Notas de Aplicación

Realizado por Carlos Pillajo

IMPLEMENTACION DE UN CONTROLADOR PI CON ADSP-2199X

Contenido

Resumen

Resumen:

Controladores PI son universalmente conocidos por su flexibilidad combinada con la afinación relativamente fácil. Esta nota de aplicación describe la conversión continua en el dominio de tiempo discreto, que es esencial para cada aplicación en un procesador digital. Una rutina se presenta a continuación, que implementa la representación en tiempo discreto del regulador PI.

1. Características del controlador PI

1.1 El dominio de tiempo continuo

Los Controladores PI (Proporcional e Integral) son en la mayoría de los casos analizados y sintonizados en el dominio de tiempo continuo. La función de transferencia correspondiente se da como:

$$U(s) = \left(K_P + \frac{K_I}{s}\right).E(s) \qquad (1)$$

donde E y U denotar la señal de error de entrada y de salida del controlador, respectivamente, s es la variable de Laplace y KP y KI son los dos parámetros del regulador PI asociado con la parte proporcional (P) e integral (I). Sin embargo, la sintonización de un controlador PI no es muy intuitivo al especificar estos dos parámetros. La ecuación anterior puede ser reescrita como sigue:

$$U(s) = K_p.\left(1 + \frac{K_I}{K_P}\frac{1}{s}\right).E(s) = K_P.\omega_{PI}.\left(\frac{\frac{s}{\omega_{PI}} + 1}{s}\right).E(s) \qquad (2)$$

Donde $\omega_{PI} = \frac{K_I}{K_P}$ (expresado en [ras/seg]). Evidentemente, esta función de transferencia presenta un polo en el origen y un cero localizado en ω_{PI} La ganacia a altas frecuencias esta dada por sí Kp.. La siguiente figura muestra un diagrama de bode típico de un controlador PI, con KP y PIω establece en 0,25 (-12 dB) y 50 Hz (314rad / s), respectivamente.

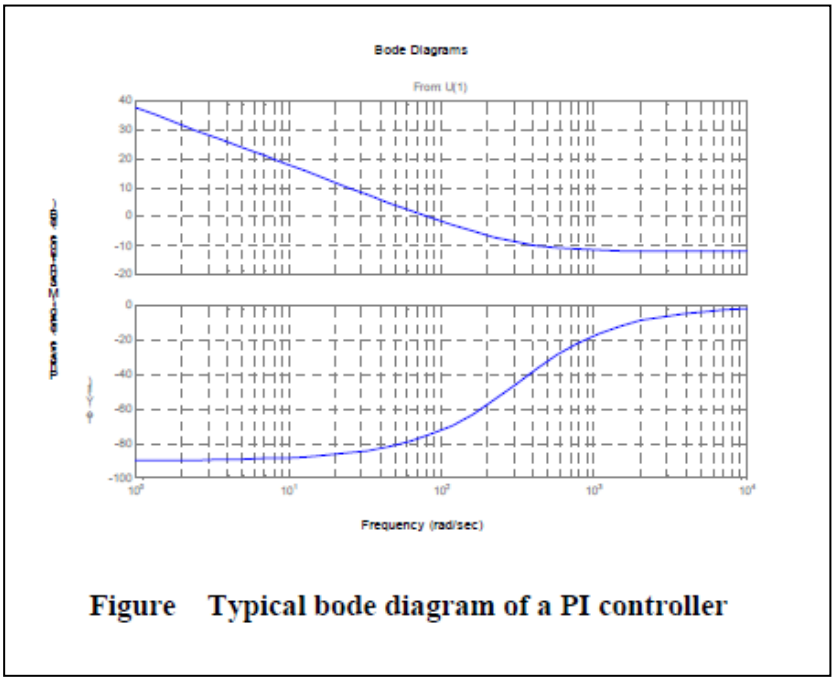


Figure Typical bode diagram of a PI controller

En lo que sigue, se supone que el controlador esté sintonizado en el dominio de tiempo continuo por KP y PIω.

1.2 El dominio de tiempo discreto

La transición de tiempo continuo para el dominio de tiempo discreto implica que la operación integral tiene que ser aproximado por una suma discreta. Hay varios métodos para la sustitución de la integral. Dos de ellos serán discutidos más adelante.

1.2.1 Retenedor de orden Zero (ZOH)

Con este enfoque, se muestrea la señal en el instante k y se mantiene constante hasta el siguiente instante de muestreo k + 1 instantánea. La siguiente figura ilustra esto.

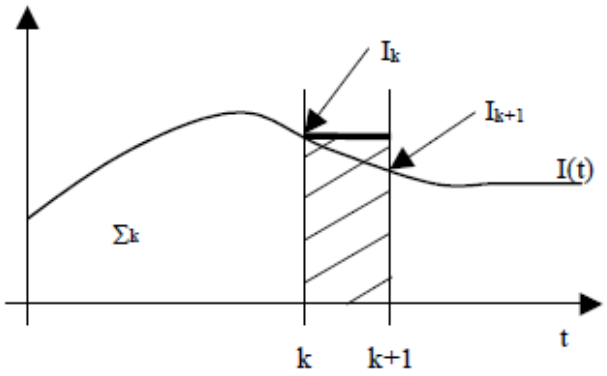


Figure Integral with ZOH approximation

La operación integral se aproxima por la acumulación de las áreas rectangulares. Denotando la suma en el instante k con Σk, la señal en el instante k con Ek y el tiempo de muestreo con TSample, la "integración" se logra a través de:

$$\sum k+1 = \sum k + E_k \,.T_{sample} \qquad (3)$$

Como es conocido, la misma ecuación puede expresarse en el dominio z por:

$$z.\sum(z) = \sum(z) + E(z).T_{sample} \qquad (4)$$

Lo que conduce a :

$$\sum(z) = \frac{T_{sample}}{z-1} \,.E(z) \qquad (5)$$

Por lo tanto el integrador continuo $\frac{1}{S}$ en la ecuación (1) se sustituye por el primero factor en la ecuación anterior. La función de transferencia en el dominio discreto se obtiene a partir de (1) y (5) como:

$$U(z) = K_P.\left(1 + \omega_{PI}.\frac{T_{sample}}{z-1}\right) E(z) = \frac{K_P z + K_P \,.(\omega_{PI}.T_{sample} - 1)}{z-1} E(z) \qquad (6)$$

Esto corresponde a la siguiente ecuación de diferencias:

$$U_{k+1} = K_P.E_{k+1} + K_P\left(\omega_{PI}.T_{sample} - 1\right).E_k + U_k \qquad (7)$$

1.2.2 Retenedor de primer orden (FOH)

Un enfoque ligeramente mejorado se muestra en la siguiente figura.

