SECO-Simulabo 2015-I

Félix Monasterio-Huelin

8 de febrero de 2015

Índice

Índice de Figuras

Índice de Tablas

1. Introducción

Con este escrito se pretende hacer una introducción al problema de representación de sistemas lineales en Matlab y Simulink. Se supone que el lector conoce la teoría que justifica cada uno de las representaciones, y que tiene a su disposición esos programas, de modo que pueda seguir estas páginas realizando simulaciones.

Se verán como ejemplos un circuito RLC en la Sección 6, un Motor DC en la Sección 7, y cómo generar señales de referencia polinómicas causales en la Sección 8.

En el Apéndice A se presenta una introducción a la manipulación de variables y funciones simbólicas y a su convensión a variables y funciones numéricas.

En lo que sigue se va a utilizar la **versión R2013a de Matlab** utilizando el sistema operativo **Linux**. Este documento se ha escrito en **Latex**. A este documento le acompaña una colección de ficheros de tipo **.mdl** y de tipo **.m**, así como dos ficheros de datos **DatosSistema.mat** y **DatosSistema.txt**, que han sido creados para su elaboración. Se encuentran en el fichero comprimido **SecoSimulabo2015-I-mdl-y-m.tar.gz**.

2. Control System Toolbox

Abriendo la ayuda (**help**) de Matlab y buscando **Control System Toolbox** aparece un conjunto de secciones dedicadas a las funciones orientadas al modelado, análisis y diseño de Sistemas de Control. En lo que sigue no se pretende sustituir la ayuda de Matlab ni abarcar toda su potencialidad.

Dedicaremos esta Sección a exponer algunas de estas funciones con el fin de hacer una introducción esquemática. Nos centraremos en la representación de sistemas lineales así como en la conversión entre unas y otras, poniendo un ejemplo sobre la cancelación de ceros y polos.

Debe tenerse en cuenta que todas estas funciones son numéricas. No obstante es posible trabajar en Matlab con funciones simbólicas. En el Apéndice A se presenta una introducción a la manipulación de variables y funciones simbólicas y a su convensión a variables y funciones numéricas.

2.1. Representaciones de sistemas en Matlab y conversión entre ellas

Hay dos representaciones fundamentales de los sistemas lineales:

1. **En el dominio del tiempo**

   Un sistema lineal puede representarse como una ecuación diferencial de orden n o como n ecuaciones diferenciales de orden uno junto con una ecuación (no diferencial) de salida. La primera la llamaremos **representación de entrada/salida** y la segunda **representación en el espacio de estados**. En ambos casos es necesario especificar las condiciones iniciales.

   La función de Matlab para la representación en el espacio de estados es **ss**.

2. **En el dominio complejo**

   Un sistema lineal suele representarse en el dominio complejo a través de una **función de transferencia**, junto con una función racional compleja de las condiciones iniciales.

   Las funciones de Matlab para la representación en el dominio complejo son **tf** y **zpk**.

En la Tabla 2.1 se muestran las funciones de Matlab para la representación de sistemas.

función Matlab				Comentario
Continuo		Muestreado		
$G1 =$	$tf(num, den)$	$G1d =$	$tf(num, den, T)$	T no especificado si $T = -1$
$G2 =$	$zpk(z, p, k)$	$G2d =$	$zpk(z, p, k, T)$	T no especificado si $T = -1$
$L =$	$ss(A, B, C, D)$	$Ld =$	$ss(A, B, C, D, T)$	T no especificado si $T = -1$

Tabla 2.1: Funciones LTI (Linear Time Invariant) para el modelado numérico en Matlab

En la Tabla 2.2 se recogen las funciones que permiten convertir una representación en otra.

función Matlab	
$[num, den] =$	$ss2tf(A, B, C, D)$
$[num, den] =$	$zp2tf(z, p, k)$
$[num, den] =$	$tfdata(G2,\ 'v')$
$[A, B, C, D] =$	$tf2ss(num, den)$
$[A, B, C, D] =$	$zp2ss(z, p, k)$
$[A, B, C, D] =$	$ssdata(G1)$
$[z, p, k] =$	$ss2zp(A, B, C, D)$
$[z, p, k] =$	$tf2zp(num, den)$
$[z, p, k] =$	$zpkdata(L,\ 'v')$

Tabla 2.2: Conversión entre formas de representación de modelos LTI

Normalmente conviene que los polinomios del numerador y del denominador sean coprimos, es decir, que no tengan factores comunes. Si $G(s)$ es una función de transferencia puede utilizarse la función de Matlab **minreal** para realizar las cancelaciones de ceros y polos existentes. Por ejemplo:

```
a=input('a=');
num=[1 a];
den=[1 1+a a];
G=tf(num,den)
G1=minreal(G)
```

Tabla 2.3: Cancelaciones de **tf**: **minreal**

Podemos observar en las ecuaciones 2.1 que $G(s)$ es una función de transferencia sin cancelaciones, mientras que $G1(s)$ es la misma con cancelación del cero y polo común, $s = -a$. Por lo tanto,

convendrá trabajar con $G1$ preferentemente,

$$G(s) = \frac{s+a}{s^2+(1+a)s+a} \tag{2.1a}$$

$$G1(s) = \frac{1}{s+1} \tag{2.1b}$$

Con la función **minreal** también es posible realizar cancelaciones cuando el sistema está representado en el espacio de estados.

```
a=input('a=');
A=[0 1; -a -(1+a)];
B=[1; -1];
C=[1 0];
D=[0];
L=ss(A,B,C,D);
L1=minreal(L);
[A1 B1 C1 D1]=ssdata(L1)
[num,den]=ss2tf[A1 B1 C1 D1];
G1=tf(num,den)
```

Tabla 2.4: Cancelaciones de **ss**: **minreal**

Las matrices A, B, C, D representan el sistema sin cancelaciones:

$$A = \begin{bmatrix} 0 & 1 \\ -a & -(1+a) \end{bmatrix} \tag{2.2a}$$

$$B = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \tag{2.2b}$$

$$C = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{2.2c}$$

$$D = \begin{bmatrix} 0 \end{bmatrix} \tag{2.2d}$$

Al ejecutar las instrucciones de la tabla 2.5 se obtienen las siguientes matrices:

$$A1 = \begin{bmatrix} -1 \end{bmatrix} \tag{2.3a}$$

$$B1 = \begin{bmatrix} -\sqrt{2} \end{bmatrix} \tag{2.3b}$$

$$C1 = \begin{bmatrix} -\frac{\sqrt{2}}{2} \end{bmatrix} \tag{2.3c}$$

$$D1 = \begin{bmatrix} 0 \end{bmatrix} \tag{2.3d}$$

Puede comprobarse que la función de transferencia $G1$ coincide con la obtenida antes en la ecuación 2.1b.

Otra forma de hacer cancelaciones con la representación en el espacio de estados es utilizar la función de Matlab **modred**. Lo que hace esta función es simplificar el modelo, por lo que puede ser más general que una cancelación ya que permite eliminar estados cuya influencia en el comportamiento del sistema sea despreciable. Se puede saber qué estados pueden despreciarse utilizando la función de Matlab **balreal**. En el anterior ejemplo podemos hacer lo siguiente:

```
a=input('a=');
A=[0 1; -a -(1+a)];
B=[1; -1];
C=[1 0];
D=[0];
L=ss(A,B,C,D);
[LL,g]=balreal(L);
g
L2=modred(L,2,'MatchDC'); % O también: L2=modred(L,(g<1e-4),'MatchDC');
[A2 B2 C2 D2]=ssdata(L2)
```

Tabla 2.5: Cancelaciones de **ss**: **modred**

Pueden eliminarse los elementos de valor pequeño (por ejemplo de valor $g < 1e-4$)) de la diagonal principal de los gramianos, dados por el vector g. En este ejemplo,

$$g = \begin{bmatrix} 0{,}5 \\ 0 \end{bmatrix} \tag{2.4}$$

por lo que puede eliminarse el estado x_2, ya que $g(2) = 0 < 1e-4$.

Las matrices que se obtienen con la cancelación son:

$$A2 = \begin{bmatrix} -1 \end{bmatrix} \tag{2.5a}$$
$$B2 = \begin{bmatrix} 1 \end{bmatrix} \tag{2.5b}$$
$$C2 = \begin{bmatrix} 1 \end{bmatrix} \tag{2.5c}$$
$$D2 = \begin{bmatrix} 0 \end{bmatrix} \tag{2.5d}$$

Puede comprobarse que la función de transferencia correspondiente a esta representación en el espacio de estados es nuevamente $G1$, si bien $L1$ y $L2$ son realizaciones distintas de la misma función de transferencia.

3. Introducción a Simulink

Simulink dispone de un conjunto de herramientas que permite obtener los mismos resultados haciendo las cosas de diferente forma. En este estudio no pretendemos abarcar toda la potencialidad de Simulink ni de Matlab.

Podemos distinguir dos formas principales de representación gráfica de un sistema:

1. **La forma PS o Physical Signal**, que consiste en utilizar los elementos físicos del sistema, como por ejemplo resistencias, bobinas, condensadores, fuentes de alimentación, etc. para construir un circuito eléctrico. En lo que sigue solo utilizaremos la librería **Simscape** de Simulink. Este entorno puede ser útil en las situaciones en que no se disponga de un modelo matemático.

2. **La forma S o Simulink**. En realidad esta es la forma de representación abstracta, como la representación mediante funciones de transferencia o ecuaciones de estado y de salida. Dispone de elementos matemáticos integradores, derivadores, sumadores, etc. Existen en Simulink bloques específicos para hacer esto además de un bloque más general que permite programar el sistema con el lenguaje de Matlab.

Existen dos bloques que permiten transformar las señales generadas mediante una forma de representación en la otra: $\boldsymbol{PS} \rightarrow \boldsymbol{S}$ y $\boldsymbol{S} \rightarrow \boldsymbol{PS}$. Por lo tanto, es posible construir un sistema que combine ambas representaciones. Puede resultar útil cuando algún subsistema pueda ser fácilmente representado mediante un modelo matemático y algún otro subsistema mediante un modelo físico.

En el entorno S hay un bloque que permite guardar el valor de las señales en variables que pueden ser utilizadas en el entorno de trabajo de Matlab: **To Workspace**. Incluyendo este bloque en el esquema de simulación es posible entonces utilizar las variables creadas para su manipulación en el entorno de trabajo de Matlab, como por ejemplo para la generación de gráficas utilizando la función **plot**.

Por fin, hay un bloque en el entorno S que permite la generación de ficheros de datos de tipo MAT: **To File**. A partir del fichero generado es posible obtener otros ficheros con formato de texto en el entorno de trabajo de Matlab, como veremos en la Sección 5.

En las Secciones 6, 7 y 8 se verán algunos ejemplos. En todos los casos se guardará el modelo de simulación en un fichero de extensión **.mdl**.

4. Ejecución de una simulación

Para realizar una simulación con Simulink es necesario seleccionar el integrador numérico que se quiera utilizar. Esta selección se realiza abriendo la ventana de la pestaña **Simulation** y después **Model Configuration Parameters** y dentro de ella la opción **Solver**.

Hay dos clases de integradores numéricos, que se seleccionan en el apartado "Solver options" en "Type". En ambos casos hay una colección de integradores numéricos que se seleccionan en "Solver".

1. **Fixed-step**. El parámetro principal es el paso de integración constante "Fixed-step size (fundamental sample time)", que no es otra cosa que un periodo de muestreo constante de integración. Cuanto menor sea más puntos se crearán para el mismo intervalo de integración ("Stop time"-"Start time"), y en general mayor será la precisión.

2. **Variable-step**. El parámetro principal es el máximo paso de integración "Max step size" que juega un papel similar al del periodo de muestreo de los integradores de paso fijo.

No vamos a entrar aquí en más detalles, sino tan solo recalcar la importancia de hacer una selección adecuada del integrador de acuerdo con los propósitos de la simulación.

En ocasiones conviene hacer simulaciones con muy baja precisión hasta lograr ajustar todo el entorno de simulación, ya que de esta forma se reduce el tiempo de cálculo, aunque a expensas de la precisión. Cuando ya se ha logrado preparar el entorno de simulación, conviene ajustar el paso de integración para obtener suficiente resolución en los datos. Una de las razones es porque conviene que las curvas que se vayan a documentar aparezcan con la mejor resolución posible.

Si se pretende guardar datos de la simulación en un fichero o en alguna variable del espacio de trabajo, hay que abrir la opción **Data Import/Export** y dentro de ella seleccionar el fomato de

datos, "Format" y asignar algún valor a "Limit data points to last". Si se ha realizado una simulación con pasos de integración muy bajos conviene poner el límite máximo a un valor suficientemente elevado, o en caso contrario pueden producirse errores al hacer operaciones con variables de diferentes dimensiones.

5. Ficheros de datos

Vamos a ver diversas formas de crear ficheros a partir de la información obtenida desde Simulink, y de incorporar ficheros de datos y datos en Simulink.

1. Utilizando el bloque **To File**.

   Supongamos que se ha creado en el entorno de Simulink el fichero **nom.mat** utilizando el bloque **To File** con la opción "Save format" en "array".

   Los comandos en Matlab de la Tabla 5.1 permiten convertirlo en un fichero de texto de nombre **nom.txt** multicolumna, donde las columnas están separadas por un espacio en blanco. La primera columna representa el tiempo y las restantes representan las variables que se hayan almacenado en Simulink y en el orden en que se haya hecho.

```
datos=load('nom.mat');          % struct 1x1
nomV=fieldnames(datos);         % cell 1x1
A=datos.(nomV{1});              % array
B=A';                           % traspuesta de A
save nom.txt B -ASCII;          % guarda en columnas el array B
```

Tabla 5.1: **To File** en array: de **.mat** a **.txt**

   Si la opción elegida en "Save format" es "timeseries" entonces puede hacerse lo que se indica en la Tabla 5.2.

```
datos=load('nom.mat');
nomV=fieldnames(datos);
A=datos.(nomV{1});              % struct
B=[A.Time A.Data];              % array
save nom.txt B -ASCII;          % guarda en columnas el array B
```

Tabla 5.2: **To File** en timeseries: de **.mat** a **.txt**

2. Utilizando el bloque **To Workspace**.

   Supongamos que se ha creado en el entorno de Simulink la variable "var" utilizando el bloque **To Workspace**.

   La variable "var" contiene los datos de las variables como una matriz, en el orden en que se hayan creado. A su vez se habrá creado la variable "tout" que contiene el tiempo. Entonces es posible generar desde el entorno de trabajo de Matlab un fichero del tipo que sea. Por ejemplo para generar un fichero de texto multicolumna de nombre **nom.txt** puede hacerse lo que se indica en la Tabla 5.3.

```
B=[tout var];
save nom.txt B -ASCII;
```

Tabla 5.3: **To Workspace**: "var" a **.txt**

Pueden guardarse solamente algunas columnas de la matriz B obtenida por cualquiera de los métodos anteriores. Por ejemplo, para generar un fichero de texto de dos columnas (la 1 y la 3) puede hacerse lo que se indica en la Tabla 5.4.

```
B13=[B(:,1) B(:,3)];
save nom13.txt B13 -ASCII;
```

Tabla 5.4: **To Workspace**: algunas columnas de "var" a **.txt**

Haciendo en la ventana de comandos de Matlab, **help colon** puede verse cómo puede utilizarse el símbolo **':'**.

Se producirá un error si el número de columna excede la dimensión de la matriz. Para conocer la dimensión puede hacerse,

```
[n,m]=size(B)
```

3. Utilizando el bloque **From File**.

   El bloque **From File** lee un fichero **.mat** que tiene que estar correctamente estructurado.

   Si se dispone de un fichero de datos en formato texto de varias columnas de datos separadas por un espacio en blanco, y donde la primera columna representa el tiempo, puede transformarse en un fichero **.mat** bien estructurado. Por ejemplo, si el fichero de texto **nom.txt** tiene dos columnas, la del tiempo y otra variable, se puede hacer lo que se indica en la Tabla 5.5.

```
htxt=fopen('nom.txt','r');
valor=fscanf(htxt,'%f',[2 inf]);
fclose(htxt);
save('nom.mat','valor');
```

Tabla 5.5: **From File**: de **.txt** a **.mat**

Este fichero **nom.mat** puede ser llamado con el bloque **From File** sin problemas.

4. Utilizando el bloque **From Workspace**.

   El bloque **From Workspace** lee una variable array creada en el espacio de trabajo, donde la primera columna debe ser el tiempo. Por ejemplo, si se tiene un fichero de datos **nom.txt** donde la primera columna es el tiempo, puede crearse la variable "tvar" utilizando la función de Matlab **load** como se indica en la Tabla 5.6.

```
tvar=load('nom.txt');
```

Tabla 5.6: **From Workspace**: de **.txt** a "tvar"

6. Circuito RLC

Consideremos el sistema eléctrico RLC de la Figura 6.1.

