Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniera y Agrimensura

Tesina de Grado - Licenciatura en Ciencias de la Computación

Implementación Autónoma de Métodos de Integración Numérica QSS

Joaquín Francisco Fernández
<joaquin.f.fernandez@gmail.com>
Legajo: F-1843/1

Director: Dr. Ernesto Kofman
<kofman@cifasis-conicet.gov.ar>

Marzo, 2012

Resumen

La resolución de ecuaciones diferenciales ordinarias, requiere el uso de métodos de integración numérica. Todos los algoritmos tradicionales de integración se basan en la discretización de la variable independiente (que generalmente representa el tiempo). Las rutinas que implementan estos algoritmos, se denominan *solvers*. Los Métodos de Integración Numérica QSS (*Quantized State System*), realizan la discretización sobre las variables de estado. En consecuencia, convierten los sistemas continuos en sistemas de eventos discretos. La mayor parte de las implementaciones actuales de los métodos de QSS, utilizan como base herramientas de simulación de eventos discretos, basados en el formalismo *DEVS* (*Discrete EVent System specification)*. Si bien estas implementaciones son simples, resultan ineficientes, dado que malgastan gran parte de la carga computacional en la transmisión de eventos entre submodelos. En este trabajo, se presentará una implementación autónoma para los métodos de integración numérica QSS, sin utilizar el formalismo *DEVS*, formularemos y desarrollaremos un motor de simulación apto para este tipo de métodos, definiendo un entorno de simulación capaz de obtener información estructural de las ecuaciones de estado y de los eventos que forman el modelo, permitiendo declarar modelos en este nuevo entorno de simulación y compararemos los resultados obtenidos con las implementaciones tradicionales de los métodos de integración numérica QSS.

Agradecimientos

Quiero agradecer a Ernesto Kofman, por su apoyo y dedicación durante todo el desarrollo de este trabajo, a todos mis amigos que estuvieron conmigo durante estos años, en especial a Javier, Ariel y Ciro y a todos los que me compañaron en este camino. Por último, quiero agregadecer especialmente a mis padres, Francisco y Gladis, y a mis hermanas, Paola y Juana por ser un ejemplo de vida a seguir.

Índice general

Capítulo 1

Introducción

La resolución de ecuaciones diferenciales ordinarias, requiere el uso de métodos de integración numérica. Todos los algoritmos tradicionales de integración se basan en la discretización de la variable independiente (que generalmente representa el tiempo). Las rutinas que implementan estos algoritmos, se denominan *solvers* y existen gran variedad de implementaciones de los mismos en diferentes lenguajes de programación (principalmente en Fortran y C).

Los Métodos de Integración Numérica QSS (*Quantized State System*) [11, 7, 13], a diferencia de los métodos de integración tradicionales [3], realizan la discretización sobre las variables de estado. En consecuencia, convierten los sistemas continuos en sistemas de eventos discretos, y tienen grandes ventajas para simular sistemas con discontinuidades.

La mayor parte de las implementaciones actuales de los métodos de QSS, utilizan como base herramientas de simulación de eventos discretos, basados en el formalismo *DEVS* (*Discrete EVent System specification).* Entre ellas encontramos:

1. PowerDEVS [2], que implementa la totalidad de los algoritmos de QSS.
2. Cell-Devs [4], donde se implementan los métodos QSS de primer orden.
3. VLE [15], donde se implementan los métodos QSS de primer orden.
4. En [1], se desarrollaron los métodos QSS basados en DEVS para OpenModelica.

Estas implementaciones, si bien son simples, resultan ineficientes, dado que malgastan gran parte de la carga computacional en la transmisión de eventos entre submodelos.

El principal obstáculo para implementar los algoritmos de QSS de forma autónoma, es la necesidad de obtener información estructural sobre las ecuaciones. Una implementación autónoma eficiente de éstos métodos requiere el conocimiento de las matrices de incidencia, que indican qué variables influyen sobre cada ecuación. Además en presencia de discontinuidades, se necesita información estructural adicional.

Hasta el momento, existe una implementación autónoma de los métodos de QSS en Java, denominada Easy Java Simulations (EJS)[5]. Sin embargo, en la

misma, toda la información estructural necesaria, la deben definir manualmente los usuarios, lo cuál es muy poco práctico. Además al estar implementado en Java, resulta muy ineficiente en lo que respecta a tiempos de simulación.

En este trabajo desarrollaremos un entorno de simulación completo para métodos de integración numérica QSS, dando la definición del motor de simulación apto para este tipo de métodos, la implementación de los distintos *solvers* QSS dentro de este marco y la representación de los modelos.

Para este desarrollo, veremos en los capítulos siguientes:

- En el Capítulo 2 veremos una breve descripción de los métodos de integración clásicos, sus principales características y propiedades, así como también la formulación de distintos motores de simulación para estos métodos.
- En el Capítulo 3 presentaremos los métodos de integración por cuantificación de estados, veremos sus principales caracterícas y propiedades, daremos una breve introducción al formalismo DEVS y veremos la definición de los métodos de integración de QSS dentro de este marco.
- En el Capítulo 4, daremos la formulación de un motor de simulación apto para los métodos de integración por cuantifiación de estado, evaluaremos las diferencias existentes con un motor de simulación tradicional, y defeniremos la información estructural adicional necesaria para poder simular sistemas de eventos discretos, sin utilizar el formalismo DEVS. Dentro de este marco, daremos una formulación para los *solvers* y la definición de los modelos a simular.
- Luego, en el Capítulo 5, evaluaremos los detalles de implementación del entorno de simulación presentado en el capítulo anterior.
- En el Capítulo 6, mostraremos distintos ejemplos en el nuevo entorno de simulación presentado, donde evaluaremos la eficiencia del simulador, utilizando como herramienta de comparación, PowerDEVS, un entorno de simulación basado en el formalismo DEVS, que además de ser el único entorno en el que están implementados los métodos de QSS en su totalidad, es el más eficiente.
- Por último en el Capítulo 7, daremos las conclusiones y describiremos las modificaciones y posibles trabajos futuros.

Capítulo 2

Métodos clásicos de integración

En este capítulo, basado principalmente en [3], se presenta una muy breve descripción de algunos de los métodos de integración de tiempo discreto, así como también, la formulación de algoritmos de simulación para los motores que utilizan estos métodos y de algunos de los algoritmos de los métodos de integración mencionados.

2.1. Conceptos básicos de integración numérica

En esta sección se introducirán características generales de todos los métodos de tiempo discreto, como así también herramientas y conceptos que se utilizarán a lo largo del capítulo.

2.1.1. Principios de integración numérica

Los métodos de integración surgen como una herramienta que permite la simulación de modelos de sistemas continuos ya que por lo general resulta muy dificil encontrar soluciones analíticas de los mismos.

Para poder comprender el principio en que se basan todos los métodos de integración, analicemos en forma general la aproximación que realizan los métodos de integración del modelo de ecuaciones de estado:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t) \tag{2.1}$$

donde $\mathbf{x}$ es el *vector de estados*, $\mathbf{u}$ es el *vector de entradas*, y t representa el tiempo, con condiciones iniciales:

$$\mathbf{x}(t = t_0) = \mathbf{x_0} \tag{2.2}$$

Una componente $x_i(t)$ del vector de estados representa la i^{th} trayectoria del estado en función del tiempo t. Siempre y cuando el modelo de ecuaciones de estado no contenga discontinuidades en $f_i(\mathbf{x}, \mathbf{u}, t)$ ni en sus derivadas, $x_i(t)$ será también una función continua. Además, la función podrá aproximarse con la precisión deseada mediante series de Taylor alrededor de cualquier punto de la

trayectoria (siempre y cuando no haya escape finito, es decir, que la trayectoria tienda a infinito para un valor finito de tiempo).

Denominando t^* al instante de tiempo en torno al cual se aproxima la trayectoria mediante una serie de Taylor, y sea $t^* + h$ el instante de tiempo en el cual se quiere evaluar la aproximación. Entonces, la trayectoria en dicho punto puede expresarse como sigue:

$$x_i(t^* + h) = x_i(t^*) + \frac{dx_i(t^*)}{dt} \cdot h + \frac{d^2x_i(t^*)}{dt^2} \cdot \frac{h^2}{2!} + \dots \tag{2.3}$$

Reemplazando con la ecuación de estado (2.1), la serie (5.4) queda:

$$x_i(t^* + h) = x_i(t^*) + f_i(t^*) \cdot h + \frac{df_i(t^*)}{dt} \cdot \frac{h^2}{2!} + \dots \tag{2.4}$$

Los distintos algoritmos de integración difieren en la manera de aproximar las derivadas superiores del estado y en el número de términos de la serie de Taylor que consideran para la aproximación.

2.1.2. Precisión de la aproximación y orden de un método

Evidentemente, la precisión con la que se aproximan las derivadas de orden superior debe estar acorde al número de términos de la serie de Taylor que se considera. Si se tienen en cuenta $n + 1$ términos de la serie, la precisión de la aproximación de la derivada segunda del estado $d^2x_i(t^*)/dt^2 = df_i(t^*)/dt$ debe ser de orden $n - 2$, ya que este factor se multiplica por h^2. La precisión de la tercer derivada debe ser de orden $n - 3$ ya que este factor se multiplica por h^3, etc. De esta forma, la aproximación será correcta hasta h^n. Luego, n se denomina *orden de la aproximación* del método de integración, o, simplemente, se dice que el método de integración es de orden n.

Mientras mayor es el orden de un método, más precisa es la estimación de $x_i(t^* + h)$. En consecuencia, al usar métodos de orden mayor, se puede integrar utilizando pasos grandes. Por otro lado, al usar pasos cada vez más chicos, los términos de orden superior de la serie de Taylor decrecen cada vez más rápidos y la serie de Taylor puede truncarse antes.

El costo de cada paso depende fuertemente del orden del método en uso. En este sentido, los algoritmos de orden alto son mucho más costosos que los de orden bajo. Sin embargo, este costo puede compensarse por el hecho de poder utilizar un paso mucho mayor y entonces requerir un número mucho menor de pasos para completar la simulación. Esto implica que hay que buscar una solución de compromiso entre ambos factores.

2.1.3. Integración de Euler

El algoritmo de integración más simple se obtiene truncando la serie de Taylor tras el término lineal:

$$\mathbf{x}(t^* + h) \approx \mathbf{x}(t^*) + \dot{\mathbf{x}}(t^*) \cdot h \tag{2.5a}$$

o:

$$\mathbf{x}(t^* + h) \approx \mathbf{x}(t^*) + \mathbf{f}(\mathbf{x}(t^*), t^*) \cdot h \tag{2.5b}$$

Este esquema es particularmente simple ya que no requiere aproximar ninguna derivada de orden superior, y el término lineal está directamente disponible del modelo de ecuaciones de estado. Este esquema de integración se denomina *Método de Forward Euler* (FE).

La Fig.2.1 muestra una interpretación gráfica de la aproximación realizada por el método de FE.

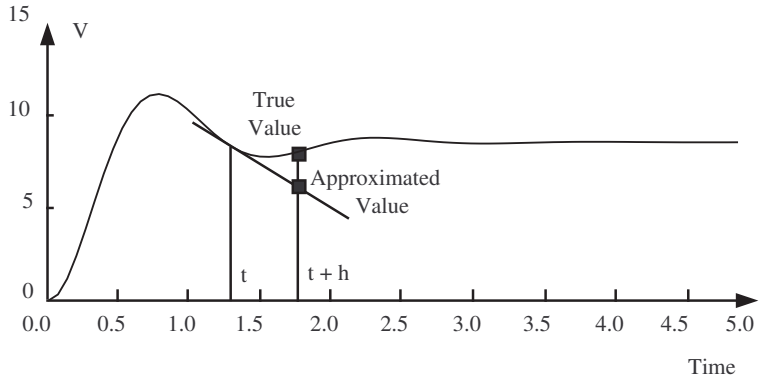

Figura 2.1: Integración numérica utilizando Forward Euler.

La simulación utilizando el método de FE se torna trivial ya que el método de integración utiliza solo valores pasados de las variables de estado y sus derivadas. Un esquema de integración que exhibe esta característica se denomina *algoritmo de integración explícito.*

Un simulador para el método de FE, se puede describir de la siguiente manera:

Algoritmo 2.1: Motor de simulación para Fordward Euler

```
1  FE(h, f, it, ft, x0)
2  begin
3     t := it;
4     x := x0;
5     While  (t ≤ ft)
6     begin
7        x := x + f(x(t), t) · h ) ;
8        t := t + h;
9     end
10 end
```

Mostraremos un ejemplo simple de cómo definir un modelo para este motor de simulación, consideremos el siguiente sistema:

$$\begin{cases} \dot{x}_1(t) = x_2(t) \\ \dot{x}_2(t) = 1 - x_1(t) - x_2(t) \end{cases} \tag{2.6}$$

Lo único que necesitamos para poder simular el sistema es la formulación de la función que representa las ecuaciones de estado del sistema, como se muestra en el Modelo 2.2.

Modelo 2.2: Simulación Fordward Euler

```
1 f(x, t)
2 begin
3    x1 := x2;
4    x2 := 1 − x1 − x2;
5    return [x1, x2]
6 end
```

La Figura 2.2 muestra un esquema con una pequeña modificación.

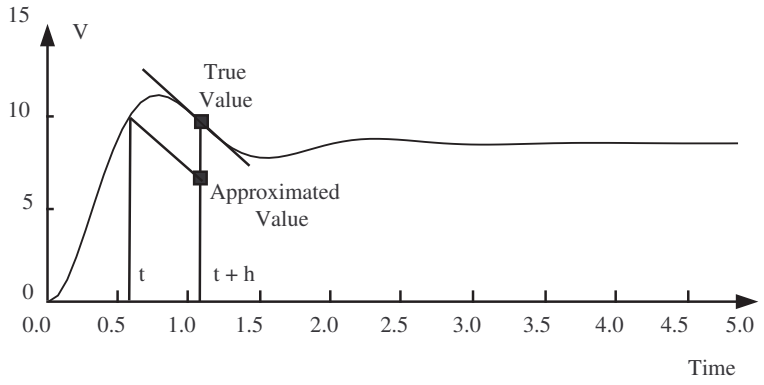

Figura 2.2: Integración numérica utilizando Backward Euler.

En este esquema, la solución $\mathbf{x}(t^* + h)$ se aproxima utilizando los valores de $\mathbf{x}(t^*)$ y $\mathbf{f}(\mathbf{x}(t^* + h), t^* + h)$ mediante la fórmula:

$$\mathbf{x}(t^* + h) \approx \mathbf{x}(t^*) + \mathbf{f}(\mathbf{x}(t^* + h), t^* + h) \cdot h \tag{2.7}$$

Este esquema se conoce como el método de integración de *Backward Euler* (BE).

Como puede verse, esta fórmula depende del valor actual y del valor futuro de las variables, lo que causa problemas. Para calcular $\mathbf{x}(t^* + h)$ en la Eq.(2.7), necesitamos conocer $\mathbf{f}(\mathbf{x}(t^* + h), t^* + h))$, pero para calcular $\mathbf{f}(\mathbf{x}(t^* + h), t^* + h))$ de la Ec.(2.1), necesitamos saber $\mathbf{x}(t^* + h)$. En consecuencia, estamos ante un *lazo algebraico* no lineal.

Este tipo de algoritmos se denominan *métodos de integración implícitos.*

Si bien los métodos implícitos son ventajosos desde el punto de vista numérico, la carga computacional adicional creada por la necesidad de resolver simultáneamente un sistema de ecuaciones algebraicas no lineales al menos una vez por cada paso de integración puede hacerlos inapropiados para su uso en software de simulación de propósito general excepto para aplicaciones específicas, tales como los sistemas stiff (ver Sección 2.2.2).

En general, en cualquier esquema de iteración para métodos implícitos, se utiliza la iteración de Newton [1], que requiere el cómputo de al menos una aproximación de la matriz Jacobiana, y la inversión (refactorización) de la matriz Hessiana. Como ambas operaciones son bastante costosas, las diferentes implementaciones varían en qué tan a menudo recalculan el Jacobiano (esto se

[1] Una explicación en detalle sobre la iteración de Newton se encuentra en [3].

denomina iteración de Newton modificada). Cuanto más no lineal sea el problema, más a menudo hay que recalcular el Jacobiano. Así mismo, al cambiar el paso de integración, hay que refactorizar el Hessiano.

Definiremos ahora un simulador para el método de BE:

Algoritmo 2.3: Motor de simulación para Backward Euler

```
1  FE(h, f, it, ft, x0)
2  begin
3     t := it;
4     x := x0;
5     While  (t ≤ ft)
6     begin
7        x_p = x;
8        J = jacobiant(f, x_p, t));
9        invH = inv(I − h ∗ J);
10       x = x_p − invH ∗ (x_p − h ∗ f(x_p, t) − x);
11       l = 0;
12       While  (|x − x_p| > |x_p|  and  l < 20)
13       begin
14          l = l + 1;
15          x_p = x;
16          J = jacobiant(f, x_p, t);
17          invH = inv(I − h ∗ J);
18          x = x_p − invH ∗ (x_p − h ∗ f(x_p, t) − x);
19       end
20       t := t + h;
21    end
22 end
```

En los ejemplos presentados en el Algoritmo 2.1 como en el Algoritmo 2.3, el método de integración está definido dentro del motor. En general, en cualquier entorno de simulación, se puede distinguir entre la definición del motor de simulación, el método de integración numérica utilizado y la definición de los modelos de las ecuaciones de estado. En las secciones siguientes, se distinguirá entre la formulación del método integración y el motor de simulación utilizado. Además se dará una breve descripción del funcionamiento del motor de simulación correspondiente.

2.2. Métodos de integración monopaso

Se denomina métodos de integración monopaso a aquellos que calculan x_{k+1} utilizando únicamente información sobre x_k.

2.2.1. Métodos Runge-Kutta

Un método de Runge–Kutta es un algoritmo que avanza la solución desde $x_k(t_k)$ hasta $x_{k+1}(t_k + h)$, usando una formula del tipo

$$\mathbf{x_{k+1}} \;=\; \mathbf{x_k} \;+ h \cdot (c_1 \cdot \mathbf{k_1} + \ldots + c_n \cdot \mathbf{k_n})$$

donde las llamadas etapas $k_1 \ldots k_n$ se calculan sucesivamente a apartir de las ecuaciones:

$$
\begin{aligned}
&\text{etapa } 0: && \mathbf{k_1} = \mathbf{f}(\mathbf{x_k} + b_{1,1} \cdot h \cdot \mathbf{k_1} + \ldots + b_{1,n} \cdot h \cdot \mathbf{k_n}, t_k + a_1 h) \\
&\vdots && \vdots \\
&\text{etapa } n-1: && \mathbf{k_n} = \mathbf{f}(\mathbf{x_k} + b_{n,1} \cdot h \cdot \mathbf{k_1} + \ldots + b_{n,n} \cdot h \cdot \mathbf{k_n}, t_k + a_n h) \\
&\text{etapa } n: && \mathbf{x_{k+1}} = \mathbf{x_k} + c_1 \cdot h \cdot \mathbf{k_1} + \ldots + c_n \cdot h \cdot \mathbf{k_n}
\end{aligned}
$$

El número n de evaluaciones de función en el algoritmo se llama 'número de etapas' y frecuentemente es considerado como una medida del costo computacional de la fórmula considerada.

Una forma muy común de representar a los algoritmos de RK es a través de la tabla de Butcher. Esta tabla toma en el caso general la siguiente forma:

$$
\begin{array}{c|ccc}
a_1 & b_{1,1} & \ldots & b_{1,n} \\
\vdots & \vdots & \ddots & \vdots \\
a_n & b_{n,1} & \ldots & b_{n,n} \\
\hline
\mathrm{x} & c_1 & \ldots & c_n
\end{array}
$$

El método más popular de estos es el de Runge–Kutta de orden cuatro (RK4) con tabla de Butcher:

$$
\begin{array}{c|cccc}
0 & 0 & 0 & 0 & 0 \\
1/2 & 1/2 & 0 & 0 & 0 \\
1/2 & 0 & 1/2 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 \\
\hline
x & 1/6 & 1/3 & 1/3 & 1/6
\end{array}
$$

o sea,

$$
\begin{aligned}
&\text{etapa } 0: && \mathbf{k_1} = \mathbf{f}(\mathbf{x_k}, t_k) \\
&\text{etapa } 1: && \mathbf{k_2} = \mathbf{f}(\mathbf{x_k} + \tfrac{h}{2} \cdot \mathbf{k_1}, t_k + \tfrac{h}{2}) \\
&\text{etapa } 2: && \mathbf{k_3} = \mathbf{f}(\mathbf{x_k} + \tfrac{h}{2} \cdot \mathbf{k_2}, t_k + \tfrac{h}{2}) \\
&\text{etapa } 3: && \mathbf{k_4} = \mathbf{f}(\mathbf{x_k} + h \cdot \mathbf{k_3}, t_k + h) \\
&\text{etapa } 4: && \mathbf{x_{k+1}} = \mathbf{x_k} + \tfrac{h}{6} \cdot [\mathbf{k_1} + 2 \cdot \mathbf{k_2} + 2 \cdot \mathbf{k_3} + \mathbf{k_4}]
\end{aligned}
$$

Este algoritmo es particularmente atractivo debido a que tiene muchos elementos nulos en la tabla de Butcher. Tiene, como puede verse, cuatro evaluaciones de la función $\mathbf{f}$.

La formulación, tanto del método, como del motor de simulación para métodos monopaso, es la siguiente:

Algoritmo 2.4: Motor de simulación para métodos monopaso

```
1 RK4( f, x, t, h )
2 begin
3    k1 := f(x, t);
4    k2 := f(x + h/2 * k1, t + h/2);
```

```
    k3 := f(x + h/2 * k2, t + h/2);
    k4 := f(x + h * k3, t + h);
    x := x + h/6 * (k1 + 2 * k2 + 2 * k3 + k4);
    return x;
end

Engine(f, it, ft, x0, method, h)
begin
    t := it;
    x := x0;
    While (t ≤ ft)
    begin
        x :=method(f, x, t, h));
        t := t + h;
    end
end
```

Donde los parámetros que recibe el motor son los siguientes:

- f El modelo con la definición de las derivadas de las variables de estado.
- it El tiempo inicial de la simulación.
- ft El tiempo final de la simulación.
- $x0$ Valores iniciales de las variables de estado.
- h El paso de integración utilizado.
- *method* El método de integración utilizado.