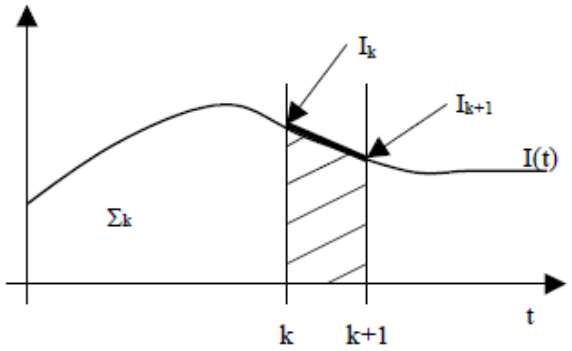


Figure Integral with FOH approximation

La operación integral se aproxima por la acumulación de las áreas trapezoidales. Denotando la suma en el instante k con Σk, la señal en el instante k con Ik y el tiempo de muestreo con TSample, la "integración" se logra a través de:

$$\sum k+1 = \sum k + \frac{I_k + I_{k+1}}{2}.T_{sample} \qquad (8)$$

donde el último término se deriva de la fórmula conocida para el área de un trapecio. Siguiendo un procedimiento similar como en la sección anterior, esto se puede expresar en el dominio z por:

$$z\sum(z) = \sum(z) + I(z).\frac{1+z}{2}.T_{sample} \qquad (9)$$

Lo que resulta en :

$$\sum(z) = \frac{T_{sample}}{2}\frac{z+1}{z-1}.I(z) \qquad (10)$$

Esta vez, el integrador continuo $\frac{1}{S}$ en la ecuación (1) se sustituye por toda el factor anterior a I (z). La función de transferencia en el dominio discreto se obtiene a partir de (1) y (10) como:

$$U(z) = K_P.\left(1 + \omega_{PI}.\frac{T_{sample}}{2}\frac{z+1}{z-1}\right)I(z)$$

$$= \frac{K_P\left(\frac{\omega_{PI}.T_{sample}}{2}+1\right)z + K_P.\left(\frac{\omega_{PI}.T_{sample}}{2}-1\right)}{z-1}I(z) \qquad (11)$$

Esto corresponde a la siguiente ecuación de diferencias:

$$U_{k+1} = K_P.\left(\frac{\omega_{PI}.T_{sample}}{2}+1\right)I_{k+1} + K_P\left(\frac{\omega_{PI}.T_{sample}}{2}-1\right).I_k + U_k \qquad (12)$$

1.3 Implementación en un procesador digital de señales

Es fácil ver que tanto las ecuaciones de diferencia de (7) y (12) son de la misma forma como se informa a continuación:

$$U_{k+1} = A_1.I_{k+1} + A_0.I_k + U_k \qquad (13)$$

Esta es una secuencia de multiplicar y acumular las operaciones que son ideales para su implementación en un DSP como el ADSP-21990. Sin embargo, se debe tener cuidado debido a la representación de punto fijo de 16 bits de los valores. Para un correcto funcionamiento, todos los coeficientes y las señales son que reducirse al formato común 15. Suponiendo que el error de entrada que ya está en este formato, los coeficientes pueden tener en general a ser escalado a estar dentro de este rango. Esto se logra mediante la introducción de un factor de escala B0 y reformular la ecuación (13) como sigue:

$$U_{k+1}.B_0 = A_1.B_0.I_{k+1} + A_0.B_0.I_k + U_k.B_0 = A_1^{sc}.I_{k+1} + A_0^{sc}.I_k + U_k.B_0 \qquad (14)$$

donde los ápice 'sc' denota los valores escalados de los coeficientes. B0 se elige de tal manera que ambos coeficientes están en el rango entre -1 y 1, con la precisión máxima. Además, si se elige como una potencia de 2, la final de descalcificación es una operación sencilla de desplazamiento. Esto se explicará con más detalle más adelante en la Sección 2.1.

1.4 Precisión de la rutina

Se desprende de la ecuación (14) que los incrementos de Uk en cada iteración se hace más pequeño a medida que disminuye de error de entrada y como Ik + 1 no cambia de manera significativa en comparación con su valor previois Ik. Además, los coeficientes de escala pueden ser pequeñas. Podría por lo tanto, en algunos casos producirse Uk que ya no se incrementa incluso con error de entrada distinto de cero, lo que conduce a un error de estado estacionario. Una forma de mejorar el controlador es aumentar la longitud de palabra de Uk. Por lo tanto, la biblioteca propuesto incluye una versión de 32 bits del algoritmo. Un ejemplo de salida será incluida en la sección 4.

1.5 Contra Wind-up

El fenómeno de Wind-up de la parte integral se puede evitar fácilmente mediante la saturación de la suma actual Uk + 1 siempre que excede 1 o es menor que -1. Esta característica se incorpora en las rutinas que aquí se presentan. Consulte la Sección 2.5 para más detalles sobre este tema.

2 Uso de las rutinas de PI

2.1 Determinación de los coeficientes

Una vez que el controlador PI se sintoniza en el dominio de tiempo continuo (por lo tanto, dado $K_P \; y \; \omega_{PI}$), el usuario tiene que elegir un tiempo de muestreo adecuado TSample. Se puede demostrar que para valores tales que $\omega_{PI}.T_{sample} \leq \frac{1}{20}$ o $\omega_{PI}.T_{sample} \leq \frac{1}{10}$ en los casos con retención de orden cero o primer orden , respectivamente, el error de aproximación es menor que 3%. Para valores que son superiores a estos límites, el controlador discreta ya no se comportan como su pendiente continua. A continuación, A1 y A0 se han de determinar por comparación de los coeficientes de la ecuación (13) con los de (7) y (12) para ZOH y FOH, respectivamente. Por fin, un factor de escala apropiado tiene que ser encontrado. Es fácil ver que todos los coeficientes de la ecuación (13) se convierten en el formato 1.15 si se dividen por el valor siguiente:

$$B_{real} = max(|A_1|, |A_0|, 1) \qquad (15)$$

Sin embargo, con el fin de simplificar el proceso de descalcificación y el procedimiento contra el wind-up, se elige el factor de escala para ser una potencia de dos, como se mencionó en la sección 1.3. Si n denota un valores enteros no negativos tales que, $B_0 = 2^{-n}$, está claro que n puede ser encontrado por :

$$n = \overline{log_2(B_{real})} \qquad (16)$$

donde la barra por encima de la expresión significa la operación de redondeo hacia el número entero inmediatamente superior. La siguiente tabla reasume todas las operaciones que se requieren para implementar el controlador PI.