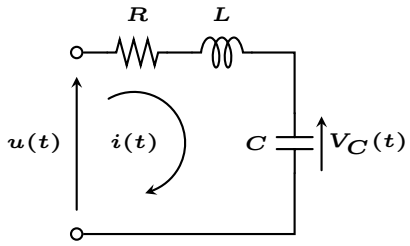

Figura 6.1: Circuito eléctrico RLC

En las siguientes subsecciones se describen diversas formas de representar en Simulink un modelo de simulación del circuito RLC. Solamente el que utiliza la librería **Simscape** se construye como circuito RLC. Los restantes son diversas formas de representar un modelo matemático análogo a este circuito. Con **Simscape** también es posible construir un modelo matemático análogo construyendo un Calculador Analógico con integradores realizados con Amplificadores Operacionales. No obstante esta última solución no la vamos a estudiar en esta Sección.

1. Con **Simscape**. Subsección 6.1.
2. Con **LTI System**, que es el bloque que se encuentra en **Control System Toolbox**. Subsección 6.2.
3. Con **State-Space** que es el bloque de representación en el espacio de estados que se encuentra en **Simulink/Continuous**. Subsección 6.3.
4. Como ecuación diferencial, con bloques integradores que se encuentran en **Simulink/Continuous**. Subsección 6.4.
5. Con **MATLAB Function**, que es un bloque que se encuentra en **Simulink/ User-Defined Functions**. Subsección 6.5.

La simulación de los ejemplos de esta Sección se han realizado en las mismas condiciones. Se ha elegido como "Solver" el método de integración "ode23t (mod.stiff/Trapezoidal)" con una "Relative tolerance" de $1e-3$ y "Stop Time" a 100. Todas las demás opciones se han dejado a "auto" y a las que vienen por defecto. También se ha elegido en "Data Import/Export" el "Format" que se ha puesto a "array" y el "limit data to last" a 100000.

No es necesario abrir el entorno de Simulink para ejecutar un modelo que se haya creado y guardado como fichero con la extensión **.mdl**. Por ejemplo, si se ha creado un fichero **Sistema.mdl** del sistema que se desea simular puede utilizarse la función de Matlab **sim** en el entorno de trabajo:

```
sim('Sistema');
```

Los ficheros de extensión **.mdl** incluyen la configuración de la simulación, por lo que si se desea modificar dicha configuración resulta más cómodo abrir el entorno de Simulink y hacerlo ahí, es decir, que aunque se puede hacer con la función de Matlab **sim** introduciendo los nuevos parámetros de configuración, es más engorroso. Para más detalles puede consultarse la ayuda de Matlab **help sim**.

6.1. Circuito RLC: Simscape

En la Figura 6.2 se muestra la ventana asociada al circuito RLC realizada con la librería **simscape**. El fichero que se ha creado se denomina **RLCSimscape.mdl**.

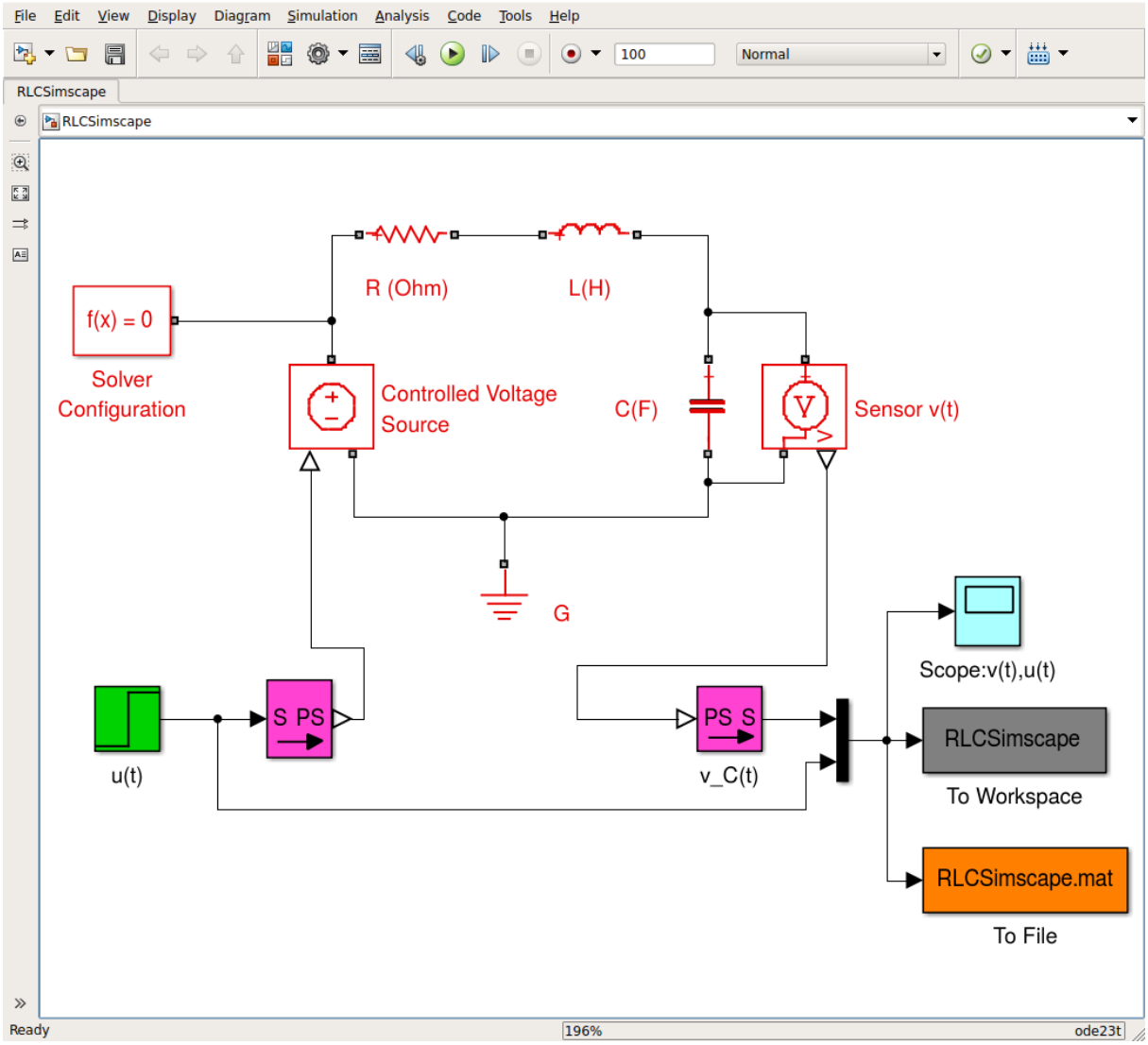

Figura 6.2: RLCSimscape.mdl

Puede observarse que los parámetros R, L y C se han dejado indefinidos por lo que la ejecuación de una simulación dará un error mientras no se especifiquen los valores numéricos concretos en el entorno de trabajo de Matlab. Por lo tanto, antes de ejecutar **RLCSimscape.mdl** deben asignarse valores a los parámetros. Lo que sí debe fijarse son las unidades que serán utilizadas, que en este ejemplo se han fijado a Ohm, H y F respectivamente. También se ha dejado indefinido la amplitud de la señal de entrada escalón U.

Por ejemplo, ejecutando en el espacio de trabajo las instrucciones de la Tabla 6.1, podrá posteriormente ejecutarse sin error la simulación en el entorno de Simulink:

```
R=input('R=');  % p.e. R =10
L=input('L=');  % p.e. L=100
C=input('C=');  % p.e: C= 1
U=input('U=');  % p.e: U= 5
```

Tabla 6.1: Parámetros R,L y C

Pulsando dos veces en el bloque **Scope** pueden verse las curvas de salida (la tensión en el condensador) y de entrada (la señal constante U).

Simultáneamente se han creado dos variables en el espacio de trabajo de Matlab utilizando el bloque **To Workspace**: "tout" y "RLCSimscape". La primera es un array del tiempo de simulación y la segunta es una array de dos columnas de las variables de entrada y salida. Puede verse la curva ejecutando la función de Matlab **plot** en el entorno de trabajo,

```
plot(tout,RLCSimscape)
```

Tabla 6.2: Plot del array de entrada/salida RLCSimscape en función de tout

También se ha creado un fichero de datos "RLCSimscape.mat" utilizando el bloque **To File**.

6.2. Circuito RLC: LTI System

Para la simulación del sistema RLC puede utilizarse cualquiera de las tres funciones de representación de modelos de la Tabla 2.1. En la Figura 6.3 se muestra la ventana asociada al circuito RLC realizada con la librería **Control System Toolbox**. El fichero que se ha creado se denomina **RLCLTISystem_ss.mdl** porque solo se muestra el ejemplo con la función de Matlab **ss**.

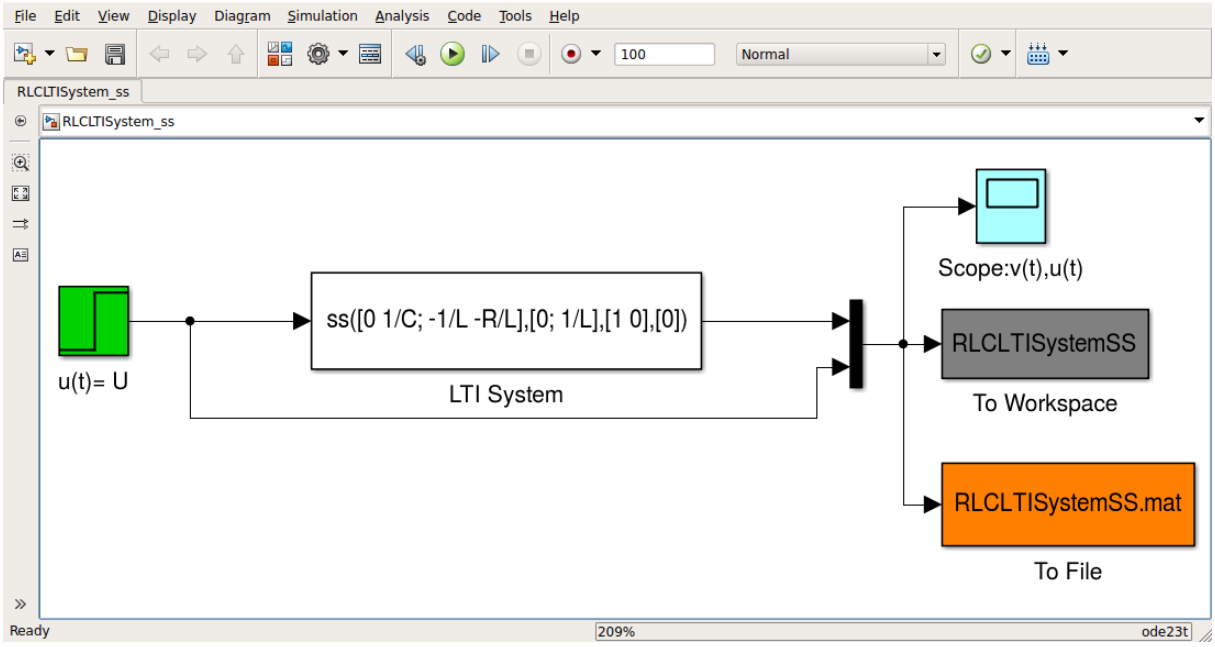

Figura 6.3: RLCLTISystem_ss.mdl

También se han dejado, como en la subsección anterior, los parámetros R, L, C y U indeterminados, por lo que deben introducirse, como antes, en el espacio de trabajo de Matlab. La variable creada se ha llamado ahora "RLCLTISystemSS", y el fichero creado "RLCLTISystemSS.mat".

6.3. Circuito RLC: State-Space

En la Figura 6.4 se muestra la ventana asociada al circuito RLC realizada con el bloque de representación en el espacio de estados que se encuentra en **Simulink/Continuous**. El fichero que se ha creado se denomina **RLCStateSpace.mdl**.

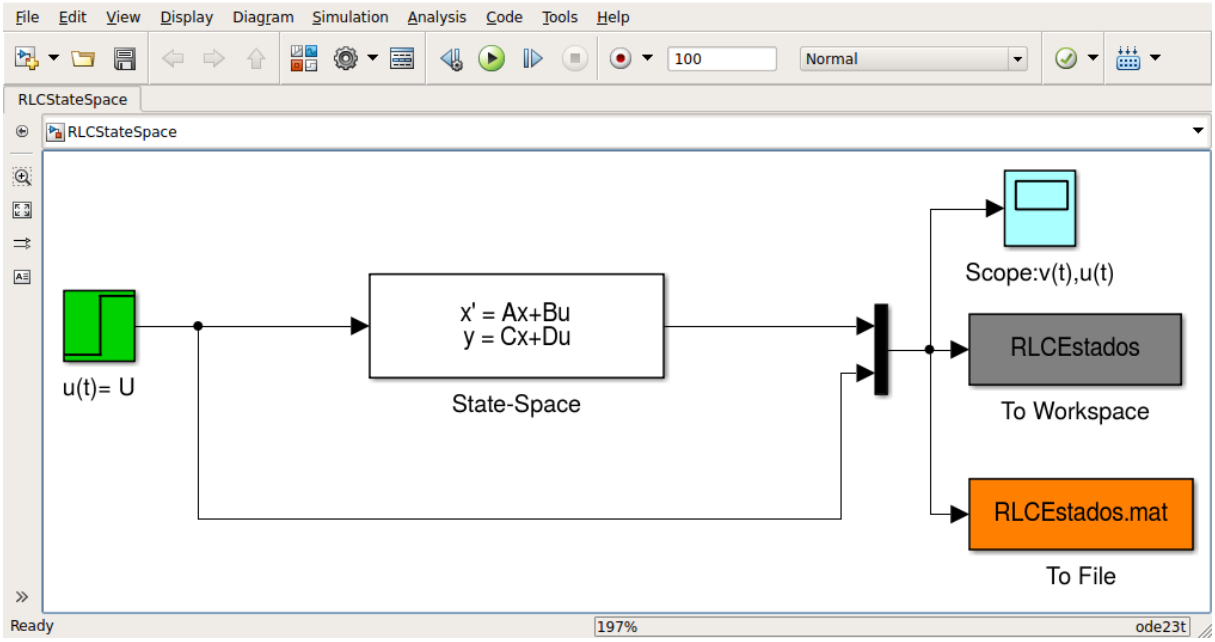

Figura 6.4: RLCStateSpace.mdl

También se han dejado, como en las subsecciones anteriores, los parámetros R, L, C y U indeterminados, por lo que deben introducirse, como antes, en el espacio de trabajo de Matlab. La variable creada se ha llamado ahora "RLCEstados", y el fichero creado "RLCEstados.mat".

6.4. Circuito RLC: Ecuación diferencial

En la Figura 6.5 se muestra la ventana asociada al circuito RLC realizada con bloques elementales que permiten construir e integrar una ecuación diferencial. Estos bloques se encuentran en **Simulink/Continuous**. El fichero que se ha creado se denomina **RLCode.mdl**.