En cada paso el motor debe:

1. Controlar el tiempo de simulación.
2. Evaluar el modelo f con el método de integración correspondiente, para obtener el nuevo valor $\mathbf{x}$ correspondiente a $t + h$ para todas las ecuaciones de estado definidas.
3. Actualizar el tiempo de simulación.

2.2.2. Sistemas stiff

Un sistema lineal y estacionario se dice que es *stiff* cuando es estable y hay autovalores cuyas partes reales son muy distintas, es decir, hay modos muy rápidos y modos muy lentos.

Daremos primero algunas definiciones básicas de los dominios de estabilidad numérica, un análisis detallado de los dominios de estabilidad de los distintos métodos de integración numérica aquí mencionados puede encontrarse en [3], para esto debemos considerar nuevamente la solución de un sistema autónomo, lineal y estacionario:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} \tag{2.8}$$

con las condiciones iniciales de la Ec.(2.2). La solución analítica es la siguiente:

$$\mathbf{x}(t) = \exp(\mathbf{A} \cdot t) \cdot \mathbf{x_0} \tag{2.9}$$

Esta solución es *analíticamente estable* si todas las trayectorias permanecen acotadas cuando el tiempo tiende a infinito. El sistema (2.8) es analíticamente estable si y sólo si todos los autovalores de $\mathbf{A}$ tienen parte real negativa:

$$\mathbb{R}\mathrm{e}\{\mathrm{Eig}(\mathbf{A})\} = \mathbb{R}\mathrm{e}\{\lambda\} < 0{,}0 \tag{2.10}$$

Se dice que un sistema lineal y estacionario de tiempo continuo integrado con un método dado de integración de paso fijo es *numéricamente estable* si y sólo si el sistema de tiempo discreto asociado es analíticamente estable.

El problema con los sistemas stiff es que la presencia de modos rápidos obliga a utilizar un paso de integración muy pequeño para no caer en la zona de inestabilidad del método. Obligando de esta manera a evaluar todas las ecuaciones cuando sólo es necesario evaluar los modos rápidos, lo que implica un costo computacional extra.

Para integrar entonces sistemas stiff sin tener que reducir el paso de integración a causa de la estabilidad, es necesario buscar métodos que incluyan en su región estable el semiplano izquierdo completo del plano $(\lambda \cdot h)$, o al menos una gran porción del mismo.

> *Definición:*
> Un método de integración que contiene en su región de estabilidad a todo el semiplano izquierdo del plano $(\lambda \cdot h)$ se denomina *absolutamente estable*, o, más simplemente, *A–estable*.

En las siguientes secciones, se presentarán métodos numéricos que son adecuados para resolver este tipo de sistemas.

2.2.3. Métodos de interpolación hacia atrás

Para poder tratar los problemas stiff se necesitan algoritmos A–estables. A continuación se mostrará una clase especial de algoritmos IRK que se denominan *métodos de backinterpolation*, o métodos de interpolación hacia atrás (métodos BI).

La idea de los Métodos BI consiste en dar un paso hacia atrás e iterar hasta que la condición 'final' del paso hacia atrás coincida con de x_k. Para poder comprender la idea observemos que el método de BE:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot \dot{\mathbf{x}}_{\mathbf{k+1}} \tag{2.11}$$

La Ec.(2.11) puede reescribirse como:

$$\mathbf{x_k} = \mathbf{x_{k+1}} - h \cdot \dot{\mathbf{x}}_{\mathbf{k+1}} \tag{2.12}$$

Luego, un paso hacia adelante utilizando BE puede interpretarse como un paso hacia atrás de valor $-h$ utilizando FE.

Una manera de implementar BE entonces es comenzar con una estimación de $\mathbf{x_{k+1}}$, integrar hacia atrás en el tiempo y luego iterar sobre la condición 'inicial' desconocida $\mathbf{x_{k+1}}$ hasta acertar el valor 'final' conocido $\mathbf{x_k}$.

2.2.4. Control de paso

Como se vio anteriormente, el uso de pasos de integración pequeños produce en general menores errores de integración pero con mayores costos computacionales. El paso de integración correcto es entonces un compromiso entre error y costo.

Como la relación error/costo con el paso de integración depende en gran medida de las propiedades numéricas del sistema a ser integrado, no está claro que un mismo paso de integración produzca el mismo error a lo largo de toda la simulación. Podría ser necesario entonces, variar el paso de integración durante la simulación para mantener el error en un valor más o menos constante. Esta observación nos lleva a la necesidad de buscar *métodos de paso variable* que a su vez requerirán de *algoritmos de control de paso*.

Básicamente la idea del control de paso consiste en tomar dos algoritmos distintos de distinto orden, y repetir dos veces el mismo paso, uno con cada algoritmo. Los resultados de ambos diferirán en ε. La diferencia entre ambas soluciones, ε, se utiliza como estima del error de integración local.

Luego se define el error relativo según

$$\varepsilon_{\text{rel}} = \frac{|x_1 - x_2|}{\text{máx}(|x_1|, |x_2|, \delta)} \tag{2.13}$$

donde δ es un factor pequeño, por ejemplo,$\delta = 10^{-10}$, introducido para evitar problemas cuando los estados están próximos a cero.

El control de paso consiste en tratar de mantener constante el valor de ε_{rel}. Para esto existen varias heurísticas, una de ellas consiste en rechazar un paso, disminuir el paso de integración y volverlo a realizar cuando el error obtenido es mayor que la tolerancia de error prefijada. Otra posibilidad consiste en aceptar un paso aunque el error sea mayor que la tolerancia de error y disminuir el paso de integración en el próximo paso.

Como se mencionó al comienzo de esta sección, para estimar el paso se precisan realizar cada paso con dos algoritmos de orden distintos. El problema de proceder así es el muy alto costo de evaluar cada algoritmo al menos una vez en cada paso (cada paso se calcula 2 veces al menos). Una idea que logra mejorar notablemente el costo computacional adicional consiste en introducido consiste en utilizar algoritmos RK empotrados (dos algoritmos RK que coinciden en sus primeras etapas). El más utilizado de estos métodos es el RK4/5 que posee la siguiente tabla de Butcher:

0	0	0	0	0	0	0
1/4	1/4	0	0	0	0	0
3/8	3/32	9/32	0	0	0	0
12/13	1932/2197	−7200/2197	7296/2197	0	0	0
1	439/216	-8	3680/513	−845/4104	0	0
1/2	−8/27	2	−3544/2565	1859/4104	−11/40	0
x_1	25/216	0	1408/2565	2197/4104	−1/5	0
x_2	16/135	0	6656/12825	28561/56430	−9/50	2/55

En este método puede verse que x_1 es un RK4 de cinco etapas y x_2 es un RK5 de 6 etapas. Sin embargo, ambos algoritmos comparten las primeras 5 etapas. Luego, el método completo con control de paso resulta ser un RK5 de 6 etapas y el único costo adicional del control de paso es el cálculo del corrector de RK4. En definitiva el control de paso es casi gratuito.

La formulación, del método de integración RK4/5 y del motor con control de paso es la siguiente:

Algoritmo 2.5: Motor de simulación con control de paso

```
1  RK45(f, x, t, h)
2  begin
3     k1 := f(x, t);
4     k2 := f(x + 1/4 * h * k1, t + 1/4 * h);
5     k3 := f(x + 3/32 * h * k1 + 9/32 * h * k2, t + 3/8 * h);
6     k4 := f(x + 1932/2197 * h * k1 − 7200/2197 * h * k2 + 7296/2197 * h * k3, t + 12/13 * h);
7     k5 := f(x + 439/216 * h * k1 − 8 * h * k2 + 3680/513 * h * k3 − 845/4104 * h * k4, t + h);
8     k6 := f(x − 8/27 * h * k1 + 2 * h * k2 − 3544/2565 * h * k3 + 1859/4104 * h * k4 − 11/40 * h * k5, t + 1/2 * h);
9     x1 := x + 25/216 * h * k1 + 1408/2565 * h * k3 + 2197/4104 * h * k4 − 1/5 * h * k5;
10    x2 := x + 16/135 * h * k1 + 6656/12825 * h * k3 + 28561/56430 * h * k4 − 9/50 * h * k5 + 2/55 * h * k6;
11    return[x1, x2];
12 end
13
14 Engine (f, it, ft, x0, method, tol, hmin, hmax)
15 begin
16    t := it;
17    x := x0;
18    k := 0;
19    h := hmin;
20    While (t ≤ ft)
21    begin
22       k := k + 1;
23       [x1, x2] :=method(f, x, t, h);
24       err := |x1 − x2|/max(|x2|, ϵ);
25       if (err > tol and h > hmin) then
26       begin
27          h := Obtener_Nuevo_Paso(tol, err, h)
28          k := k − 1;
29       else
30          t := t + h;
31          x := x2;
32          h := Obtener_Nuevo_Paso(tol, err, h)
33       end
34    end
35    if (h < hmin) then
36    begin
37       h := hmin;
38    end
39    if (h > hmax) Then
40    begin
41       h := hmax;
42    end
43 end
```

Donde, además de los parámetros definidos en el Algoritmo 2.4, se agregan:

- *tol* Error relativo.
- *hmin* Valor mínimo del paso de integración.

- *hmax* Valor máximo del paso de integración.

En cada paso el motor debe:

1. Controlar el tiempo de simulación.
2. Evaluar el modelo f con el método de integración correspondiente, para obtener el nuevo valor **x1** y **x2** correspondiente a $t + h$ para todas las ecuaciones de estado definidas.
3. Ejecutar el control de paso, en caso de que error obtenido sea mayor a la tolerancia prefijada, o que el paso de integración sea menor que el mínimo definido, se rechaza el paso, se debe recalcular el paso de integración y se vuelve a ejecutar la iteración (se repite el paso hasta que sea aceptado).
4. Si paso es aceptado, se debe recalcular h, se actualiza el tiempo de simulación y se actualiza el valor de $\mathbf{x}$.
5. Por último, se debe chequear, en cada paso, que el h obtenido en el caso de que se acepte el paso, se encuentre dentro de los límites definidos por *hmin* y *hmax*.

2.3. Métodos de integración multipaso

En la sección 2.2, se mostraron métodos de integración que, de una forma u otra, tratan de aproximar la expansión de Taylor de la solución desconocida en torno al instante de tiempo actual. La idea básica era no calcular las derivadas superiores en forma explícita, sino reemplazarlas por distintas evaluaciones de la función f en varios puntos diferentes dentro del paso de integración.

Una desventaja de este enfoque es que cada vez que se comienza un nuevo paso, se deja de lado todas las evaluaciones anteriores de la función f en el paso anterior.

Los métodos multipaso, en lugar de evaluar varias veces la función en un paso para incrementar el orden de la aproximación, tratan de aprovechar las evaluaciones realizadas en pasos anteriores. En tal sentido, estos métodos utilizan como base polinomios de interpolación o de extrapolación de orden alto.

Dentro de las familias de métodos multipaso, se pueden pueden distinguir:

- Fórmulas Explícitas de Adams–Bashforth. Un ejemplo dentro de la familia de los métodos Adams–Bashforth es el método de Adams–Bashforth de tercer orden (AB3):

$$\mathbf{x}(t_{k+1}) = \mathbf{x}(t_k) + \frac{h}{12}\left(23\mathbf{f}_k - 16\mathbf{f}_{k-1} + 5\mathbf{f}_{k-2}\right) \tag{2.14}$$

- Fórmulas Implícitas de Adams–Moulton. En el caso de esta familia de métodos se puede mostrar a modo de ejemplo el caso del algoritmo implícito de Adams–Moulton de tercer orden:

$$\mathbf{x}(t_{k+1}) = \mathbf{x}(t_k) + \frac{h}{12}\left(5\mathbf{f}_{k+1} + 8\mathbf{f}_k - \mathbf{f}_{k-1}\right) \tag{2.15}$$

- Fórmulas de Diferencias Hacia Atrás (BDF). Esta familia de métodos implícitos son aptos para simular sistemas stiff. Presentaremos la fórmula de tercer orden de diferencias hacia atrás (BDF3, por *Backward Difference Formula*).

$$\mathbf{x}_{k+1} = \frac{18}{11}\mathbf{x}_k - \frac{9}{11}\mathbf{x}_{k-1} + \frac{2}{11}\mathbf{x}_{k-2} + \frac{6}{11} \cdot h \cdot \mathbf{f}_{k+1} \tag{2.16}$$

2.3.1. Control de paso

El control de paso en estos métodos no es ni sencillo ni económico, ya que las fórmulas multipaso se basan en considerar que los puntos en el tiempo están equiespaciados. Luego, si se cambia el paso de integración durante la simulación, los puntos en el tiempo no están más equiespaciados y se deben utilizar polinomios de interpolación o extrapolación para poder calcular valores equiespaciados temporalmente y poder así continuar usando los métodos multipaso. El cálculo mediante estos polinomios se debe realizar cada vez que se cambia el paso. Debido al costo que esto implica, en estos métodos se trata de variar lo menos posible el paso.

2.3.2. Inicialización de los métodos

Un problema que hay que resolver en los métodos multipaso es el arranque. Normalmente, se conoce el estado inicial en tiempo t_0, pero no hay datos anteriores disponibles, y no se puede entonces comenzar utilizando métodos multipaso de orden mayor que uno (en el caso implícito).

Para los métodos de tipo ABM y AB la solución que se usa generalmente es comenzar dando los primeros pasos con algoritmos de Runge–Kutta del mismo orden de método multipaso a utilizar para no comenzar con problemas de precisión.

El caso de BDF es un poco más problemático ya que al tratar con sistemas stiff, RK deber utilizar pasos excesivamente pequeños. En este caso lo que se hace es utilizar BRK en el arranque para no tener limitaciones de estabilidad en el arranque.

A modo de ejemplo de ejemplo, definiremos un motor sencillo para métodos de integración multipaso, con paso fijo (h constante).

Algoritmo 2.6: Motor de simulación para métodos multipaso

```
1  Engine (f, ti, tf, x0, h, order, init, method)
2  begin
3    length := (tf − ti)/h;
4    x := x0;
5    k := 1;
6    While (k ≤ order)
7    begin
8      (x_0, ..., x_order−1) := init(f, x, t, h);
9    end
10   k := order + 1;
11   While (k ≤ length)
12   begin
13     x := method(f, x_0, ..., x_order−1, t, h);
14     x_0 := x_1
```

```
15      ⋮
16      x_{order-1} := x
17    end
18  end
```

Donde los parámetros que recibe el motor son los siguientes:

- f El modelo con la definición de las derivadas de las variables de estado.
- it El tiempo inicial de la simulación.
- ft El tiempo final de la simulación.
- **x0** Valores iniciales de las variables de estado.
- h El paso de integración utilizado.
- *order* El orden del método.
- *init* El método de integración utilizado para inicializar la simulación.
- *method* El método de integración utilizado.

El motor debe hacer lo siguiente:

1. Calcular la cantidad de pasos que va a dar el motor (en este caso, el paso de integración es fijo).
2. Obtener los valores iniciales de $\mathbf{x_0}, ..., \mathbf{x_{order-1}}$, utilizando el metodo de inicalización indicado.
3. Luego de obtener los valores iniciales, para el resto de los pasos ($[order + 1, ..., lenght]$), se debe evaluar f con el método correspondiente, para definir el nuevo valor de $\mathbf{x}$.
4. Por último, se debe actualizar la memoria, es decir, la cantidad de valores de $\mathbf{x}$ que el método necesita recordar, esto depende del orden del método.

2.4. Métodos linealmente implícitos

Los métodos linealmente implícitos o semi–implícitos explotan el hecho de que los métodos implícitos aplicados a sistemas lineales pueden implementarse directamente mediante inversión matricial.

Uno de los métodos de este tipo más utilizados es la fórmula de Euler semi–implícita:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot [\mathbf{f}(\mathbf{x_k}, t_k) + \mathcal{J}_{\mathbf{x_k}, t_k} \cdot (\mathbf{x_{k+1}} - \mathbf{x_k})] \tag{2.17}$$

donde

$$\mathcal{J}_{\mathbf{x_k}, t_k} = \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{\mathbf{x_k}, t_k} \tag{2.18}$$

es la matriz Jacobiana evaluada en $(\mathbf{x_k}, t_k)$.

Notar que

$$\mathcal{J}_{\mathbf{x_k}, t_k} \cdot (\mathbf{x_{k+1}} - \mathbf{x_k}) \approx \mathbf{f}(\mathbf{x_{k+1}}, t_{k+1}) - \mathbf{f}(\mathbf{x_k}, t_k) \tag{2.19}$$

y entonces:

$$\mathbf{f}(\mathbf{x_k}, t_k) + \mathcal{J}_{\mathbf{x_k},t_k} \cdot (\mathbf{x_{k+1}} - \mathbf{x_k}) \approx \mathbf{f}(\mathbf{x_{k+1}}, t_{k+1}) \tag{2.20}$$

Es decir, el método de Euler linealmente implícito se aproxima al método de Backward Euler. Más aún, en el caso lineal:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} \tag{2.21}$$

tenemos:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot [\mathbf{A} \cdot \mathbf{x_k} + \mathcal{J}_{\mathbf{x_k},t_k} \cdot (\mathbf{x_{k+1}} - \mathbf{x_k})] = \mathbf{x_k} + h \cdot \mathbf{A} \cdot \mathbf{x_{k+1}} \tag{2.22}$$

que coincide exactamente con Backward Euler.

La Ecuación (2.17) puede reescribirse como:

$$(I - h \cdot \mathcal{J}_{\mathbf{x_k},t_k}) \cdot \mathbf{x_{k+1}} = (I - h \cdot \mathcal{J}_{\mathbf{x_k},t_k}) \cdot \mathbf{x_k} + h \cdot \mathbf{f}(\mathbf{x_k}, t_k) \tag{2.23}$$

lo que muestra que $\mathbf{x_{k+1}}$ puede obtenerse resolviendo un sistema lineal de ecuaciones.

El valor de $\mathbf{x_{k+1}}$ puede también obtenerse como:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot (I - h \cdot \mathcal{J}_{\mathbf{x_k},t_k})^{-1} \cdot \mathbf{f}(\mathbf{x_k}, t_k) \tag{2.24}$$

Esta fórmula es similar a la de Forward Euler, pero difiere en la presencia del término $(I - h \cdot \mathcal{J}_{\mathbf{x_k},t_k})^{-1}$.

Este término implica que el algoritmo debe calcular el Jacobiano en cada paso e invertir una matriz.

2.5. Simulación de sistemas discontinuos

Como se vio en las secciones anteriores de este capítulo, todos los métodos de integración de tiempo discreto se basan, explícita o implícitamente, en expansiones de Taylor. Las trayectorias siempre se aproximan mediante polinomios o mediante funciones racionales en el paso h en torno al tiempo actual t_k.

Esto trae problemas al tratar con modelos discontinuos, ya que los polinomios nunca exhiben discontinuidades, y las funciones racionales sólo tienen polos aislados, pero no discontinuidades finitas. Entonces, si un algoritmo de integración trata de integrar a través de una discontinuidad, sin dudas va a tener problemas.

Dado que el paso h es finito, el algoritmo de integración no reconoce una discontinuidad como tal. Lo único que nota es que la trayectoria de pronto cambia su comportamiento y actúa como si hubiera un gradiente muy grande.

El siguiente ejemplo consiste en una pelota rebotando contra el piso.

$$\dot{x}(t) = v(t) \tag{2.25a}$$

$$\dot{v}(t) = -g - s_w(t) \cdot \frac{1}{m}(k \cdot x(t) + b \cdot v(t)) \tag{2.25b}$$

donde

$$s_w = \begin{cases} 0 & \text{si } x(t) > 0 \\ 1 & \text{en otro caso} \end{cases} \tag{2.26}$$

En este modelo, se considera que cuando $x(t) > 0$ la pelotita está en el aire y responde a una ecuación de caída libre ($s_w = 0$). Cuando la pelotita entra en contacto con el piso, en cambio, sigue un modelo *masa–resorte–amortiguador*, lo que produce el rebote.

Los parámetros usados en el mismo son: $m = 1$, $b = 30$, $k = 1 \times 10^6$ y $g = 9{,}81$.

Simulando este modelo varias veces utilizando el método RK4, durante 5 segundos a partir de la condición inicial $x(0) = 1$, $v(0) = 0$. Se comenzaron a tener resultados *decentes* con un paso de integración $h = 0{,}002$. En la Fig 2.3 se muestran los resultados de la simulación con pasos $h = 0{,}002$, $h = 0{,}001$ y $h = 0{,}0005$.

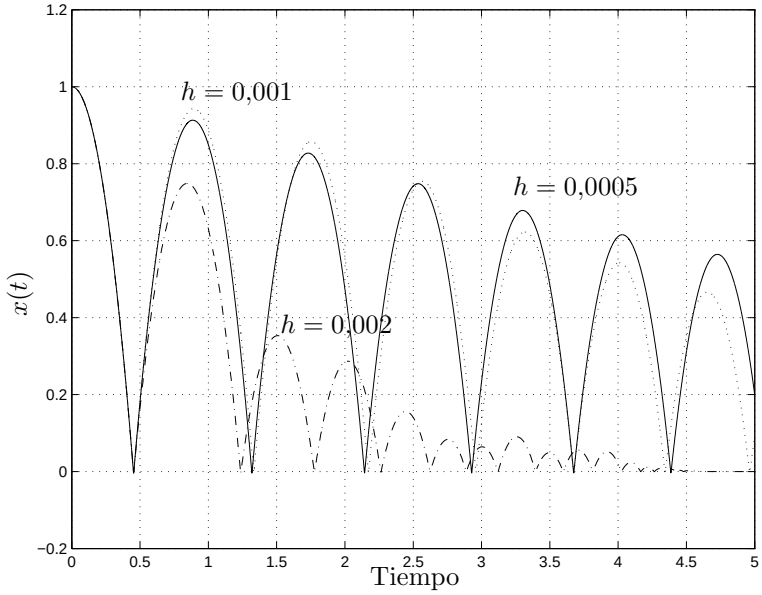

Figura 2.3: Simulación con RK4 de la pelotita rebotando.

Mirando el comienzo de la simulación, no hay error apreciable hasta el primer pique y a partir del mismo las soluciones difieren notablemente entre sí. Además, en la simulación con paso $h = 0{,}002$ luego del séptimo pique (apróximadamente a los 3 segundos) la pelota gana más altura de la que tenía anteriormente lo cual es evidentemente erróneo. El problema tiene que ver con la discontinuidad haciendo que no se pueda confiar en los resultados de simulación usando paso fijo.