Table 1 Determination of the coefficients

Parameter	*ZOH*	*FOH*
T_{sample}	$T_{sample} \leq \frac{1}{20} \cdot \omega_{PI}^{-1}$	$T_{sample} \leq \frac{1}{10} \cdot \omega_{PI}^{-1}$
A_1	$A_1 = K_P$	$A_1 = K_P\left(\frac{\omega_{PI} \cdot T_{sample}}{2} + 1\right)$
A_0	$A_0 = K_P\left(\omega_{PI} \cdot T_{sample} - 1\right)$	$A_0 = K_P\left(\frac{\omega_{PI} \cdot T_{sample}}{2} - 1\right)$
n	$n = \overline{\log_2\left(\max\left(\left\|A_1\right\|, \left\|A_0\right\|, 1\right)\right)}$	
B_0	$B_0 = 2^{-n}$	
A_1^{sc}	$A_1^{sc} = A_1 \cdot B_0$	
A_0^{sc}	$A_0^{sc} = A_0 \cdot B_0$	

2.2 Uso de la rutina de controlador

Las rutinas se desarrollan como una librería fácil de usar, que tiene que estar vinculado a la aplicación del usuario. La librería consta de dos archivos. El "pi.dsp" archivo contiene el código ensamblador para las subrutinas. Este paquete tiene que ser compilado y puede ser ligado a una aplicación. El usuario tiene que incluir el archivo de cabecera "pi.h", que proporciona una llamada de función como de las rutinas. El archivo de ejemplo en la siguiente sección demostrará el uso del controlador. La siguiente tabla reasume el conjunto de macros definidos en esta librería.

Table 2 Implemented routines

Precision	*Operation*	*Usage*	*Input*	*Output*
32 bit	Initialisation	**PI32_Init**(Delay_line, Initial_value);	none	none
	PI	**PI32**(Delay_line, Coefficients, Scale_Shift);	ar	sr1

Como se verá con más detalle en la Sección 2.5, la rutina de control PI requiere los coeficientes A1, A0 almacenado (en este orden) en la memoria de programa. Esto se hace con un buffer circular llamados coeficientes en la Tabla 2 Del mismo modo, Ik + 1 y Uk+1 (MSW y LSW) necesitan ser almacenados en un buffer circular llamado Delay_line (en memoria de datos y en este orden) al final de la computación, ya que son necesarios para la siguiente llamada a la rutina. Scale_Shift denota el valor entero n que se ha definido por la ecuación (2.2). Initial_value es la salida inicial de la PI (correspondiente al valor inicial del integrador en el caso continuo). Una llamada a PI32_Init inicializa el Delay_line búfer en la limpieza de Ik y cargar Uk con este parámetro (restablece la PI si initial_value es igual a cero). Por fin, se requiere que el error de entrada para la rutina PI (definida como diferencia entre el valor de referencia y la señal de realimentación) para ser almacenado en el registro ar (formato float). El valor de salida (formato float) está contenida en el registro sr1 después de ejecutar la rutina

2.3 Uso de los registros de DSP

En esta librería, dos macros se definen, como se muestra en la Tabla 2, laTabla 3 ofrece una visión general de los registradores de núcleo DSP que son modificados por ellos. Además, la línea de retardo también se modifica.

Table 3 Usage of DSP core registers for the subroutines

Precision	*Usage*	*Modified registers*
32 bit	**PI32_Init**(Delay_line, Initial_value);	I3, L3, ar, B3, M3
	PI32(Delay_line, Coefficients, Scale_Shift);	I3, L3, ax0, B3, M3, I7, L7, ay0, mx1, mx0, mr1, mr0, ar, se, sr, my0

2.4 El acceso a la librería a través del archivo de cabecera: pi.h

La Liberia se puede acceder mediante la inclusión del archivo de cabecera "pi.h" en el código de aplicación. El archivo de cabecera está destinado a proporcionar llamadas de función como a las rutinas de PI. En él se definen las llamadas que se muestran en la Tabla 2 El uso de ellos requiere que los buffers Delay_line y Coeficientes de ser puesta en marcha correctamente. el parámetro Initial_value sirve como valor de reposición para la rutina de inicialización. Observe cómo se utiliza I3 señalar el buffer de línea de retardo en la memoria de datos y I7 señala el buffer de coeficiente en la memoria del programa. Para obtener más información, por favor consulte los comentarios en el archivo "pi.h".

2.5 El código de programa: pi.dsp

En este archivo, se define la rutina PI32_Control_. Suponiendo que los registros del DAG I3 e I7 se han establecido correctamente (en la macro "PI32"), entonces la línea de retardo se carga en los registros centrales. La multiplicación por B0 del último término de la ecuación (14) se logra por un cambio en el valor entero de 32 bits Uk. A continuación, la ecuación (14) se ejecuta. A continuación la rutina comprueba si el cambio final por n (contenida en ay0) producirán un resultado Uk + 1 que es más grande que 1 en valor absoluto. Si no, el desplazamiento se ejecuta y el valor de salida se almacena en la línea de retardo para el siguiente ciclo. Si el resultado desplazado sería más grande que uno, Uk + 1 está saturado a 1 o -1 (dependiendo del signo del resultado) y este valor se almacena. Para obtener más detalles, consulte "pi.dsp".

3 Resolución de Ecuaciones Diferenciales Ordinarias (ODEs)

Todo problema que involucre ecuaciones diferenciales, puede ser reducido a un conjunto de ecuaciones diferenciales de primer orden.

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x)$$

$$\frac{dy}{dx} = z(x)$$
$$\frac{dz}{dx} = r(x) - q(x)z(x)$$

- Cualquier problema genérico se reduce entonces, al estudio de un conjunto de *N* ecuaciones diferenciales acopladas de primer orden, de la forma general:

$$\frac{dy_i(x)}{dx} = f_i(x, y_1, \ldots, y_N), \qquad i = 1, \ldots, N$$

donde las funciones f_i son conocidas.