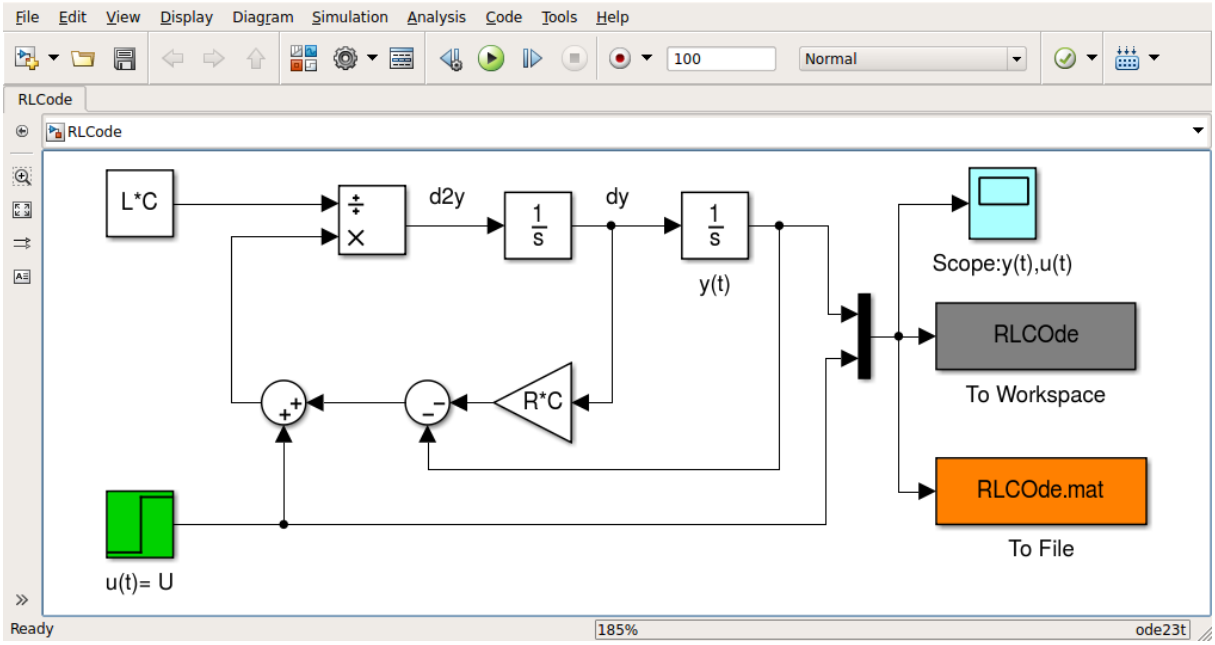

Figura 6.5: RLCode.mdl

También se han dejado, como en las subsecciones anteriores, los parámetros R, L, C y U indeterminados, por lo que deben introducirse, como antes, en el espacio de trabajo de Matlab. La variable creada se ha llamado ahora "RLCOde", y el fichero creado "RLCOde.mat".

6.5. Circuito RLC: MATLAB Function

En esta subsección se crea un programa de simulación utilizando el bloque **MATLAB Function** que se encuentra en **Simulink/User-Defined Functions**. En la Figura 6.6 se muestra la ventana principal del fichero **RLCMatlabFunction.mdl**, cuya diferencia con los ejemplos de las anteriores subsecciones es que se han creado subsistemas para la programación de la entrada (subsistema "Entrada y Parámetros") y de la salida de datos (subsistema "Presentación y Memoria"). En la Figura 6.8 se muestra el subsistema de salida de datos, que comentamos a continuación. En la Figura 6.7 se muestra el subsistema de entrada, que como en la subsecciones anteriores se han dejado los parámetros indefinidos, por lo que deben asignarse en el espacio de trabajo de Matlab.

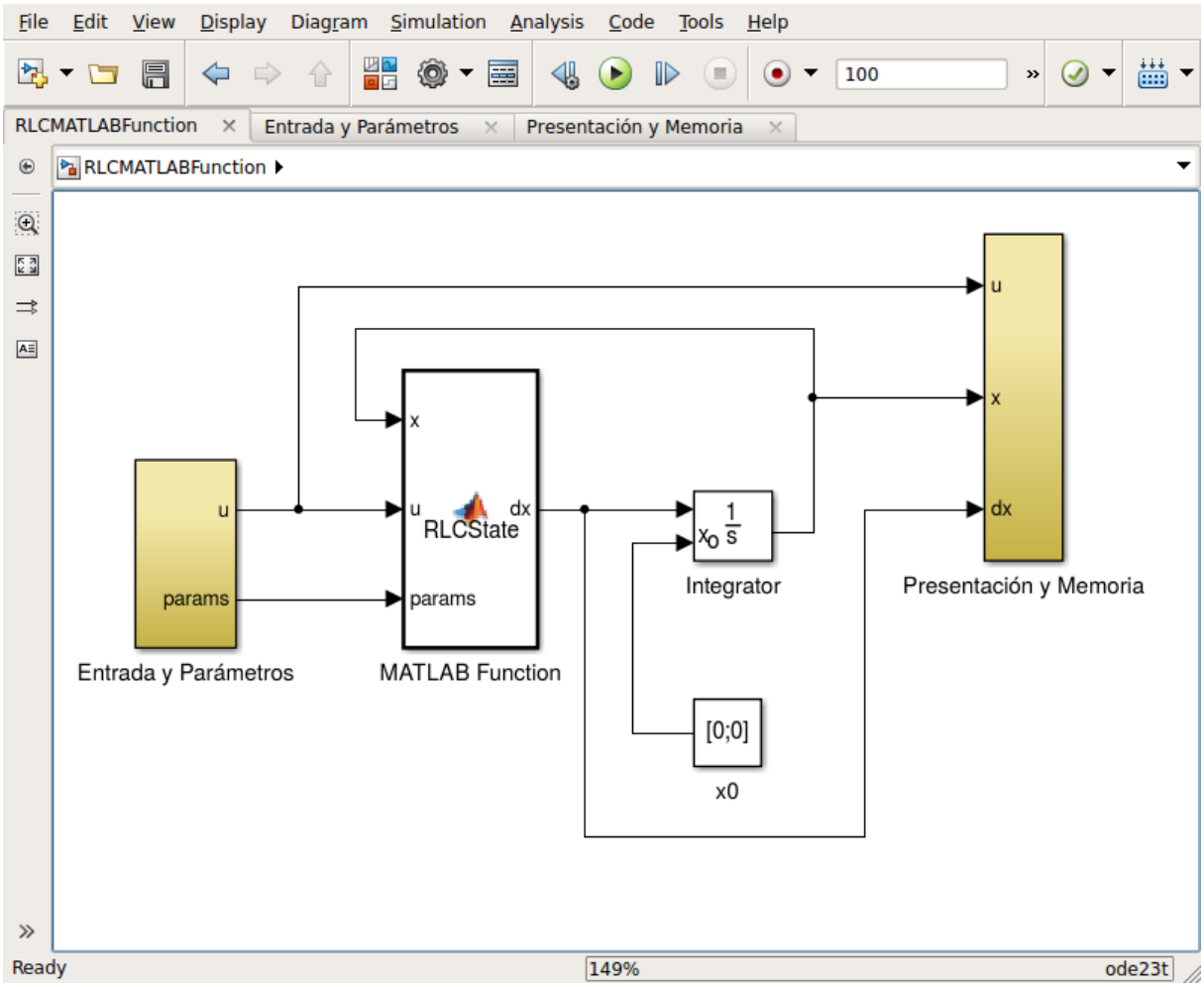

Figura 6.6: RLCMatlabFunction.mdl

Pulsando dos veces el bloque **MATLAB Function** se abre la ventana del editor de funciones, en la que se ha escrito el siguiente código de la función matemática asociada al circuito RLC. En este ejemplo se ha programado la ecuación de estado por lo que la salida debe ser $dx = \dot{x}(t)$. Para obtener el vector de estado $x(t)$ será necesario integrar la salida de esta función, como se muestra en la Figura 6.6. Se ha utilizado un integrador al que se le pueden definir condiciones iniciales.

```
function dx RLCState(x,u,params)
        %#codegen
        % x1=v_C; x2= i_L
        R=params(1);
        L=params(2);
        C=params(3);
        A=[0 1/C;-1/L -R/L];
        B=[0; 1/L]
        dx=A*x+B*u;
end
```

Tabla 6.3: Editor: Matlab Function del circuito RLC

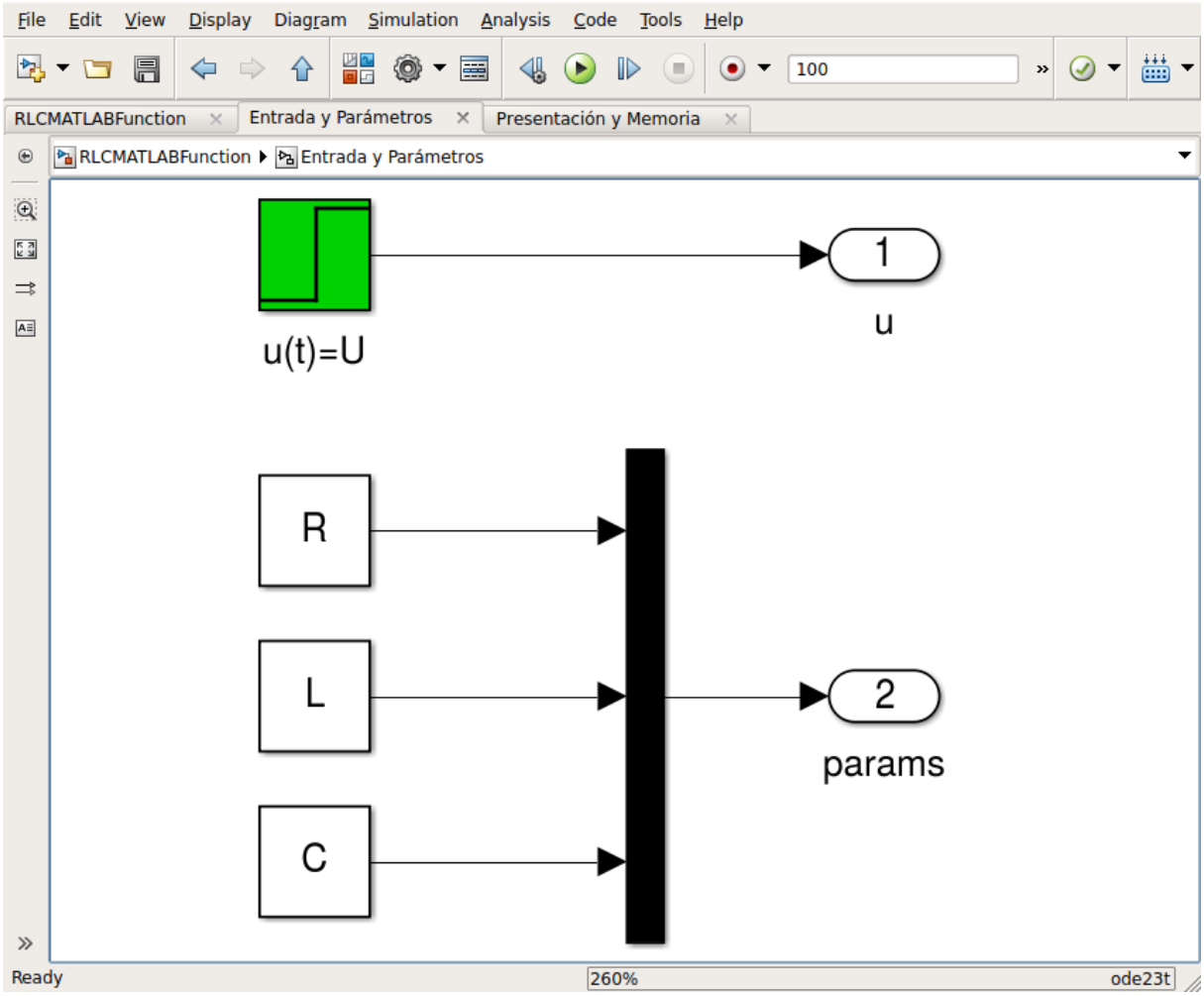

Figura 6.7: RLCMatlabFunction-Entrada

El subsistema de la Figura 6.8 de presentación y de generación de la variable "RLC" y del fichero "RLCFunction.mat" es algo distinto al de las subsecciones anteriores. La diferencia radica en que se guardan más variables, y no solo la salida y la entrada. Lo que se almacena es el vector de estados $x(t)$ y su derivada $\dot{x}(t)$ así como la entrada $u(t)$. Por lo tanto son cinco señales, que con el fin de hacer una gráfica de las curvas que sea legible, se han incluido unas ganancias constantes $(K, Kd1, Kd2)$ que también deben asignarse en el espacio de trabajo, antes de realizar la simulación.

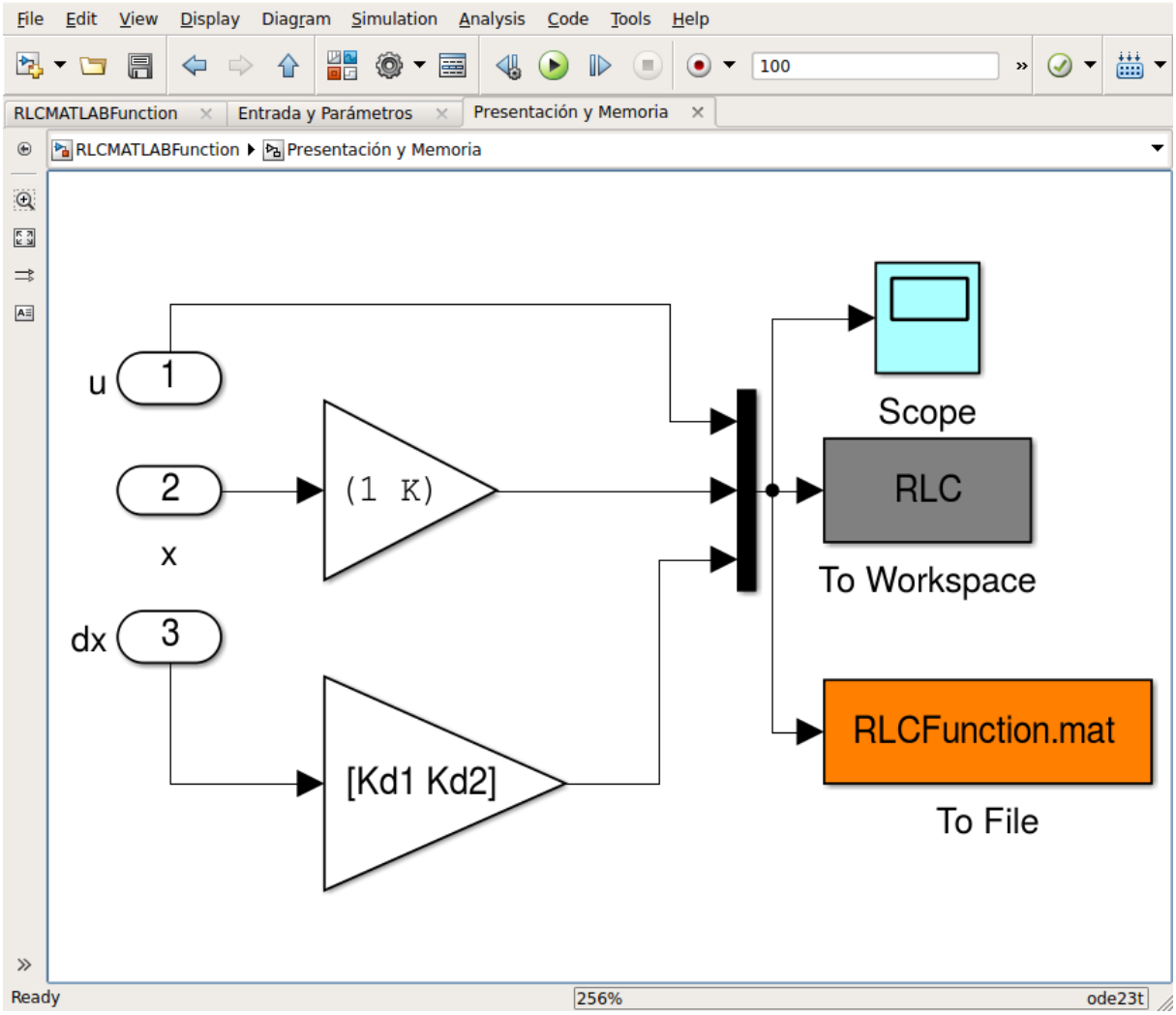

Figura 6.8: RLCMatlabFunction-Presentacion

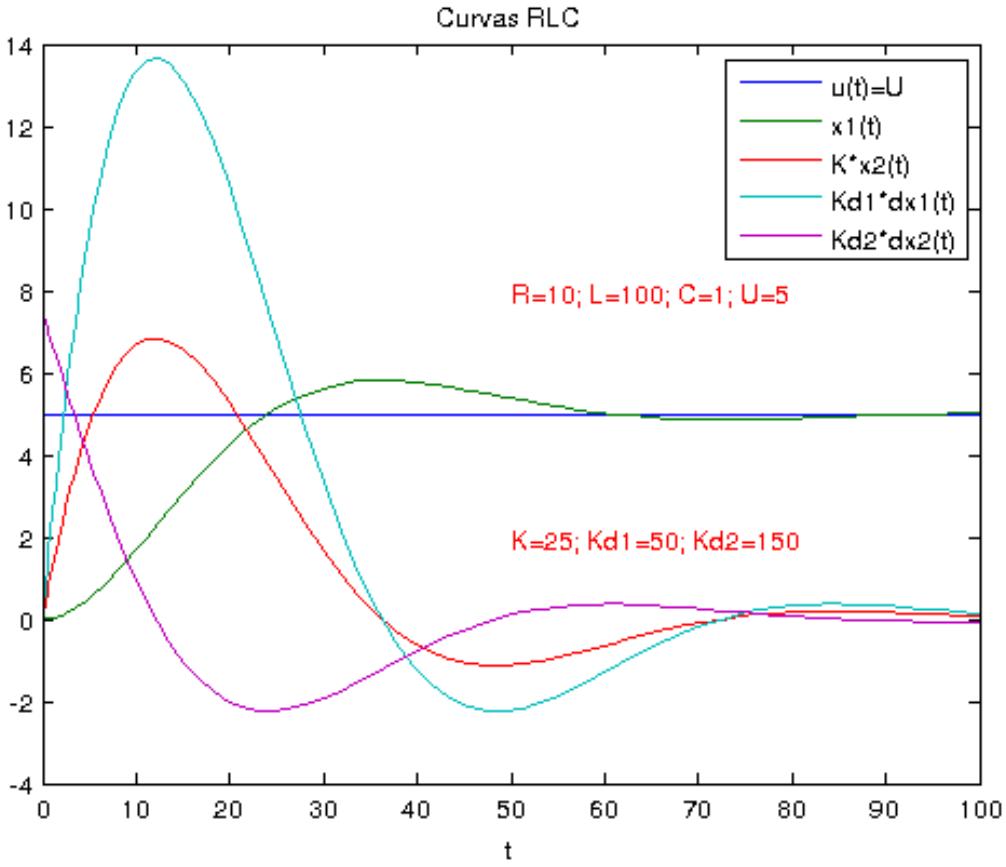

Figura 6.9: RLCMatlabFunction-Grafica

En la Figura 6.9 se muestra la gráfica de las curvas obtenida en el espacio de trabajo ejecutando las instrucciones de la Tabla 6.4.

```
plot(tout,RLC);
title('Curvas RLC');
legend({'u(t)=U','x1(t)','K*x2(t)','Kd1*dx1(t)','Kd2*dx2(t)'});
text(50,2,'K=25; Kd1=50; Kd2=150','Color','r');
text(50,8,'R=10; L=100; C=1; U=5','Color','r');
xlabel('t');
```

Tabla 6.4: Gráfica de RLC

7. Motor DC

Consideremos el motor DC sin carga de la Figura 7.1.

