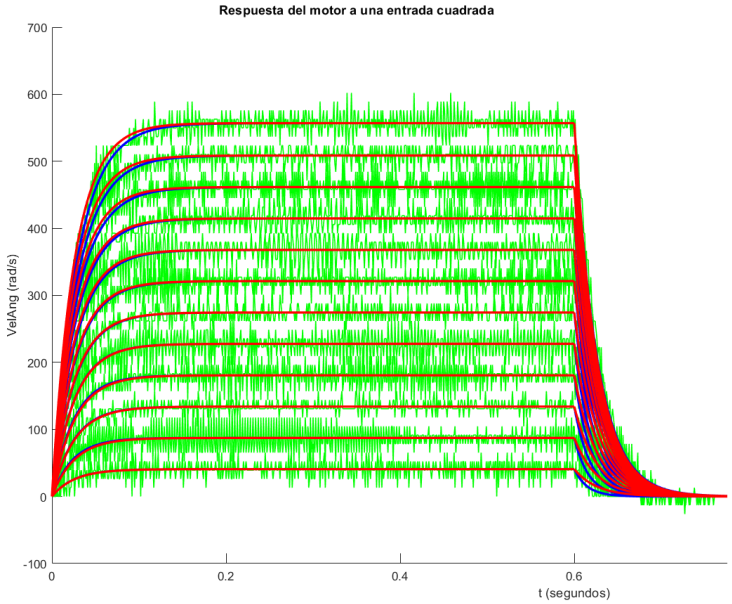
Con esto obtenemos los resultados, en un primer intento: tenemos esto en la gráfica de la respuesta al motor a una tensión cuadrada:



Respuesta al motor a una tensión cuadrada (primer intento)

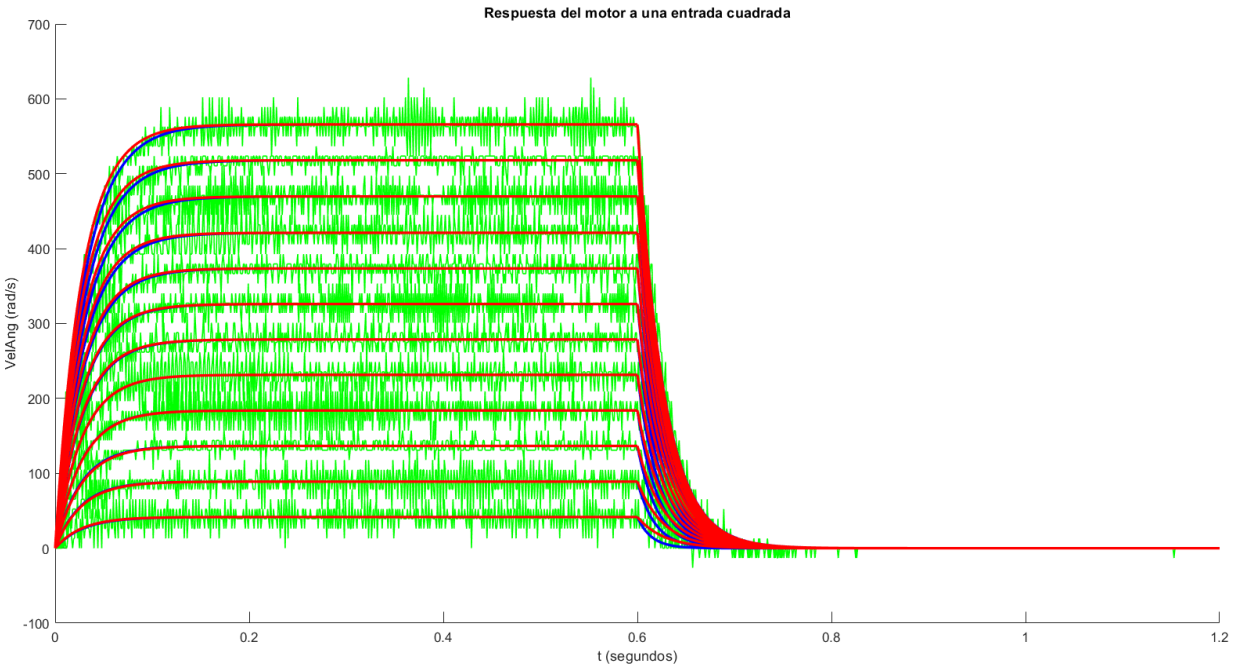
Si lo comparamos con el ejemplo de la documentación **[10]**, vemos que hay presente mucho error (representado en verde), con lo que revisamos los datos y el código en Arduino.

Primero, al realizar las pruebas en el motor, notamos que en la primera prueba de cada experimento la velocidad del motor no era constante con lo que decidimos añadir una prueba más (11 en total) y omitir esta primera al hacer la media. Segundo, los datos los recogíamos con variables tipo *int* que no utilizan decimales con lo que habría saltos en las medidas muy grandes que en la gráfica se observa como errores escalonados (todos los errores parecen ir por escalones en la gráfica), decidimos cambiarlos por *double*. Una vez hecho esto obtuvimos lo siguiente:



Respuesta al motor a una tensión cuadrada (segundo intento)

Observamos que el error se ha reducido, pero no al nivel del ejemplo con lo que seguimos buscando maneras de reducirlo. Observamos que el puerto serie no nos estaba dando mucho decimales, implementamos una manera de que nos proporcionase los datos con 6 decimales, el resultado fue este:



Respuesta al motor a una tensión cuadrada (tercer intento)

Aun existiendo todavía error, consideramos que en esto punto es culpa de la calidad del motor en sí, más que en la captura de datos realizada. Una manera de que hubiese menor error, aunque sería de forma relativa, es cambiando el período de muestreo a uno mayor.