- La idea general en la que se basan los distintos métodos de resolución de ODEs es la misma:

Reescribir los *dy's* y *dx's* como saltos finitos D*y* y D*x,* y multiplicar las ecuaciones por D*x*.

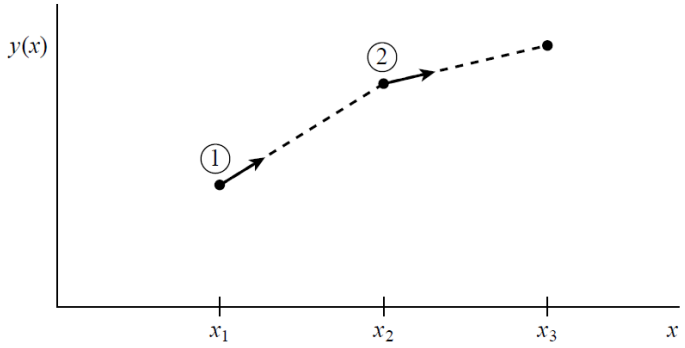
3.1 Método de Euler

La implementación literal del procedimiento descrito da lugar al método de Euler, el cual resulta:

$$y_{n+1} = y_n + hf(x_n, y_n)$$

- donde:
- D*x* = *h* y D*y* = $y_{n+1} - y_n$

Gráficamente



- El método de Euler resulta conceptualmente importante, aunque no funciona adecuadamente en muchos sistemas prácticos.
- Las mejoras a este método dan lugar a un gran número de algoritmos de resolución de ecuaciones.
- Todos se basan, sin embargo, en el mismo concepto original que consiste en multiplicar las funciones de derivadas por pequeños incrementos en la variable *x*.

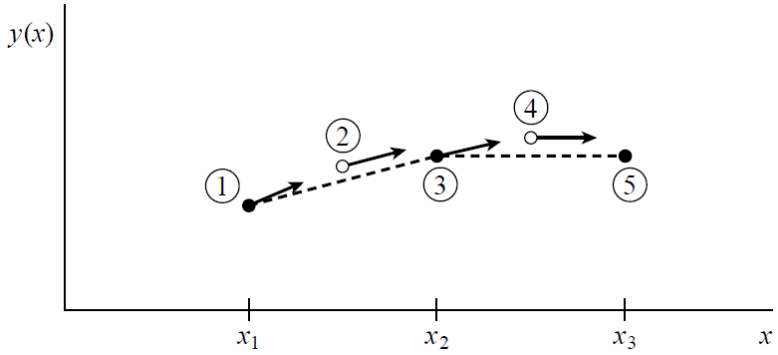
3.2 Métodos de Runge-Kutta

Si se toma un paso intermedio antes de realizar el cálculo definitivo de la función, se obtiene:

$$k_1 = hf(x_n, y_n)$$
$$k_2 = hf\left(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1\right)$$
$$y_{n+1} = y_n + k_2 + O(h^3)$$

- Puede demostrarse que este paso da como resultado un error menor al cometido con el método de Euler y se lo denomina Runge – Kutta de 2° orden.

Gráficamente

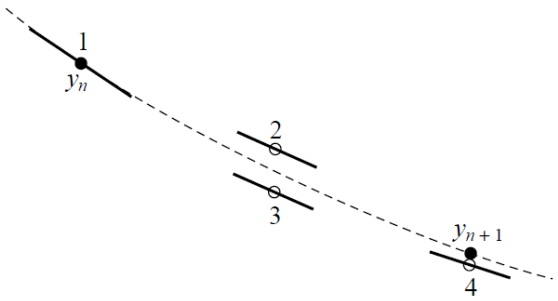


- El proceso puede extenderse y realizar múltiples saltos previos para obtener un promedio final.

- El método de Runge – Kutta de 4° orden resulta:

$$k_1 = hf(x_n, y_n)$$
$$k_2 = hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2})$$
$$k_3 = hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2})$$
$$k_4 = hf(x_n + h, y_n + k_3)$$
$$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)$$

- Gráficamente



- Este método requiere de 4 evaluaciones de la función derivada antes de realizar el cálculo de la función *y* en el instante *n+1*.

- Es, por lejos, el método de Runge – Kutta más utilizado.

- Diversas mejoras dan lugar a métodos de Runge – Kutta de mayor orden, aunque la reducción del error obtenida no siempre justifique el incremento en la cantidad de cálculo requerido.

4. Software de la implementación y simulación

Los siguientes ejemplos son simulaciones de la resolución de ecuaciones diferenciales ordinarias aplicadas al modelo matemático del comportamiento de una máquina de DC de iman permanente, con diferentes métodos.

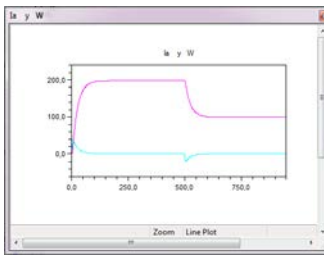
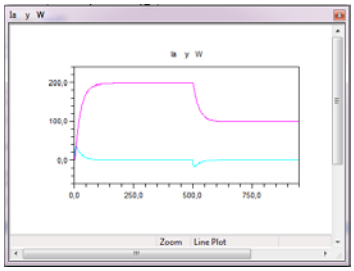
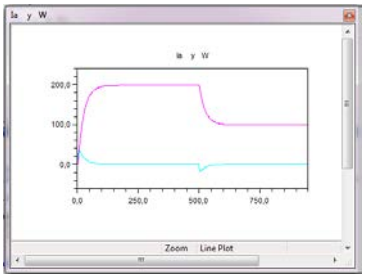
Ejemplo N1.- MaqDC1_0.c

En el ejemplo N1, se simula el arranque de una maquina DC, se visualiza el comportamiento de la corriente de armadura Ia, y la velocidad angular del motor w, con un voltaje de alimentación de 100v y unos parámetros de la máquina: K=0.5, Ra=2.4,La=4.1e-3,B=0.001,J=0.0027; y un tiempo de muestreo h=1e-3, simulación implementada para visualizar el comportamiento para 1000 muestras, se resuelve la simulación por tres métodos: EU (Euler), RK2 (Runge-Kutta de 2do orden) y RK4 (Runge-Kutta de 4to orden)