En la figura 2.4 se muestran los resultados al simular el sistema utilizando un método de paso variable, en este caso un método RK23 (utiliza un RK de segundo orden y para controlar el paso utiliza el mismo método implementado con dos semipasos de $h/2$). El método es de segundo orden, y el término del error es de tercer orden. Para la simulación se utilizaron las tolerancias relativas 10^{-3}, 10^{-4} y 10^{-5}.

Si bien algunos piques se resuelven *bien* (al menos el método hace lo mismo con las tres tolerancias), en otros piques el error se torna inaceptable.

El problema de las simulaciones realizadas es que en algunos pasos los méto-

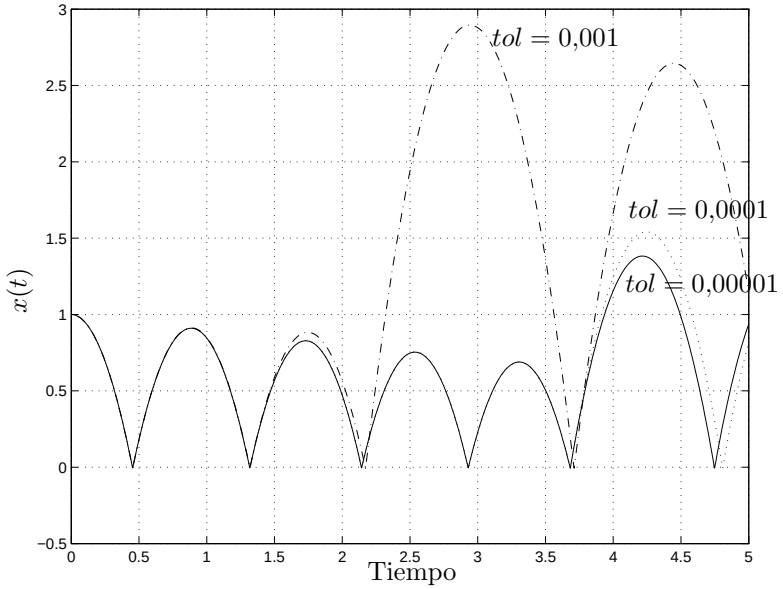

Figura 2.4: Simulación con RK23 de la pelotita rebotando.

dos integraban a través de una discontinuidad. Es decir, calculaban como si tuvieran una función continua entre t_k y $t_k + h$, pero en realidad en el medio (en algún punto t^* en dicho intervalo) ocurre una discontinuidad.

La forma de evitar esto es en principio muy simple: lo que se necesita es un método de paso variable que dé un paso exactamente en el instante t^* en el que ocurre la discontinuidad. De esa forma, siempre se está integrando una función continua antes de t^* y otra función continua después de t^*. Este es el principio básico de todos los métodos que realizan *manejo de discontinuidades*.

2.5.1. Eventos temporales

Se denomina *Eventos temporales* a las discontinuidades de las cuales se sabe con cierta anticipación el tiempo de ocurrencia de las mismas. La forma de tratar eventos temporales es muy sencilla. Dado que se conoce cuando ocurrirán, simplemente se le debe avisar al algoritmo de integración el tiempo de ocurrencia de los mismos. El algoritmo deber entonces *agendar* dichos eventos y cada vez que de un paso debe tener cuidado de no saltearse ninguno. Cada vez que el paso de integración h a utilizar sea mayor que el tiempo que falta para el siguiente evento, deber utilizar un paso de integración que sea exactamente igual al tiempo para dicho evento.

Notar que ninguna discontinuidad tiene lugar mientras el evento es localizado. La discontinuidad no se debe codificar directamente en el modelo, sino sólo la condición para que esta ocurra. Así, las trayectorias vistas por el método de integración serán perfectamente continuas.

Una vez que el tiempo del siguiente evento ha sido localizado, la simulación continua se debe detener por un momento y se debe actualizar la parte discreta del modelo.

De esta manera, una corrida de simulación de un modelo discontinuo puede interpretarse como una secuencia de varias corridas de simulaciones continuas separadas mediante eventos discretos.

2.5.2. Eventos de estado

Muy frecuentemente, el tiempo de ocurrencia de una discontinuidad no se conoce de antemano. Por ejemplo, en la pelotita rebotando, no se conoce el tiempo en el que se producen los piques. Todo lo que se sabe es que los piques se dan cuando la altura es cero. Es decir, se conoce la *condición del evento* en lugar del *tiempo del evento*.

Las condiciones de los eventos se suelen especificar como *funciones de cruce por cero*, que son funciones que dependen de las variables de estado del sistema y que se hacen cero cuando ocurre una discontinuidad. En el caso de la pelotita rebotando, una posible función de cruce por cero es $\mathcal{F}_0(x, v) = x$.

Se dice que ocurre un *evento de estado* cada vez que una función de cruce por cero cruza efectivamente por cero. En muchos casos, puede haber varias funciones de cruce por cero.

Las funciones de cruce por cero deben evaluarse continuamente durante la simulación. Las variables que resultan de dichas funciones normalmente se colocan en un vector y deben ser monitoreadas. Si una de ellas pasa a través de cero, debe comenzarse una iteración para determinar el tiempo de ocurrencia del cruce por cero con una precisión predeterminada.

Así, cuando una condición de evento es detectada durante la ejecución de un paso de integración, debe actuarse sobre el mecanismo de control de paso del algoritmo para forzar una iteración hacia el primer instante en el que se produjo el cruce por cero durante el paso actual.

Una vez localizado este tiempo, la idea es muy similar a la del tratamiento de eventos temporales.

En el caso de tratar con modelos discontínuos como los descriptos en esta sección, se debe agregar al motor de simulación la capacidad de controlar las condiciones de los eventos definidos en el sistema, además de incorporar las variables discretas definidas en el sistema. El motor se puede definir de la siguiente manera:

Algoritmo 2.7: Motor de simulación para sistemas híbridos

```
1  Engine (f, it, ft, x0, d0, ev_zc, ev_handler, method, tol, hmin, hmax)
2  begin
3     t := it;
4     x := x0;
5     d := d0;
6     k := 0;
7     h := hmin;
8     While  (t ≤ ft)
9     begin
10       k := k + 1;
11       [x1, x2] :=method(f, x, d, t, h);
12       err := |x1 − x2|/max(|x2|, ε);
13       if  (err > tol  and  h > hmin)  then
14       begin
15          h := Obtener_Nuevo_Paso(tol, err, h)
```

```
16          k := k − 1;
17        else
18          zc_tmp := ev_zc;
19          zc := ev_zc(x, d, t);
20          cross :=Chequear_Signo(zc, zc_tmp)
21          if (cross ≠ ∅) then
22          begin
23            ev_handler(x, d, t);
24            t* :=Detectar_Discontinuidad(ev_zc, x, d);
25            t := t*;
26          else
27            ev_zc := zc_tmp;
28            t := t + h;
29            x := x2;
30            h := Obtener_Nuevo_Paso(tol, err, h)
31          end
32        end
33    end
34    if (h < hmin) then
35    begin
36      h := hmin;
37    end
38    if (h > hmax) then
39    begin
40      h := hmax;
41    end
42  end
```

Donde, además de los parámetros ya definidos en el Algoritmo 2.5, los parámetros adicionales que recibe el motor son los siguientes:

- **d0** Valores iniciales de las variables discretas.
- **ev_zc** Definición de las funciones de cruce por cero correspondientes a los eventos.
- **ev_handler** Definición de funciones de los handlers correspondientes a los eventos.

En cada paso, el motor debe hacer lo siguiente:

1. Controlar el tiempo de simulación.
2. Evaluar el modelo f con el método de integración correspondiente, para obtener el nuevo valor **x1** y **x2** correspondiente a $t + h$.
3. Ejecutar el control de paso, en caso de que el error obtenido sea mayor a la tolerancia prefijada, o que el paso de integración sea menor que el mínimo definido, se rechaza el paso, se debe recalcular el paso de integración y se vuelve a ejecutar la iteración (se repite el paso hasta que sea aceptado).
4. Si el paso es aceptado, se debe evaluar las funciones de cruce **ev_fc** correspondientes a cada uno de los eventos definidos en el modelo para $t + h$.
5. Verificar el cambio de signo en cada una de las funciones de cruce.

6. En caso de existir un cambio de signo en una o más funciones de cruce (lo cuál indica que pasamos por lo menos por una discontinuidad en el sistema), se debe detectar el tiempo exacto mínimo de todas las discontinuidades t^*, ejecutar el handler correspondiente, rechazar el paso de simulación, y volver a arrancar la simulación desde el nuevo tiempo.

7. Si no se detecta ningún cambio, se debe recalcular h, se actualiza el tiempo de simulación, se actualiza el valor de $\mathbf{x}$ y de **ev_zc** .

8. Por último, se debe chequear, en cada paso, que el h obtenido en el caso de que se acepte el paso, se encuentre dentro de los límites definidos por $hmin$ y $hmax$.

En el Modelo 2.8 vemos una posible formulación del modelo de la pelotita picando definido mediante el sistema Ec. (2.25), que puede ser simulada con el motor definido en el Algoritmo 2.7. Donde, como vimos en esta sección, debemos dar por separado, la representación de las ecuaciones de estado, y la función de cruce y el handler correspondiente al evento del sistema híbrido.

Modelo 2.8: Modelo pelotita picando

```
1  pelotita_picando=(x, d, t)
2  begin
3     m := 1;
4     k := 1e6;
5     b := 30;
6     g := 9,8;
7     dx1 = x2;
8     dx2 = −g − d ∗ (b/m ∗ x2 + k/m ∗ x1)
9     return [dx1, dx2];
10 end
11
12 fcross(x, d, t)
13 begin
14    return x1;
15 end
16
17 handler(x, d, t)
18 begin
19    return 1 − d1;
20 end
```

Capítulo 3

Métodos de integración por cuantificación

Como se vio en el capítulo anterior, los sistemas continuos se representan generalmente mediante ecuaciones diferenciales ordinarias (ODEs).

Para poder simular estos modelos, los métodos de integración tradicionales, realizan una discretización del tiempo de manera de transformar el modelo de tiempo continuo en un modelo de ecuaciones en diferencias 'equivalente'. De este modo, la solución del modelo de tiempo discreto en los instantes de muestreo se aproxima a la solución del modelo original en dichos instantes.

Si bien los métodos de integración tradicionales son los más difundidos y han dado respuesta a muchos problemas de simulación, existen también muchos modelos para los cuales la solución brindada por los mismos no ha sido del todo satisfactoria desde el punto de vista de eficiencia. Además, existen modelos de sistemas que, según como se formulen, pueden llegar a ser imposibles de simular a 'ciegas', entendiéndose por esto, cargando simplemente el modelo matemático en un simulador sin tener mucha noción del comportamiento del modelo.

A fines de los 90's se comenzó a desarrollar una metodología alternativa a la clásica discretización temporal para poder aproximar los sistemas continuos. En la misma, en lugar de discretizar el tiempo, se cuantifican las variables de estado manteniendo continua la variable tiempo. De este modo, se verá que en lugar de obtenerse un modelo de tiempo discreto equivalente, se llega a un modelo de eventos discretos equivalente. Y que esta discretización puede representarse fácilmente mediante el formalismo DEVS.

En este capítulo, basado en [12], se presentarán los principios de esta metodología y se dará una descripción de los métodos de integración basados en la cuantificación de los estados que se implementaron para este trabajo.

3.1. Ejemplo introductorio de discretización espacial

Un oscilador armónico puede ser representado mediante el modelo de un sistema continuo de segundo orden:

$$\begin{aligned} \dot{x}_{a_1}(t) &= x_{a_2}(t) \\ \dot{x}_{a_2}(t) &= -x_{a_1}(t). \end{aligned} \tag{3.1}$$

Si se saben las condiciones iniciales $x_{a_1}(t_0)$ y $x_{a_2}(t_0)$, dado que el sistema es lineal, es muy fácil encontrar la solución analítica del modelo, siendo ésta $x_{a_i}(t) = c_i \sin(t) + d_i \cos(t)$ con c_i y d_i constantes.

Veamos ahora que ocurre si se modifica el sistema (3.1) de la siguiente manera:

$$\begin{aligned} \dot{x}_1(t) &= \text{floor}[x_2(t)] \triangleq q_2(t) \\ \dot{x}_2(t) &= -\text{floor}[x_1(t)] \triangleq -q_1(t) \end{aligned} \tag{3.2}$$

donde floor(x_i) es una función que da como resultado el valor entero más cercano a x_i y que es a su vez menor o igual a x_i.

A pesar de que este nuevo sistema es no lineal y discontinuo, se lo puede simular fácilmente. Tomemos como condiciones iniciales $x_1(0) = 4{,}5$ y $x_2(0) = 0{,}5$.

Con estos valores iniciales se tiene que $q_1(0) = 4$ y $q_2(0) = 0$ y se mantienen en esos valores hasta que $x_1(t)$ o $x_2(t)$ cruce a través de alguno de los valores enteros más próximos a ella. En consecuencia, $\dot{x}_1(0) = 0$ y $\dot{x}_2(0) = -4$ por lo que x_1 se mantiene constante y x_2 decrece con pendiente -4.

Luego de $0{,}5/4 = 0{,}125$ segundos (instante $t_1 = 0{,}125$), x_2 cruza por 0 y q_2 toma el valor -1. Entonces, cambia la pendiente de x_1 tomando el valor $\dot{x}_1(t_1^+) = -1$

Luego, x_2 cruza por -1 en el instante $t_2 = t_1 + 1/4$, y q_2 toma el valor -2. En ese instante, $x_1(t_2) = 4{,}5 - 1/4 = 4{,}25$ y $\dot{x}_1(t_2^+) = -2$.

El próximo cambio ocurre cuando x_1 cruza por 4 en el instante de tiempo $t_3 = t_2 + 0{,}25/2$. Luego, $q_1(t_3^+) = 3$ y la pendiente de x_2 pasa a ser -3. Continuando con el análisis de la misma manera, se obtienen los resultados de la *simulación* mostrados en las figuras 3.1–3.2. Los resultados son muy similares a la solución analítica del sistema original3.1.

Cuando se reemplaza x_i por $q_i = \text{floor}(x_i)$ en un sistema como el de la Ec.(3.1), se obtiene una aproximación del problema original pero que puede ser resuelta en un número finito de pasos conservando un comportamiento similar al del sistema original.

Mas aún, se puede asociar la solución del sistema (3.2) con el comportamiento de un sistema de eventos discretos (ya que realizan un número finito de cambios), pero no al de un sistema de tiempo discreto. En este caso, los eventos corresponden a los cruces de las variables de estado por los niveles de discretización de las mismas, cuando ocurren pueden provocar cambios en las derivadas de los estados, que conducen a una reprogramación del tiempo en que ocurre el próximo cruce de nivel.

Para poder ver como se puede generalizar esta idea, se deberá introducir antes algunas herramientas que permiten representar y simular sistemas como el de la Ec.(3.2).

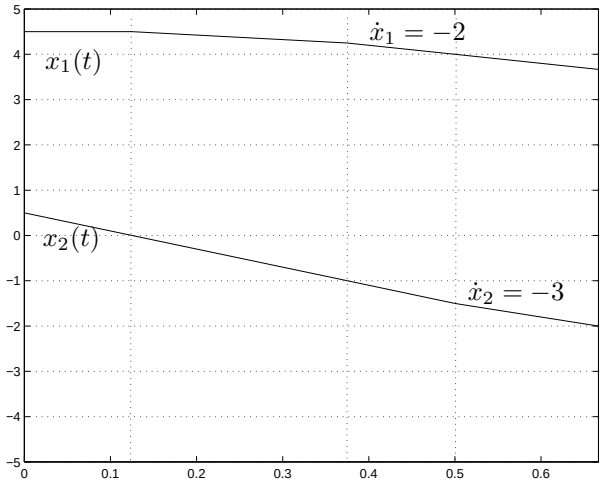

Figura 3.1: Simulación de las trayectorias del sistema de la Ec.(3.2) (Comienzo).

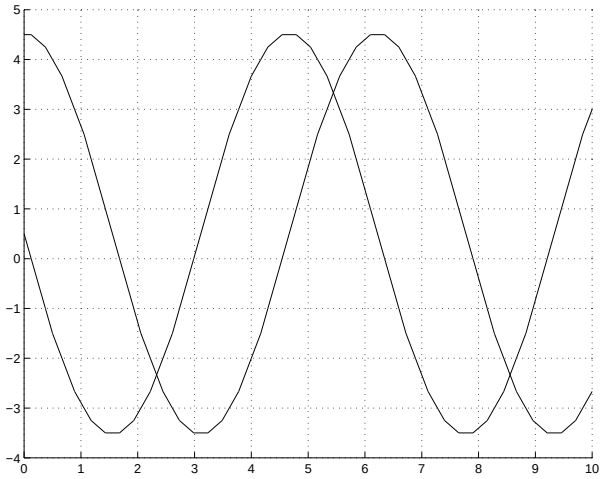

Figura 3.2: Simulación de las trayectorias del sistema de la Ec.(3.2).

3.2. Sistemas de eventos discretos y DEVS

Todos los métodos de tiempo discreto aproximan las ecuaciones diferenciales por sistemas de tiempo discreto (ecuaciones en diferencia) de la forma:

$$\mathbf{x}(t_{k+1}) = \mathbf{f}(\mathbf{x}(t_k), t_k) \tag{3.3}$$

Con la nueva idea de cuantificar las variables de estado, sin embargo, se obtienen sistemas que son discretos (ya que realizan un número finito de cambios), pero no son de tiempo discreto.

Se verá que esta nueva manera de aproximar las ecuaciones nos lleva a *sistemas de eventos discretos*, dentro del formalismo DEVS.

DEVS, cuyas siglas provienen de *Discrete EVent System specification*, fue

introducido por Bernard Zeigler a mediados de la década de 1970. DEVS permite representar todos los sistemas cuyo comportamiento entrada/salida pueda describirse mediante secuencias de eventos.

En este contexto, un *evento* es la representación de un cambio instantáneo en alguna parte del sistema. Como tal, puede caracterizarse mediante un valor y un tiempo de ocurrencia. El valor puede ser un número, un vector, una palabra, o en general, un elemento de algún conjunto.

La trayectoria definida por una secuencia de eventos toma el valor ϕ (o *No Event*) en casi todos los instantes de tiempo, excepto en los instantes en los que hay eventos. En estos instantes, la trayectoria toma el valor correspondiente al evento. La Figura 3.3 muestra una trayectoria de eventos que toma los valores x_2 en el tiempo t_1, luego toma el valor x_3 en t_2, etc.

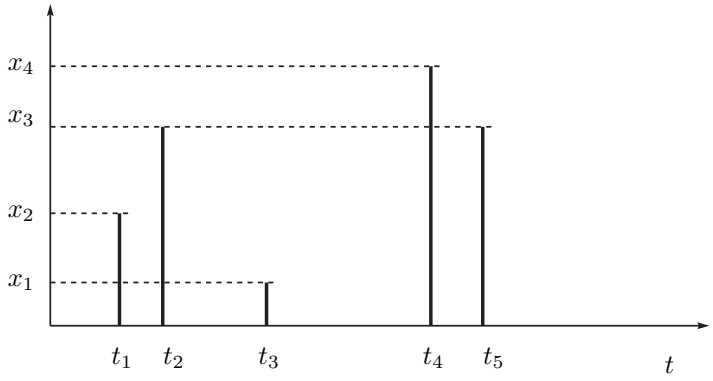

Figura 3.3: Trayectoria de eventos.

Un modelo DEVS procesa una trayectoria de eventos y, de acuerdo a dicha trayectoria y a sus propias condiciones iniciales, provoca una trayectoria de eventos de salida. Este comportamiento entrada/salida se muestra en la Figura 3.4.

Figura 3.4: Comportamiento Entrada/Salida de un modelo DEVS.

El comportamiento de un modelo DEVS *atómico* se explicita mediante distintas funciones y conjuntos, definidos en la siguiente estructura:

$$M = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta)$$

donde:

- X es el conjunto de los valores de los eventos de entrada, es decir el conjunto de todos los valores que un evento de entrada puede tomar;
- Y es el conjunto de valores de los eventos de salida;
- S es el conjunto de los valores del estado;
- δ_{int}, δ_{ext}, λ y ta son funciones que definen la dinámica del sistema.

La Figura 3.5 nos muestra el comportamiento de un modelo DEVS.

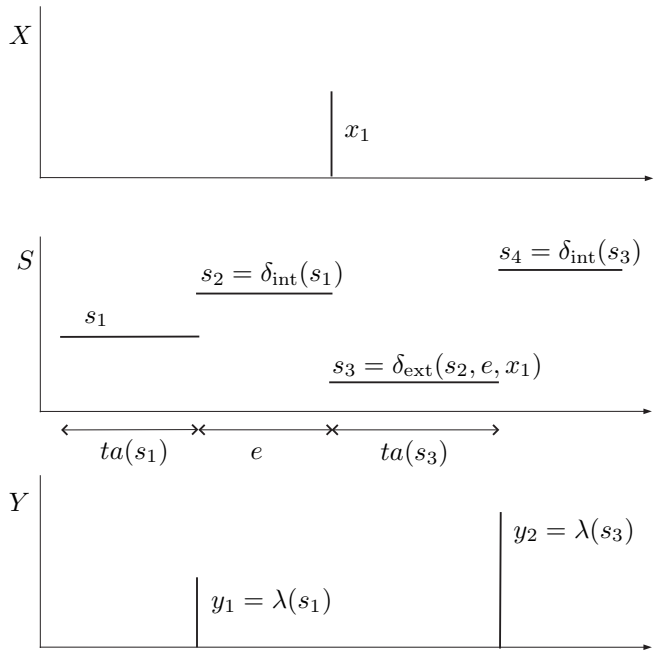

Figura 3.5: Trayectorias de un modelo DEVS.

Cada estado posible s ($s \in S$) tiene asociado un *tiempo de avance* calculado por la *función de avance de tiempo* $ta(s)$ ($ta(s) : S \to \Re_0^+$). El avance de tiempo es un número real no negativo, que determina cuánto tiempo debe permanecer el sistema en un estado en ausencia de eventos de entrada.