El modelado final del motor presentado mediante su función de transferencia $G_{\dot{\theta}}(s)$ en velocidad es:

$$G_{\dot{\theta}}(s) = \frac{K_m}{s + p_m} = \frac{1719{,}9114}{s + 36{,}72}$$

En posición que tiene un polo en el origen será:

$$G_{\theta}(s) = \frac{K_m}{s(s + p_{m)}} = \frac{1719{,}9114}{s(s + 36{,}72)}$$

Tenemos finalmente un polo de valor **s = 36,72** y una ganancia **K_m = 1719,9114**. Otras gráficas de interés sobre el modelo experimental se muestran en el anexo.

3. Análisis, diseño e implementación de un controlador P para un control de posición angular.

El diseño del controlador se hará sujeto a unas especificaciones de régimen permanente (seguimiento de la señal) y de régimen transitorio (valores de sobreelongación M_p y tiempo de establecimiento t_s) que consideremos nosotros correctas.

Primero vamos a analizar el sistema antes de diseñar e implementar el controlador, el diagrama es el siguiente:

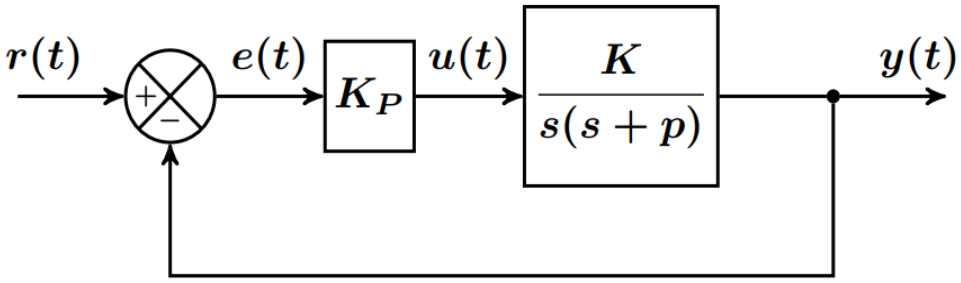


Diagrama del controlador P simplificado

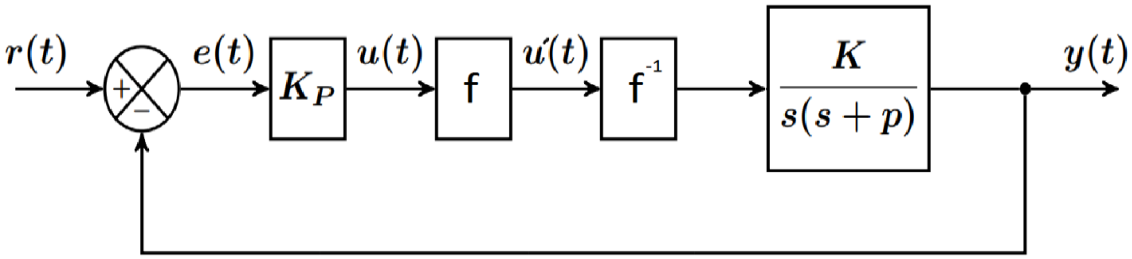


Figura: Diagrama del controlador P teniendo en cuenta no linealidades.

Estudiaremos tanto para el régimen permanente como el transitorio y usaremos el modelo simplificado (omitiendo la parte de las no linealidades f^{-1} y f).

→ *Régimen permanente*

Para abordar el problema de seguimiento en régimen permanente se parte de la función de transferencia en posición del motor, la cual se resta a uno dando a función de transferencia del error $H_e(s)$:

$$H_e(s) = 1 - H_s(s) = \frac{s(s+p)}{s^2 + sp + KKp}$$

Siendo la señal del error:

$$E(s) = R(s) * H_e(s)$$

Donde R(s) es la señal referencia.

Ahora aplicamos el teorema del valor final con E(s) (ya que es un sistema estable), si tiende a cero el controlador será capaz de seguir la señal, como señal de referencia usaremos el escalón, con lo que tendremos:

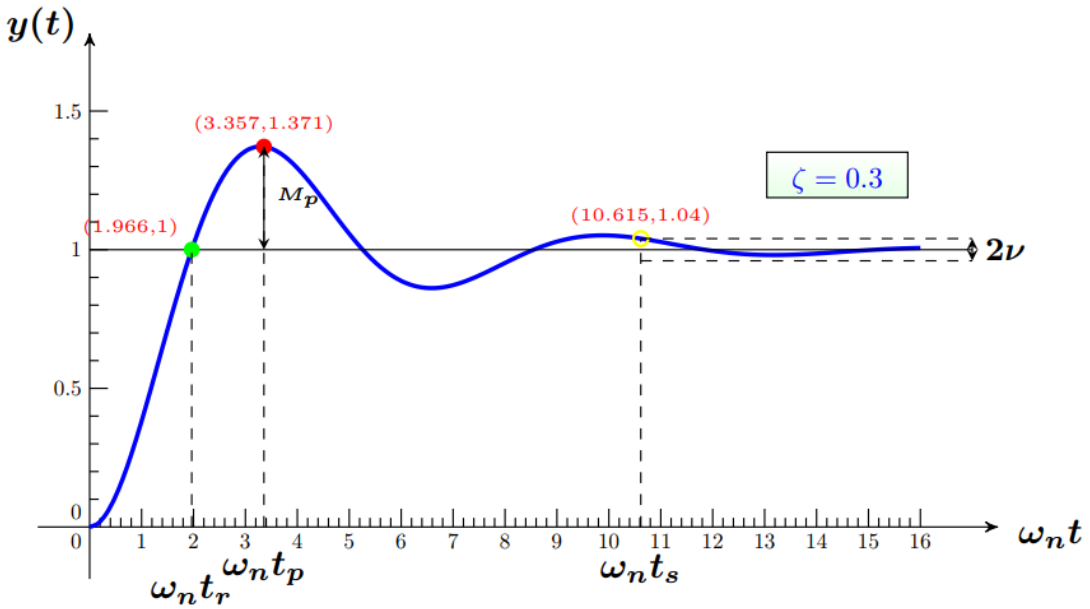
$$R(s) = \frac{1}{s}$$

$$\lim_{t\to\infty} e(t) = \lim_{s\to 0} s * E(s) = s * \frac{s(s+p)}{s^2 + sp + KKp} * \frac{1}{s} = 0$$

Como el límite tiende a cero, la señal de referencia de escalón asegura que no haya error en la región permanente (en tiempo tendiendo a infinito).

→ *Régimen transitorio*

Nuestro diseño está sujeto a dos especificaciones: la primera el tiempo de establecimiento, t_s, tiempo en el que la señal pasa de régimen transitorio a régimen permanente (depende también de una tolerancia *ν*), y segundo la sobreelongación máxima, M_p, importante en problemas de saturación. En el siguiente ejemplo se pueden observar ambos factores:



Características del régimen transitorio.

Teniendo en cuenta esto, procedemos a calcular K_p, representada en la figura del principio del apartado, es el principal parámetro característico del controlador que nos ayudará a diseñarlo.

Para ello, partimos de la frecuencia natural y coeficiente de amortiguamiento (ξ) de este sistema:

En estos sistemas de segundo orden de lazo cerrado la función de transferencia se puede escribir tal que:

$$H_s(s) = \frac{w_n^2}{s^2 + 2\xi\, w_n s + w_n^2}$$

ξ el amortiguamiento y wn la frecuencia natural

Ahora lo igualamos a nuestra función de transferencia de lazo cerrado:

$$H_s(s) = \frac{KK_p}{s^2 + ps + KK_p}$$

Y tendremos que la frecuencia natural es:

$$w_n = \sqrt{K_p * K}$$

$$w_n = \frac{p}{2 * \xi}$$

Igualando y despejando K_p:

$$K_p = \frac{p^2}{4\xi^2 \mathrm{K}}$$

K y p hallados en el modelo experimental

Una vez que tenemos como calcular K_p, enunciamos M_p y t_s (igual que en **[11]** ya que se trata del mismo sistema) y calculamos los tres valores para una tolerancia *v*=2% y tres valores diferentes de amortiguamiento correspondientes a amortiguamiento crítico, subamortiguamiento y sobreamortiguamiento.

$$t_s \approx \frac{\ln\left(\frac{1}{\nu * \sqrt{1-\xi^2}}\right)}{\xi * w_n}$$

$$M_p = e^{-\left(\frac{\xi * \pi}{\sqrt{1-\xi^2}}\right)}$$

Amortiguamiento	M_p	t_s	K_p
0,3	0,3723	0,2156	162,9568
0,707	0,0432	0,2319	29,3411
1	-	-	14,6661

Valores de Kp multiplicados por el factor de la reductora igual a 75

Como era de esperar con sobreamortiguamiento máximo o ξ =1 no se pueden calcular los valores de M_p ni t_s. Estos serán los datos que probaremos en nuestro controlador.

→ *Implementación software*

En la implementación software del controlador utilizamos las interrupciones periódicas de la librería *DueTimer*. Para llegar a la posición del motor deseada, cada 1ms se ejecuta la función *controladorP()* **[5]**, la cual corrige el voltaje introducido al motor en función del error.

El error está definido como la diferencia entre el valor de referencia de posición angular (el que se desea alcanzar) y el valor actual de la posición angular del motor. La posición actual del motor se obtiene mediante los encoders y es transformada a radianes con la función *to_rad()* **[5]** para tener todos los parámetros en las unidades del Sistema Internacional.

En un principio intentamos introducir un factor para corregir las no linealidades del motor utilizando los coeficientes del vector de parámetros del polinomio invertido obtenidos en *Matlab*.