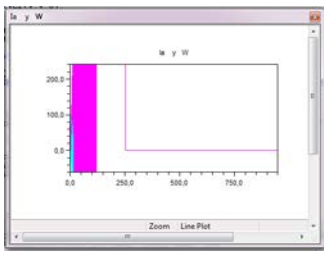
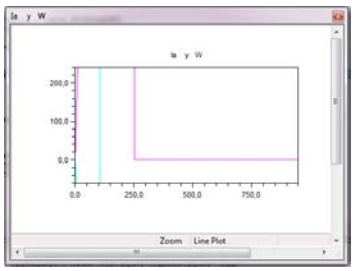
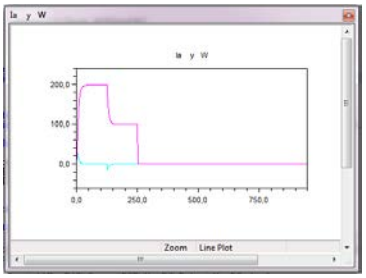
```
//Simulación del arranque de una máquina DC de imán permanente.
//Para la resolución de las ODE se puede escoger entre los
//métodos: EU (Euler), RK2 (Runge-Kutta de 2do orden) y RK4 (Runge-Kutta de 4to orden)

#define EU

float K=0.5,Va=100;
float t=0,h=1e-3;
float Ia,w;
float data1[1000],data2[1000];

void modelo(float *Y, float *dY); //declaración de la función "modelo"
void solver(float *y,float *y_0); //declaración de la función "solver"

void main(void)
{
int i=0;
float t0;
float y[2]={0,0},y_0[2]={0,0};

        do
        {
        t+=h;
        i++;

        if (t>0.5)
                Va=50;

        solver(y,y_0);

        data1[i]=Ia=y_0[0]=y[0];
        data2[i]=w=y_0[1]=y[1];

        } while (t<1);

for(;;)
asm("nop;");
```

```
}

#ifdef EU
void solver(float *y,float *y_0) //definición de la función "solver" (Euler)
{
 float dy[2];

 modelo(y_0,dy);
 y[0]=y_0[0]+h*dy[0];
 y[1]=y_0[1]+h*dy[1];

}
#endif

#ifdef RK2
void solver(float *y,float *y_0) //definición de la función "solver" (Runge-Kutta 2do orden)
{
 float dy[2],y_aux[2];

 modelo(y_0,dy);
 y_aux[0]=y_0[0]+0.5*h*dy[0];
 y_aux[1]=y_0[1]+0.5*h*dy[1];

 modelo(y_aux,dy);
 y[0]=y_0[0]+h*dy[0];
 y[1]=y_0[1]+h*dy[1];

}
#endif

#ifdef RK4
void solver(float *y,float *y_0) //definición de la función "solver" (Runge-Kutta 4to orden)
{
 float dy1[2],dy2[2],dy3[2],dy4[2],y_aux[2];

 modelo(y_0,dy1);
 y_aux[0]=y_0[0]+h*dy1[0]/2.0;
 y_aux[1]=y_0[1]+h*dy1[1]/2.0;

 modelo(y_aux,dy2);
 y_aux[0]=y_0[0]+h*dy2[0]/2.0;
 y_aux[1]=y_0[1]+h*dy2[1]/2.0;

 modelo(y_aux,dy3);
 y_aux[0]=y_0[0]+h*dy3[0];
 y_aux[1]=y_0[1]+h*dy3[1];

 modelo(y_aux,dy4);
 y[0]=y_0[0]+h*(dy1[0]+2.0*dy2[0]+2.0*dy3[0]+dy4[0])/6.0;
 y[1]=y_0[1]+h*(dy1[1]+2.0*dy2[1]+2.0*dy3[1]+dy4[1])/6.0;
}
#endif
```

```
void modelo(float *Y, float *dY) //definición de la función "modelo"
{
float Ra=2.4,La=4.1e-3,B=0.001,J=0.0027;

dY[0]=(1/La)*(Va-K*Y[1]-Y[0]*Ra);    //Y[0]=Ia
dY[1]=(1/J)*(K*Y[0]-B*Y[1]);         //Y[1]=w
                                                    //dY[0]=dIa_dt;
                                                    //dY[1]=dw_dt;

}
```

EU	RK2	RK4
		

Con float t=0,h=4e-3;

Como se puede visualizar en los gráficos respectivos la señal de voltaje Va (color rosado) y la señal de la corriente Ia (color celeste), casi no existe diferencia de simulación en los métodos de resolución, cuando se aumenta el tiempo de muestreo a h=4e-3, se torna critico el método por el cual se resuelve las OEDs del modelo matemático que corresponde al comportamiento de una máquina DC

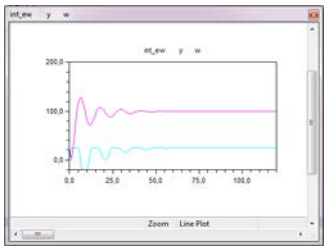
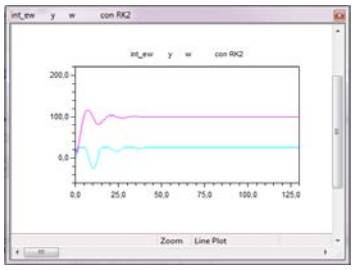
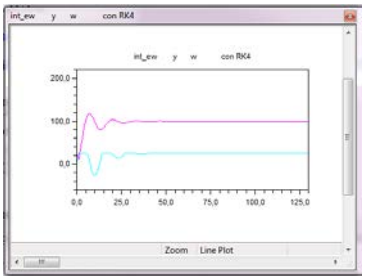
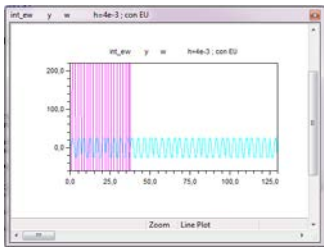
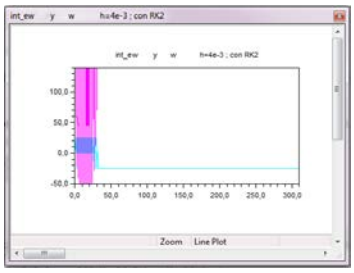
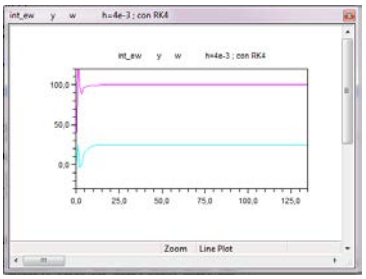
Ejemplo N2.- MaqDC2_0