Luego, si el estado toma el valor s_1 en el instante t_1, tras $ta(s_1)$ unidades de tiempo (o sea, en el instante $t_1+ta(s_1)$), el sistema realiza una *transición interna*, alcanzando el nuevo estado s_2. El nuevo estado se calcula como $s_2 = \delta_{\text{int}}(s_1)$. La Función δ_{int} ($\delta_{\text{int}} : S \to S$) se denomina *función de transición interna*.

Cuando el estado cambia su valor desde s_1 a s_2, se produce un evento de salida con el valor $y_1 = \lambda(s_1)$. La función λ ($\lambda : S \to Y$) se denomina *función de salida*. De esta manera, las funciones ta, δ_{int} y λ definen el comportamiento autónomo de un modelo DEVS.

Cuando llega un evento de entrada, el estado cambia instantáneamente. El nuevo valor del estado depende no sólo del valor del evento de entrada, sino también del valor anterior del estado y del tiempo transcurrido desde la última transición. Si el sistema toma el valor s_2 en el instante t_2, y llega un evento

en el instante $t_2 + e < ta(s_2)$ con valor x_1, el nuevo estado se calcula como $s_3 = \delta_{\text{ext}}(s_2, e, x_1)$. En este caso, se dice que el sistema realiza una *transición externa*. La función δ_{ext} ($\delta_{\text{ext}} : S \times \Re_0^+ \times X \to S$) se denomina *función de transición externa*. Durante la transición externa, no se produce ningún evento de salida.

En muchos casos, además, los conjuntos X e Y (de donde toman los valores los eventos de entrada y salida) se forman por el producto cartesiano de un conjunto arbitrario (que contiene los valores de los eventos propiamente dichos) y un conjunto que contiene un número finito de *puertos* de entrada y salida respectivamente (luego se verá que estos puertos se utilizarán para acoplar modelos DEVS atómicos).

Sea por ejemplo un sistema que calcula una función estática $f(u_0, u_1)$, donde u_0 y u_1 son trayectorias seccionalmente constantes.

Una trayectoria seccionalmente constante puede ser representada mediante una secuencia de eventos si se relacion cada evento con un cambio en el valor de la trayectoria. Utilizando esta idea, se puede construir el siguiente modelo DEVS atómico:

$$
\begin{aligned}
&M_F = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta), \text{ donde} \\
&X = \Re \times \{0, 1\} \\
&Y = \Re \times \{0\} \\
&S = \Re^2 \times \Re_0^+ \\
&\delta_{\text{int}}(s) = \delta_{\text{int}}(u_0, u_1, \sigma) = (u_0, u_1, \infty) \\
&\delta_{\text{ext}}(s, e, x) = \delta_{\text{ext}}(u_0, u_1, \sigma, e, x_v, p) = \tilde{s} \\
&\lambda(s) = \lambda(u_0, u_1, \sigma) = (f(u_0, u_1), 0) \\
&ta(s) = ta(u_0, u_1, \sigma) = \sigma
\end{aligned}
$$

con:

$$
\tilde{s} = \begin{cases} (x_v, u_1, 0) & \text{si } p = 0 \\ (u_0, x_v, 0) & \text{en otro caso} \end{cases}
$$

En este modelo, se incluyó una variable σ en el estado s que coincide con la función ta. Esto se suele hacer siempre, ya que así se hace más fácil la obtención de un modelo DEVS.

Como se dijo anteriormente, cada evento de entrada y de salida incluye un número entero que indica el correspondiente puerto de entrada o salida. En los eventos de entrada, el puerto p puede ser 0 o 1 (es decir, hay dos puertos de entrada, uno para u_0 y el otro para u_1). En los eventos de salida, hay un sólo puerto de salida (0).

3.2.1. Modelos DEVS acoplados

DEVS es un formalismo muy general, que puede utilizarse para describir sistemas muy complejos. Sin embargo, representar un sistema muy complejo utilizando funciones de transición y avance de tiempo es muy difícil ya que para describir dichas funciones se debe tener en cuenta todas las posibles situaciones en el sistema.

Los sistemas complejos generalmente se piensan como el acoplamiento de sistemas más simples. A través del acoplamiento, los eventos de salida de unos subsistemas se convierten en eventos de entrada de otros subsistemas. La teoría de DEVS garantiza que el modelo resultante de acoplar varios modelos DEVS atómicos es equivalente a un nuevo modelo DEVS atómico, es decir, DEVS es cerrado frente al acoplamiento (clausura del acoplamiento). Esto permite el acoplamiento jerárquico de modelos DEVS, o sea, la utilización de modelos acoplados como si fueran modelos atómicos que a su vez pueden acoplarse con otros modelos atómicos o acoplados.

Existen diversas formas de acoplar modelos DEVS. Una de ellas, y es la que utiliza el software PowerDEVS, es el acoplamiento mediante puertos de entrada/salida.

La Figura 3.6 muestra un modelo DEVS acoplado N, resultado de acoplar los modelos M_a y M_b. De acuerdo a la propiedad de clausura de DEVS, el modelo N puede usarse de la misma forma que si fuera un modelo atómico, y puede ser acoplado con otros modelos atómicos y/o acoplados.

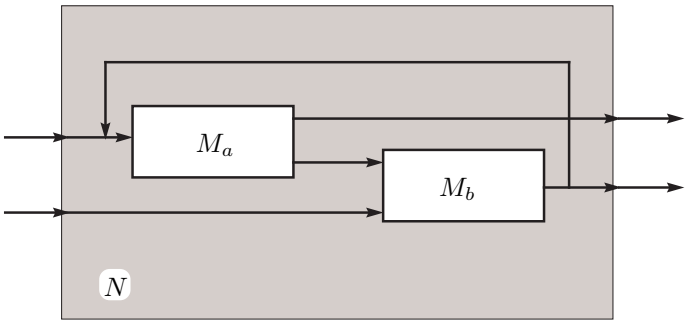

Figura 3.6: Modelo DEVS acoplado.

Para poder mostrar las conexiones entre los distintos modelos DEVS acoplados, se utilizarán *Diagramas en Bloques* similares al mostrado en la Figura 3.6.

3.2.2. Simulación de modelos DEVS y PowerDEVS

Una de las características más importantes de DEVS es su facilidad para implementar simulaciones. Un modelo DEVS puede simularse con un programa ad–hoc escrito en cualquier lenguaje. De hecho, la simulación de un modelo DEVS no es mucho más complicada que la de un modelo de tiempo discreto.

Un algoritmo básico que puede utilizarse para simular un modelo DEVS acoplado puede describirse por los siguientes pasos:

1. Identificamos el modelo atómico que, de acuerdo a su tiempo de avance y al tiempo transcurrido, deba ser el próximo en realizar la transición interna. Llamamos d^* a este modelo y t_n al tiempo de dicha transición.

2. Avanzamos el reloj de la simulación t a $t = t_n$, y ejecutamos la función de transición interna del modelo d^*.

3. Propagamos el evento de salida provocado por d^* a todos los modelos conectados al puerto de salida y ejecutamos las funciones de transición externas correspondientes. Luego, volvemos al paso 1.

Aunque, la implementación de una simulación de DEVS es muy simple, en general los modelos que se utilizan en la práctica están compuestos por muchos subsistemas y, hacer un programa ad–hoc de todos estos modelos suele tornarse muy trabajoso.

Utilizaremos la herramienta de software denominada PowerDEVS para la simulación de los modelos DEVS. Toda la familia de métodos QSS está implementada en PowerDEVS, como se describió en 1. A pesar de ser un simulador de DEVS de propósito general, fue concebido especialmente para facilitar la simulación de sistemas híbridos basados en los métodos QSS [3].

PowerDEVS tiene una interfase gráfica similar a Simulink que permite editar diagrama de bloques. A un nivel inferior, cada bloque tiene una descripción en lenguaje C++ del atómico DEVS que puede ser editada (usando la herramienta para edición de atómicos de PowerDEVS). Además, la distribución del mismo cuenta con librerías que tienen todos los bloques necesarios para modelar sistemas continuos e híbridos (integradores, funciones matemáticas, bloques para manejo de discontinuidades, fuentes, etc.). En consecuencia, un usuario puede modelar y simular usando los métodos QSS sin necesidad de saber nada de DEVS, simplemente como si modelara o simulara en Simulink.

En las figuras 3.7 y 3.8 pueden verse dos capturas de pantalla en las que se ve la librera de bloques continuos de PowerDEVS y el modelo de un motor de corriente continua con un control de velocidad implementado a través de PWM con la ventana de simulación.

PowerDEVS puede intercambiar parámetros con Scilab durante la simulación del mismo modo que lo hacen Simulink y Matlab. Esta interacción permite explotar toda herramientas de procesamiento de datos, visualización, matemáticas y de manipulación de matrices de Scilab, dándole a su vez al usuario un espacio de trabajo interactivo.

3.2.3. DEVS y simulación de sistemas continuos

En la Sección 3.2, se vio que las trayectorias seccionalmente constantes podían ser representadas mediante secuencias de eventos. Esta simple idea constituye en realidad la base del uso de DEVS en la simulación de sistemas continuos.

En esa sección se vio que un modelo DEVS puede representar el comportamiento de una función estática con una entrada seccionalmente constante. En los sistemas continuos las trayectorias no tienen esta forma, pero pueden ser alterados para que las trayectorias sean así. De hecho, esto es lo que se hizo con el sistema de la Ec.(3.1), donde se utilizó la función 'floor' para convertirlo en el sistema de la Ec.(3.2).

En este ejemplo, se puede dividir la Ec.(3.2) de la siguiente forma:

$$\dot{x}_1(t) = q_2(t) \tag{3.4a}$$
$$\dot{x}_2(t) = d_1(t) \tag{3.4b}$$
$$q_i(t) = \text{floor}[x_i(t)] \tag{3.4c}$$

y:

$$d_1(t) = -q_1(t) \tag{3.5}$$

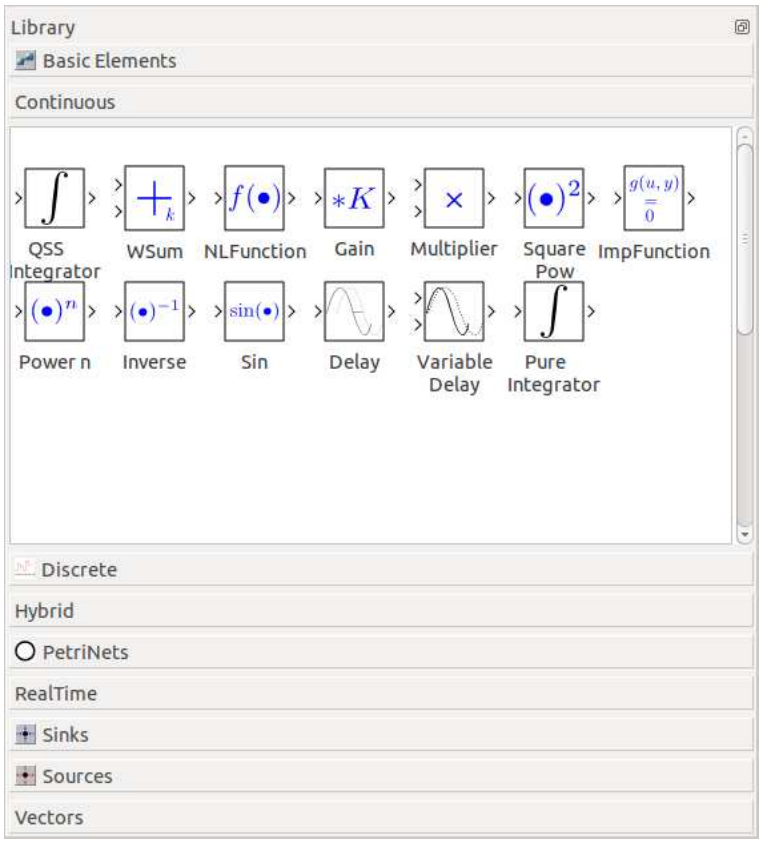

Figura 3.7: Librerías de PowerDEVS.

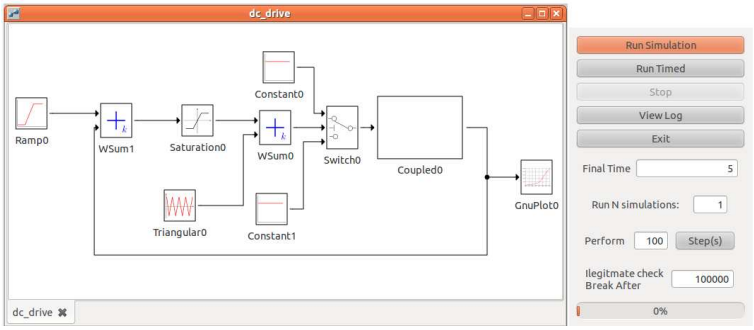

Figura 3.8: Modelo de un motor con control de Velocidad en PowerDEVS

Se puede representar este sistema usando el Diagrama en Bloques de la Figura 3.9.

Como se dijo anteriormente, el subsistema F_1 (ecuación (3.5)) -siendo una ecuación estática- puede ser representado por la función M_F presentada en la sección 3.2.

Cada subsistema QI_i integra una trayectoria de entrada $\dot{x}_i(t)$ seccionalmente

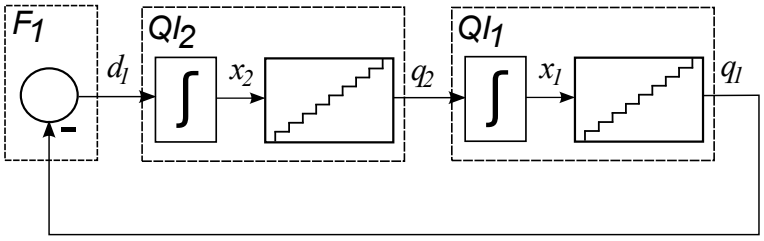

Figura 3.9: Representación en diagrama en Bloques de las Ecs.(3.4)-(3.5).

constante para calcular la trayectoria seccionalmente lineal del estado $x_i(t)$ y genera una trayectoria de salida cuantificada $q_i(t) = \text{floor}[x_i(t)]$ seccionalmente constante. Esta salida cambia solamente cuando la trayectoria de estado $x_i(t)$ seccionalmente lineal cruza algún nivel de discretización.

Este comportamiento también puede ser traducido fácilmente en un modelo DEVS.

$$
\begin{aligned}
&M_{QI} = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta), \text{ donde} \\
&X = Y = \Re \times \mathbb{N} \\
&S = \Re^2 \times \mathbb{Z} \times \Re_0^+ \\
&\delta_{\text{int}}(s) = \delta_{\text{int}}(x, d_x, q, \sigma) = (x + \sigma \cdot d_x, d_x, q + \text{sign}(d_x), \frac{1}{|d_x|}) \\
&\delta_{\text{ext}}(s, e, x) = \delta_{\text{ext}}(x, d_x, q, \sigma, e, x_v, p) = (x + e \cdot d_x, x_v, q, \tilde{\sigma}) \\
&\lambda(s) = \lambda(x, d_x, q, \sigma) = (q + \text{sign}(d_x), 0) \\
&ta(s) = ta(x, d_x, q, \sigma) = \sigma
\end{aligned}
$$

con:

$$
\tilde{\sigma} = \begin{cases} \frac{q+1-x}{x_v} & \text{si } x_v > 0 \\ \frac{q-x}{x_v} & \text{si } x_v < 0 \\ \infty & \text{en otro caso} \end{cases}
$$

El subsistema QI_j formado por un integrador junto con el bloque en forma de escalera (cuantificador) representa las ecuaciones

$$
\begin{aligned}
\dot{x}_i &= q_i \\
q_i &= \text{floor}[x_i]
\end{aligned}
$$

y es lo que Zeigler llamó *integrador cuantificado*. Aquí, la función 'floor' actúa como una *función de cuantificación*. En general, una función de cuantificación mapea un dominio continuo de números reales en un conjunto discreto de los reales.

Un sistema que relacione su entrada y su salida mediante cualquier tipo de función de cuantificación ser entonces llamado *cuantificador*. En el caso mostrado, el bloque con forma de escalera es un caso particular de cuantificador con cuantificación uniforme.

Un integrador cuantificado es entonces un integrador concatenado con un cuantificador.

Luego, si se tiene un sistema general invariante en el tiempo [1]:

$$\begin{array}{rcl} \dot{x}_{a_1} & = & f_1(x_{a_1}, x_{a_2}, \cdots, x_{a_n}, u_1, \cdots, u_m) \\ \dot{x}_{a_2} & = & f_2(x_{a_1}, x_{a_2}, \cdots, x_{a_n}, u_1, \cdots, u_m) \\ & \vdots & \\ \dot{x}_{a_n} & = & f_n(x_{a_1}, x_{a_2}, \cdots, x_{a_n}, u_1, \cdots, u_m) \end{array} \tag{3.6}$$

con trayectorias de entrada seccionalmente constantes $u_j(t)$, puede transformarse en:

$$\begin{array}{rcl} \dot{x}_1 & = & f_1(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m) \\ \dot{x}_2 & = & f_2(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m) \\ & \vdots & \\ \dot{x}_n & = & f_n(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m) \end{array} \tag{3.7}$$

donde $q_i(t)$ se relaciona con $x_i(t)$ mediante alguna función de cuantificación.

Las variables q_i se llaman *variables cuantificadas*. Este sistema de ecuaciones puede representarse mediante el diagrama de bloques de la Figura 3.10, donde **q** y **u** son los vectores formados por las variables cuantificadas y por las variables de entrada respectivamente.

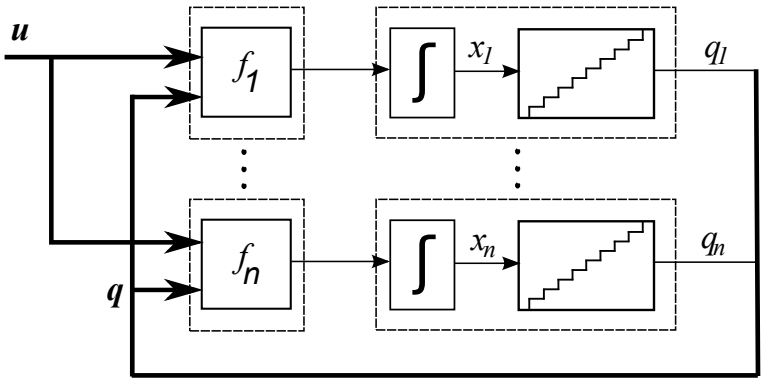

Figura 3.10: Diagrama en Bloques de la Ec.(3.7).

Cada subsistema en la Figura 3.10 puede representarse exactamente por un modelo DEVS, ya que todos los subsistemas son funciones estáticas o integradores cuantificados. Estos modelos DEVS pueden acoplarse, y, debido a la clausura bajo acoplamiento, el sistema completo formar también un modelo DEVS.

Entonces, cuando un sistema se modifica agregando cuantificadores en la salida de todos los integradores, el sistema resultante es equivalente a un modelo DEVS acoplado que puede simularse, siempre y cuando todas las entradas sean seccionalmente constantes.

Esta idea constituye la primer aproximación a un método para simular sistemas continuos mediante eventos discretos.

[1] Utilizando x_a para referirse a las variables de estado del sistema original, tal que $x_a(t)$ es la solución analítica.

Desafortunadamente, esta idea no funciona ....

Este método conduce, en la mayor parte de los casos, a un modelo DEVS *ilegítimo*, es decir, un modelo que realiza un número infinito de transiciones en un intervalo acotado de tiempo. De tal forma, la simulación no puede avanzar luego de cierto tiempo. Por ejemplo, dada la siguiente ecuación diferencial de primer orden:

$$\dot{x}(t) = -x(t) - 0{,}5 \tag{3.8}$$

con condiciones iniciales $x(0) = 0$. Si se procede, como se describió anteriormente, reemplazando $x(t)$ por $q(t) = \text{floor}[x(t)]$ en el lado derecho de (3.8) se obtiene

$$\dot{x}(t) = -q(t) - 0{,}5 \tag{3.9}$$

Si se simula este sistema, se obtiene el siguiente resultado. En el instante $t = 0$ se tiene $x(0) = 0$ y por lo tanto $q(0) = 0$. Entonces, la derivada $\dot{x}(0) = -0{,}5$ y $x(0^+)$ es negativa. Luego, $q(0^+) = -1$ y $\dot{x}(0^+) = +0{,}5$.

Como ahora la derivada es positiva, $x(t)$ vuelve inmediatamente a cero y $q(t)$ también. Ahora, estamos nuevamente en las condiciones en que se comenzó a simular el sistema y el tiempo de simulación no avanzó.

Este comportamiento da como resultado una oscilación permanente en la cual $q(t)$ cambia entre 0 y -1. El problema reside en que estas oscilaciones tienen periodo 0 (frecuencia infinita).

A pesar de que la metodología presentada no funciona, sirvió de base para el desarrollo de la familia de métodos de integración que se muestra en las siguientes secciones de este capítulo.

3.3. Método de los sistemas de estados cuantificados (QSS)

Si se analizan las oscilaciones infinitamente rápidas en el sistema de la ecuación (3.8) se puede ver que las mismas se deben a los cambios en $q(t)$. Un cambio infinitesimal en $x(t)$ puede producir, debido a la cuantificación, una oscilación importante con una frecuencia infinita en $q(t)$.

Una solución a esto es utilizar histéresis en la cuantificación. Agregando histéresis a la relación entre $x(t)$ y $q(t)$, las oscilaciones en esta última pueden ser sólo debidas a oscilaciones grandes en $x(t)$. Si la derivada $\dot{x}(t)$ es finita, una oscilación grande en $x(t)$ no puede ocurrir instantáneamente sino que tiene una frecuencia máxima acotada.

En base al cambio de la función de cuantificación sin histéresis por una con histéresis surgió toda una familia de métodos de integración por cuantificación denominados QSS (Quantaized State System). Dentro de esta familia existen métodos de primer, segundo y tercer orden denominados QSS, QSS2 y QSS3 respectivamente.

3.3.1. Método de integración de estados cuantificados QSS

Para poder mostrar formalmente los métodos QSS es conveniente definir primero el concepto de *función de cuantificación con histéresis.*

Sea que $x(t) : \Re \to \Re$ es una trayectoria continua en el tiempo, y $q(t) : \Re \to \Re$ una trayectoria seccionalmente constante. Luego, $x(t)$ y $q(t)$ se relacionan mediante una función de cuantificación con histéresis uniforme si se cumple que:

$$q(t) = \begin{cases} \text{floor}[x(t_0)/\Delta Q] \cdot \Delta Q & \text{si } t = t_0 \\ x(t) & \text{si } |q(t^-) - x(t)| \geq \Delta Q \\ q(t^-) & \text{en otro caso} \end{cases} \tag{3.10}$$

El parámetro de la cuantificación ΔQ se denomina *quántum*.