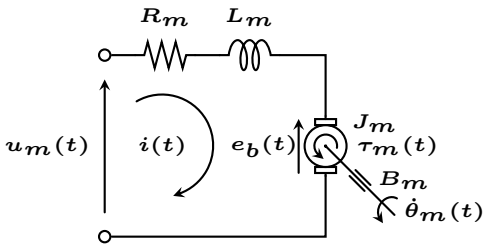

Figura 7.1: MotorDC

En las siguientes subsecciones se describen diversas formas de representar en Simulink un modelo de simulación del Motor DC.

1. Con **Simscape**. Subsección 7.1.
2. Con **MATLAB Function**, que es un bloque que se encuentra en **Simulink/User-Defined Functions**. Subsección 7.2.

En la Tabla 7.1 se muestra el código del fichero **MotorDCParam.m** que incluye los parámetros que serán utilizados en las simulaciones de las siguientes subsecciones.

```
%parameters
% Electrical Torque, by equivalent circuit pasrameters
Rm=5.3; %Ohm
Lm=5.8e-4; % F
kb=2.2e-2; % V/(rad/s)
% Mechanical
Jm=1.4e-6; % kg*m^2
Bm=2.01e-6; % N*m/(rad/s))
% entrada
U=5; % serán V
%%%
tmax=0.2;
%%% Gananciaqs para presentación
K1=1/(2*pi);
K2=1/25;
K3=10;
Kd1=K2;
Kd2=1/1000;
Kd3=1/1000;
```

Tabla 7.1: MotorDCParam.m

La simulación de los ejemplos de esta Sección se han realizado en las mismas condiciones. Abriendo la pestaña "Simulations" y dentro de ella "Model Configuration Parameters", se ha elegido como "Solver" el método de integración "ode45 (Dormand-Prince)" con una "Relative tolerance" de $1e-3$ y "Stop Time" igual a $tmax$, variable a la que debe darse algún valor antes de realizar una simulación. Esta variable se ha incluido en el script **MotorDCParam.m**. También se ha modificado en "Solver Options" el "Type" que se ha puesto a "Variable-step" y el "Max step size" que se ha puesto a $1e-4$. Todas las demás opciones se han dejado a "auto" y a las que vienen por defecto. También se ha elegido en "Data Import/Export" el "Format" que se ha puesto a "array" y el "limit data to last" a 100000.

7.1. Motor DC: Simscape

Las instrucciones de la Tabla 7.2 se corresponden con el fichero **MotorDCSimscapeM.m**. Al ejecutar este script se realiza una simulación del Motor DC creado en Simulink utilizando la librería **Simscape**, que hemos llamado **MotorDCSimscapeSensores.mdl**, y cuyas curvas resultantes se muestran en las Figuras 7.2 y 7.3.

```
clc
clear all
delete(findall(0,'Type','figure')) % cierra todas las figuras
MotorDCParam
sim('MotorDCSimscapeSensores');
plot(tout,Mot)
title('CurvaMotDC-Simscape')
xlabel('t')
legend({'K1*theta(t)','K2*w(t)','K3*i(t)','u(t)'})
%%%%%%%%%%%%%%%%%%%%%%%%%
figure
plot(tout,Mot(:,3))
xlim([0 5e-3])
xlabel('t')
legend('K3*i(t)')
title('CurvaMotDC-SimscapeB')
%%%%%%%%%%%%%%%%%%%%%%%%%
```

Tabla 7.2: MotorDCSimscapeM.m

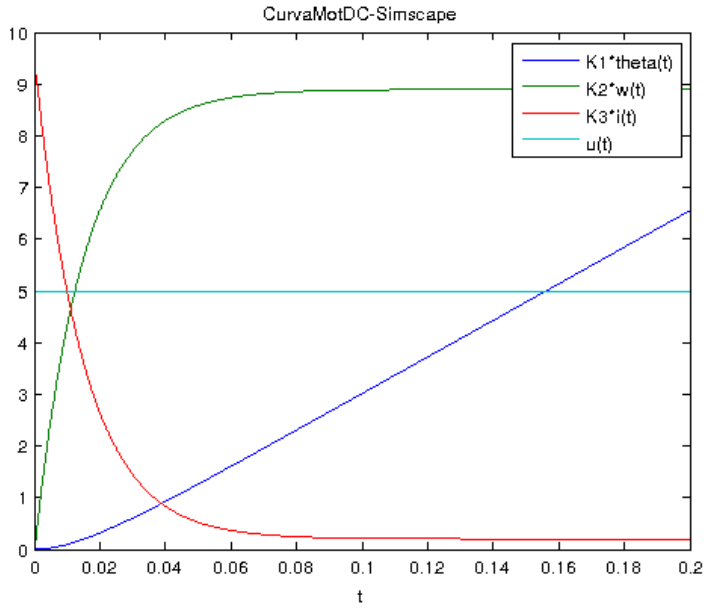

Figura 7.2: MotDCSimscape-Grafica

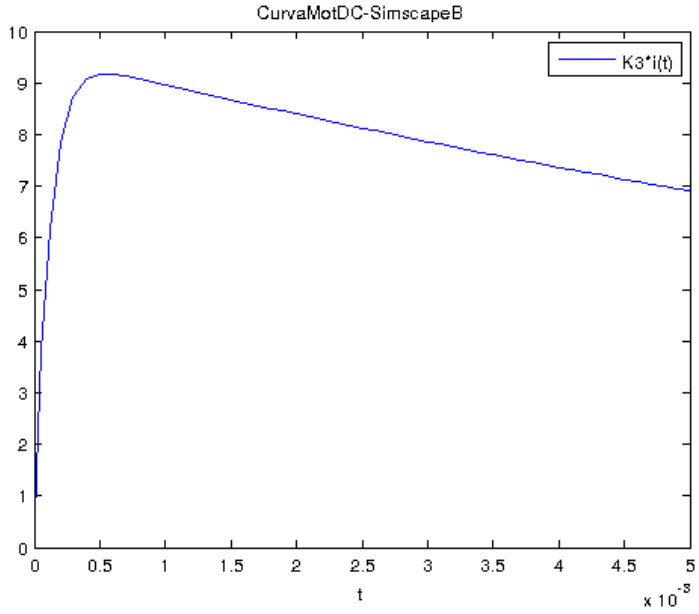

Figura 7.3: MotDCSimscapeB-Grafica

En la Figura 7.4 se muestra la ventana principal de **MotorDCSimscapeSensores.mdl**, y en la Figura 7.5 la ventana del subsistema de presentación. Para la simulación se ha escogido el bloque **DC Motor** que se encuentra en la librería **SimElectronics/Actuators & Drivers/Rotational Actuators**. Esto mismo podía haberse hecho utilizando el bloque básico **Rotational Electromechanical Converter**, como se hace en el ejemplo que ofrece Matlab, y que puede verse ejecutando en el espacio de trabajo **ssc_dcmotor**.

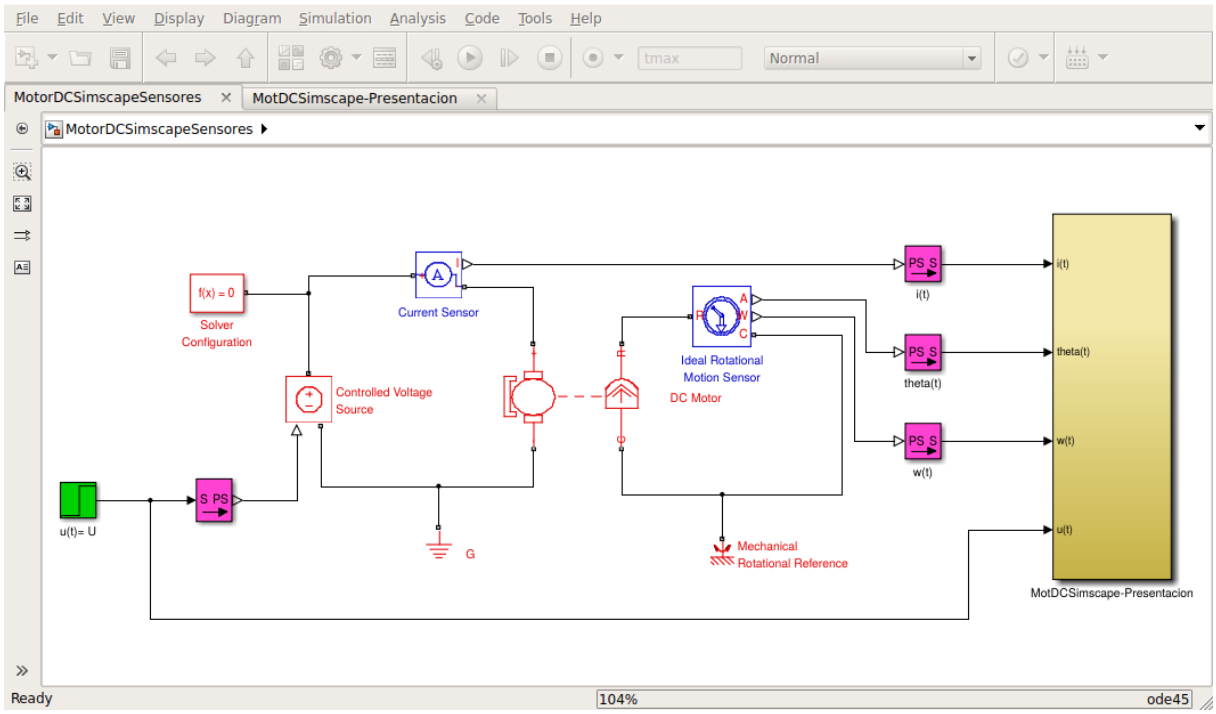

Figura 7.4: MotorDCSimscapeSensores.mdl

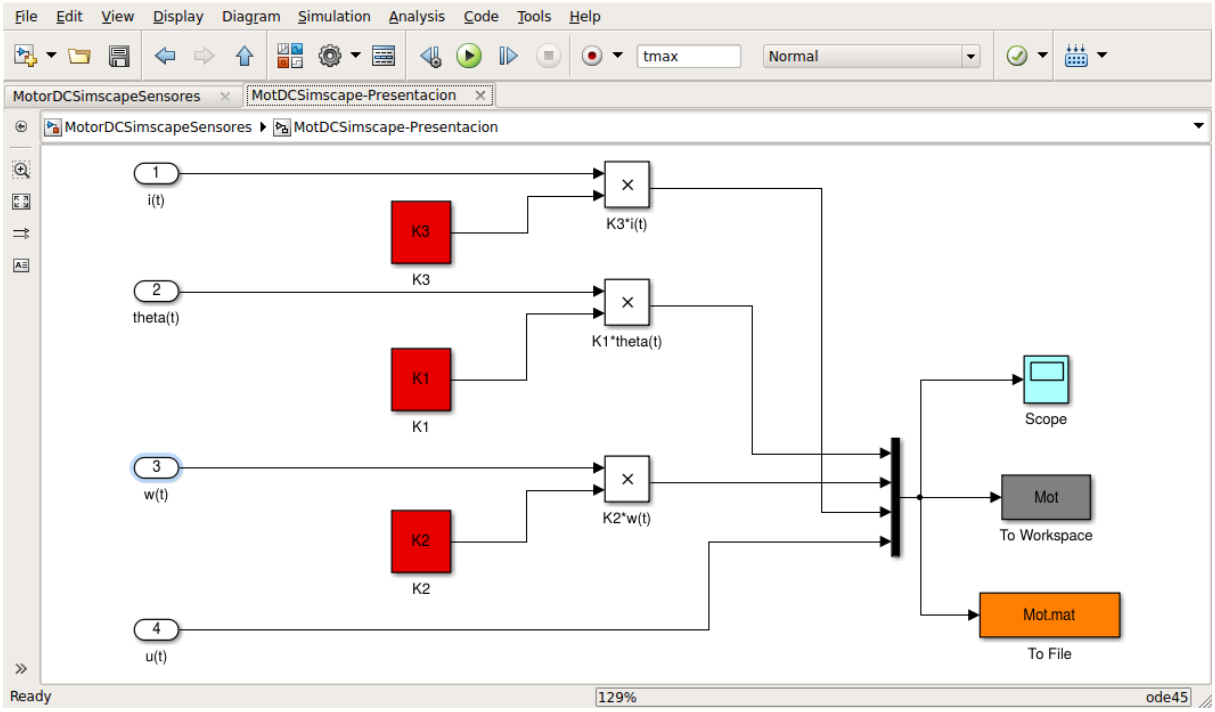

Figura 7.5: MotorDCSimscapeSensores-Presentacion (en MotorDCSimscapeSensores.mdl)