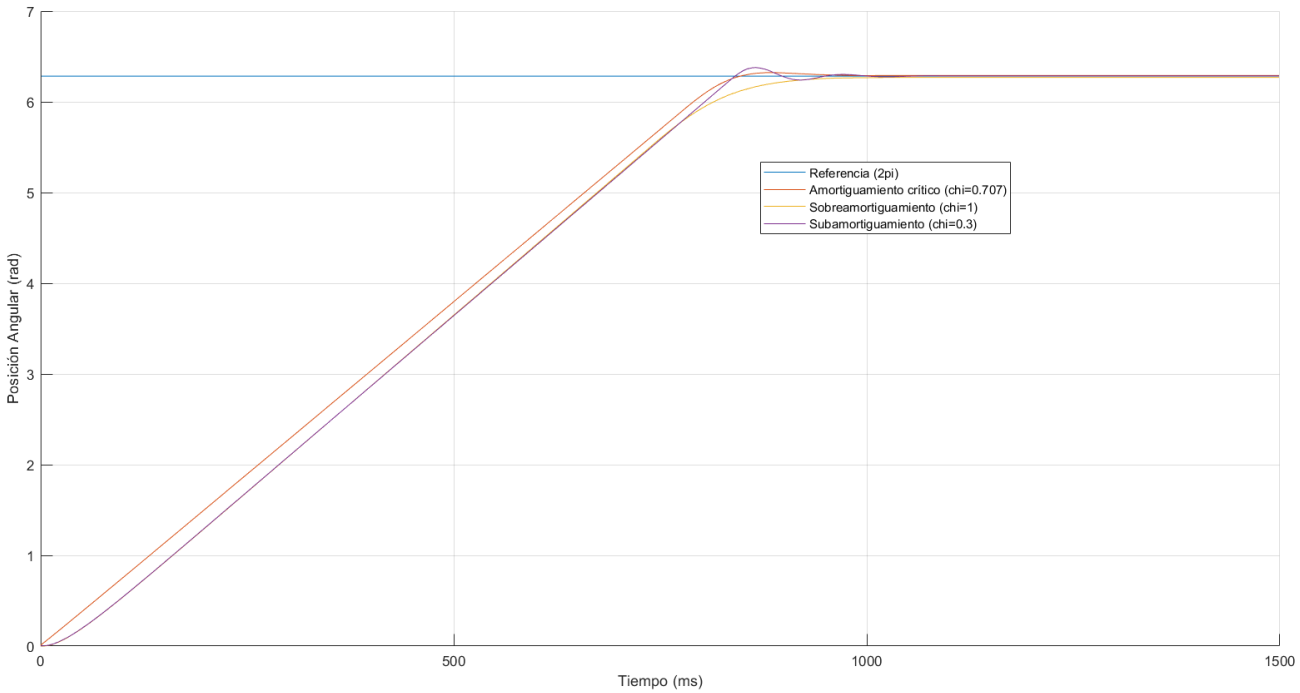
$$V_j = b_0 u + b_1 u^3 + b_2 u^5 + \cdots + b_{11} u^{21}$$

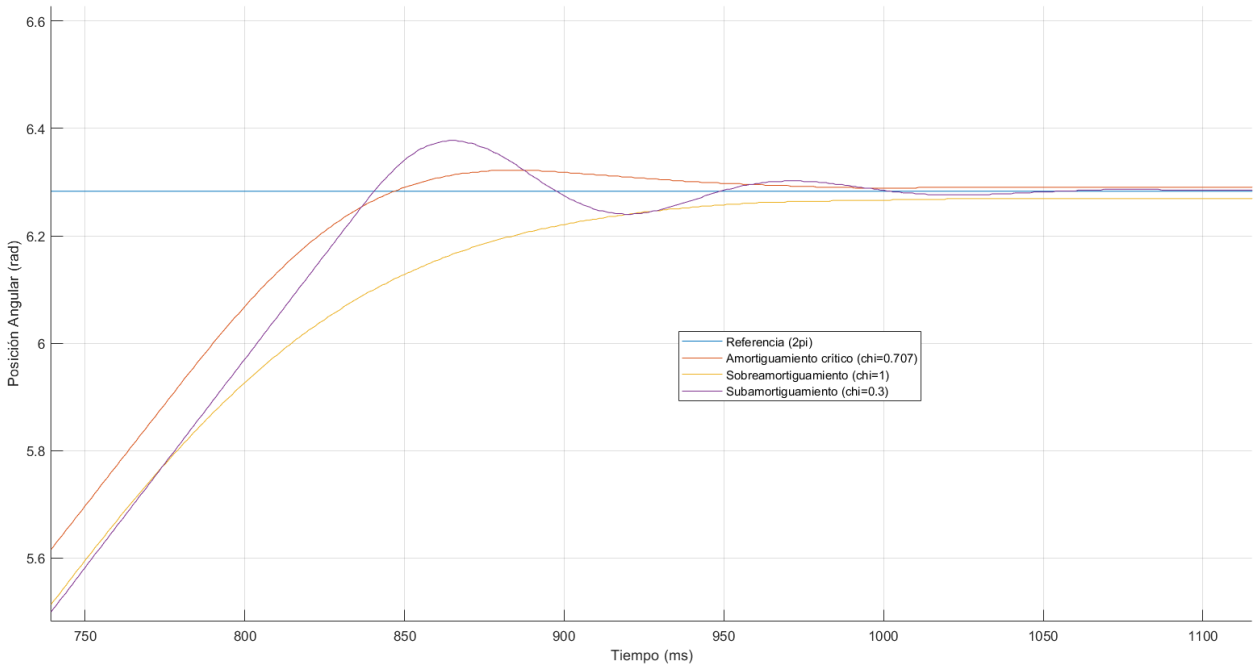
Sin embargo, el cálculo de este polinomio de grado 21 es una carga demasiado pesada para que el procesador lo realice cada 1ms. De modo que no lo funcionaba de manera correcta.

Para este problema teníamos dos posibles soluciones. Una, aumentar significativamente el periodo de la interrupción, perdiendo gran precisión. Y la segunda solución era ignorar las no linealidades del motor, e implementar el controlador únicamente con el parámetro de corrección K_p.

Finalmente decidimos por implementar la segunda solución. Ésta consiste en calcular el error, multiplicarlo por el valor calculado de K_p e introducirlo como tensión con la función *setPWMvolt()* **[5]** explicada anteriormente.