Las Figuras 3.11–3.12 muestran una función de cuantificación con histéresis de orden cero y dos variables relacionadas mediante dicha función

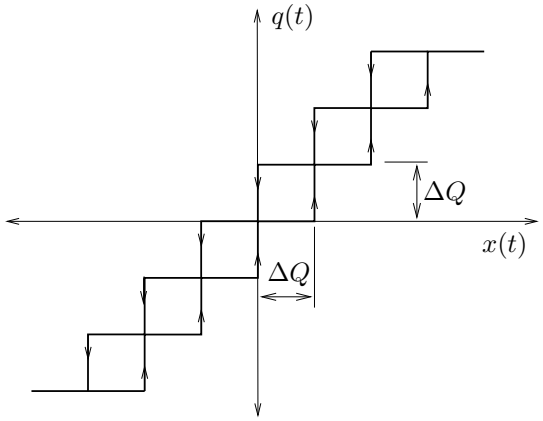

Figura 3.11: Función de Cuantificación con Histéresis de orden cero.

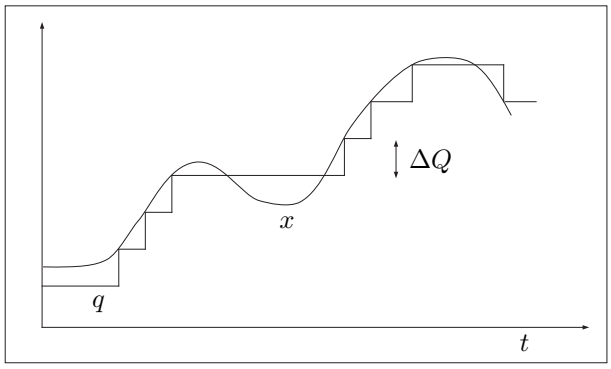

Figura 3.12: Variables Relacionadas con una Función de Cuantificación con Histéresis de orden cero.

Notar que un cuantificador con histéresis tiene memoria. Es decir, calcula $q(t)$ no solo en función del valor actual de $x(t)$ sino también de su valor pasado.

Una vez definida la función de cuantificación con histéresis se puede definir el método QSS.

Dado un sistema como el de la Ec.(5.1), el método de QSS lo transforma en un *sistema de estados cuantificados* de la forma de la Ec.(3.7), donde cada par de variables $x_i(t)$ y $q_i(t)$ se relaciona mediante funciones de cuantificación con histéresis de orden cero.

Es decir, para utilizar el método de QSS simplemente se debe elegir el *quántum* a utilizar ΔQ_i para cada variable de estado.

En la figura 3.13 se ve la representación en diagrama de bloques de un sistema de estados cuantificados usando esta idea. Tal como se hizo anteriormente, el sistema puede ser separado en integradores cuantificados con histéresis y funciones estáticas.

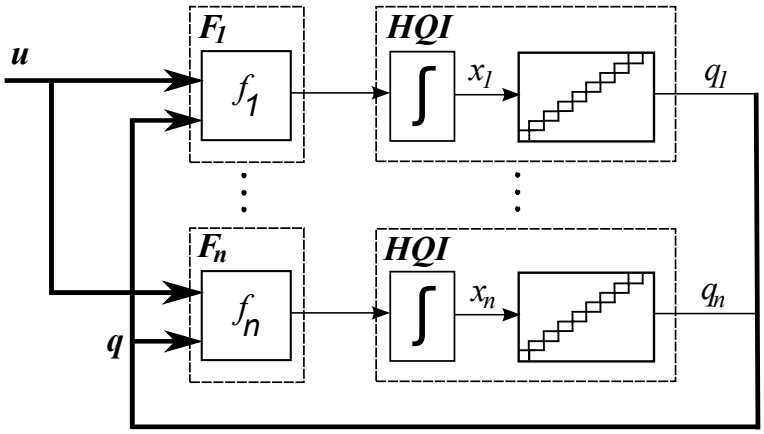

Figura 3.13: Representación en diagrama en bloques de un sistema de estados cuantificados.

Se puede demostrar que las variables cuantificadas y de estado son siempre seccionalmente constantes y seccionalmente lineales respectivamente. Por esto, la simulación con QSS no puede trabarse como ocurría con la cuantificación sin histéresis.

Para llevar a cabo la simulación, al igual que antes, podemos construir un modelo DEVS acoplando los subsistemas correspondientes a las funciones estáticas y a los *integradores cuantificados con histéresis*.

Modelo DEVS del integrador cuantificado de primer orden

Un integrador cuantificado con histéresis es un integrador cuantificado donde la función de cuantificación sin memoria es reemplazada por una con histéresis. Esto es equivalente a cambiar la Ec.(3.4c) por la Ec.(3.10) en el sistema de la Ec.(3.4).

Con esta modificación, el integrador cuantificado con histéresis puede representarse por el modelo DEVS:

$M_{HQI} = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta)$, donde
$X = Y = \Re \times \{0\}$
$S = \Re^3 \times \Re_0^+$
$\delta_{\text{int}}(s) = \delta_{\text{int}}(x, d_x, q, \sigma) = (x + \sigma \cdot d_x, d_x, x + \sigma \cdot d_x, \sigma_1)$
$\delta_{\text{ext}}(s, e, x_u) = \delta_{\text{ext}}(x, d_x, q, \sigma, e, x_v, p) = (x + e \cdot d_x, x_v, q, \sigma_2)$
$\lambda(s) = \lambda(x, d_x, q, \sigma) = (x + \sigma \cdot d_x, 0)$
$ta(s) = ta(x, d_x, q, \sigma) = \sigma$

con:

$$\sigma_1 = \frac{\Delta Q}{|d_x|}$$

y:

$$\sigma_2 = \begin{cases} \dfrac{q + \Delta Q - (x + e \cdot d_x)}{x_v} & \text{si } x_v > 0 \\ \dfrac{(x + e \cdot d_x) - (q - \Delta Q)}{|x_v|} & \text{si } x_v < 0 \\ \infty & \text{si } x_v = 0 \end{cases}$$

3.3.2. Método de integración de estados cuantificados QSS2

El método QSS si bien es el método que pudo simular sistemas continuos aproximándolos por sistemas cuantificados en forma segura (sin generar modelos ilegítimos), está muy limitado por ser un método de primer orden. Esta limitación hace que para simular un sistema con una precisión medianamente alta el número de pasos sea inaceptable.

El método QSS2 se basa en los mismos principios que QSS solo que en lugar de utilizar una función de cuantificación de orden cero utiliza una de primer orden.

Para poder definir entonces este método se debe ver en primer lugar cómo es una función de cuantificación de primer orden.

Usando la idea de la función de cuantificación de orden cero, se puede definir una función de cuantificación de primer orden como una función que genera una trayectoria de salida seccionalmente lineal cuyo valor y pendiente cambia solamente cuando la diferencia entre esta salida y la entrada se vuelve mayor que un valor predeterminado. Este valor, es el quántum. Esta idea puede verse en la figura 3.14.

La trayectoria de salida seccionalmente lineal comienza con el valor de la entrada y con pendiente igual a la entrada y luego, cuando la trayectoria de entrada y de salida difieren en ΔQ, la trayectoria de salida se representa por un nuevo segmento que comienza en el valor de la entrada. Denominando $q(t)$ a la trayectoria de salida, mq a la pendiente de la misma y $x(t)$ a la entrada, el comportamiento descripto se puede escribir formalmente como:

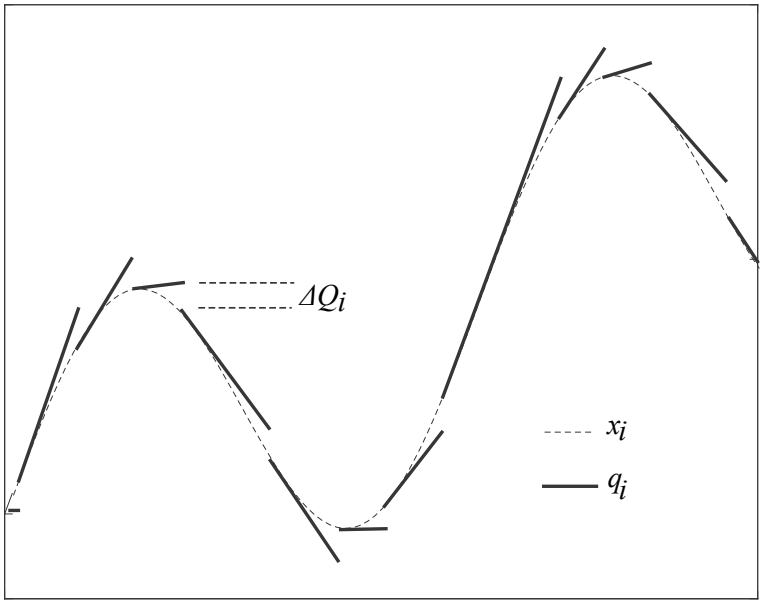

Figura 3.14: Función de cuantificación de primer orden.

$$
\begin{aligned}
q(t) &= \begin{cases} x(t) & \text{si } t = t_0 \vee |q(t^-) - x(t^-)| \geq \Delta Q \\ q(t_j) + mq_j \cdot (t - t_j) & \text{en otro caso} \end{cases} \\
mq_j &= \begin{cases} \dot{x}(t_j^-) & \text{si } q(t_j) \neq q(t_j^-) \\ 0 & \text{si } t = t_0 \end{cases}
\end{aligned}
$$

Una propiedad fundamental de la función de cuantificación de primer orden descripta es que

$$|x(t) - q(t)| \leq \Delta Q \quad \forall t \geq 0 \tag{3.11}$$

Reemplazando las funciones de cuantificación de orden cero de QSS por funciones de cuantificación de primer orden, se obtiene un método QSS de segundo orden (QSS2).

Utilizando este método, se obtienen sistemas de estados cuantificados de segundo orden. Estos nuevos sistemas también pueden ser representados por las ecuaciones (3.7):

$$
\begin{aligned}
\dot{x}_1 &= f_1(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m) \\
\dot{x}_2 &= f_2(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m) \\
&\vdots \\
\dot{x}_n &= f_n(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m)
\end{aligned} \tag{3.12}
$$

La definición de QSS2 es prácticamente la misma que la de QSS. Los dos métodos difieren únicamente en el modo en que se calcula $q_i(t)$ a partir de $x_i(t)$. Teniendo en cuenta que las trayectorias de las variables cuantificadas $q_i(t)$ en QSS2 son seccionalmente lineales, entonces las trayectorias $\dot{x}(t)$ de las derivadas

de los estados también son seccionalmente lineales [2] y entonces las trayectorias $x_i(t)$ de las variables de estado resultan seccionalmente parabólicas.

Como en el caso del método QSS, el método QSS2 también puede ser implementado por modelos DEVS de integradores cuantificados y funciones estáticas. Sin embargo, los nuevos modelos atómicos deben tener en cuenta además del valor de la trayectoria de entrada y de salida, la pendiente de las mismas.

En la figura 3.15 puede verse el diagrama en bloques de un sistema cuantificado de segundo orden representado mediante modelos atómicos DEVS. En el mismo se representan las trayectorias de salida de cada modelo atómico mediante su valor y pendiente.

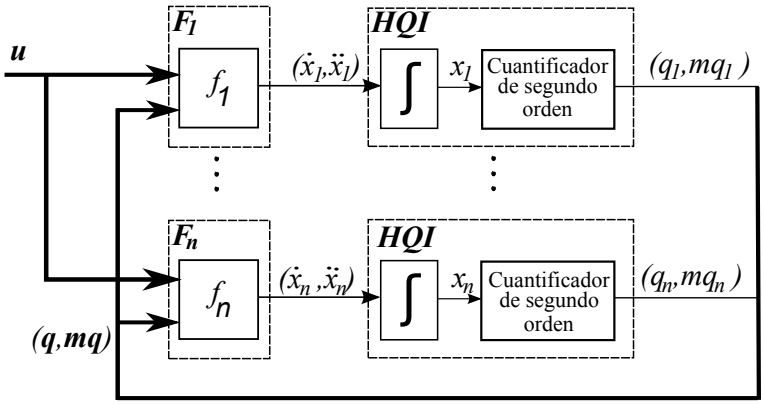

Figura 3.15: Diagrama en Bloques de un QSS2 genérico

Modelo DEVS del integrador cuantificado de segundo orden

El siguiente modelo DEVS representa el comportamiento de un integrador cuantificado de primer orden con trayectorias de entrada y salida seccionalmente lineales.

$$
\begin{aligned}
HQI &= (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta), \text{ donde} \\
X &= (\Re \times \Re) \times \{inports\} = \Re \times Re \times \{1\} \\
Y &= (\Re \times \Re) \times \{outports\} = \Re \times Re \times \{1\} \\
S &= \Re^5 \times \Re_0^+ \\
\delta_{\text{int}}(s) &= \delta_{\text{int}}(x, d_x, dd_x, q, mq, \sigma) \\
&= (\widetilde{x}, dx + ddx \cdot \sigma, dd_x, \widetilde{x}, dx + dd_x \cdot \sigma, \widetilde{\sigma}) \\
\delta_{\text{ext}}(s, e, v) &= \delta_{\text{ext}}(x, d_x, dd_x, q, mq, \sigma, e, v, mv, port) \\
&= (\hat{x}, v, mv, q + mq \cdot e, mq, \hat{\sigma}) \\
\lambda(s) &= \lambda(x, d_x, dd_x, q, mq, \sigma) = (\widetilde{x}, d_x + dd_x \cdot \sigma, 1) \\
ta(s) &= ta(x, d_x, q, \sigma) = \sigma
\end{aligned}
$$

[2] Para ser rigurosos, esto solo es cierto para sistemas LTI. En los demás casos, las derivadas de los estados se aproximan mediante trayectorias seccionalmente lineales.

donde:

$$\begin{aligned}
\widetilde{x} &= x + d_x \cdot \sigma + dd_x \cdot \sigma^2/2 \\
\widetilde{\sigma} &= \begin{cases} \sqrt{\frac{2\Delta Q}{dd_x}} & \text{si } dd_x \neq 0 \\ \infty & \text{en otro caso} \end{cases} \qquad (3.13) \\
\hat{x} &= x + d_x \cdot e + dd_x \cdot e^2/2 \qquad (3.14)
\end{aligned}$$

y $\hat{\sigma}$ se puede calcular como la menor solución positiva de:

$$\left|\hat{x} + v \cdot \hat{\sigma} + mv \cdot \hat{\sigma}^2 - (q + mq \cdot e + mq \cdot \hat{\sigma})\right| = \Delta Q \qquad (3.15)$$

Los eventos de entrada y salida están constituidos por números que representan el valor, la pendiente y el puerto de la trayectoria a los que están asociados. El estado $s \in S$ guarda el valor y pendiente (v y mv) del último valor a la entrada, el valor actual de la variable de estado (x), el valor y pendiente (q y mq) del último valor que se sacó y el tiempo (σ) que falta para que se genere el próximo evento de salida.

La función de transición interna recalcula el estado s después de cada evento de salida y la función de transición externa hace lo mismo pero luego de que se recibe un evento de entrada. Las ecuaciones (3.13) y (3.15) calculan cuánto tiempo falta hasta el próximo evento de salida, o sea, hasta que la diferencia entre $x(t)$ y $q(t)$ sea igual a ΔQ. Finalmente, la función de salida (λ) genera el evento de salida con el valor y pendiente correspondientes.

Modelo DEVS de las funciones estáticas

Como se mencion anteriormente, las funciones estáticas de los QSS2 deben tener en cuenta no solo el valor de las trayectorias de entrada sino también la pendiente de la misma. De igual modo, la salida de las mismas esta compuesta por el valor y pendiente de la trayectoria de salida.

Teniendo en cuenta esto el modelo DEVS de una función estática genérica $f_i(u_1, \ldots, u_n)$ resulta en este caso:

$$\begin{aligned}
&F_i = (X, Y, S, \delta_{\text{int}}, \delta_{\text{ext}}, \lambda, ta), \text{ donde} \\
&X = \Re \times \Re \times \{inports\} = \Re \times Re \times 1, \ldots, n \\
&Y = (\Re \times \Re) \times \{outports\} = (\Re \times Re) \times \{1\} \\
&S = (\Re \times \Re \times \Re)^n \times \Re_0^+ \\
&\delta_{\text{int}}\left((u_1, mu_1, c_1), \ldots, (u_n, mu_n, c_n), \sigma\right) = \\
&\qquad = ((u_1, mu_1, c_1), \ldots, (u_n, mu_n, c_n), \infty) \\
&\delta_{\text{ext}}\left((u_1, mu_1, c_1), \ldots, (u_n, mu_n, c_n), \sigma, e, v, mv, port\right) = \\
&\qquad = ((\hat{u}_1, \hat{mu}_1, \hat{c}_1), \ldots, (\hat{u}_n, \hat{mu}_n, \hat{c}_n), 0) \\
&\lambda\left((u_1, mu_1, c_1), \ldots, (u_n, mu_n, c_n), \sigma\right) = \\
&\qquad = (f_i(\hat{u}_1, \ldots, \hat{u}_n), c_1 \cdot mu_1 + \ldots + c_n \cdot mu_n, 1) \\
&ta\left((u_1, mu_1, c_1), \ldots, (u_n, mu_n, c_n), \sigma\right) = \sigma
\end{aligned}$$

con

$$
\begin{aligned}
\hat{u}_j &= \begin{cases} v & \text{si } j = port \\ u_j + mu_j \cdot e & \text{en otro caso} \end{cases} \\
\hat{mu}_j &= \begin{cases} mv & \text{si } j = port \\ mu_j & \text{en otro caso} \end{cases} \\
\hat{c}_j &= \begin{cases} \frac{f_i(u+mu\cdot e)-f_i(\hat{u})}{u_j+mu_j\cdot e-\hat{u}_j} & \text{si } j = port \wedge u_j + mu_j \cdot e - \hat{u}_j \neq 0 \\ c_j \text{en otro caso} \end{cases}
\end{aligned} \tag{3.16}
$$

Si la función f_i es lineal, este modelo DEVS representa exactamente su comportamiento. La ecuación (3.16) calcula los coeficientes que multiplican a la pendiente de las trayectorias de entrada. En el caso lineal, los mismos coincidirán con los coeficientes A_{ij} y B_{ij} correspondientes a las matrices de evolución y de entrada respectivamente.

En caso de ser f_i no lineal, la trayectoria de salida de la misma no será seccionalmente lineal. Sin embargo, las trayectorias generadas por el modelo DEVS serán seccionalmente lineales y serán una buena aproximación a la salida real.

3.3.3. Método de integración de estados cuantificados QSS3

Para poder obtener una aproximación de tercer orden, se debe tener en cuenta no solo la primera derivada de las trayectorias del sistema (como en QSS2) sino que además se debe tener en cuenta la segunda derivada. En este sentido, se debe redefinir la función de cuantificador mostrado en la figura 3.14 de manera que las trayectorias de salida sean seccionalmente parabólicas. Luego, dado un sistema de ecuaciones de estado como el de la Ec.(5.1):

$$
\begin{aligned}
\dot{x}_{a_1} &= f_1(x_{a_1}, x_{a_2}, \cdots, x_{a_n}, u_1, \cdots, u_m) \\
\dot{x}_{a_2} &= f_2(x_{a_1}, x_{a_2}, \cdots, x_{a_n}, u_1, \cdots, u_m) \\
&\vdots \\
\dot{x}_{a_n} &= f_n(x_{a_1}, x_{a_2}, \cdots, x_{a_n}, u_1, \cdots, u_m)
\end{aligned}
$$

el método QSS3 lo aproxima por las Ec.(3.7):

$$
\begin{aligned}
\dot{x}_1 &= f_1(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m) \\
\dot{x}_2 &= f_2(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m) \\
&\vdots \\
\dot{x}_n &= f_n(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m)
\end{aligned}
$$

donde x y q se relacionan componente a componente mediante funciones de cuantificación de segundo orden.

Una función de cuantificación de segundo orden genera una trayectoria de salida seccionalmente parabólica cuyo valor, pendiente y segunda derivada cambian cuando la diferencia entre su salida y su entrada difieren en más de un valor predeterminado (Figura 3.16).

Formalmente, se dice que las trayectorias $x_i(t)$ y $q_i(t)$ estn relacionadas mediante una función de cuantificación de segundo orden si $q_i(t_0) = x_i(t_0)$ y

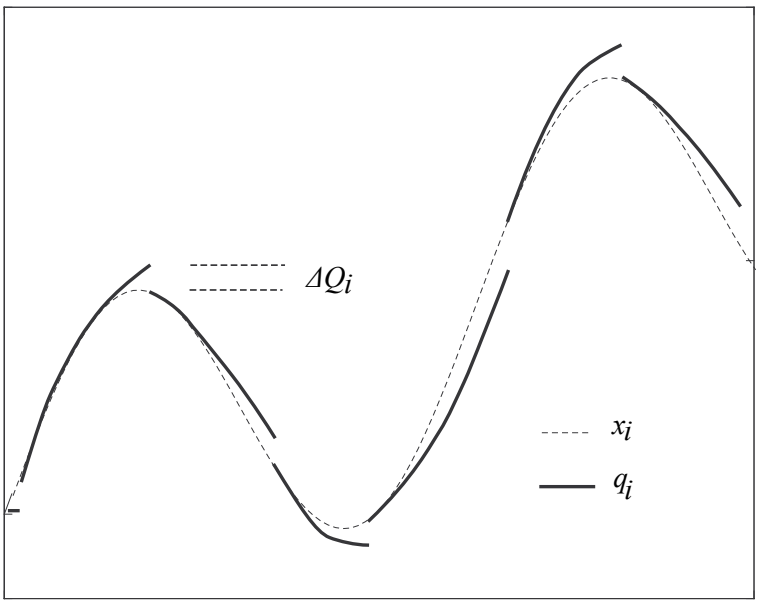

Figura 3.16: Función de cuantificación de Segundo orden.

$$q_i(t) = \begin{cases} x_i(t) & \text{si } |q_i(t^-) - x_i(t^-)| = \Delta Q_i \\ q_i(t_j) + mq_{i_j} \cdot (t - t_j) + pq_{i_j} \cdot (t - t_j)^2 & \text{en otro caso} \end{cases} \tag{3.17}$$

con $t_j \leq t < t_{j+1}$, la secuencia $t_0, \ldots, t_j, \ldots$ tal que t_{j+1} es el mínimo $t > t_j$ en el cual

$$\left| x_i(t_j) + dx_i(t - t_j + ddx_i(t - t_j)^2 - q_i(t)) \right| = \Delta Q_i$$

y pendientes

$$\begin{array}{ll} mq_{i_0} = 0; & mq_{i_j} = \dot{x}_i(t_j^-), \quad j = 1, 2, \ldots \\ pq_{i_0} = 0; & pq_{i_j} = \ddot{x}_i(t_j^-), \quad j = 1, 2, \ldots \end{array}$$

Luego, los integradores cuantificados en QSS3 y las funciones estáticas tienen trayectorias de entrada y salida seccionalmente parabólicas. Una vez diferenciado el integrador cuantificado QSS3 y las correspondientes funciones estáticas, se procede de forma similar a lo hecho en QSS y QSS2 para construir los modelos DEVS de los mismos. Una explicación y desarrollo más profundo de los mismos puede encontrarse en [8].