7.2. Motor DC: MATLAB Function

Las instrucciones de la Tabla 7.3 se corresponden con el fichero **MotorDCMatlabFunctionM.m**. Al ejecutar este script se realiza una simulación del Motor DC creado en Simulink utilizando el bloque **MATLAB Function**, que hemos llamado **MotorDCMatlabFunction.mdl**, y cuyas curvas resultantes se muestran en las Figuras 7.6 y 7.7.

```
clc
clear all
delete(findall(0,'Type','figure')) % cierra todas las figuras
MotorDCParam
sim('MotorDCMatlabFunction');
figure
plot(tout,MotEstados)
title('CurvaMotDC-MatlabFunction')
xlabel('t')
legend({'u(t)','K1*theta(t)','K2*w(t)','K3*i(t)','Kd1*xd1(t)','Kd2*xd2(t)','Kd3*xd3(t)'})
%%%%%%%%%%%%%%%%%%%%%%%%%
figure
plot(tout,[MotEstados(:,4) MotEstados(:,6) MotEstados(:,7)])
xlim([0 5e-3])
xlabel('t')
legend({'K3*i(t)','Kd2*alpha(t)','Kd3*di(t)/dt'})
title('CurvaMotDC-MatlabFunctionB')
%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Tabla 7.3: MotorDCMatlabFunctionM.m

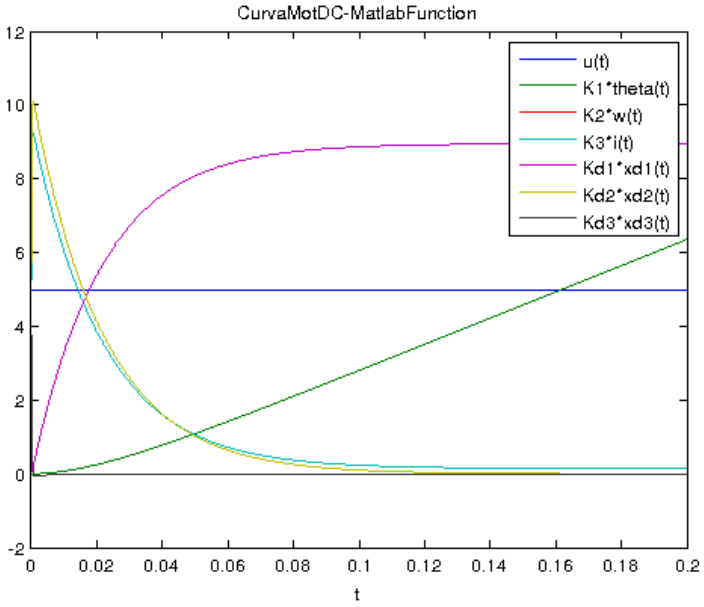

Figura 7.6: MotDCMatlabFunction-Grafica

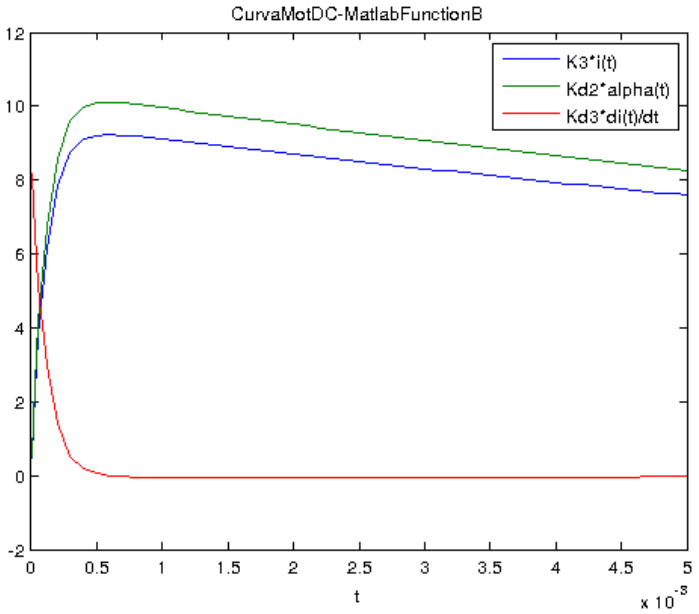

Figura 7.7: MotDCMatlabFunctionB-Grafica

En la Figura 7.8 se muestra la ventana principal de **MotorDCMatlabFunction.mdl**, y en la Figuras 7.9 y 7.10 las ventanas de los subsistemas de entrada y presentación. En la Tabla 7.4 se indica la función programada en el bloque **MATLAB Function**.

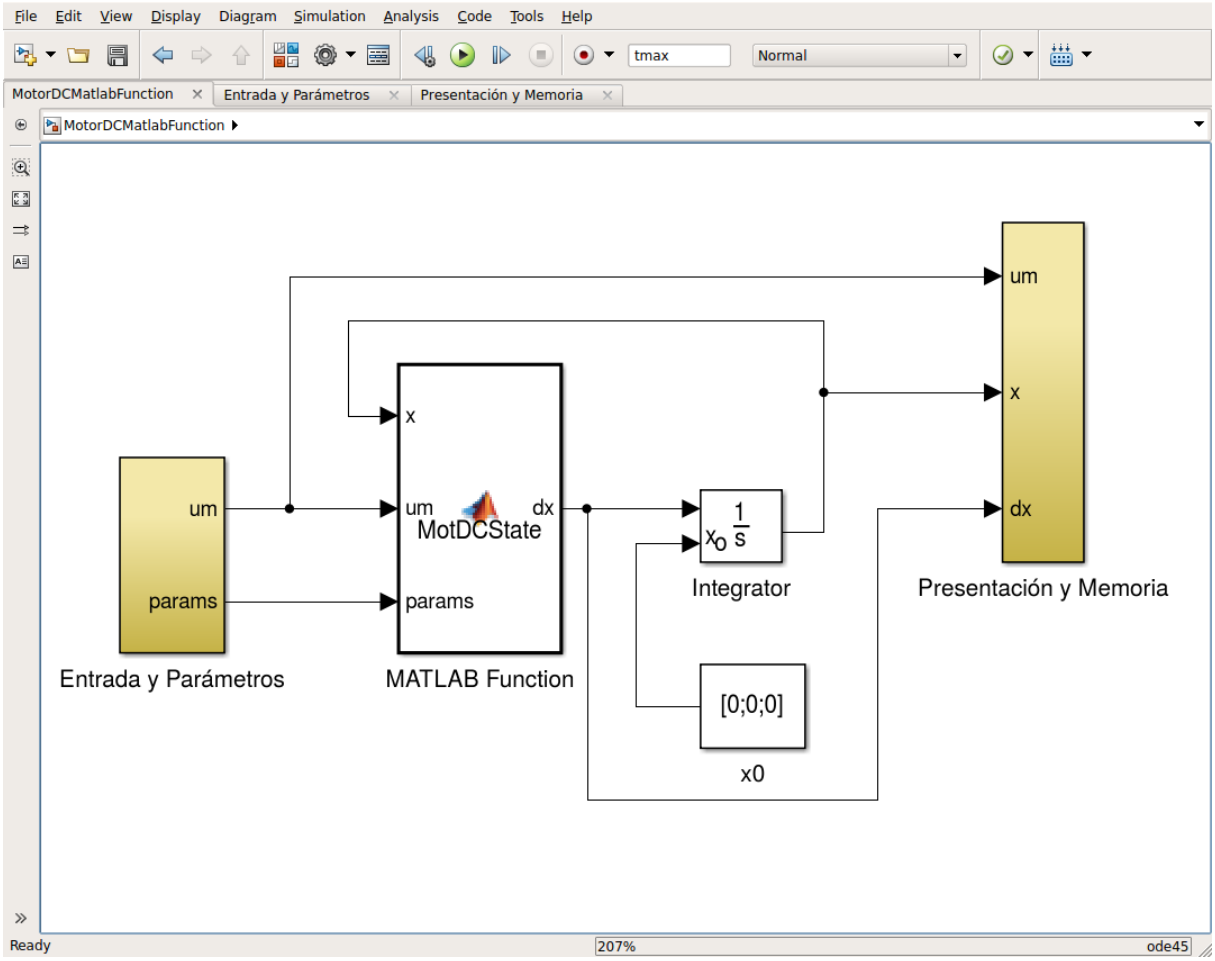

Figura 7.8: MotorDCMatlabFunction.mdl