En el ejemplo N2, se simula el arranque de una maquina DC con control PI en el lazo de velocidad, se visualiza el comportamiento de la velocidad angular del motor w, y de su variación int_ew con un voltaje de alimentación de 100v y unos parámetros de la máquina: K=0.5, Ra=2.4,La=4.1e-3,B=0.001,J=0.0027; y un tiempo de muestreo h=1e-3, simulación implementada para visualizar el comportamiento para 1000 muestras, se resuelve la simulación por tres métodos: EU (Euler), RK2 (Runge-Kutta de 2do orden) y RK4 (Runge-Kutta de 4to orden)

```
//Simulación del arranque de una máquina DC de imán permanente
//con control proporcional integrativo en el lazo de velocidad.
//Para la resolución de las ODE se puede escoger entre los
//métodos: EU (Euler), RK2 (Runge-Kutta de 2do orden) y RK4 (Runge-Kutta de 4to orden)

#define EU
#define Kp      0.05
#define Ki      0.02

float K=0.5,Va=100,Tm=0.0;
float t=0,h=1e-3,Uk;
float Ia,w,w_ref,ew,int_ew=0;
float data1[1000],data2[1000];

void modelo(float *Y, float *dY); //declaración de la función "modelo"
void solver(float *y, float *y_0); //declaración de la función "solver"

void main(void)
{
int i=0;
float t0;
float y[2]={0,0},y_0[2]={0,0};

w_ref=100;

        do
        {
        t+=h;
        i++;

        ew=w_ref-w;
        int_ew+=ew;

        if (int_ew>25) int_ew=25;
        if (int_ew<-25) int_ew=-25;

        Uk=Kp*ew+Ki*int_ew;

//      if (Uk>1) Uk=1;
//      if (Uk<-1) Uk=-1;

        Va=Uk*100;

        solver(y,y_0);
```

```
            y_0[0]=y[0];
            y_0[1]=y[1];

            Ia=y_0[0];
            w=y_0[1];

            data1[i]=int_ew;
            data2[i]=w;

            } while (i<1000);

for(;;)
asm("nop;");

}

#ifdef EU
void solver(float *y,float *y_0) //definición de la función "solver" (Euler)
{
 float dy[2];

 modelo(y_0,dy);
 y[0]=y_0[0]+h*dy[0];
 y[1]=y_0[1]+h*dy[1];

}
#endif

#ifdef RK2
void solver(float *y,float *y_0) //definición de la función "solver" (Runge-Kutta 2do
orden)
{
 float dy[2],y_aux[2];

 modelo(y_0,dy);
 y_aux[0]=y_0[0]+0.5*h*dy[0];
 y_aux[1]=y_0[1]+0.5*h*dy[1];

 modelo(y_aux,dy);
 y[0]=y_0[0]+h*dy[0];
 y[1]=y_0[1]+h*dy[1];

}
#endif

#ifdef RK4
void solver(float *y,float *y_0) //definición de la función "solver" (Runge-Kutta 4to
orden)
{
 float dy1[2],dy2[2],dy3[2],dy4[2],y_aux[2];

 modelo(y_0,dy1);
 y_aux[0]=y_0[0]+h*dy1[0]/2.0;
 y_aux[1]=y_0[1]+h*dy1[1]/2.0;
```

```
 modelo(y_aux,dy2);
 y_aux[0]=y_0[0]+h*dy2[0]/2.0;
 y_aux[1]=y_0[1]+h*dy2[1]/2.0;

 modelo(y_aux,dy3);
 y_aux[0]=y_0[0]+h*dy3[0];
 y_aux[1]=y_0[1]+h*dy3[1];

 modelo(y_aux,dy4);
 y[0]=y_0[0]+h*(dy1[0]+2.0*dy2[0]+2.0*dy3[0]+dy4[0])/6.0;
 y[1]=y_0[1]+h*(dy1[1]+2.0*dy2[1]+2.0*dy3[1]+dy4[1])/6.0;
}
#endif

void modelo(float *Y, float *dY) //definición de la función "modelo"
{
float Ra=2.4,La=4.1e-3,B=0.001,J=0.0027;

dY[0]=(1/La)*(Va-K*Y[1]-Y[0]*Ra);    //Y[0]=Ia
dY[1]=(1/J)*(K*Y[0]-B*Y[1]-Tm);                  //Y[1]=w
                                                                  //dY[0]=dIa_dt;
                                                                  //dY[1]=dw_dt;

}
```

EU	RK2	RK4
		
		

Como se puede visualizar en los gráficos anteriores el control PI realiza su acción sobre la velocidad angular w (color rojo), también se visualiza la señal del error de la velocidad int_ew (color celeste), este control es uno de los mas utilizados, en los respectivos gráficos se visualiza que casi no existe diferencia de simulación en los métodos de resolución aun tiempo de muestreo de h=1e-3, cuando se aumenta el tiempo de muestreo a h=3e-3, se torna critico el método por el cual se resuelve las

OEDs del modelo matemático que corresponde al comportamiento de una máquina DC y se torna mas crítico el control de la máquina DC.

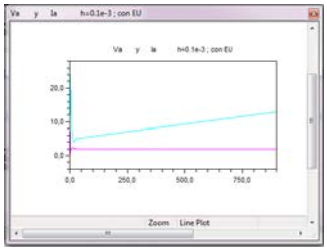
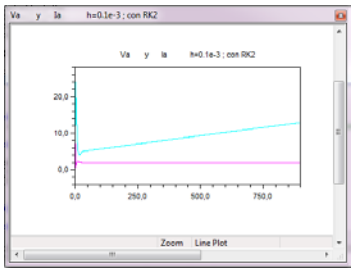
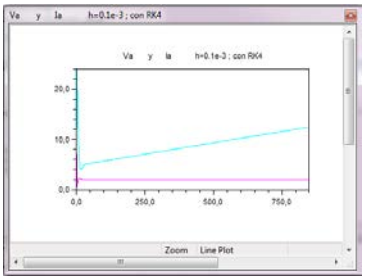
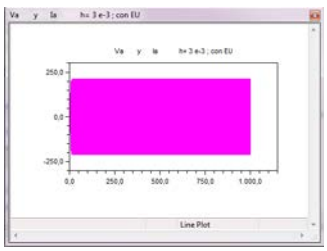
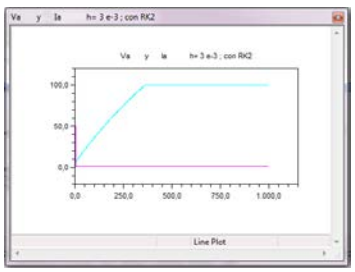
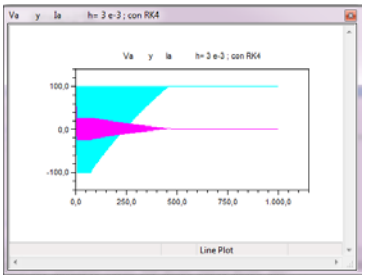
Ejemplo N3.- MaqDC3_0