3.3.4. Control de error relativo en los métodos QSS

Uno de los problemas aún no explicados es cómo se puede seleccionar el quántum. En esta sección se mostrar un método para tener control de error relativo [10] en los métodos QSS.

Usando cuantificación logarítmica, es decir, haciendo que el quántum sea proporcional a la magnitud de las variables cuantificadas, se logra que los métodos QSS tengan un control de error relativo.

La idea básica de la cuantificación logarítmica consiste en tomar el valor del quántum ΔQ_i proporcional al valor de x_i en el momento en que se alcanza una condición de evento.

El problema de escoger el quántum de esa manera se da cuando x_i es próximo a cero. En este caso, ΔQ_i resultar demasiado pequeño y se producirán una gran cantidad de eventos innecesarios. Entonces, la elección correcta del quántum es:

$$\Delta Q_i(t) = \max\left(E_{rel_i} \cdot |x_i(t_k)|, \Delta Q_{i_m in}\right) \tag{3.18}$$

donde t_k es el último instante de tiempo en que ocurrió un evento en x_i

Para poder ver que de esta forma se logra un control del error relativo, se deber hacer un anlisis del error. Dado el siguiente sistema LTI:

$$\dot{\mathbf{x}}_\mathbf{a}(t) = A \cdot \mathbf{x}_\mathbf{a}(t) + B \cdot \mathbf{u}(t) \tag{3.19}$$

definiendo $\mathbf{\Delta x}(t) \triangleq \mathbf{q}(t) - \mathbf{x}(t)$, los métodos QSS lo aproximan por

$$\dot{\mathbf{x}}(t) = A \cdot (\mathbf{x}(t) + \mathbf{\Delta x}) + B \cdot \mathbf{u}(t) \tag{3.20}$$

Haciendo ahora la diferencia entre las Ecs.(3.19) y (3.20), se obtiene la ecuación para el error $\mathbf{e}(t) = \mathbf{x}(t) - \mathbf{x}_\mathbf{a}(t)$:

$$\dot{\mathbf{e}}(t) = A \cdot (\mathbf{e}(t) + \mathbf{\Delta x}(t)) \tag{3.21}$$

donde $|\mathbf{\Delta x}(t)| \leq \mathbf{\Delta Q}(t)$ para todo $t \geq t_0$. Luego, trabajando con las ecuaciones y aplicando el teorema 2.2 de [9] se demostró que:

$$|\mathbf{e}(t)| \leq (I - R\, E_{rel})^{-1}\, R \max\left(E_{rel} \cdot |x_{max}|, \Delta Q_{min}\right) \tag{3.22}$$

donde A es una matriz Hurwitz con forma canónica de Jordan $\Lambda = V^{-1}AV$ y $R \triangleq |V|\ \left|[Re(\Lambda)]^{-1}V^{-1}H\right|$

En esta ecuación se ve claramente que la cota de error es proporcional al máximo valor de $\mathbf{x}_a(t)$, Es decir, se logra un control del error relativo.

Además, en la mayoría de las aplicaciones se elije E_{rel} muy pequeño (un valor tpico es $E_{rel} = 0{,}01$). En este caso se puede aproximar la ecuación (3.22) por:

$$|\mathbf{e}(t)| \leq R \max\left(E_{rel} \cdot |x_{max}|, \Delta Q_{min}\right) \tag{3.23}$$

3.3.5. QSS y sistemas stiff

En las secciones anteriores se dio una descripción de los métodos de integración de estados cuantificados QSS, QSS2 y QSS3. En esta sección se describirá los problemas que tienen estos métodos para simular sistemas stiff, cuyas características se definieron en 2.2.2 y en las secciones siguientes, se darán las definiciones de métodos de estados cuantificados adecuados para resolver este tipo de sistemas, así como también su definición en el formalismo DEVS. Para mostrar los inconvenientes que presentan los métodos de QSS para simular sistemas stiff consideremos el siguiente ejemplo.

Sea el sistema

$$\begin{aligned} \dot{x}_1(t) &= 0{,}01\, x_2(t) \\ \dot{x}_2(t) &= -100\, x_1(t) - 100\, x_2(t) + 2020 \end{aligned} \tag{3.24}$$

que tiene autovalores $\lambda_1 \approx -0{,}01$ y $\lambda_2 \approx -99{,}99$, por lo que puede considerarse stiff.

El método de QSS aproximará este sistema según

$$\begin{aligned} \dot{x}_1(t) &= 0{,}01\, q_2(t) \\ \dot{x}_2(t) &= -100\, q_1(t) - 100\, q_2(t) + 2020 \end{aligned} \tag{3.25}$$

Considerando condiciones iniciales $x_1(0) = 0$, $x_2(0) = 20$, y cuantificación $\Delta Q_1 = \Delta Q_2 = 1$, la integración por QSS hará los siguientes pasos:

En $t = 0$ tenemos $q_1(0) = 0$ y $q_2(0) = 20$. Por lo tanto $\dot{x}_1(0) = 0{,}2$ y $\dot{x}_2(0) = 20$. Esta situación se mantiene hasta que $|q_i - x_i| = \Delta Q_i = 1$.

El próximo cambio en q_1 es entonces agendado para $t = 1/0{,}2 = 5$ mientras que el cambio en q_2 se agenda para $t = 1/20 = 0{,}05$.

Por lo tanto en $t = 0{,}05$ hay un nuevo paso, tras el cual, $q_1(0{,}05) = 0$, $q_2(0{,}05) = 21$, $x_1(0{,}05) = 0{,}01$, $x_2(0{,}05) = 21$. Las derivadas quedan $\dot{x}_1(0{,}05) = 0{,}21$ y $\dot{x}_2(0{,}05) = -80$.

El próximo cambio en q_1 se reagenda para $t = 0{,}05 + (1 - 0{,}01)/0{,}21 = 4{,}764$ mientras el siguiente cambio en q_2 se agenda para $0{,}05 + 1/80 = 0{,}0625$. Por lo tanto, el siguiente paso se hace en $t = 0{,}0625$.

En $t = 0{,}0625$ tenemos $q_1(0{,}0625) = 0$, $q_2(0{,}0625) = x_2(0{,}0625) = 20$, $x_1(0{,}0625) \approx 0{,}0126$ y las derivadas son las mismas que en $t = 0$.

Esto se repite cíclicamente hasta que efectivamente se produce el cambio en q_1 (esto se da en $t \approx 4{,}95$, tras 158 cambios en el valor de q_2, oscilando entre 20 y 21).

La simulación continua de la misma manera. En las Figs.3.17–3.18 puede observarse la evolución de $q_1(t)$ y $q_2(t)$ hasta $t = 500$ seg.

Como puede verse, hay oscilaciones rápidas en q_2 que provocan un total de 15995 transiciones en dicha variable, mientras que q_1 realiza sólo 21 cambios. En definitiva hay más de 16000 pasos para completar la simulación (es del orden de los 25000 pasos que debe realizar como mínimo el método de Euler para obtener un resultado estable).

Evidentemente, el método de QSS no es capaz de integrar eficientemente el sistema (3.24).

3.3.6. Método de integración de estados cuantificados CQSS

La idea básica de CQSS (QSS Centrado) consiste en utilizar el valor medio entre el q_i que utilizara QSS y el valor futuro estimado de $q_i + \Delta Q_i$ para el calculo de las derivadas de los estados x_i. El método CQSS resulta ser un compromiso de estas dos situaciones extremas.

En la Figura 3.19 se puede ver el comportamiento de esta nueva función de cuantificación. A su vez, esta figura permite comprender más claramente el nombre otorgado a este nuevo método. Mientras que en QSS, el estado cuantificado q_i permanece siempre "atrás" del verdadero estado x_i , el valor de $q_i + \Delta Q_i$, siempre está por "delante" del mismo.

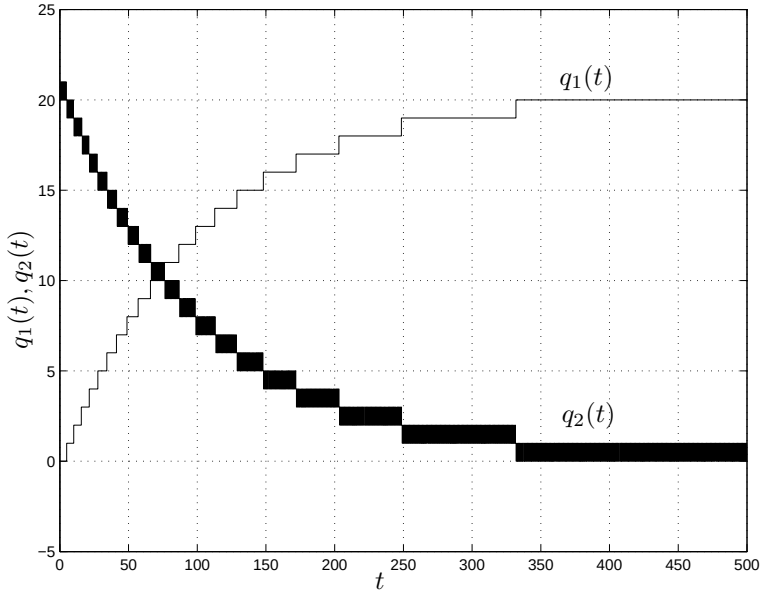

Figura 3.17: Simulación con QSS

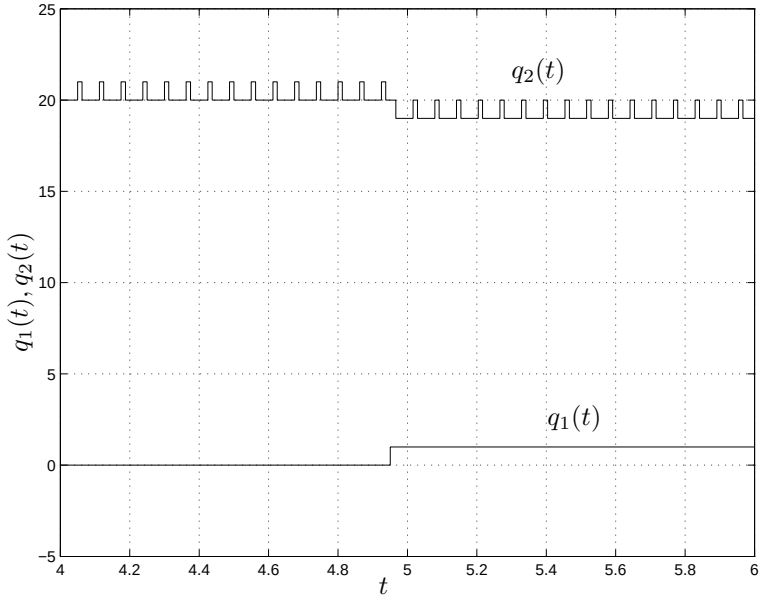

Figura 3.18: Simulación con QSS (detalle)

El método de CQSS tiene en sí la misma definición que QSS difiriendo únicamente en que q_i se mantiene centrado en los intervalos. La definición del modelo DEVS es la misma que la dada en la Sección 3.3.1, con la diferencia de que ahora, el nuevo valor de q_i en δ_{int} se acualiza de acuerdo al valor de $\delta_{\text{int}}(s) = \delta_{\text{int}}(x, d_x, q, \sigma) = (x + \sigma \cdot d_x, d_x, \hat{q}, \sigma_1)$

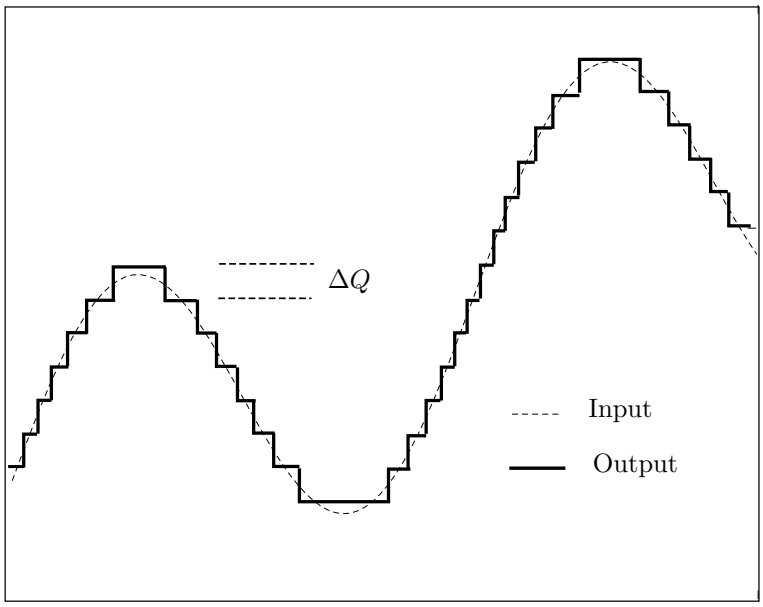

Figura 3.19: Función de cuantificación centrada

con:

$$\hat{q} = \begin{cases} q + \Delta Q/2 & \text{si } d_x >= 0 \\ q - \Delta Q/2 & \text{si } d_x < 0 \end{cases}$$

3.3.7. Método de integración de estados cuantificados LIQSS

Para mostrar la idea del desarrollo del método LIQSS, veamos paso a paso cómo se comporta el algoritmo sobre el ejemplo introductorio de la Sección 3.3.5:

$$\begin{aligned} \dot{x}_1(t) &= 0{,}01\, x_2(t) \\ \dot{x}_2(t) &= -100\, x_1(t) - 100\, x_2(t) + 2020 \end{aligned} \tag{3.26}$$

Recordemos que los autovalores son $\lambda_1 \approx -0{,}01$ y $\lambda_2 \approx -99{,}99$ por lo que es claramente stiff y tomemos las mismas condiciones iniciales y la misma cuantificación de la sección 3.3.5, es decir $x_1(0) = 0$, $x_2(0) = 20$ y cuantificación $\Delta Q_1 = \Delta Q_2 = 1$.

En $t = 0$, podemos elegir $q_2 = 19$ o $q_2 = 21$ según cuál sea el signo de $\dot{x}_2(t)$. En ambos casos, $\dot{x}_1(0) > 0$ por lo que el valor cuantificado futuro de x_1 ser $q_1(0) = 1$.

Por otro lado, si elegimos $q_2(0) = 21$ entonces $\dot{x}_2(0) = -180 < 0$ y si elegimos $q_2(0) = 19$ resulta $\dot{x}_2(0) = 20 > 0$ por lo tanto, existe un punto $19 < \hat{q}_2(0) < 21$ en el cual $\dot{x}_2(0) = 0$. Se puede calcular (explotando la dependencia lineal de $\dot{x}_2$ con q_2) el valor de $\hat{q}_2(0)$ según

$$\hat{q}_2(0) = 21 - \frac{-180}{-100} = 19{,}2$$

Finalmente, las derivadas de los estados resultan: $\dot{x}_1(0) = 0{,}192$, $\dot{x}_2(0) = 0$.

El siguiente cambio en q_1 es agendado para $t = 1/0{,}192 \approx 5{,}2083$ mientras que el próximo cambio en q_2 se agenda para $t = \infty$.

El siguiente paso es entonces en $t = 1/0{,}192 \approx 5{,}2083$. En ese instante $x_1 = 1$ y $x_2 = 0$. Luego tendremos $q_1(5{,}2083) = 2$ (ya que $\dot{x}_1(5{,}2083) > 0$). Si reevaluamos $\dot{x}_2$ para $q_2 = 19$ y $q_2 = 21$ resulta que en ambos casos es menor que cero, entonces el valor correcto es $q_2(5{,}2083) = 19$ porque de esta manera x_2 evoluciona hacia q_2.

Con estos valores de q_1 y q_2 se tiene $\dot{x}_1(5{,}2083) = 0{,}19$ y $\dot{x}_2(5{,}2083) = -80$. El próximo cambio en q_1 se agenda para $t = 1/0{,}192 + 1/0{,}19 \approx 10{,}47149$, mientras que el siguiente cambio en q_2 se agenda para $t = 1/0{,}192 + 1/80 \approx 5{,}22083$. Por lo tanto, el siguiente paso se da en $t = 5{,}22083$, cuando x_2 alcanza a q_2.

Los cálculos continúan de la misma manera. En la Figura 3.20 se puede ver la evolución de $q_1(t)$ y $q_2(t)$ hasta $t = 500$.

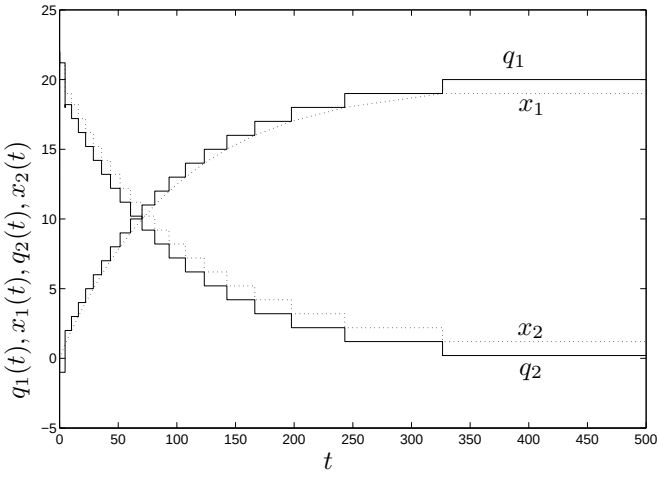

Figura 3.20: Resultado de la simulación con LIQSS

Como puede verse, las oscilaciones rápidas de q_2 no están presentes al usar este método por lo que la simulación llevó sólo 46 cambios en total (21 cambios en q_1 y 25 cambios en q_2) lo cual es un resultado más que bueno para un sistema stiff. Cabe recordar que el mismo sistema al simularse con QSS tomó 16016 pasos (ver 3.3.5).

Modelo DEVS del integrador cuantificado LIQSS

Como se dijo anteriormente, la diferencia principal entre LIQSS y QSS reside en el modo en el cual se obtiene $\mathbf{q}$ a partir de $\mathbf{x}$. Siguiendo la idea de la implementación en DEVS de QSS, es decir, mediante el acoplamiento de *integradores cuantificados* y *funciones estáticas*, pero teniendo en cuenta las diferencias mencionadas, podemos encontrar el modelo DEVS para el algoritmo LIQSS.

La estructura de su implementación se puede ver en la Figura 3.21.

Dado que las trayectorias de $u_j(t)$ y $q_j(t)$ son seccionalmente constantes, las funciones estáticas son las mismas que las de QSS.

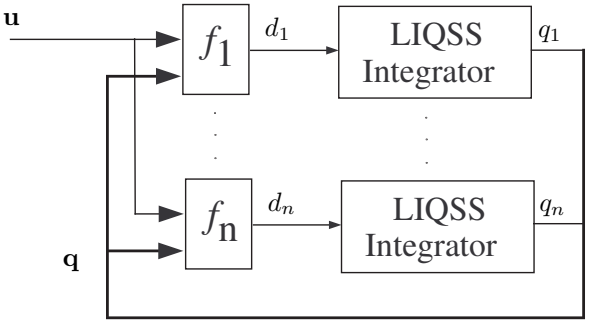

Figura 3.21: Diagrama en bloques de LIQSS1

En consecuencia, sólo se debe definir el integrador LIQSS de manera que calcule q_j según la definición de LIQSS.

Para poder construir el modelo DEVS del integrador cuantificado LIQSS, analizaremos primero su comportamiento.

Supongamos que en el instante de tiempo t el estado x_j alcanza al valor $\overline{q}_j$ con una pendiente positiva ($\dot{x}_j(t^-) > 0$). Entonces, se deben actualizar las funciones de cuantificación superior e inferior ($\overline{q}_j$ y $\underline{q}_j$) incrementándolas en ΔQ_j. Con los valores de $\underline{q}_j(t)$, $\overline{q}_j(t)$ y una estima de A_{jj} podemos prever cuánto resultará $\dot{x}_j(t)$ evaluado en cada uno de ellos. Esta estima se puede realizar a partir de la siguiente ecuación:

$$\dot{x}_j(\overline{q}_j(t)) = \dot{x}(t^-) - A_{jj}q_j(t^-) + A_{jj}\overline{q}_j(t) \tag{3.27}$$

$$\dot{x}_j(\underline{q}_j(t)) = \dot{x}(t^-) - A_{jj}q_j(t^-) + A_{jj}\underline{q}_j(t) \tag{3.28}$$

donde A_{jj} puede ser fácilmente estimado a partir de los valores de $\dot{x}(t^-)$, $q_j(t^-)$.

Luego, si $\dot{x}_j(\overline{q}_j) > 0$ y $\dot{x}_j(\underline{q}_j) > 0$ tomamos $q_j = \overline{q}_j$ y generamos un evento de salida con el valor q_j. Contrariamente, si resulta $\dot{x}_j(\overline{q}_j) \leq 0$ y $\dot{x}_j(\underline{q}_j) \leq 0$ tomamos $q_j(t) = \underline{q}_j(t)$ y generamos un evento de salida con el valor $q_j(t)$.