```
function dx  = MotDCState(x,um,params)
%#codegen
% x1=theta(t); x2=w(t); x3=i(t)
Rm=params(1);
Lm=params(2);
Bm=params(3);
Jm=params(4);
kb=params(5);
km=kb;
A=[0 1 0;0 -Bm/Jm km/Jm;0 -kb/Lm -Rm/Lm];
B=[0;0; 1/Lm];
dx = A*x +B*um;
end
```

Tabla 7.4: Editor: Matlab Function del motor DC

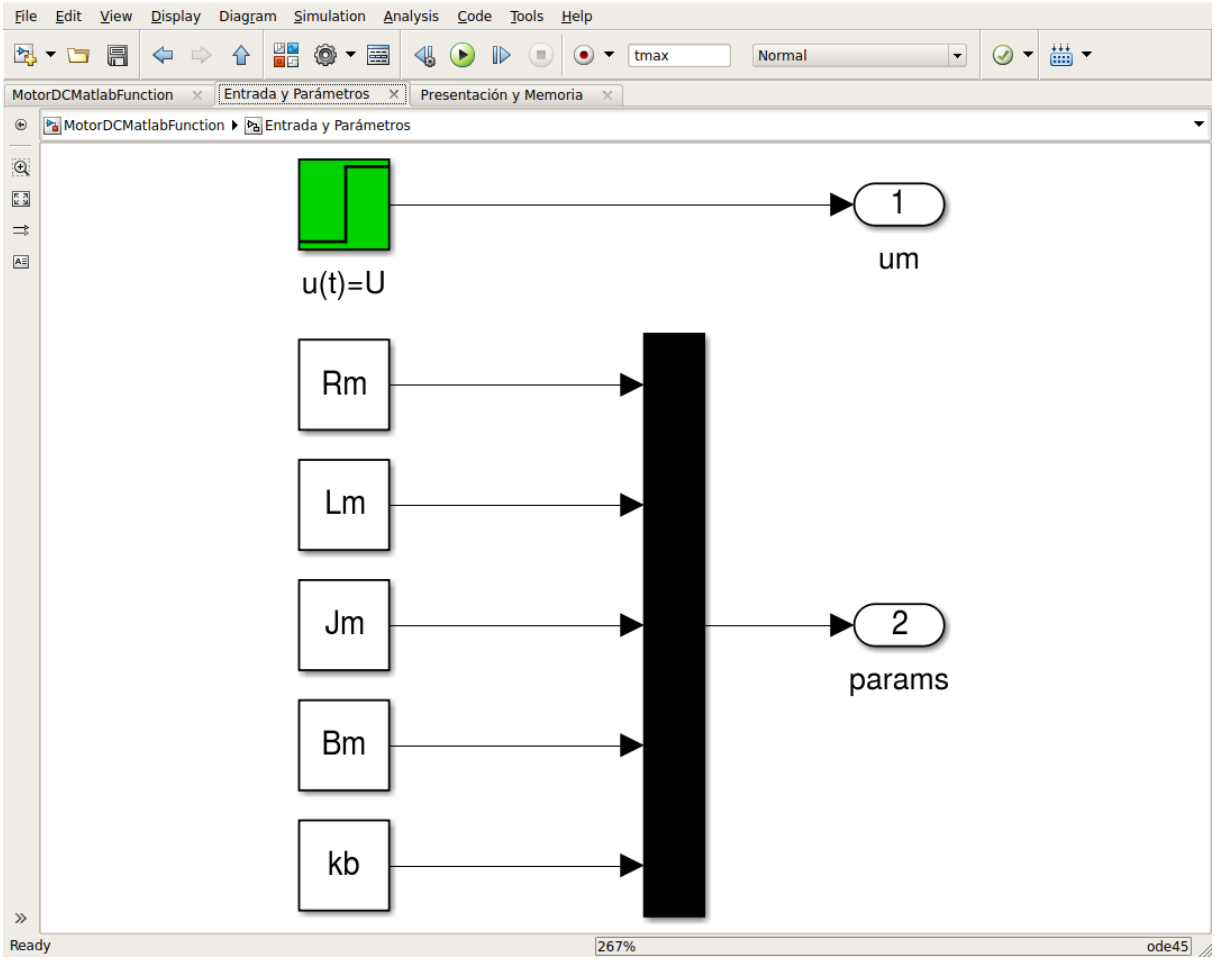

Figura 7.9: MotorDCMatlabFunction-Entrada (en MotorDCMatlabFunction.mdl)

El subsistema de la Figura 7.10 de presentación y de generación de la variable "MotEstados" y del fichero "MotFunction.mat" es algo distinto al de las subsecciones anteriores. La diferencia radica en que se guardan más variables, y no solo la salida y la entrada. Lo que se almacena es el vector de estados $x(t)$ y su derivada $\dot{x}(t)$ así como la entrada $u(t)$. Por lo tanto son siete señales, que con el fin de hacer una gráfica de las curvas que sea legible, se han incluido unas ganancias constantes $(K1, K2, K3, , Kd1, Kd2, Kd3)$ que también deben asignarse en el espacio de trabajo, antes de realizar la simulación.

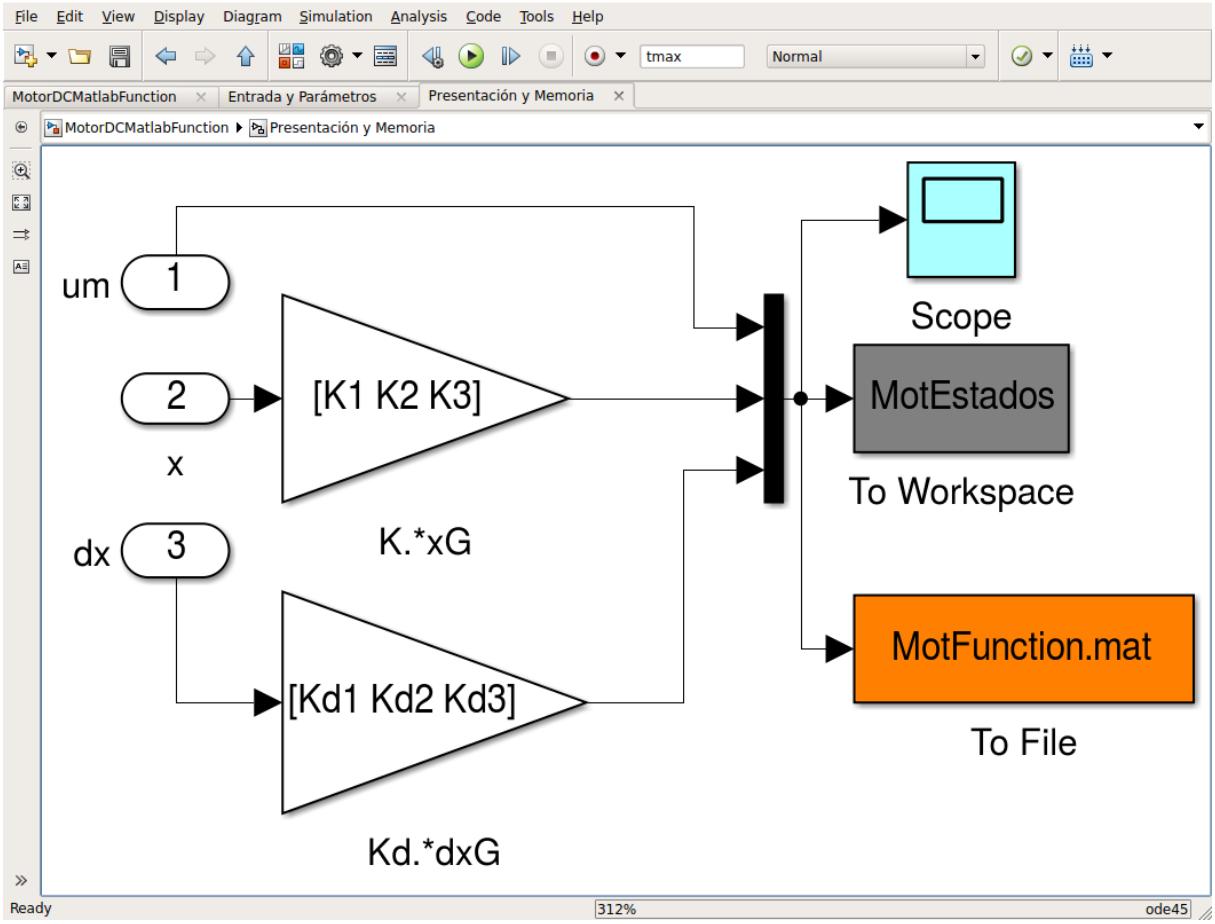

Figura 7.10: MotorDCMatlabFunction-Presentacion (en MotorDCMatlabFunction.mdl)

8. Señales de referencia polinómicas

Las señales de referencia monómicas son funciones causales, es decir, con valor nulo para $t \leq 0$. Una combinación lineal de ellas permite obtener cualquier función causal polinómica.

En esta Sección vamos a ver cómo generar con Simulink una función impulso que simula aproximadamente una función delta de Dirac $\delta(t)$, y cómo generar una función trapezoidal y su integral.

Para ello utilizaremos los bloques **step** y **ramp** con retardos temporales.

1. Función delta de Dirac. Es posible construir una delta de Dirac aproximada que funcione bien en las simulaciones. Deberá ser un pulso de anchura ΔT y de amplitud $\frac{1}{\Delta T}$, donde ΔT debe ser un valor pequeño. Subsección 8.1.
2. Función trapezoidal. Subsección 8.2.

8.1. Delta de Dirac

En la Figura 8.1 se muestra la ventana principal de **RefDirac.mdl**. Se simula una función delta de Dirac aproximada, basada en la idea de generar un pulso de anchura muy pequeña y amplitud muy grande, puesto que la derivada del escalón unidad es la delta de Dirac,

$$\delta(t) = \lim_{\Delta T \to 0} \frac{r_0(t) - r_0(t - \Delta T)}{\Delta T} \tag{8.1}$$

donde $r_0(t)$ es la función escalón unidad.

El valor escogido ha sido $\Delta T = e^{-4}$, aunque este valor depende del que se escoja como paso de integración. Se ha incluido una constante de retardo "ret" para poder generar el pulso en cualquier instante de tiempo, es decir, deltas de Dirac retardadas en el tiempo, $\delta(t - ret)$.

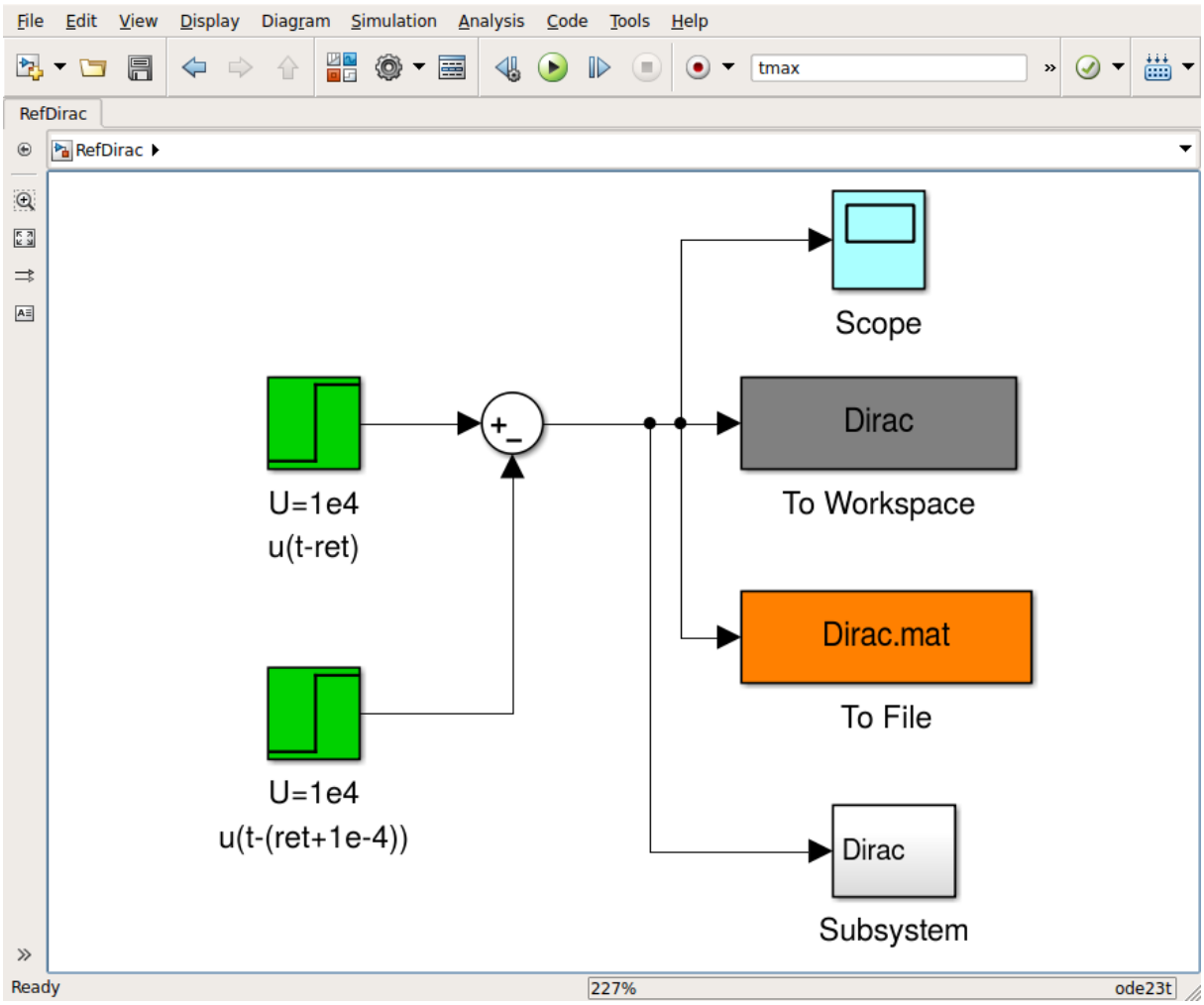

Figura 8.1: RefDirac.mdl

En la Figura 8.2 se muestra la ventana de un subsistema que sirve como ejemplo de aplicación. Se trata de un sistema de primer orden y orden relativo la unidad expresado en la forma de una función de transferencia, cuya respuesta al impulso se muestra en la Figura 8.3.

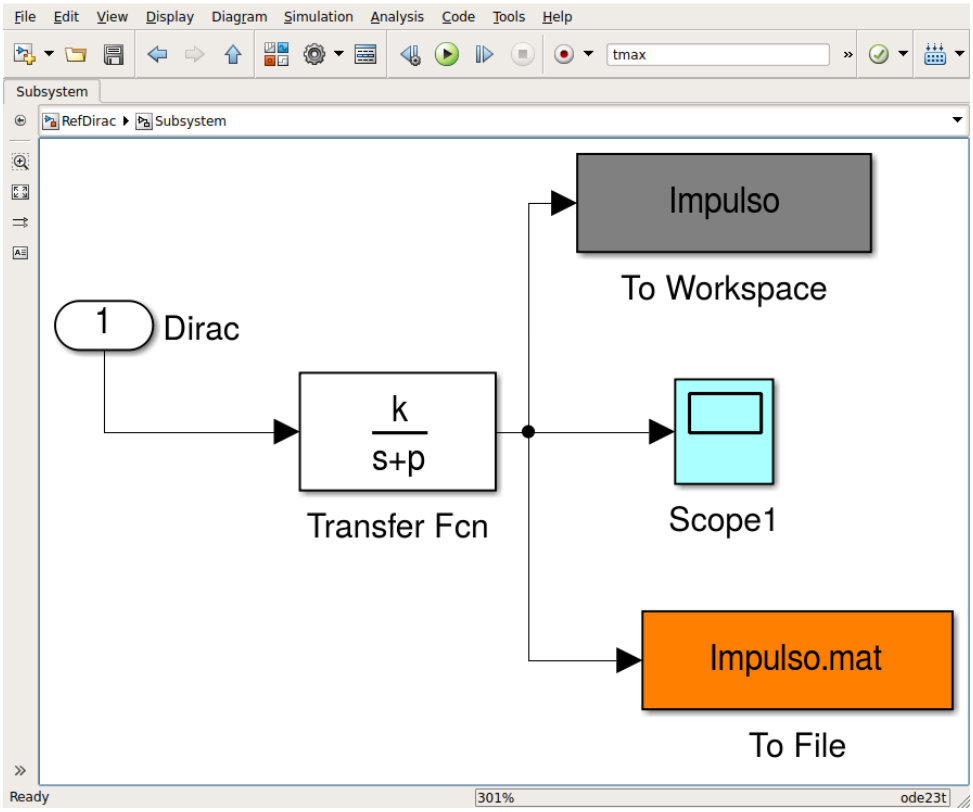

Figura 8.2: RefDirac-Subsistema (en RefDirac.mdl)

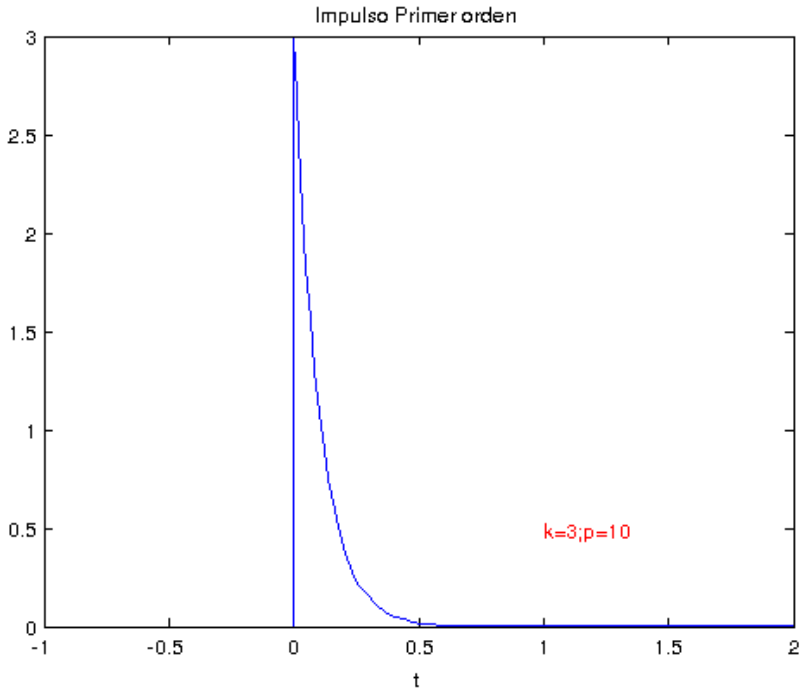

Figura 8.3: ImpulsoPrimerOrden

8.2. Trapecio y su integral

En la Figura 8.4 se muestra la ventana principal del programa **RefTrapecio.mdl** que realiza una señal de tipo tracezoidal utilizando señales de tipo rampa con retardos temporales, además de su integral. La salida de la integral se ha multiplicado por una ganancia con el fin de hacer una presentación gráfica legible. Se ha tenido también en cuenta que la pendiente de bajada del trapecio puede ser cero. En la Tabla 8.1 se muestra el fichero de parámetros **RefTrapecioParam.m** que vamos a utilizar en la simulación.

Una señal trapezoidal causal puede representare en la forma

$$y(t) = \begin{cases} m1\,(r_1(t-t1) - r_1(t-t2)) - m2\,(r_1(t-t2) - r_1(t-t3)) & ,m2 \neq 0 \\ m1\,(r_1(t-t1) - r_1(t-t2)) - m1t1r_0(t-t2) & ,m2 = 0 \end{cases} \tag{8.2}$$

donde $r_1(t)$ es la función rampa de pendiente unidad, $r_0(t)$ es la función escalón unidad y

$$t3 = \begin{cases} t2 + t1\dfrac{m1}{m2} & ,m2 \neq 0 \\ t2 & ,m2 = 0 \end{cases} \tag{8.3}$$

Para la programación de la señal trapezoidal se han fijado los parámetros $\{t1, t2, m1, m2\}$ y después se ha calculado $t3$ utilizando la ecuación dada por 8.3.

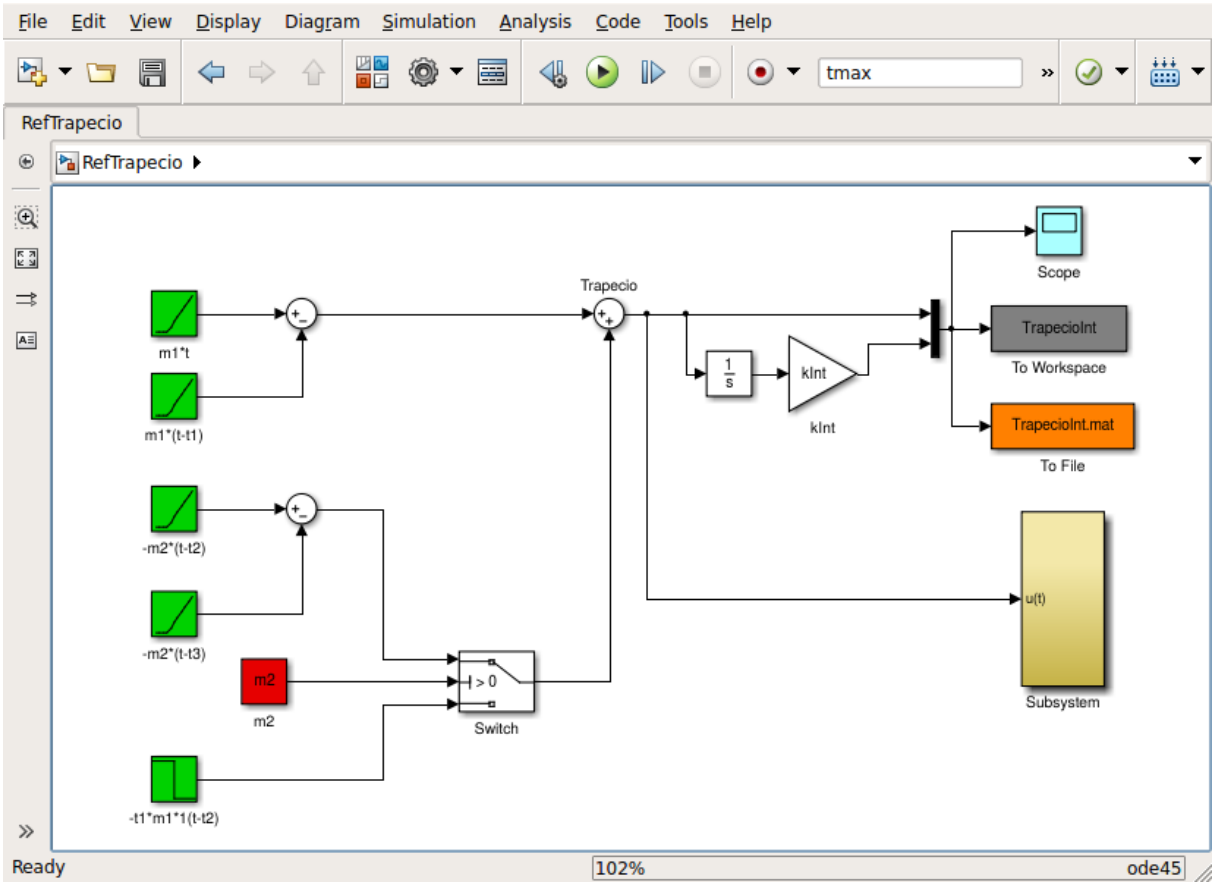

Figura 8.4: RefTrapecio.mdl

```
t1=50;
t2=200;
m1=0.15;
m2=7.5e-2;
if m2==0 t3=t2;
else t3=t2+t1*m1/m2;
end
%%%%%
tmax=1.2*t3;
%%%%%
kInt=2/(-t1+t2+t3);
```

Tabla 8.1: RefTrapecioParam.m

En la Tabla 8.2 se muestra el código del script **RefTrapecioM.m** con el que se realizará una simulación, obteniendo la curva de la Figura 8.5. En este script aparecen los valores Km, p porque el subsistema que se ha programado es un sistema de primer orden, aunque en esta subsección no lo vamos a utilizar.