Realizamos pruebas para el valor de referencia 2π y obtuvimos los siguientes valores:





Respuesta del controlador P (posición angular para una referencia = 2 π)

Para no sobrecargar el documento principal, en el anexo se han incluido las gráficas obtenidas con referencias de π y – π.

Podemos observar que los valores experimentales son coherentes con los cálculos teóricos:

- Con un amortiguamiento crítico alcanzamos antes el valor de referencia (±2% de tolerancia) y obtenemos una ligera oscilación
- Con un sobreamortiguamiento eliminamos las oscilaciones, pero tardamos más tiempo en alcanzar el punto deseado e incluso obtenemos un error constante que nos impide alcanzar completamente la referencia exacta.
- Con un subamortiguamiento alcanzamos rápidamente la referencia, pero tenemos demasiadas oscilaciones, por lo que no se alcanza una estabilidad hasta pasado un tiempo.

4. Conclusiones

Como apuntes finales nos gustaría destacar de cada parte:

- ## Diseño e implementación de la arquitectura software:

La principal conclusión que hemos podido aprender a la hora de implementar el software es que la función que imprime por el puerto serie (Serial.print()) es muy lenta, por lo que es recomendable realizar esta tarea al final de la ejecución. Además de intentar que las interrupciones del sistema se atiendan en el menor tiempo posible.

- ## Modelado experimental del motor:

Aun habiendo obtenido una función de transferencia final después de haber probado varias maneras de mejorar la calidad de los datos capturados, nos hubiera gustado probar otros métodos para mejorar la calidad de las representaciones y valores finales obtenidos. Quizá cambiando el código de Matlab hubiéramos encontrado otro camino de mejora, pero nuestros conocimientos son limitados en este programa.

Sin embargo, nuestro tiempo era bastante limitado para probar todas estas cosas.

- ## Análisis, diseño e implementación del controlador:

Las medidas realizadas teóricamente y luego probadas en el controlador podrían haber sido de mayor número para tener una mejor caracterización del controlador, aun así, consideramos que eran las mínimas suficientes para entender el comportamiento del controlador y que queda comprobado con lo expuesto, con más tiempo nos hubiese gustado hacer más pruebas con esto.

Además, nos hubiese gustado obtener datos y estudiar el comportamiento del controlador cuando introducíamos el factor para las no linealidades, ya que únicamente lo pudimos observar experimentalmente por falta de tiempo.

En general, aunque hemos llevado el trabajo al día y no dejándolo para el final, si hubiésemos tenido más tiempo para probar más cosas, hacer más experimentos… nos habría quedado un trabajo más robusto. Con todo esto dicho, éramos conscientes de que estas limitaciones se iban a dar y estamos satisfechos en general con el trabajo realizado.

5. Bibliografía

[1] Motor 12V + Encoder 48 CPR + Reductora 75:1 (Pololu)

[2] Arduino Due

[3] Etapa de potencia X-NUCLEO-IHM04A1

[4] Puente en H

[5] Código entregado junto con la memoria

[6] Datasheet Atmel SAM3X

[7] Arduino Due PWM frequency Change

[8] Apuntes de modelado en Robalabo (Pág. 11 a 18)

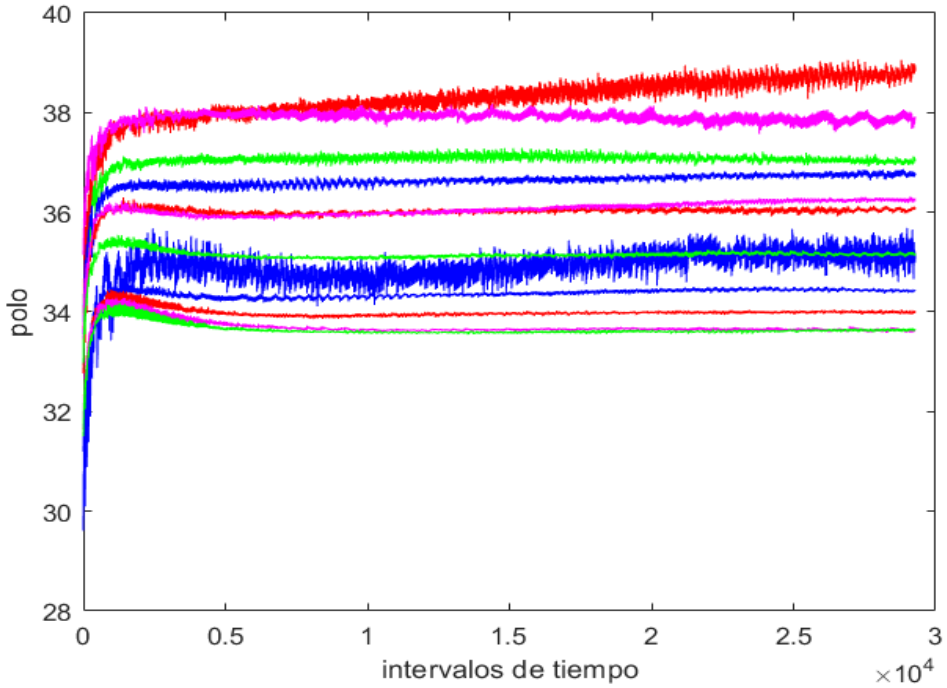
[9] Scripts de Matlab para cálculo y visualización del modelado experimental en Robolabo