En el ejemplo N3, se simula el arranque de una maquina DC con control PI en el lazo de corriente, se visualiza el comportamiento de la señal de voltaje y la señal de corriente Ia, con un voltaje de alimentación de 100v y unos parámetros de la máquina: K=0.5, Ra=2.4,La=4.1e-3,B=0.001,J=0.0027; y un tiempo de muestreo h=0.1e-3, simulación implementada para visualizar el comportamiento para 1000 muestras, se resuelve la simulación por tres métodos: EU (Euler), RK2 (Runge-Kutta de 2do orden) y RK4 (Runge-Kutta de 4to orden)

```
//Simulación del arranque de una máquina DC de imán permanente
//con control proporcional integrativo en el lazo de corriente.
//Para la resolución de las ODE se puede escoger entre diversos
//métodos: EU (Euler), RK2 (Runge-Kutta de 2do orden) y RK4 (Runge-Kutta de 4to orden)

#define EU
#define Kp      10
#define Ki      2

float K=0.5,Va=100,Tm=0.5;
float t=0,h=0.1e-3;
float Ia,w,Ia_ref,eIa=0,int_eIa=0;
float data1[1000],data2[1000],data3[1000],data4[1000];

void modelo(float *Y, float *dY); //declaración de la función "modelo"
void solver(float *y, float *y_0); //declaración de la función "solver"

void main(void)
{
int i=0;
float t0;
float y[2]={0,0},y_0[2]={0,0};

Ia_ref=2;

        do
        {
        t+=h;
        i++;

        eIa=Ia_ref-Ia;
        int_eIa+=eIa;
        Va=Kp*eIa+Ki*int_eIa;

        if(Va>100) Va=100;
        if(Va<-100) Va=-100;

        solver(y,y_0);
```

```
            y_0[0]=y[0];
            y_0[1]=y[1];

            Ia=y_0[0];
            w=y_0[1];

            data1[i]=Va;
            data2[i]=Ia;
            data3[i]=w;
            data4[i]=Ki*int_eIa;

            } while (i<1000);

for(;;)
asm("nop;");

}

#ifdef EU
void solver(float *y,float *y_0) //definición de la función "solver" (Euler)
{
 float dy[2];

 modelo(y_0,dy);
 y[0]=y_0[0]+h*dy[0];
 y[1]=y_0[1]+h*dy[1];

}
#endif

#ifdef RK2
void solver(float *y,float *y_0) //definición de la función "solver" (Runge-Kutta 2do
orden)
{
 float dy[2],y_aux[2];

 modelo(y_0,dy);
 y_aux[0]=y_0[0]+0.5*h*dy[0];
 y_aux[1]=y_0[1]+0.5*h*dy[1];

 modelo(y_aux,dy);
 y[0]=y_0[0]+h*dy[0];
 y[1]=y_0[1]+h*dy[1];

}
#endif

#ifdef RK4
void solver(float *y,float *y_0) //definición de la función "solver" (Runge-Kutta 4to
orden)
{
 float dy1[2],dy2[2],dy3[2],dy4[2],y_aux[2];

 modelo(y_0,dy1);
 y_aux[0]=y_0[0]+h*dy1[0]/2.0;
```

```
 y_aux[1]=y_0[1]+h*dy1[1]/2.0;

 modelo(y_aux,dy2);
 y_aux[0]=y_0[0]+h*dy2[0]/2.0;
 y_aux[1]=y_0[1]+h*dy2[1]/2.0;

 modelo(y_aux,dy3);
 y_aux[0]=y_0[0]+h*dy3[0];
 y_aux[1]=y_0[1]+h*dy3[1];

 modelo(y_aux,dy4);
 y[0]=y_0[0]+h*(dy1[0]+2.0*dy2[0]+2.0*dy3[0]+dy4[0])/6.0;
 y[1]=y_0[1]+h*(dy1[1]+2.0*dy2[1]+2.0*dy3[1]+dy4[1])/6.0;
}
#endif

void modelo(float *Y, float *dY) //definición de la función "modelo"
{
float Ra=2.4,La=4.1e-3,B=0.001,J=0.0027;

dY[0]=(1/La)*(Va-K*Y[1]-Y[0]*Ra);    //Y[0]=Ia
dY[1]=(1/J)*(K*Y[0]-B*Y[1]-Tm);                  //Y[1]=w
                                                                    //dY[0]=dIa_dt;
                                                                    //dY[1]=dw_dt;

}
```

EU	RK2	RK4
		
		

Como se puede visualizar en los gráficos anteriores el control PI realiza su acción sobre la señal de la corriente Ia (color rosado), también se visualiza la señal del voltaje Va (color celeste), este control PI es aplicado al lazo de corriente, en los respectivos

gráficos se visualiza que casi no existe diferencia de simulación en los métodos de resolución aun tiempo de muestreo de h=0.1e-3, cuando se aumenta el tiempo de muestreo a h=3e-3, se torna critico el método por el cual se resuelve las OEDs del modelo matemático que corresponde al comportamiento de una máquina DC y se torna más crítico el control de la máquina DC.