```
clc
clear all
delete(findall(0,'Type','figure')) % cierra todas las figuras
RefTrapecioParam
%%%%%%%%%%%%%%%%%%%%%%%%
Km=100;
p=66;
%%%%%%%%%%%%%%%%%%%%%%%%
sim('RefTrapecio');
figure
plot(tout,TrapecioInt)
title('CurvaTrapecio')
xlabel('t')
legend({'y(t)','kInt*Int(t)'},'Location','Best')
text(t2/2,2,'kInt=2/(-t1+t2+t3)=0.0044','Color','r')
text(t2/2,1.5,'t1=50; t2=200; m1=0.15; m2=0.075; t3=t2+t1*m1/m2=300','Color','r')
%%%%%%%%%%%%%%%%%%%%%%%%
```

Tabla 8.2: RefTrapecioM.m

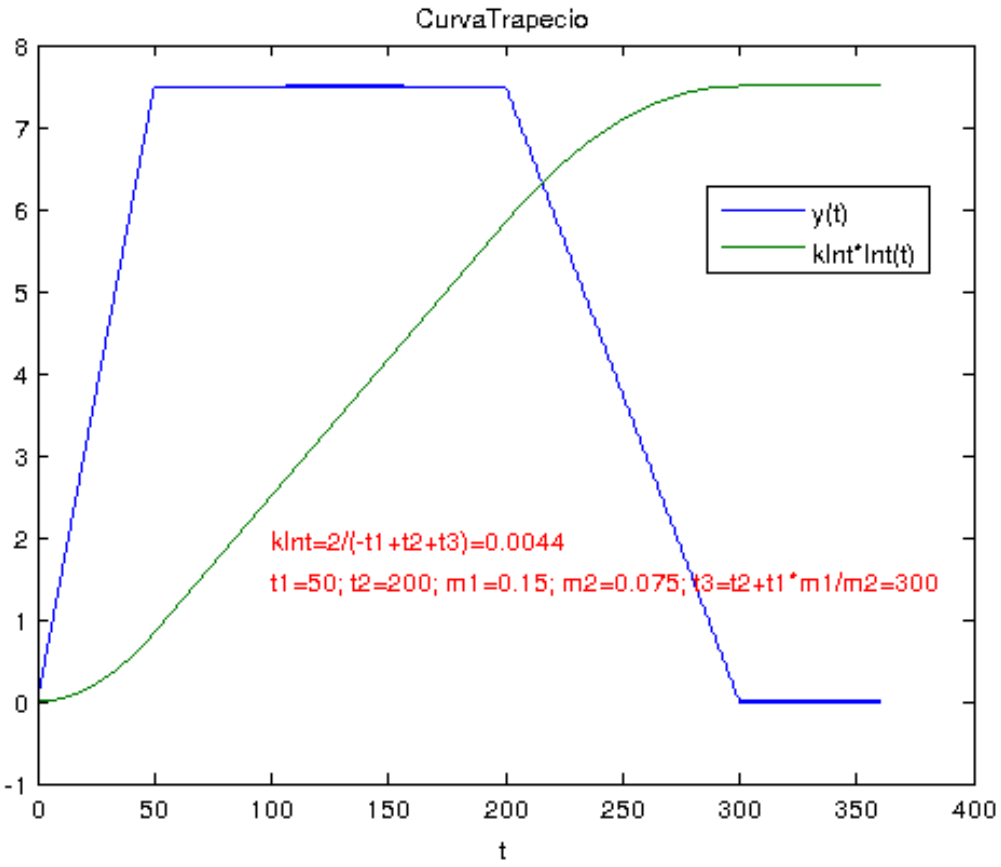

Figura 8.5: CurvaRefTrapecio

8.3. Error de modelado de un sistema de primer orden

Supongamos que se quiere obtener el modelo matemático de un sistema real desconocido realizando un experimento de laboratorio.

Supongamos que el sistema real obedece a la función de transferencia

$$G(s) = \frac{K_m}{s+p} \tag{8.4}$$

donde K_m, p son parámetros desconocidos.

Para obtener el modelo matemático se ha realizado un experimento en el cual se ha inyectado al sistema una entrada trapezoidal como la del programa descrito en la Subsección 8.2, en la que $t3 = t2$, es decir $m2 = 0$, una caída escalón al final del trapecio desde un valor conocido U. Si se supone que el sistema real se comporta como un sistema de primer orden, la respuesta a la parte final de este trapecio debe seguir una ley exponencial de la forma

$$y(t) = \frac{UK_m}{p} e^{-p(t-t3)}, \quad t \geq t3 \tag{8.5}$$

Esto es cierto si el sistema ha alcanzado el régimen permanente en las proximidades de $t = t3$, cuyo valor debe ser la ganancia a bajas frecuencias multiplicado por el valor del escalón $U = m1 * t1$, es decir, $K_0 = \frac{UK_m}{p}$. Por esta razón el tramo del trapecio $[t2, t3]$ debe ser suficientemente largo.

Supongamos también que disponemos de los datos numéricos del experimento real guardados en un fichero de texto "DatosSistema.txt", y que se ha utilizado alguna técnica de modelado que ha permitido estimar los valores de K_m y p.

Lo que vamos a hacer a continuación es obtener el error de modelado suponiendo que se ha estimado correctamente la ganancia a bajas frecuencias K_0, pero se ha estimado un valor del polo

$p' = p - \Delta p$. Como consecuencia la función de transferencia modelada tiene la forma,

$$G'(s) = \frac{K'_m}{s + p'} \tag{8.6}$$

donde se cumple, por hipótesis de este ejemplo, que

$$K_0 = \frac{K'_m}{p'} \tag{8.7}$$

En la Figura 8.6 se muestra la ventana del subsistema programado del fichero **RefTrapecioModelado.mdl**, cuya ventana principal es idéntica a la de la Figura 8.4. Lo que se ha programado es la ecuación 8.5 y se ha comparado con la variable "DatosArray" obtenida al cargar el fichero de texto "DatosSistema.txt". Se ha incluido también, con el fin de que sirva de ejemplo, la incorporación del fichero "DatosSistema.mat" que es la versión estructurada del fichero de datos en formato texto. En la Sección 5 se explica como convertir ficheros de un formato a otro.

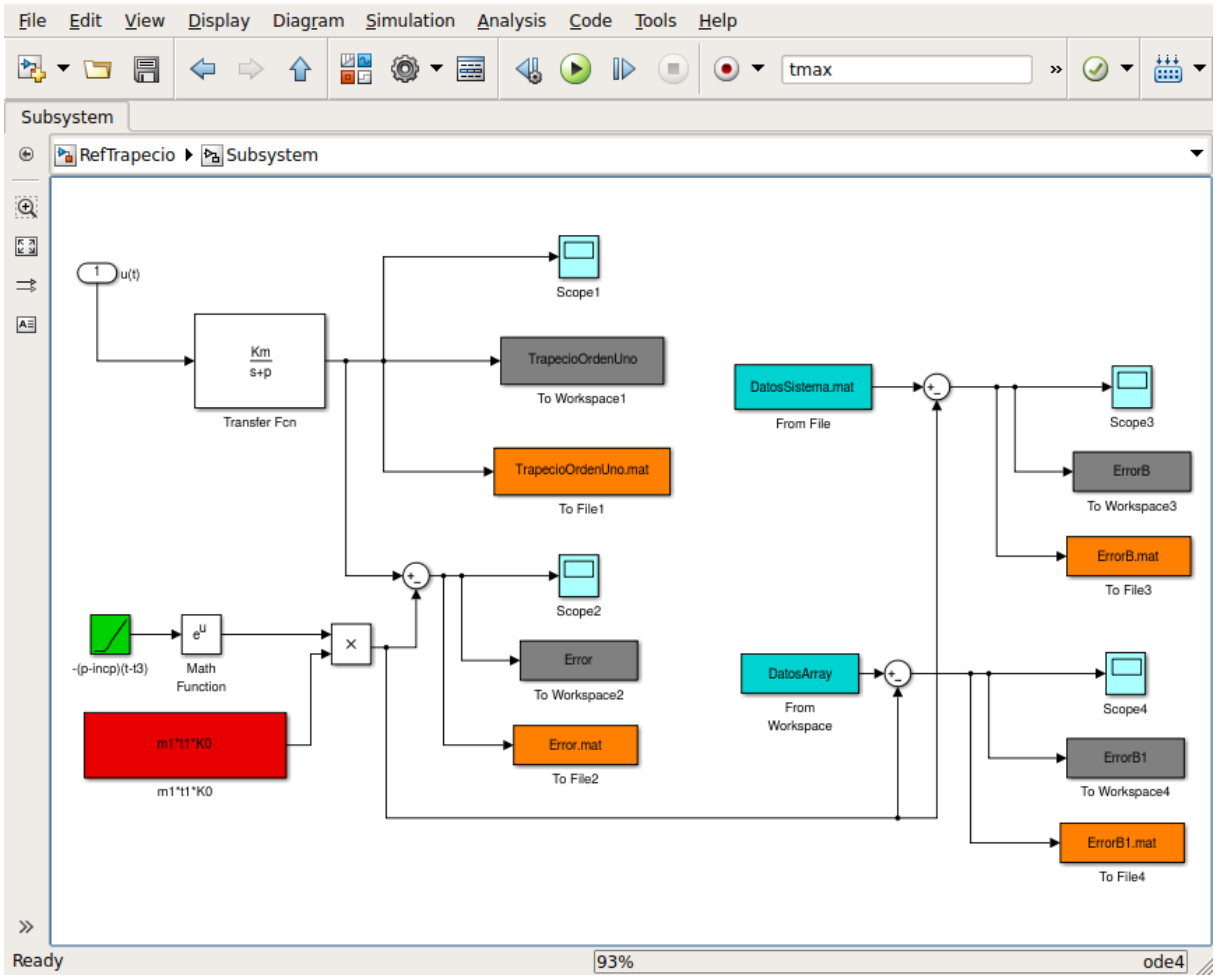

Figura 8.6: RefTrapecioModelado-Subsistema (en RefTrapecioModelado.mdl)

Se han ejecutado las instrucciones de la Tabla 8.3 obteniendo las Figuras 8.7 y 8.8. La Figura 8.8 representa el error de modelado cuando $\Delta p = incp = 2$ y la ganancia a bajas frecuencias se ha estimado sin error.

La simulación se ha realizado con el integrador "ode4 (Runge-Kutta)" que es de tipo "Fixed-step", al que se le ha fijado un "fixed-step size (fundamental sample time)" de $1e-3$. El número de puntos máximo que se pueden guardar en las variables y ficheros se ha fijado a 500000.

```
clc
clear all
delete(findall(0,'Type','figure')) % cierra todas las figuras
t1=50;
t2=200;
m1=0.15;
m2=0;
t3=t2;
%%%%
tmax=1.2*t3;
%%%%
kInt=2/(-t1+t2+t3);
%%%%%%%%%%%%%%%%%%%%%%%%
Km=100;
p=66;
K0=Km/p;
incp=2;
%%%%%%%%%%%%%%%%%%%%%%
DatosArray=load('DatosSistema.txt');
%%%%%%%%%%%%%%%%%%%%%%%%
figure
plot(tout,TrapecioOrdenUno)
xlim([t2-1e-2 t2+0.15])
xlabel('t')
%legend({'K3*i(t)','Kd2*alpha(t)','Kd3*di(t)/dt'})
title('CurvaOrdenUno')
text(t2+0.05,2,'Km=100; p=66','Color','r')
%%%%%%%%%%%%%%%%%%%%%%%%%
figure
plot(tout,ErrorB1)
xlim([t2-1e-2 t2+0.15])
xlabel('t')
title('CurvaErrorOrdenUno')
text(t2+0.05,-0.1,'K0=Km/p=1.5152; pm=64','Color','r')
%%%%%%%%%%%%%%%%%%%%%%%%%
```

Tabla 8.3: RefTrapecioModeladoM.m

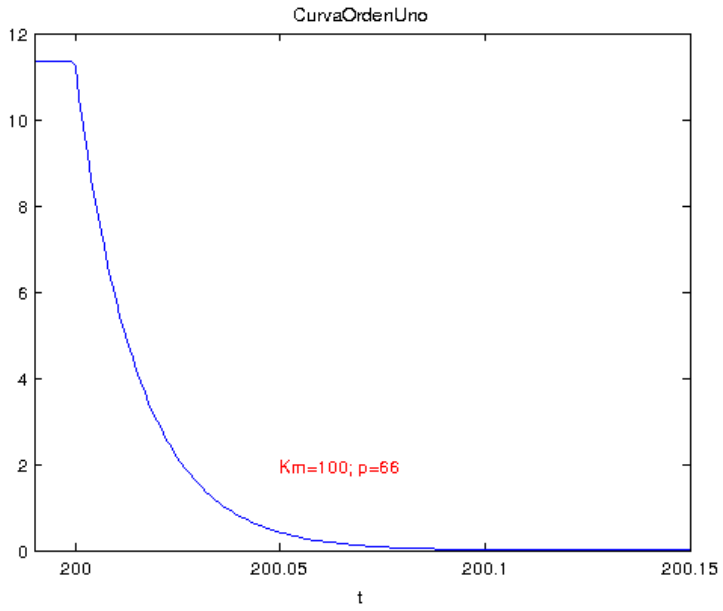

Figura 8.7: CurvaOrdenUno

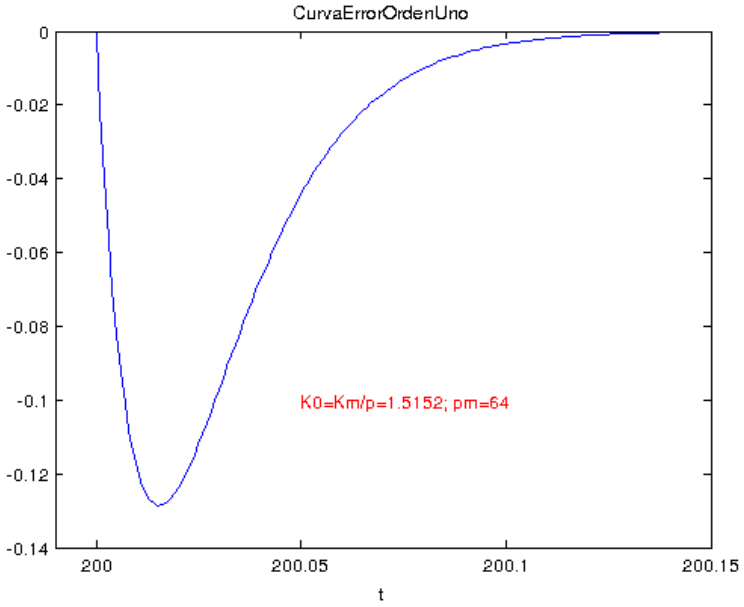

Figura 8.8: CurvaErrorOrdenUno

A. Manipulación simbólica y conversión a numérica

En ocasiones conviene manipular matrices, polinomios y funciones simbólicamente, y después convertirlas a numéricas.

A.1. Matrices y polinomios

El siguiente ejemplo permite manejar la matriz A dependiente de un parámetro h, calculando su inversa $Ainv$, sus autovalores l y su determinante $Adet$. Posteriormente pueden realizarse operaciones numéricas dando algún valor concreto a h (en el ejemplo $h \leftarrow h1$ donde $h1 = 8$, siendo $h1$ una variable numérica y h una variable simbólica).

Para la conversión simbólica a numérica es necesario que la matriz esté expresada con símbolos numéricos, es decir, debe realizarse una substitución de todos los parámetros indeterminados por números utilizando la función de Matlab **subs**. El resultado seguirá siendo simbólico, aunque con parámetros numéricos. Entonces ya puede transformarse en una matriz numérica utilizando, por ejemplo, la función de Matlab **double**. En la Tabla A.1 se muestra cómo hacer esto.

La matriz simbólica A que utilizaremos en los ejemplos tiene la forma siguiente:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & h \end{bmatrix} \tag{A.1}$$

```
syms h real;
A=[1 2; 3 h];            % A es simbólica
Ainv=inv(A);             % Ainv es simbólica
Adet=det(A);             % Adet es simbólica
l=eig(A);                % l es simbólica
h1=8;                    % h1 es numérica o simbólica
B= subs(A,h,h1);         % B es simbólica
AN=double(B);            % AN es numérica
lN=double(subs(l,h,h1)); % lN es numérica
lS=eig(B);               % lS es simbólica
lN2=double(lS);          % lN2=lN
lN3=eig(AN);             % lN3=lN
```

Tabla A.1: **subs** y **double**

Supongamos que se quiere expresar la matriz inversa simbólica $Ainv$ en función de alguna variable que sea común a todos los elementos de la matriz, entonces puede definirse una nueva variable simbólica $Idet$ y obtener una subexpresión utilizando la función de Matlab **subexpr**, a partir de la cual pueden realizarse operaciones simbólicas. En la Tabla A.2 se obtiene simbólicamente la matriz adjunta traspuesta o matriz de cofactores, $Aadj$, es decir, $Aadj = Ainv * Adet$.

```
Ainv=inv(A);                          % Ainv es simbólica
[Ainv2, Idet]=subexpr(Ainv,'Idet');
Aadj2=Ainv2/'Idet';                   % Aadj2 es simbólica
Aadj= Ainv*Adet;                      % Aadj=Aadj2
```

Tabla A.2: **subexpr**

La función **subexpr** de Matlab asigna a la variable simbólica $Idet$ la inversa del determinante. Hay que tener en cuenta que ahora $Idet$ es la inversa del determinate de A, pero 'Idet' (entrecomillada) es el nombre de la variable simbólica que aparece en $Ainv2$. Es por esto que debe hacerse $Aadj2 = Ainv2/$'Idet' y no $Aadj2 = Ainv2/Idet$.

Los autovalores pueden obtenerse, como se ha hecho antes, utilizando la función de Matlab **eig** o resolviendo simbólicamente la ecuación característica $det(A - sI) = 0$ donde I es la matriz identidad, que en Matlab es $eye(n)$ siendo n el orden de la matriz, y s son los autovalores. Puede hacerse calculando el polinomio carcaterístico P a partir de la definición, o utilizando la función de Matlab **charpoly** como se muestra en la Tabla A.3.

```
syms s;
P=det(A-s*eye(2));
l2=solve(P==0);      % l2 vector simbólico de autovalores de A
P2=charpoly(A,s);    % P2=collect(P) y P=expand(P2)
```

Tabla A.3: **solve** y **charpoly**

En general $l2$ y $l = eig(A)$ representan los mismos autovalores de A, pero pueden aparecer en orden distinto, es decir, que $l(1)$ puede ser $l2(2)$ y $l(2)$ puede ser $l2(1)$. No obstante pueden ordenarse de mayor a menor utilizando la función **sort** de Matlab. Entonces haciendo $l = sort(l)$ y $l2 = sort(l2)$ se cumple que $l = l2$, y por lo tanto $l(i) = l2(i)$ para el i-ésimo autovalor.

Hay algunas funciones en Matlab para la simplificación y manipulación de expresiones simbólicas, como por ejemplo **collect**, **factor** y **expand**. Puede consultarse la ayuda de Matlab para comprender qué es lo que hacen y cómo deben utilizarse. En el ejemplo anterior $P2 = collect(P)$ y también $P = expand(P2)$, es decir, que los polinomios característicos aparecen con los coeficientes expandidos o agrupados según las potencias de la variable s.

Es posible obtener los coeficientes de un polinomio P de manera simbólica con la función de Matlab **coeffs**, pero hay que tener en cuenta que no solo el resultado es simbólico sino que aparece en orden de grados crecientes de los monomios del polinomio. Puede interesar que aparezcan en orden de grados decrecientes de los monomios, ya que esta es la forma habitual de representar los polinomios numéricos a partir de sus coeficientes.

Para la obtención correcta del polinomio numérico también puede utilizarse directamente la función de Matlab **sym2poly** una vez se hayan substituido todos los parámetros indeterminados por parámetros numéricos, como se muestra en la Tabla A.4.