[10] Apuntes de modelado en Robalabo (Pág. 22)

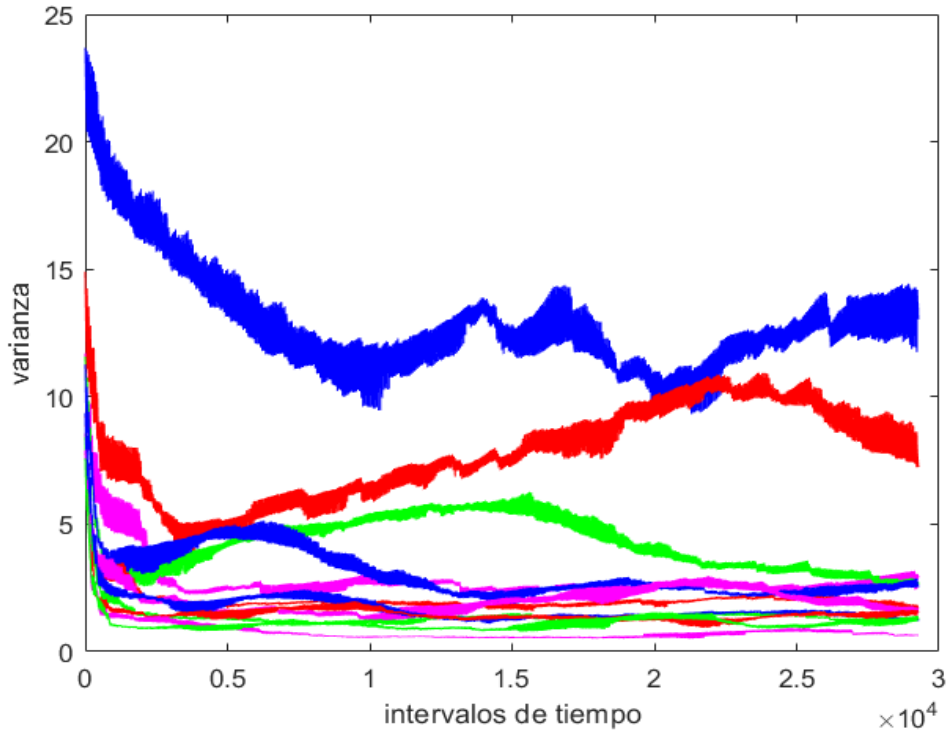
[11] Apuntes de análisis en Robolabo (Pág. 13)

6. Anexo

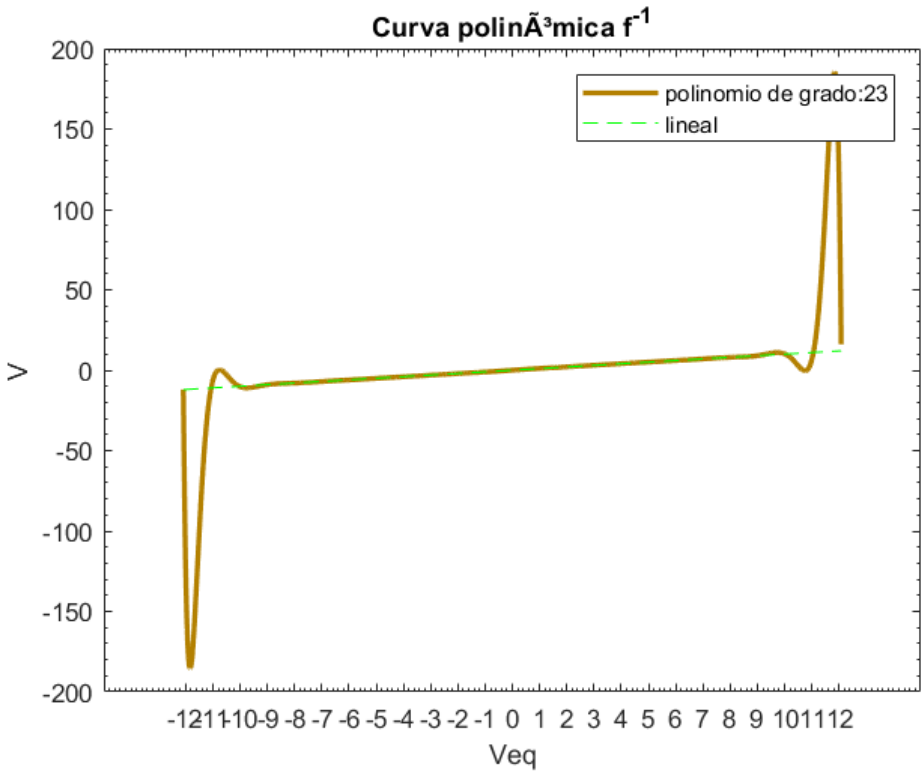
→ *Gráficas sobre el modelado experimental del motor DC*