Podría darse la situación en la cual los signos de $\dot{x}_j(\overline{q}_j)$ y $\dot{x}_j(\underline{q}_j)$ fuesen distintos. En este caso se puede asegurar por el teorema del valor medio que existe un valor $\widetilde{q}_j$ entre $\underline{q}_j$ y $\overline{q}_j$ en el cual $\dot{x}_j(\widetilde{q}_j) = 0$. Este valor puede ser estimado, dando el valor exacto en los casos en que $\dot{x}_j$ depende linealmente de q_j. Tomamos entonces $q_j(t) = \widetilde{q}_j$ y generamos un evento de salida con el valor $q_j(t)$.

Además, la única situación en que se genera un evento de salida es cuando $x_j(t)$ alcanza ya sea a $\overline{q}_j(t)$ o a $\underline{q}_j(t)$.

Lo único que falta hacer una vez que se ha calculado q_j es agendar en qué instante de tiempo se producirá el próximo evento. En cualquiera de estas situaciones, el tiempo σ_j hasta el próximo evento está determinado por el primer cruce x_j ya sea con $\overline{q}_j$ o $\underline{q}_j$, pudiéndose calcular como:

$$\sigma_j = \begin{cases} (\overline{q}_j(t) - x_j(t))/d_j & \text{si } d_j > 0 \\ (x_j - \underline{q}_j(t))/d_j & \text{si } d_j < 0 \\ \infty & \text{en otro caso} \end{cases}$$

Notar que una vez que se ha generado un evento de salida, no se generará ningún nuevo evento de salida en ese instante aunque se reciba un evento de entrada que cambie el signo de $\dot{x}_j$ El comportamiento descripto para el integrador cuantificado puede ser fácilmente traducido en el siguiente modelo DEVS correspondiente al integrador j-simo:

$$M = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

donde

$$X = Y = \mathbb{R} \times \mathbb{N};\ S = \mathbb{R}^4 \times \mathbb{R}^+$$

$$\begin{aligned} \delta_{\text{int}}(s) &= \delta_{\text{int}}(dX, X, q, dq, \sigma) = \left(dX, \widetilde{X}, \widetilde{X}, dq_1, \sigma_1\right) \\ \delta_{ext}(s, e, u) &= \delta_{ext}(dX, X, q, dq, e, v) = \left(v, \hat{X}, q, dq, \sigma_2\right) \\ \lambda(s) &= \lambda(dX, X, q, dq, \sigma) = X + dX.\sigma + dq_1 \\ ta(s) &= \sigma \end{aligned}$$

con

$$\begin{aligned} \widetilde{X} &= X + dX.\sigma \\ \sigma_1 &= \begin{cases} \Delta Q/dX & \text{si } dX \neq 0 \\ \infty & \text{en otro caso} \end{cases} \\ dq_1 &= \begin{cases} \Delta Q & \text{si } dX > 0 \wedge A_{jj}.(X + dX.\sigma + \Delta Q + In) \geq 0 \\ -\Delta Q & \text{si } dX \leq 0 \wedge A_{jj}.(X + dX.\sigma - \Delta Q + In) \leq 0 \\ \frac{-In}{A_{jj}} - q & \text{en otro caso} \end{cases} \\ \hat{X} &= X + dx.e \end{aligned}$$

donde $In = dX - A_{jj}.(q + dq)$,y σ_2 se calcula como la mínima solución positiva de

$$|X + dX.e + v.\sigma_2 - q| = \Delta Q$$

3.3.8. Método de integración de estados cuantificados LIQSS2

Combinando las ideas de QSS2 y LIQSS se desarrolló un nuevo método de integración linealmente implícito denominado LIQSS2.

Las trayectorias de las variables cuantificadas de este nuevo método son seccionalmente lineales en lugar de seccionalmente constantes. Como en QSS2 buscaremos que las pendientes de las variables cuantificadas coincidan con las de las variables de estado correspondientes (en los instantes de cambio al menos).

Por otro lado, mientras que en LIQSS se elegía q_j de manera de evitar que cambie el signo de la derivada $\dot{x}_j$ (buscando la condición $\dot{x}_j = 0$), en LIQSS2 elegiremos q_j de manera de evitar que cambie el signo de la derivada segunda $\ddot{x}_j$. En forma análoga a LIQSS, buscaremos que se cumpla $\ddot{x}_j \cdot (q_j - x_j) \geq 0$.

En la figura 3.22 se muestra un ejemplo general de la forma de las trayectorias siguiendo esta idea.

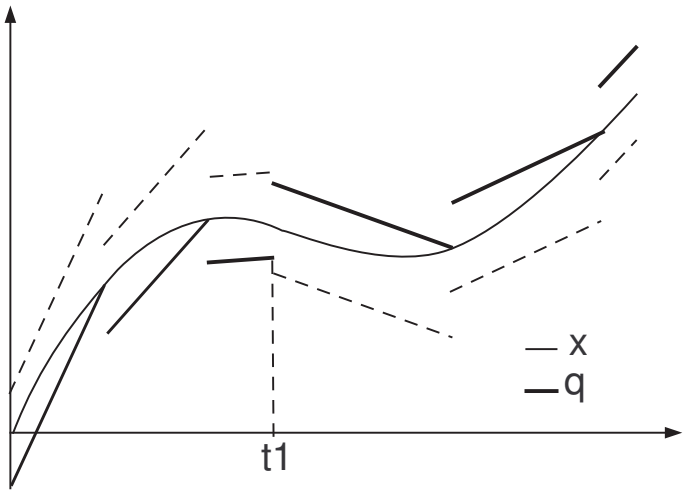

Figura 3.22: Trayectorias de LIQSS2

Modelo DEVS del integrador cuantificado LIQSS2

Un modelo DEVS para el integrador cuantificado LIQSS2 puede ser fácilmente traducido de la siguiente manera:

$$M = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

donde

$$X = Y = \mathbb{R}^2 \times \mathbb{N};\ S = \mathbb{R}^6 \times \mathbb{R}^+$$

$$\begin{aligned}
\delta_{\text{int}}(s) &= \delta_{\text{int}}(ddx, dx, x, q, mq, dq, \sigma) \\
&= \left(ddx, dx + 2ddx.\sigma, \widetilde{x}, \widetilde{x}, \widetilde{mq}, \widetilde{dq}, \sigma_1\right) \\
\delta_{ext}(s, e, u) &= \delta_{ext}(ddx, dx, x, q, mq, dq, \sigma, e, v, mv) \\
&= (mv/2, v, \hat{x}, \hat{q}, mq, dq, \sigma_2) \\
\lambda(s) &= \lambda(ddx, dx, x, q, mq, dq, \sigma) = (\widetilde{x} + \widetilde{dq}, \widetilde{mq}) \\
ta(s) &= \sigma
\end{aligned}$$

con

$$
\begin{aligned}
\widetilde{x} &= x + dx.\sigma + ddx.\sigma^2 \\
\widetilde{mq} &= A_{jj}\left(\widetilde{x} + \widetilde{dq}\right) + In \\
\sigma_1 &= \begin{cases} \sqrt{|\Delta Q/ddx|} & \text{si } ddx \neq 0 \\ \infty & \text{en otro caso} \end{cases} \\
\widetilde{dq} &= \begin{cases} \Delta Q & \text{si } ddx_{est} \geq 0 \wedge mIn + A_{jj}.(dx + 2.ddx.\sigma) < 0 \\ -\Delta Q & \text{si } ddx_{est} \leq 0 \wedge mIn + A_{jj}.(dx + 2.ddx.\sigma) \geq 0 \\ \dfrac{-mIn/A_{jj} - In}{A_{jj}} - \widetilde{x} & \text{en otro caso} \end{cases} \\
\hat{x} &= x + e.dx + ddx.e^2 \\
\hat{q} &= q + mq.e
\end{aligned}
$$

donde

$$
\begin{aligned}
In &= dx + 2.ddx.\sigma - A_{jj}\left(q + dq + mq.\sigma\right) \\
mIn &= ddx - A_{jj}mq \\
ddx_{est} &= \begin{cases} -A_{jj}\left(A_{jj}\left(\widetilde{x} + dQ\right) + In\right) - mIn & \\ \qquad\qquad \text{si } mIn + A_{jj}.(dx + 2.ddx.\sigma) < 0 & \\ -A_{jj}\left(A_{jj}\left(\widetilde{x} - dQ\right) + In\right) - mIn \quad \text{en otro caso} & \end{cases}
\end{aligned}
$$

y σ_2 se calcula como la mnima solución positiva de

$$\left|\hat{q} + mq.\sigma_2 - \hat{x} - v.\sigma_2 - mv.\sigma_2^2/2\right| = \Delta Q$$

3.3.9. Método de integración de estados cuantificados LIQSS3

De manera similar a lo hecho al extender el método LQSS para obtener LIQSS2, combinando las ideas de QSS3 y LIQSS2 se desarrolló un nuevo método de integración linealmente implícito denominado LIQSS3.

Las trayectorias de las variables cuantificadas de este método son seccionalmente parabólicas en lugar de seccionalmente lineales de modo que pueden ser caracterizadas por la terna de valores (q,mq y pq). Como en QSS3 buscaremos que tanto las pendientes mq como las segundas derivadas de las variables cuantificadas (pq) coincidan con las de las variables de estado correspondientes. Por otro lado, mientras que en LIQSS2 se elegía q_j de manera de evitar que cambie el signo de la derivada segunda $\ddot{x}_j$ (buscando la condición $\ddot{x}_j = 0$), en LIQSS3 elegiremos q_j de manera de evitar que cambie el signo de la derivada tercera $\dddot{x}_j$. En forma análoga a LIQSS, buscaremos que en todo momento se cumpla $\dddot{x}_j \cdot (q_j - x_j) \geq 0$.

En la figura 3.23 se muestra un ejemplo general de la forma de las trayectorias siguiendo esta idea.

Modelo DEVS del integrador cuantificado LIQSS3

Dado el sistema de ecuaciones

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \tag{3.29}$$

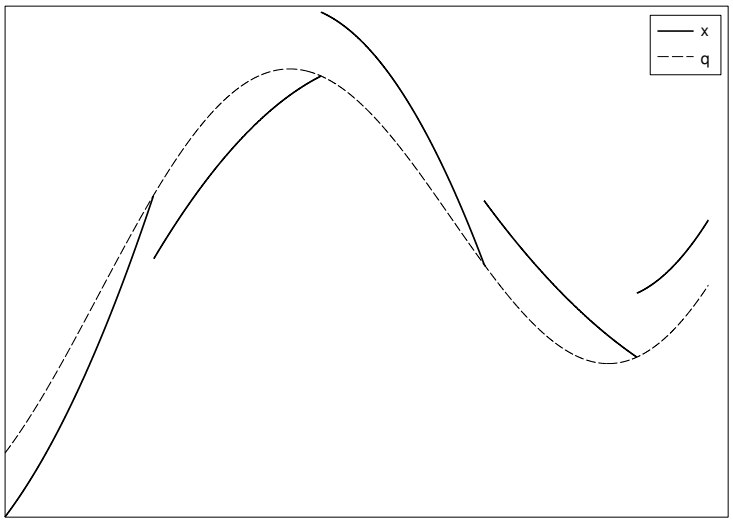

Figura 3.23: Trayectorias de LIQSS3

el método LIQSS3 lo aproxima por

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{q}(t), \mathbf{u}(t)) \tag{3.30}$$

donde cada componente q_j se define según la siguiente función

$$q_j(t) = \begin{cases} \overline{q}_j(t) & \text{si } \ddot{x}_j(\overline{\mathbf{q}}^j) > 0 \wedge \ddot{x}_j(\underline{\mathbf{q}}^j) > 0 \wedge \overline{q}_j(t^-) \neq \overline{q}_j(t) \\ \underline{q}_j(t) & \text{si } \ddot{x}_j(\overline{\mathbf{q}}^j) \leq 0 \wedge \ddot{x}_j(\underline{\mathbf{q}}^j) \leq 0 \wedge \overline{q}_j(t^-) \neq \overline{q}_j(t) \\ \widetilde{q}_j(t) & \text{si } \ddot{x}_j(\overline{\mathbf{q}}^j) \cdot \ddot{x}_j(\underline{\mathbf{q}}^j) < 0 \wedge \overline{q}_j(t^-) \neq \overline{q}_j(t) \\ q_j(t^-) & \text{en otro caso} \end{cases} \tag{3.31}$$

con $\overline{\mathbf{q}}^j = \mathbf{q}$ salvo que $q_j = \overline{q}_j$, $\underline{\mathbf{q}}^j = \mathbf{q}$ salvo que $q_j = \underline{q}_j$ y

$$\underline{q}_j(t) = \begin{cases} x_j(t_0) - \Delta Q_j & \text{si } t = t_0 \\ \underline{q}_i(t^-) + \Delta Q_j & \text{si } (x_j(t) = \underline{q}_j(t^-) + 2\Delta Q_j \\ \underline{q}_i(t^-) - \Delta Q_j & \text{si } (x_j(t) = \underline{q}_j(t^-) \\ \underline{q}_j(t_j) + mq_j(t_j) \cdot (t - t_j) + pq_j(t_j) \cdot (t - t_j)^2 & \text{en otro caso} \end{cases} \tag{3.32}$$

$$\overline{q}_j(t) = \underline{q}_j(t) + 2 \cdot \Delta Q_j \tag{3.33}$$

$$\widetilde{q}_j(t) = \begin{cases} \frac{-pIn_j(t)}{A_{jj}^3} - \frac{mIn_j(t)}{A_{jj}^2} - \frac{In_j(t)}{A_{jj}} & \text{si } A_{j,j} \neq 0 \\ q_j(t^-) & \text{en otro caso} \end{cases} \tag{3.34}$$

y

$$mq_j(t) = \begin{cases} A_{jj}q_j(t) + In_j(t) & \text{si } (q_j(t^-) \neq q_j(t) \\ mq_j(t_j) + pq_j(t_j) \cdot (t - t_j) & \text{en otro caso} \end{cases} \tag{3.35}$$

$$pq_j(t) = \begin{cases} pq_j(t^-) & \text{si } (q_j(t^-) = q_j(t) \\ \frac{A_{jj} \cdot mq_j(t)}{2} + \frac{mIn_j(t)}{2} & \text{en otro caso} \end{cases} \tag{3.36}$$

Las funciones $In_j(t)$,$mIn_j(t)$ y $pIn_j(t)$ se calculan según

$$In_j(t) = \dot{x}(t^-) - A_{jj} \cdot q_j(t^-) \tag{3.37}$$
$$mIn_j(t) = \ddot{x}(t^-) - A_{jj} \cdot mq_j(t^-) \tag{3.38}$$
$$In_j(t) = \dddot{x}(t^-) - A_{jj} \cdot pq_j(t^-) \tag{3.39}$$

Para aclarar un poco más esta definición, observar que en (3.31), $q_j(t) = \widetilde{q}_j$ sólo se da cuando $\dddot{x}_j(t) = 0$ y $A_{jj} \neq 0$. Además, cuando $q_j(t) = \widetilde{q}_j(t)$, $pq_j(t)$ es tal que la tercera derivada de x_j resulta cero, $\dot{x}_j(t) = mq_j(t)$ y $\ddot{x}_j(t) = pq_j(t)$.

3.4. Manejo de discontinuidades

En los métodos de tiempo discreto se deben detectar exactamente los instantes en los cuales ocurren las discontinuidades, ya que no se puede integrar pasando a través de una discontinuidad. En el caso de los eventos temporales simplemente se debe ajustar el paso para que éste coincida con el tiempo de ocurrencia del evento. Frente a eventos de estado, en cambio, se debe iterar para encontrar el instante preciso en el que ocurre dicho evento.

En el caso de los métodos de QSS, todos estos problemas desaparecen. Para poder ver esto, se analizar un ejemplo sencillo.

La figura 3.24 muestra dos posibles situaciones en que puede estar una pelota rebotando. Mientras la pelota está en el aire se tiene en cuenta la influencia de la fuerza de gravedad y del rozamiento del aire. Cuando la pelota está en contacto con el suelo, se considera a la misma como el modelo de un resorte comprimido con un amortiguador

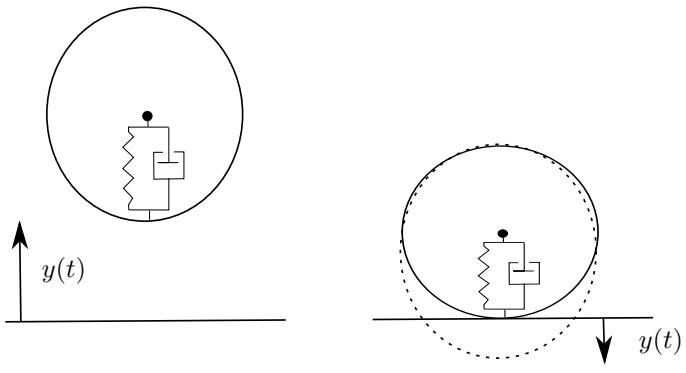

Figura 3.24: Pelota Rebotando

Este sistema puede ser modelado mediante las siguientes ecuaciones:

$$\dot{y}(t) = v(t) \tag{3.40a}$$
$$\dot{v}(t) = -g - sw \cdot \frac{1}{m}(k \cdot y(t) + b \cdot v(t)) \tag{3.40b}$$

donde:

$$sw = \begin{cases} 0 & \text{si } y(t) > 0 \\ 1 & \text{en otro caso} \end{cases} \tag{3.41a}$$

donde g es la aceleración de la gravedad, m es la masa de la pelota, k es la constante del resorte y b es el coeficiente de amortiguamiento del amortiguador.

El problema con este modelo es que el lado derecho de la última ecuación es discontinuo. Si un método de integración numérico realizase un paso a través de la discontinuidad, se obtendrá un resultado cuyo error sera inaceptable.

Como se mencionó antes, los métodos de integración convencionales resuelven este problema encontrando el instante exacto en que ocurre la discontinuidad. Luego, avanzan la simulación hasta ese instante y reinician la simulación a partir de ese instante con las condiciones iniciales obtenidas en el momento de la discontinuidad .

A pesar de que esta metodología generalmente funciona bien, agrega un costo computacional adicional importante, ya que encontrar el instante de ocurrencia de una discontinuidad implica realizar algunas iteraciones y además, reiniciar la simulación también puede ser costoso.

Como se vio en este capítulo, las trayectorias de los métodos QSS son seccionalmente constantes, lineales o parabólicas según sea el orden del método. En consecuencia, el instante de ocurrencia de un evento puede ser detectado analíticamente.

A pesar de que el evento asociado a una discontinuidad debe ocurrir en el instante exacto de ocurrencia de la misma, los métodos QSS no precisan ser reinicializados luego del mismo. El motivo es simplemente que las discontinuidades ocurren regularmente en los métodos QSS dado que las trayectorias $q_i(t)$ en los mismos son discontinuas. En consecuencia, en los métodos QSS las discontinuidades tienen el mismo efecto que un paso normal [7].

Analizaremos la implementación PowerDEVS del ejemplo de la pelota rebotando mostrado en la Figura 3.25.

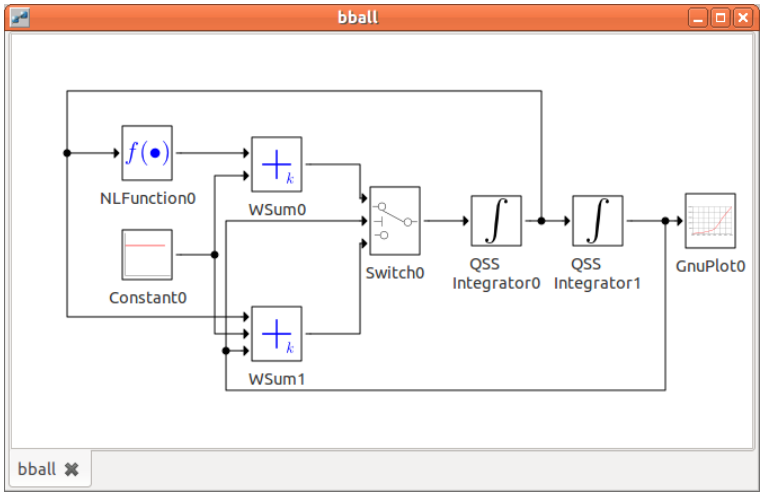

Figura 3.25: Modelo PowerDEVS de una pelota rebotando

El bloque 'Switch0' es el encargado de modelar y manejar la discontinuidad del modelo. Cada vez que recibe los sucesivos valores y pendientes de $y(t)$ por su segundo puerto de entrada, resuelve una ecuación cuadrática (se utilizaron integradores QSS3 en los cuales las trayectorias son seccionalmente parabólicas) para calcular el instante del próximo cruce y agenda para ese instante su

próxima transición interna. Mientras tanto, el resto del sistema continua con la simulación. Cuando llega un nuevo valor de $y(t)$, el switch reagenda el tiempo para su próximo evento de salida. Cuando finalmente se llega a ese tiempo, el switch genera un evento con el nuevo valor de la derivada $\dot{v}(t)$. El integrador que calcula $v(t)$ recibe ese evento como un cambio normal en $\dot{v}(t)$ y lo trata como un evento normal.

En otras palabras, el bloque 'Switch0' maneja localmente las discontinuidades mientras que el resto de los bloques no saben de su presencia.

Esta es la razón por la cual los métodos QSS son en general más eficientes que los métodos convencionales en el manejo de discontinuidades. En aquellos sistemas en los que ocurren discontinuidades más rápidamente que la dinámica del sistema continuo, lo cual suele ser muy común en los modelos de circuitos de electrónica de potencia, los métodos QSS reducen significativamente el tiempo computacional requerido en la simulación en comparación con los métodos tradicionales de simulación de ODEs [7].

3.5. Propiedades teóricas de los métodos QSS

Usando notación vectorial, el sistema:

$$\begin{aligned} \dot{x}_{a_1} &= f_1(x_{a_1}, x_{a_2}, \cdots, x_{a_n}, u_1, \cdots, u_m) \\ \dot{x}_{a_2} &= f_2(x_{a_1}, x_{a_2}, \cdots, x_{a_n}, u_1, \cdots, u_m) \\ &\vdots \\ \dot{x}_{a_n} &= f_n(x_{a_1}, x_{a_2}, \cdots, x_{a_n}, u_1, \cdots, u_m) \end{aligned}$$

toma la forma:

$$\dot{\mathbf{x}}_\mathbf{a}(t) = \mathbf{f}(\mathbf{x}_\mathbf{a}(t), \mathbf{u}(t)) \tag{3.42}$$

mientras que la aproximación QSS

$$\begin{aligned} \dot{x}_1 &= f_1(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m) \\ \dot{x}_2 &= f_2(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m) \\ &\vdots \\ \dot{x}_n &= f_n(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m) \end{aligned}$$

se puede escribir como

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t), \mathbf{u}(t)) \tag{3.43}$$

Definiendo $\mathbf{\Delta x}(t) = \mathbf{q}(t) - \mathbf{x}(t)$, esta última se puede reescribir como:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t) + \mathbf{\Delta x}(t), \mathbf{u}(t)) \tag{3.44}$$

Esta última ecuación sólo difiere del sistema original (3.42) por la presencia del termino de perturbación $\mathbf{\Delta x}(t)$.