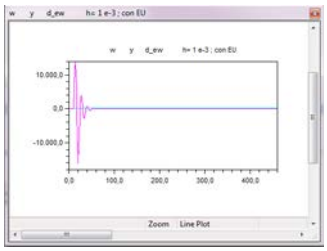
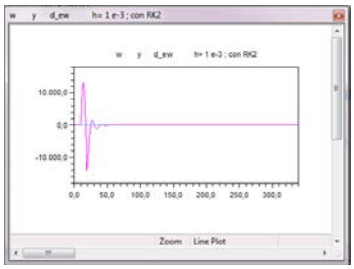
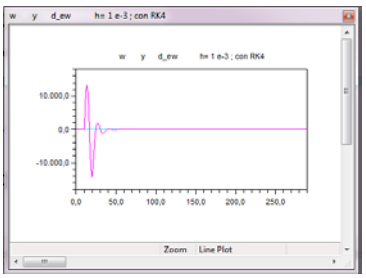
Ejemplo N4.- MaqDC4_0

En el ejemplo N4, se simula el arranque de una maquina DC con control PI en el lazo de velocidad, se visualiza el comportamiento de la velocidad angular del motor w, y de su variación d_ew con un voltaje de alimentación de 100v y unos parámetros de la máquina: K=0.5, Ra=2.4,La=4.1e-3,B=0.001,J=0.0027; y un tiempo de muestreo h=1e-3, simulación implementada para visualizar el comportamiento para 1000 muestras, se resuelve la simulación por tres métodos: EU (Euler), RK2 (Runge-Kutta de 2do orden) y RK4 (Runge-Kutta de 4to orden)

```
//Simulación del arranque de una máquina DC de imán permanente
//con control PID en el lazo de velocidad.
//Para la resolución de las ODE se puede escoger entre los
//métodos: EU (Euler), RK2 (Runge-Kutta de 2do orden) y RK4 (Runge-Kutta de 4to orden)

#include<filters.h>

#define RK2
#define Kp      0.025
#define Ki      0.010
#define Kd      0.00004

#define TAPS 9

/*Segunda fila de la matriz Coeff*/
float pm f1_coeffs[]={-0.06666666666667,-0.05,-0.03333333333333,-0.01666666666667,

        0,0.01666666666667,0.03333333333333,0.05,0.06666666666667};


float K=0.5,Va=100,Tm=0.0;
float t=0,h=1e-3,Uk;
float Ia,w,w_ref,ew,int_ew=0,d_ew;
float data1[1000],data2[1000];


void modelo(float *Y, float *dY); //declaración de la función "modelo"
void solver(float *y, float *y_0); //declaración de la función "solver"

void main(void)
{
int i=0;
float t0;
float y[2]={0,0},y_0[2]={0,0};
```

```
float state[TAPS+1];

w_ref=100;

for(i=0;i<TAPS+1;i++)
        state[i]=0.0;

        do
        {
        t+=h;
        i++;

        ew=w_ref-w;
        int_ew+=ew;
        d_ew=(1/h)*fir(ew,f1_coeffs,state,TAPS);

        if (int_ew>50) int_ew=50;
        if (int_ew<-50) int_ew=-50;


        Uk=Kp*ew+Ki*int_ew+Kd*d_ew;

//      if (Uk>1) Uk=1;
//      if (Uk<-1) Uk=-1;


        Va=Uk*100;

        solver(y,y_0);

        y_0[0]=y[0];
        y_0[1]=y[1];

        Ia=y_0[0];
        w=y_0[1];

        data1[i]=w;//int_ew;
        data2[i]=d_ew;


        } while (i<1000);

for(;;)
asm("nop;");

}

#ifdef EU
void solver(float *y,float *y_0) //definición de la función "solver" (Euler)
{
 float dy[2];

 modelo(y_0,dy);
 y[0]=y_0[0]+h*dy[0];
 y[1]=y_0[1]+h*dy[1];
```

```
}
#endif

#ifdef RK2
void solver(float *y,float *y_0) //definición de la función "solver" (Runge-Kutta 2do orden)
{
 float dy[2],y_aux[2];

 modelo(y_0,dy);
 y_aux[0]=y_0[0]+0.5*h*dy[0];
 y_aux[1]=y_0[1]+0.5*h*dy[1];

 modelo(y_aux,dy);
 y[0]=y_0[0]+h*dy[0];
 y[1]=y_0[1]+h*dy[1];

}
#endif

#ifdef RK4
void solver(float *y,float *y_0) //definición de la función "solver" (Runge-Kutta 4to orden)
{
 float dy1[2],dy2[2],dy3[2],dy4[2],y_aux[2];

 modelo(y_0,dy1);
 y_aux[0]=y_0[0]+h*dy1[0]/2.0;
 y_aux[1]=y_0[1]+h*dy1[1]/2.0;

 modelo(y_aux,dy2);
 y_aux[0]=y_0[0]+h*dy2[0]/2.0;
 y_aux[1]=y_0[1]+h*dy2[1]/2.0;

 modelo(y_aux,dy3);
 y_aux[0]=y_0[0]+h*dy3[0];
 y_aux[1]=y_0[1]+h*dy3[1];

 modelo(y_aux,dy4);
 y[0]=y_0[0]+h*(dy1[0]+2.0*dy2[0]+2.0*dy3[0]+dy4[0])/6.0;
 y[1]=y_0[1]+h*(dy1[1]+2.0*dy2[1]+2.0*dy3[1]+dy4[1])/6.0;
}
#endif

void modelo(float *Y, float *dY) //definición de la función "modelo"
{
float Ra=2.4,La=4.1e-3,B=0.001,J=0.0027;

dY[0]=(1/La)*(Va-K*Y[1]-Y[0]*Ra);    //Y[0]=Ia
dY[1]=(1/J)*(K*Y[0]-B*Y[1]-Tm);                  //Y[1]=w
                                                                        //dY[0]=dIa_dt;
                                                                        //dY[1]=dw_dt;
}
```

EU	RK2	RK4
		

Como se puede visualizar en los gráficos anteriores se pretende realizar un control PID sobre la señal de velocidad (color celeste), también se visualiza la variación de la señal del error de velocidad (color rosado), luego de pasar por un filtro de Savitsky-Golay, este control PID no esta sintonizado con los parámetos Kp= 0.025, Ki=0.010 y Kd=0.00004, queda de tarea sintonizar este control, tomar en cuenta el tiempo de muestreo y el método de resolver las OEDs del modelo matemático que corresponde al comportamiento de una máquina DC

5. Referencias Bibliográficas

1. William H. Press, Saul A. Teukolsky, "Numerical Recipes in C The Art of Scientific Computing", Capitulo 16, 2da. Ed, Cambridge University Press , 2002
2. Analog Devices Inc.," ADSP-21990: Implementation of PI Controllers",2001
3. Viola J, Restrepo J, "Apuntes de curso de DPS ", 2014