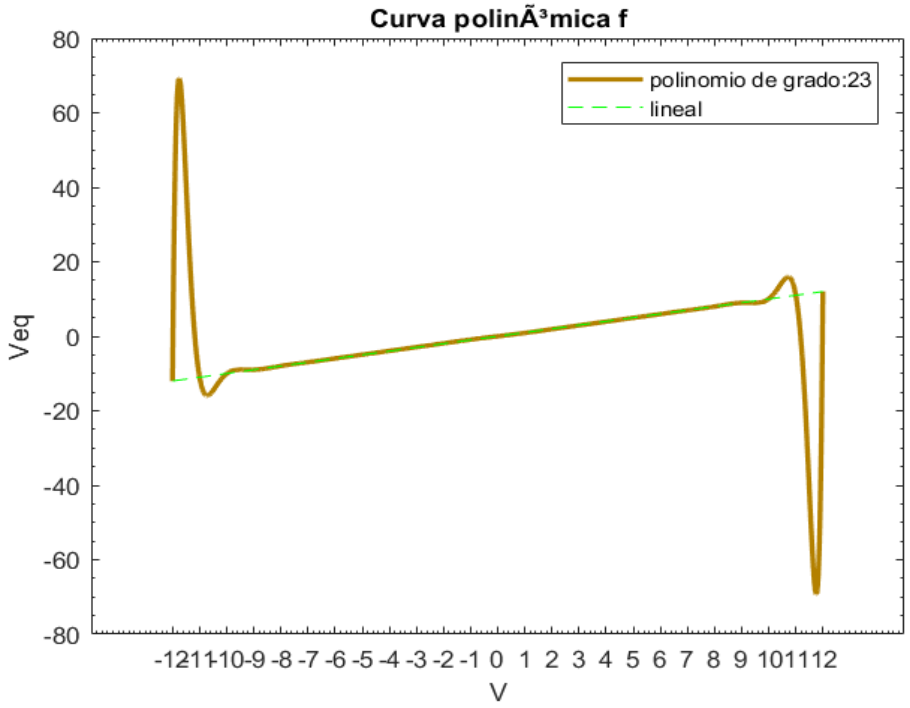
Polo medio pSj en cada intervalo de tiempo



Varianza en cada intervalo de tiempo

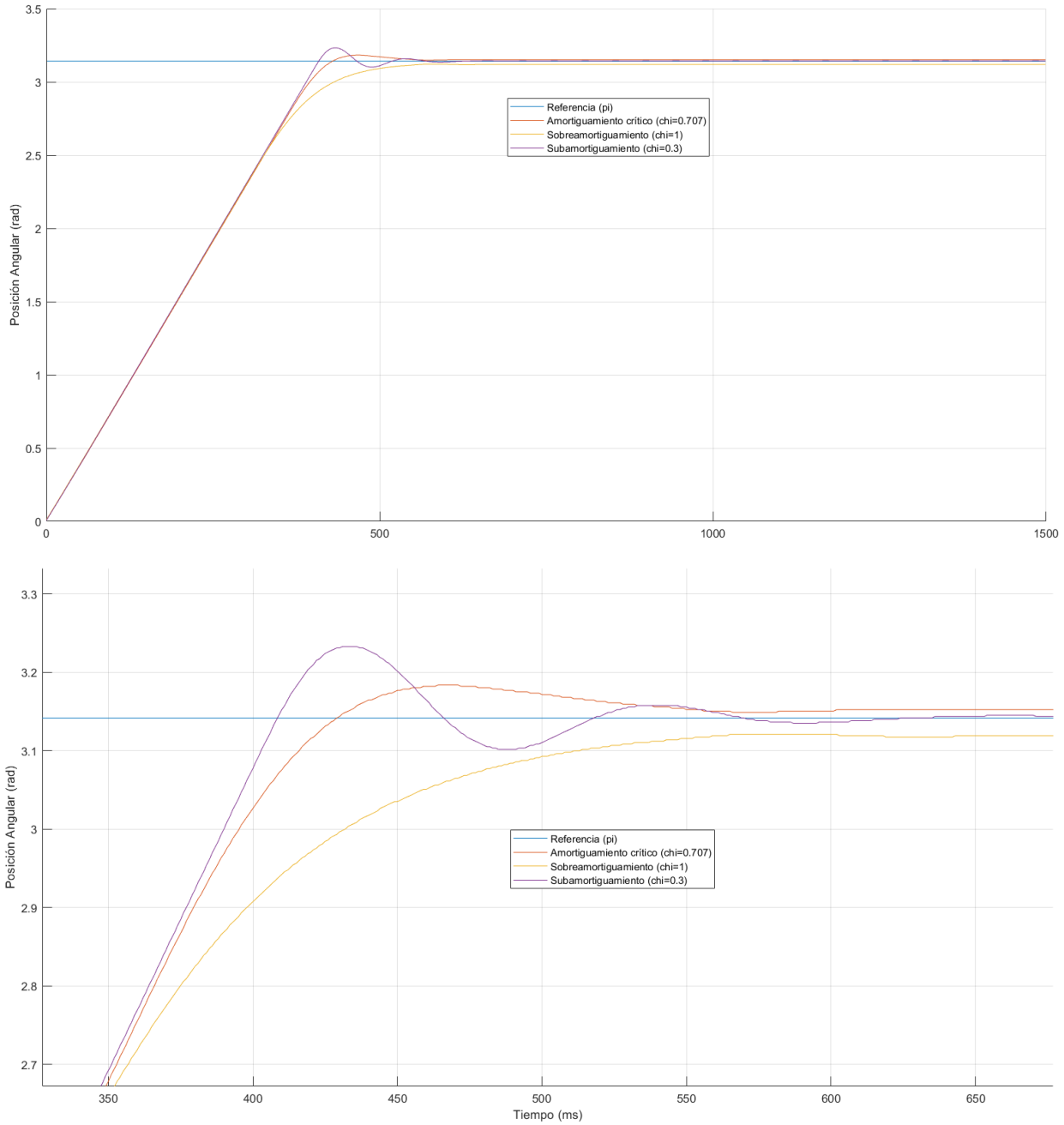


Curva polinómica f-1 (V-Veq)