```
P=collect(P);                    % no es necesario hacer esto
Pcoefs=coeffs(P,s);                   % simbólico
Pcoefs2=sort(Pcoefs,'ascend');        % simbólico
h1=8;                                 % numérico
PNcoefs=double(subs(Pcoefs2,h,h1));   % numérico
PNcoefsMal=double(subs(Pcoefs,h,h1)); % numérico
PS=subs(P,h,h1);                      % simbólico
PNcoefs2=sym2poly(PS);           % PNcoefs2=PNcoefs<>PNcoefsMal
PS2=poly2sym(PNcoefs,s);              % PS2=PS
```

Tabla A.4: **sort**, **coeffs**, **sym2poly** y **poly2sym**

Los polinomios numéricos *PNcoefs* y *PNcoefsMal* son distintos. El primero, *PNcoefs* representa correctamente el polinomio numérico correspondiente a P, mientras que *PNcoefsMal* es incorrecto. En el ejemplo

$$P = s\hat{}2 + (-h - 1) * s + h - 6 \tag{A.2a}$$

$$PS = s\hat{}2 - 9 * s + 2 \tag{A.2b}$$

$$PNcoefs = \begin{bmatrix} 1 & -9 & 2 \end{bmatrix} \tag{A.2c}$$

$$PNcoefsMal = \begin{bmatrix} 2 & -9 & 1 \end{bmatrix} \tag{A.2d}$$

Los elementos de los vectores numéricos, cuando representan polinomios, están ordenados de mayor a menor grado de los monomios, por lo que

$$PNcoefs \equiv x^2 - 9x + 2 \tag{A.3a}$$

$$PNcoefsMal \equiv 2x^2 - 9x + 1 \tag{A.3b}$$

donde x es una variable genérica.

En la Tabla A.5 se muestra un ejemplo más general de un polinomio simbólico de orden dos y su conversión a un polinomio numérico, así como las soluciones de sus ecuaciones correspondientes.

```
syms a0 a1 a2 s;
a=[a0 a1 a2];                % simbólico
an=[1 0 4];                  % numérico
p= a0*s^2+ a1*s + a2;        % simbólico
pn= sym2poly(subs(p,a,an));  % numérico
l=solve(p==0);               % simbólico
ln=double(subs(l,a,an));     % numérico
```

Tabla A.5: Polinomio p y solución de la ecuación $p = 0$

Con la función de Matlab **coeffs** es posible obtener los vectores simbólicos que representan a los coficientes y a los monomios asociados a ellos. En el ejemplo de la Tabla A.6 se reconstruye el polinomio p a partir de estos dos vectores simbólicos.

```
syms a0 a1 a2 s;
a=[a0 a1 a2];
p= a0*s^2+ a1*s + a2;
[pc,m]=coeffs(p,s);
mm=transpose(m);
p2=pc*mm;                      % p2=p
an=[1 0 4];                    % numérico
pn=double(subs(pc,a,an));      % numérico
pc2=coeffs(p,s);
pnMal=double(subs(pc2,a,an)); % numérico
```

Tabla A.6: Reconstrucción simbólica de un polinomio utilizando **coeffs**

Utilizando la función **coeffs** en la forma [pc,m]=coeffs(p,s) los coeficientes están ordenados de forma más conveniente para la conversión a polinomios numéricos:

$$pc = \begin{bmatrix} a0, & a1, & a2 \end{bmatrix} \tag{A.4a}$$

$$m = \begin{bmatrix} s\hat{}2, & s, & 1 \end{bmatrix} \tag{A.4b}$$

$$pn = \begin{bmatrix} 1 & 0 & 4 \end{bmatrix} \tag{A.4c}$$

$$pc2 = \begin{bmatrix} a2, & a1, & a0 \end{bmatrix} \tag{A.4d}$$

$$pnMal = \begin{bmatrix} 4 & 0 & 1 \end{bmatrix} \tag{A.4e}$$

A.2. Funciones: ecuación diferencial lineal de parámetros constantes de segundo orden

En esta subsección vamos a introducir la forma de resolver simbólicamente una ecuación diferencial lineal de parámetros constantes de segundo orden, cuyo parámetro independiente es el tiempo t. La ecuación diferencial tiene la forma

$$\ddot{y}(t) + 2\zeta w_n \dot{y}(t) + w_n^2 y(t) = w_n^2 u(t) \tag{A.5}$$

Aunque es posible resolver la ecuación diferencial en la forma de entrada/salida, lo haremos mediante la representación en el espacio de estados. La ecuaciones de estado y salida son,

$$\dot{x}(t) = Ax(t) + Bu(t) \tag{A.6a}$$
$$y(t) = Cx(t) \tag{A.6b}$$

con condiciones iniciales x_0, donde

$$x(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} \tag{A.7a}$$
$$x_0 = \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} \tag{A.7b}$$
$$A = \begin{bmatrix} 0 & 1 \\ -w_n^2 & -2\zeta w_n \end{bmatrix} \tag{A.7c}$$
$$B = \begin{bmatrix} 0 \\ w_n^2 \end{bmatrix} \tag{A.7d}$$
$$C = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{A.7e}$$

Las instrucciones en Matlab de la Tabla A.7 permiten representar y resolver de manera genérica las ecuaciones A.6 en el caso en que $u(t)$ sea constante.

Puede observarse que se han introducido suposiciones en los parámetros: $t \geq 0$ y $w_n > 0$ utilizando la función de Matlab **assume**. Puede comprobarse cuáles son los supuestos realizados sobre los parámetros utilizando la función de Matlab **assumptions**. Hay que tener en cuenta que Matlab considera que las variables son simplemente simbólicas, por lo que al resolver ecuaciones los resultados serán demasiado generales si no se especifica explícitamente el tipo.

```
syms x1(t) x2(t)
x=[x1; x2];
syms wn ze t real
assume(wn>0 & t>=0);
syms u real % si se pone u(t) se complica dsolve
%%%%%
a1=2*wn*ze;
a2=wn^2;
b0=a2;
A=[0 1; -a2 -a1];
B=[0; b0];
f=diff(x)==A*x+B*u;
syms x01 x02 real
x0=[x01;x02];
F=dsolve(f,x(0)==x0);
y(t)=F.x1
symvar(y) % parámetros de y
argnames(y) % argumento de y, que será t
assumptions(symvar(y))
```

Tabla A.7: Resolución simbólica de un sistema de segundo orden

Puede verse al ejecutar en Matlab estas instrucciones que la función resultante $y(t)$ es una expresión muy larga que conviene simplificar de alguna forma. Puesto que depende de condiciones iniciales arbitrarias lo que haremos es separar la parte de la solución que incluye el vector de condiciones iniciales x_0, simplificando posteriormente cada una de las partes por separado. Para ello haremos lo siguiente:

```
y1=subs(y,x0,[0 ;0]);
yfA=simplify(real(y1));
yf0=simplify(real(y-y1));
```

Tabla A.8: Separación de la parte de condiciones iniciales: $y = yfA + yf0$

En lo que sigue solo simplificaremos las expresiones de la parte de condiciones iniciales nulas. En general la simplificación de expresiones simbólicas no es una tarea sencilla.

Sabemos que un sistema de segundo orden tiene dos soluciones cualitativamente distintas, según que los polos sean complejos conjugados o reales. Sabemos que esto depende del valor del coeficiente de amortiguamiento, ζ. Por lo tanto, nos interesa obtener las expresiones simbólicas para cada caso imponiendo las condiciones correspondientes utilizando la función de Matlab **assume**.

Para el caso de polos complejos conjugados pueden ejecutarse las instrucciones de la Tabla A.9.

```
assume(1>ze>0);
yf=simplify(real(yfA));
[yf1,a]=subexpr(yf,'a');
yf2(t)=subs(yf1,a,'a');
[yf3,b(t)]=subexpr(yf2,'b(t)');
yf4=subs(yf3,b,'b(t)');
yf5=collect(yf4,'b(t)');
yf6(t)=collect(yf5,u)
a
b
```

Tabla A.9: Caso polos complejos conjugados, $\zeta \in [0, 1)$

A partir de estas instrucciones se obtiene la función simpliflicada $yf6(t)$ que tiene la forma reconocible siguiente:

$$yf6(t) = (1 - b(t) * (a * cos(a * t * wn) + ze * sin(a * t * wn)))/a) * u \tag{A.8}$$

donde

$$a = (1 - ze\hat{}2)\hat{}(1/2) \tag{A.9a}$$
$$b(t) = exp(-t * wn * ze) \tag{A.9b}$$

Para el caso de polos reales puede hacerse como en la Tabla A.10.

```
assume(1<ze) % CASO: reales
yfR=simplify(real(yfA));
[yfR1,aR]=subexpr(yfR,'aR');
yfR2(t)=subs(yfR1,aR,'aR');
[yfR3(t),bRn(t)]=subexpr(yfR2,'bRn(t)');
[yfR3(t),bRp(t)]=subexpr(yfR3,'bRp(t)');
yfR4(t)=collect(collect(yfR3,'bRp(t)'),'bRn(t)');
yfR5(t)=collect(yfR4,u)
aR
bRp
bRn
```

Tabla A.10: Caso polos reales, $\zeta \geq 1$

A partir de estas instrucciones se obtiene la función simpliflicada $yfR5(t)$ que tiene la forma reconocible siguiente:

$$yfR5(t) = (1 - (bRn(t) * (aR - ze))/(2 * aR) - (bRp(t) * (aR + ze))/(2 * aR)) * u \quad \text{(A.10)}$$

donde

$$aR = (ze\hat{}2 - 1)\hat{}(1/2) \quad \text{(A.11a)}$$
$$bRn(t) = exp(t * wn * (aR - ze)) \quad \text{(A.11b)}$$
$$bRp(t) = exp(-t * wn * (aR + ze)) \quad \text{(A.11c)}$$

A.3. Gráficas a partir de representaciones simbólicas

En Matlab pueden crearse gráficas a partir de representaciones simbólicas utilizando la función **ezplot**, y también haciendo, primero, una conversión de la función simbólica al tipo "function_handle" utilizando la función de Matlab **matlabFunction** y después utilizando la función **fplot**. También puede utilizarse la función **plot**.

En cualquier caso para obtener curvas es necesario sustituir todos los parámetros, a excepción de uno (que en nuestro caso será el tiempo t), por valores numéricos.

En las siguientes instrucciones de la Tabla A.11 se obtienen curvas a partir de la función $yf6(t)$ obtenida en la subsección anterior, y dada por A.8, para los valores $w_n = 4$ y $\zeta \in \{0.2, 0.4, 0.7\}$. Para el valor de $\zeta = 0{,}2$ utilizaremos la función **ezplot**, para $\zeta = 0{,}4$ la función **plot** y para $\zeta = 0{,}7$ la función **fplot**, con el fin de que se vea cómo hacer las cosas con los tres métodos. Dibujaremos las tres curvas en una única gráfica para el valor de $u = 1$ y $t \in [0, 5]$, utilizando la función de Matlab **hold**.

Para dibujar las curvas no es necesario haber hecho simplificaciones, por lo que se podía haber utilizado la función $y(t)$ o $y1(t)$ obtenidas en la subsección anterior, haciendo las sustituciones numéricas correspondientes.

```
g=subs(yf6,{'a','b(t)'},{a,b});          % simbólico
g02=subs(g,{'ze','wn','u'},{0.2,4,1});   % simbólico
g04=subs(g,{'ze','wn','u'},{0.4,4,1});   % simbólico
g07=subs(g,{'ze','wn','u'},{0.7,4,1});   % simbólico
g07M=matlabFunction(g07);                % function_handle
H=ezplot(g02,[0 5]);
set(H,'Color','r');
hold on
fplot(g07M,[0 5],'b');
tt=0:0.01:5;
plot(tt,g04(tt),'m');
hold off
```

Tabla A.11: Gráfica a partir de $yf6(t)$ dada por A.8

La función **plot** también admite una 'function_handle'. Es decir, que en vez de utilizar **fplot** se podía haber hecho **plot(tt,g07M(tt),'b');**.

La gráfica resultante es la de la Figura A.1.

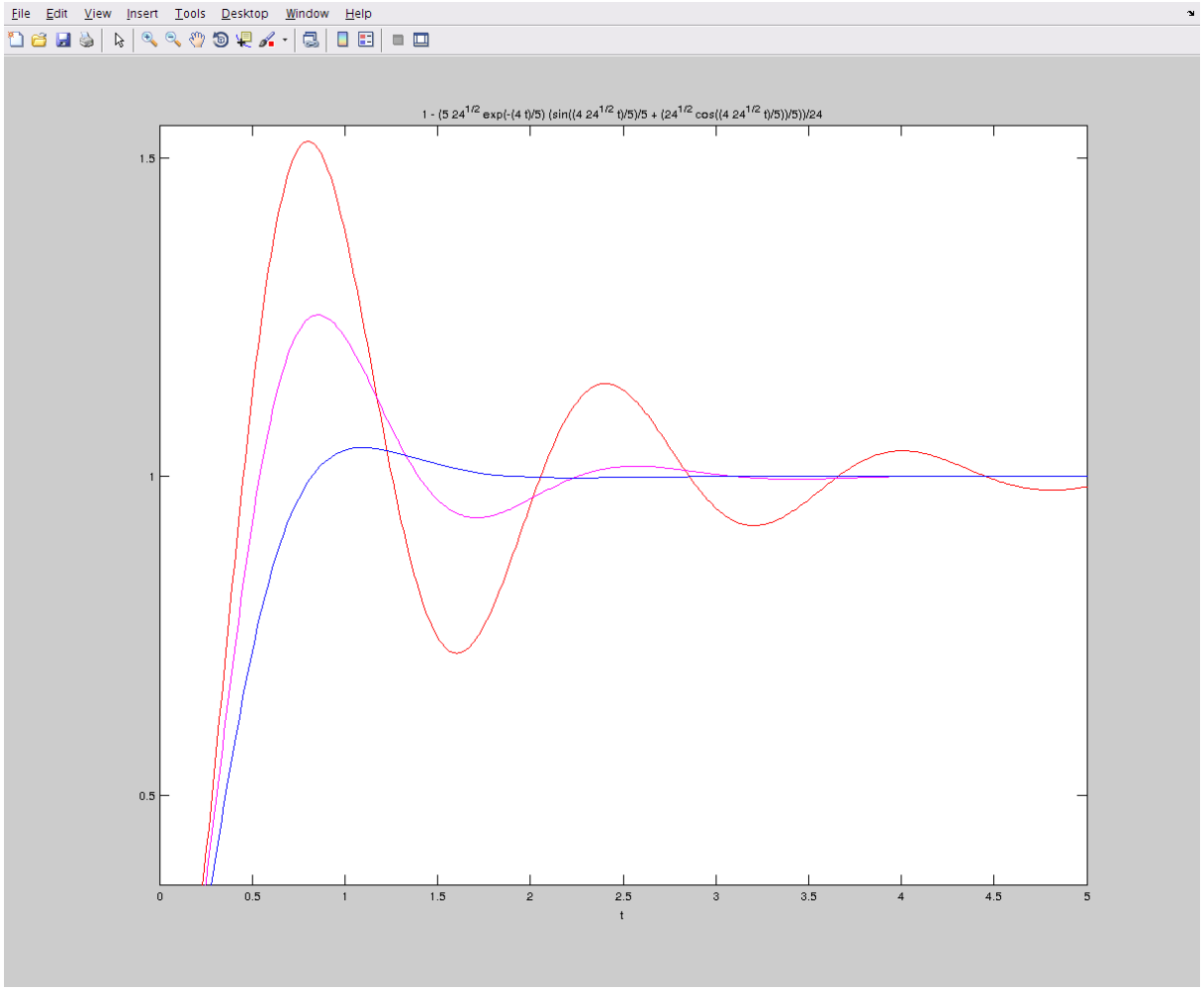

Figura A.1: Curvas de $yf6(t)$