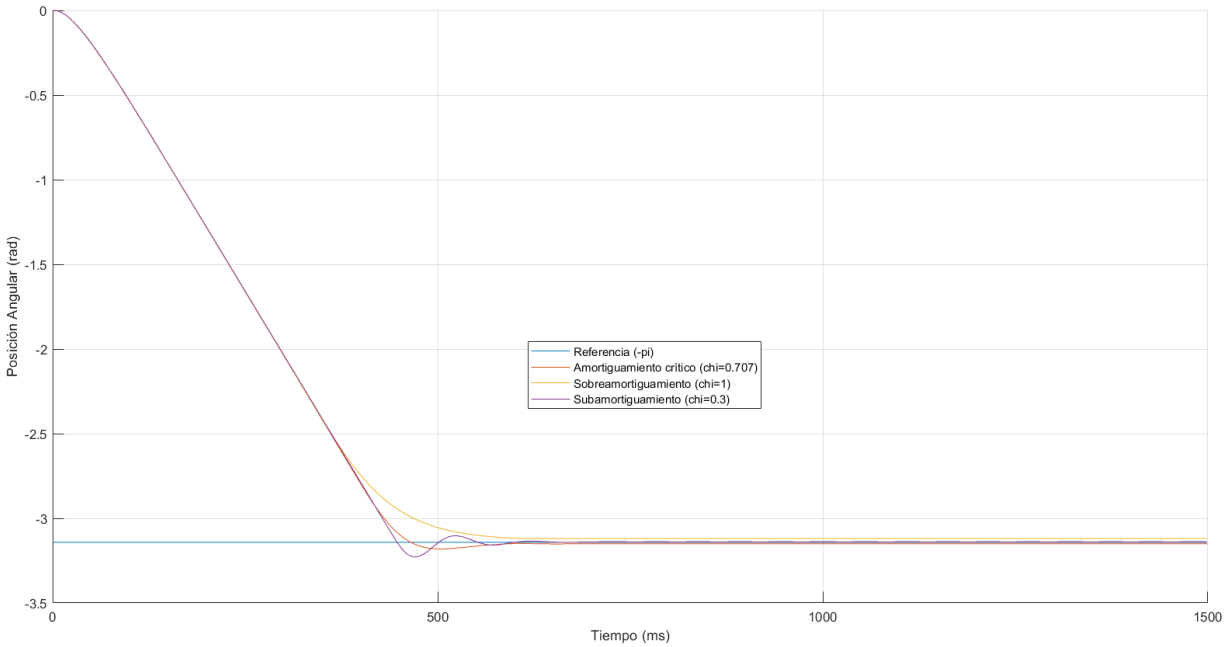


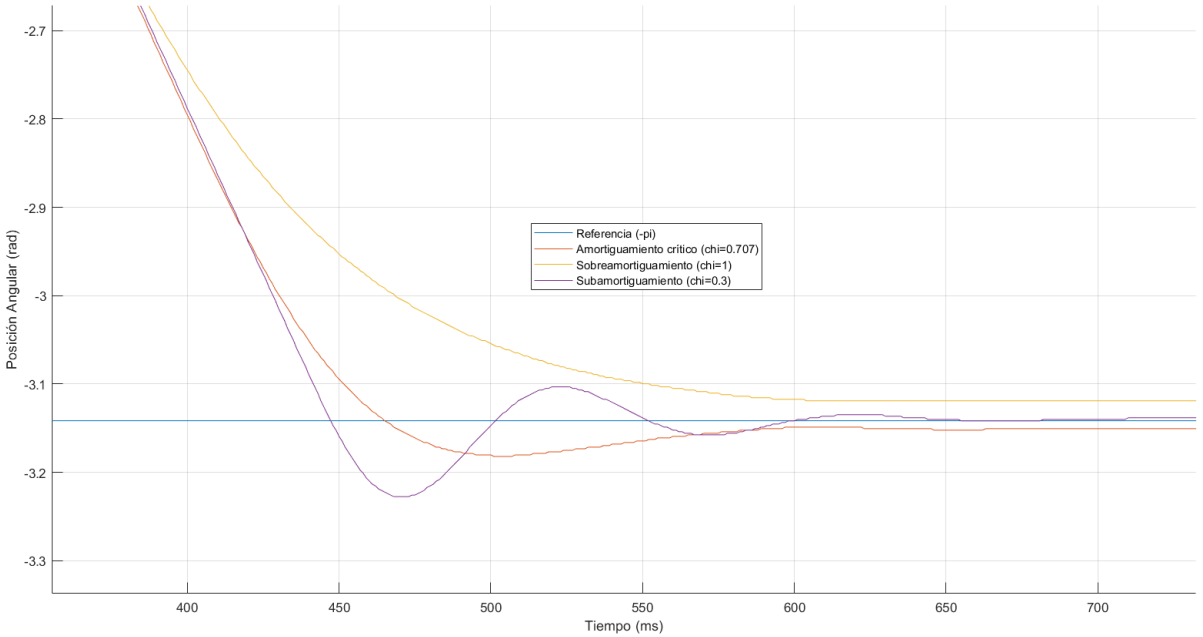
Curva polinómica f (Veq-V)

→ *Gráficas sobre la implementación del controlador tipo P*



Posición angular para una referencia = π





Posición angular para una referencia = - π