Una propiedad fundamental de las funciones de cuantificación usadas por los métodos QSS es que $x_i(t)$ y $q_i(t)$ nunca difieren entre sí en más que el quántum ΔQ_i. Entonces, cada componente $\Delta x_i(t)$ del vector $\mathbf{\Delta x}(t)$ está acotado por el quántum ΔQ_i.

Como consecuencia de este hecho, se puede analizar el efecto de usar las aproximaciones QSS como un problema de ODEs con perturbaciones acotadas.

Basándose en esta idea, la primera propiedad que se probó en el contexto de los métodos QSS fue la de *convergencia* [11]. El análisis muestra que la solución de la Ec.(3.43) aproxima a la solución de la Ec.(3.42) cuando el quántum ΔQ_i más grande se elige lo suficientemente chico. La importancia de esta propiedad reside en que se puede lograr un error de simulacinó arbitrariamente chico utilizando una cuantificación lo suficientemente chica.

Una condición suficiente que asegura que las trayectorias del sistema de ecuaciones (3.43) converge a las trayectorias de las Ecs.(3.42) es que la función $\mathbf{f}(\mathbf{x}(t), \mathbf{u}(t))$ sea localmente Lipschitz.

A pesar de que la convergencia es una propiedad teórica importante, no ofrece ninguna información cuantitativa sobre la relación entre el quántum y el error, y además no establece ninguna condición sobre el dominio de estabilidad.

Las propiedades de estabilidad de los métodos QSS fueron estudiadas en [11] buscando una función de Lyapunov para el sistema perturbado. Este análisis muestra que cuando el sistema de la Ec.(3.42) tiene un punto de equilibrio asintóticamente estable, para cualquier región arbitrariamente pequeña entorno al punto de equilibrio se puede encontrar una cuantificación tal que la solución de la Ec.(3.43) termine dentro de esa región. Además, a partir de este análisis se puede obtener un algoritmo que permite calcular el quántum necesario.

Una condición suficiente que asegura la estabilidad numérica de los métodos QSS cerca de un punto de equilibrio analíticamente estable es que la función $\mathbf{f}$ sea continua y continuamente diferenciable. Por lo tanto, la condición de estabilidad es más fuerte que la de convergencia.

Entonces, los métodos de QSS poseen herramientas que pueden ser aplicadas a sistemas no lineales para elegir el quántum de manera de asegurar que el error de simulación en régimen permanente sea menor que una cota prefijada. A pesar de que este resultado representa una gran ventaja sobre los métodos de tiempo discreto clásicos, en los cuales la estabilidad se estudia usualmente en el contexto de sistemas lineales invariantes en el tiempo (LTI) , el algoritmo es algo complicado y requiere usar una función de Lyapunov de la Ec.(3.42) que en general no se puede obtener fácilmente. Luego, este análisis de estabilidad es más importante desde el punto de vista teórico que práctico.

Tal como en los métodos de tiempo discreto, la propiedad más interesante de los métodos QSS se obtiene del análisis de aplicar los mismos a sistemas LTI.

El resultado principal de este análisis se encuentra en [6] y establece que el error en la simulación QSS de un sistema LTI asintóticamente estable está siempre acotado. Esta cota de error puede ser calculada a partir del quántum y algunas propiedades geométricas del sistema y no depende ni de las condiciones iniciales, ni de las trayectorias de entrada. Además, permanece constante durante la simulación.

Un sistema LTI se puede escribir como:

$$\dot{\mathbf{x}}_\mathbf{a}(t) = \mathbf{A} \cdot \mathbf{x}_\mathbf{a}(t) + \mathbf{B} \cdot \mathbf{u}(t) \tag{3.45}$$

con $\mathbf{A}$ y $\mathbf{B}$ matrices de valores reales de $n \times n$ y $m \times n$ respectivamente, donde n es el orden del sistema y m es el número de entradas del mismo.

Una aproximación QSS de este sistema está dada por:

$$\dot{\mathbf{x}}(t) = \mathbf{A} \cdot \mathbf{q}(t) + \mathbf{B} \cdot \mathbf{u}(t) \tag{3.46}$$

Sea $\mathbf{x_a}(t)$ la solución de la Ec.(3.45) y $\mathbf{x}(t)$ la solución de la aproximación QSS de la Ec.(3.46) comenzando desde las mismas condiciones iniciales ($\mathbf{x_a}(t_0) = \mathbf{x}(t_0)$).

Para un sistema LTI analíticamente estable, el error global de simulación está acotado por la siguiente desigualdad:

$$|\mathbf{x}(t) - \mathbf{x_a}(t)| \preceq |\mathbf{V}| \cdot |\mathbb{R}\mathrm{e}\{\mathbf{\Lambda}\}^{-1} \cdot \mathbf{\Lambda}| \cdot |\mathbf{V}^{-1}| \cdot \mathbf{\Delta Q} \tag{3.47}$$

para todo $t \geq t_0$.

donde $\Lambda = V^{-1}AV$ es la descomposición modal de A, el vector $\mathbf{\Delta Q} \triangleq [\Delta Q_1, \cdots, \Delta Q_n]^T$ está compuesto por el conjunto de los quántum usados para cada estado, el símbolo $|\cdot|$ representa el valor absoluto componente a componente de un vector o matriz y el símbolo '$\preceq$' compara componente a componente vectores de igual largo.

Entonces, la Ec.(3.47) brinda una cota de error global de simulación para cada componente del vector de estado del sistema. Esta cota de error depende linealmente del quántum.

Se puede ver entonces que:

> Los métodos QSS tienen un control de error intrínseco, sin necesidad de recurrir a normas de adaptación.

Cabe notar que esta cota de error global implica que las soluciones numéricas no pueden divergir. Finalmente, una observación muy importante es:

> Los métodos QSS permanecen numéricamente estables sin necesidad de utilizar fórmulas implícitas.

Además del análisis basado en perturbaciones que se mostró en esta sección, en [14] se desarrolló un modo alternativo para demostrar la estabilidad de los métodos QSS haciendo uso de un equivalente de las aproximaciones QSS usando multirate de tiempo discreto.

3.6. Eficiencia

En este capítulo se presentaron los diferentes métodos de integración por cuantificación de estados, y vimos cómo pueden ser representados dentro del formalismo DEVS. En la Sección 3.2.1 observamos cómo dentro este formalismo se pueden especificar sistemas complejos de una manera sencilla mediante el acoplamiento de sistemas más simples, donde los eventos de salida de un subsistema se convierten en eventos de entrada de otros subsistemas y en la Sección 3.2.2 dimos la descripción de un algoritmo básico para simular este tipo de sistemas. En esta sección evaluaremos la eficiencia de las implementaciones de los métodos de QSS basados en el formalismo DEVS.

En un modelo DEVS, las ecuaciones de estado son representadas a través del acoplamiento de bloques atómicos agregando un integrador cuantificado a la salida. La principal desventaja de este tipo de implentaciones es el costo computacional que lleva la transmisión de eventos entre los distintos submodelos. Ya que cuando un modelo atómico realiza una transición interna, el evento de salida provocado debe ser propagado a todos los modelos atómicos conectados

al puerto de salida, ejecutando la función de transición externa correspondiente, lo que implica recalcular el tiempo de la próxima transición.

En la Figura 3.26 vemos nuevamente el modelo PowerDEVS de la pelotita picando, como ejemplo, estudiaremos los eventos que se generan al producirse una salida en el bloque *QSSIntegrator0*.

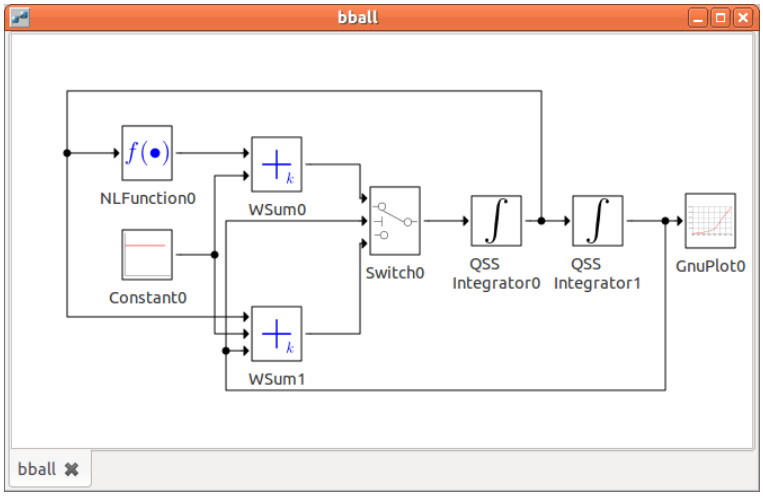

Figura 3.26: Modelo PowerDEVS de una pelota rebotando

Cuando el integrador *QSSIntegrator0* produce un evento de salida, se generan los siguientes eventos:

- Se propaga la salida a los bloques *NLFunction0*, *WSum1* y *QSSIntegrator1*, ejecutando una transición externa en cada uno de los bloques.
- Cuando el bloque *QSSIntegrator1* produce un evento de salida, se ejecuta una transición externa en los bloques *WSum1* *Switch0*, *GnuPlot0*.
- Cuando el bloque *NLFunction0* produce un evento de salida, se ejecuta una transición externa en el bloque *WSum0*.
- Luego, cuando *WSum0*, *WSum1* o *QSSIntegrator1* producen un evento de salida, se ejecuta una transición externa en el bloque *Switch0*.
- Por último, cuando se produce un evento de salida en *Switch0*, se ejecuta una transición externa en *QSSIntegrator0* y el ciclo se vuelve a repetir.

A medida que aumenta la complejidad en la representación de las ecuaciones de estado, se generan más eventos, y esto impacta en la eficiencia de la simulación. En los capítulos siguientes, presentaremos un motor de simulación que no está basado en el formalismo DEVS, y veremos cómo se pueden definir modelos para este motor sin la necesidad de generar eventos adicionales, mejorando de manera significativa la eficiencia de la simulación.

Capítulo 4

Formulación autonónoma.

En el Capítulo 2 se presentaron las características y propiedades de los métodos de integración clásicos, luego en el Capítulo 3 se describieron los métodos de integración por cuantificación, analizando sus principales propiedades y ventajas para simular sistemas contínuos híbridos, además en la Sección 3.2 se presentó el formalismo mátematico DEVS que se utiliza para especificar esta clase de métodos. En este capítulo y el siguiente se presentará la formulación autónoma de los métodos de integración por cuantificación y su implementación, que son la mayor contribución de este trabajo.

4.1. Idea básica

Todos los métodos de QSS se basan en resolver la ecuación $\dot{x} = f(q, t)$ donde q es una aproximación seccionalmente polinomial de x. Por lo tanto, la forma más simple de implementar en forma autónoma los distintos métodos es:

1. Implementar un motor de simulación que, asumiendo conocidas las secciones polinomiales de $q(t)$ integre la ecuación anterior.
2. Implementar *solvers* que calculen efectivamente $q(t)$ a partir de $x(t)$ según la definición de cada método de integración (QSS, QSS2, etc).

Cabe mencionar que, dado que el motor tiene que tener la posibilidad de evaluar individualmente las distintas derivadas y funciones de cruce, un modelo como el presentado en el Modelo 2.8 no sirve, sino que hay que usar modelos especiales. Como veremos luego, permitiremos el uso de modelos convencionales como el antes mencionado (ya que son mucho más simples para que un usuario los defina) a través de un parser que los convierte a la forma requerida por los algoritmos de QSS.

A continuación describiremos las características del motor de simulación, de los modelos, de los *solvers* y del parser requeridos para una implementación autónoma de los algoritmos.

4.2. Motor de simulación

En el Capítulo 2, vimos distintas formulaciones de motores de simulación adecuados para métodos de integración numérica de discretización temporal,

primero se presentaron los algoritmos 2.1 y 2.3 donde el método de integración estaba empotrado en el motor, luego presentamos las diferentes implementaciones para motores de simulación aptos para los distintos métodos de integración descriptos en ese capítulo. En las siguientes secciones veremos la formulación de un motor de simulación para métodos de integración por cuantificación de estados, comenzando por los sistemas autónomos sin manejar discontinuidades ni entradas, luego veremos el como manejar las discontuidades y por último, llegaremos a la formulación final, donde se tiene en cuenta el caso de los sistemas no autónomos.

4.2.1. Sistemas autónomos

En los algoritmos 2.4, 2.5 y 2.6 se presentaron distintos motores de simulación adecuados para los diferentes métodos de integración por discretización temporal. En estos ejemplos, en cada paso, el motor debe reevaluar cada una de las ecuaciones que definen el modelo:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t) \tag{4.1}$$

Como vimos en en el Capítulo 3, al utilizar métodos de integración por cuantificación, se obtiene un modelo de eventos discretos equivalente al modelo original, y en el caso de los métodos de integración tradicionales, se convierte el modelo original en un modelo de ecuaciones en diferencias equivalente.

Para el motor de simulación autónomo, la diferencia está en cuándo es necesario evaluar una ecuación de estado, dado que en el caso de los modelos de eventos discretos, solamente se evalúa una ecuación de estado cuándo cambia el valor de una de las variables que influencian la ecuación, por lo tanto, el motor debe ser capaz de diferenciar las definiciones de las ecuaciones de estado, para poder evaluar individualmente cada una de ellas, según sea necesario.

Las siguientes definiciones son necesarias para poder formular un algoritmo de simulación para métodos de cuantificación de estado.

Definición:
Dada una ecuación $eq = f(\mathbf{x}, \mathbf{d}, t)$

donde $\mathbf{x}$ es el *vector de estados*, $\mathbf{d}$ es el *vector de variables discretas*, y t representa el tiempo.

Llamaremos $InfX(eq)$ al conjunto de variables de estado que influyen en la definición de eq, es decir, a todas las variables de estado que aparecen en la definición $f(\mathbf{x}, \mathbf{d}, t)$.

Definición:
Dado un vector $\mathbf{x}$. Llamaremos $\#\mathbf{x}$ al número de elementos del vector, es decir, su dimensión.

Definición:
Dado:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t) \tag{4.2}$$

donde $\mathbf{x}$ es el *vector de estados*, y t representa el tiempo, llamaremos $MI^{n \cdot n}$ a la matriz de incidencia del sistema, donde $n = \#\mathbf{x}$, definida de la siguiente manera:

$$MI(i,j) = \begin{cases} 1 & si\ x_j \in InfX(\dot{x_i}) \\ 0 & en\ caso\ contrario. \end{cases} \tag{4.3}$$

Teniendo en cuenta la definición de los métodos de cuantificación de estados presentados en la Sección 3.3.1 y las definiciones dadas anteriormente, un motor de simulación para sistemas autónomos se puede formular de la siguiente manera:

Algoritmo 4.1: Sistemas autónomos

```
1  Engine(f, method, it, ft, x0, dq)
2  begin
3    t := it;
4    x := x0;
5    q := x0;
6    tx := it;
7    (t, i) := Proximo_Tiempo_Integracion(tx);
8    While (t ≤ ft)
9    begin
10     x_i := Actualizar_Tiempo(x_i);
11     q_i := Actualizar_Variable_Cuantificada(method, x_i, q_i);
12     tx_i := Proximo_Tiempo(method, x_i, q_i, t, dq_i);
13     for j:=1 to #x do
14     begin
15       if (MI[i, j] = 1) then
16       begin
17         x_i := Actualizar_Tiempo(x_i);
18         x_i := Actualizar_Derivada(f, j, q, t);
19         tx_i := Proximo_Tiempo(method, x_j, q_j, t, dq_j);
20       end
21     end
22     (t, i) := Proximo_Tiempo_Integracion(tx);
23   end
24 end
```

Donde:

- f Es la definición de las ecuaciones de estado del modelo.
- *method* Es el método de integración QSS utilizado.
- *it* Es el tiempo inicial de simulación.
- *ft* Es el tiempo final de simulación.
- **x0** Son los valores iniciales de las variables de estado.
- **dq** Vector que contiene los valores de ΔQ correspondientes a cada variable de estado.

Para inicializar el motor, se debe inicializar las variables de estado, las variables cuantificadas y obtener el tiempo del primer paso de integración junto con la variable que da el paso, luego, en cada paso el motor tiene que hacer lo siguiente:

1. Controlar el tiempo de simulación.
2. Actualizar el tiempo de la variable de estado x_i.
3. Actualizar el valor de la variable cuantificada $q_i(t)$, asociada a la variable que da el paso, $x_i(t)$, a través de la función de cuantificación con histéresis correspondiente al método utilizado, como las definidas en la Sección 3.3.
4. Calcular el próximo tiempo en el que la trayectoria de la variable de estado $x_i(t)$, cruza el nivel de discretización.
5. $\forall x_j \in InfX(x_i)$ actualizar la derivada $\dot{x_j}$ y luego recalcular el tiempo en que la trayectoria de la variable de estado x_j, cruza el nivel de discretización, esto depende del método de integración utilizado.
6. Obtener el tiempo del próximo paso, es decir, el menor de los tiempos asociados a las variables de estado (**tx**).

Ejemplo

Veamos en un ejemplo simple, la información que debemos obtener a partir de las ecuaciones de estado para poder ejecutar el algoritmo anteriormente descripto.

$$\begin{cases} \dot{x}_1(t) = x_2(t) \\ \dot{x}_2(t) = 1 - x_1(t) - x_2(t) \end{cases} \tag{4.4}$$

En este caso quedan definidos:

$$\begin{aligned} InfX(x_1) &= \{x_2\} \\ InfX(x_2) &= \{x_1, x_2\} \\ MI &= \begin{pmatrix} & x_1 & x_2 \\ x_1 & 0 & 1 \\ x_2 & 1 & 1 \end{pmatrix} \end{aligned} \tag{4.5}$$

Una vez definida la matriz de incidencia (4.5) resta definir las ecuaciones de estado del modelo.

Modelo 4.2: Ecuaciones de estado

```
1  f(j, q, t)
2  begin
3     if (j = 1) then
4     begin
5        return x2;
6     end
7     if (j = 2) then
8     begin
9        return 1 − x1 − x2;
10    end
11 end
```

En el Modelo 4.2 se da una definición adecuada para las ecuaciones de estado del sistema (4.4).

A partir de esta representación, se puede simular el sistema de la misma manera que con un motor de simulación de métodos de integración por discretización temporal.

4.2.2. Sistemas híbridos

En la Sección 2.5 presentamos el Algoritmo 2.7, donde se definió un motor de simulación que contemplaba el manejo de discontinuidades. Veremos ahora la formulación para métodos de estados cuantificados.

Básicamente, debemos considerar la misma situación que para los sistemas autónomos definidos en la sección anterior, pero teniendo en cuenta el manejo de discontinuidades, es decir, un motor de simulación para métodos tradicionales evalúa todas las funciones de cruce por cero correspondientes a los eventos definidos en el sistema en cada paso que da la simulación. Con los modelos de eventos discretos, tenemos dos situaciones posibles:

1. Solamente cuando una variable de estado que influye en una ecuación de cruce por cero, se debe actualizar el valor de la misma.

2. Cuando un paso en la simulación se da debido a un evento:

   a) En el caso de tratarse de un evento de estado, solamente se deben evaluar las ecuaciones del estado que son influenciadas por las variables modificadas en el handler del evento.

   b) Se deben evaluar las funciones de cruce por cero influenciadas por las variables modificadas en el handler del evento.

Por último, debemos poder diferenciar entre el conjunto de variables de estado y el conjunto de eventos definidos en el sistema, ya que las acciones a tomar en el caso de que una transición sea debido a un cambio en las variables de estado o debido a un evento son diferentes.

Daremos algunas definiciones previas a la formulación del motor para sistemas híbridos.

Definición:
Dado:

$$ev\_zc = \mathbf{f}(\mathbf{x}(t), \mathbf{d}, t) \tag{4.6}$$

donde $\mathbf{x}$ es el *vector de estados*, $\mathbf{d}$ es el *vector de variables discretas*, y t representa el tiempo, llamaremos $MIE^{n \cdot m}$ a la matriz de incidencia de eventos del sistema, donde $n = \#\mathbf{x}$ y $m = \#ev\_zc$, es decir, las funciones de cruce influenciadas por la variable de estado x_i, definida de la siguiente manera:

$$MIE(i,j) = \begin{cases} 1 & si\ x_i \in InfX(ev\_zc_j) \\ 0 & en\ caso\ contrario. \end{cases} \tag{4.7}$$

Definición:
Dada una ecuación $eq = f(\mathbf{x}, \mathbf{d}, t)$

donde $\mathbf{x}$ es el *vector de estados*, $\mathbf{d}$ es el *vector de variables discretas*, y t representa el tiempo.

Llamaremos $InfD(eq)$ al conjunto de variables discretas que influyen en la definición de eq, es decir, a todas las variables discretas que aparecen en la definición $f(\mathbf{x}, \mathbf{d}, t)$.

Definición:
Dado:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{d}(t), t) \tag{4.8}$$

donde $\mathbf{x}$ es el *vector de estados*, $\mathbf{d}$ es el *vector de variables discretas* y t representa el tiempo, llamaremos $MIDE^{n \cdot m}$ a la matriz de incidencia de variables discretas del sistema, donde $n = \#\mathbf{d}$ y $m = \#\mathbf{x}$, es decir, las variables de estado influenciadas por la variable discreta d_i, definida de la siguiente manera:

$$MIDE(i,j) = \begin{cases} 1 & si\ d_i \in InfD(\dot{x_j}) \\ 0 & en\ caso\ contrario. \end{cases} \tag{4.9}$$

Definición:
Dado:

$$ev\_zc = \mathbf{f}(\mathbf{x}(t), \mathbf{d}, t) \tag{4.10}$$

donde $\mathbf{x}$ es el *vector de estados*, $\mathbf{d}$ es el *vector de variables discretas*, y t representa el tiempo, llamaremos $MIFC^{n \cdot m}$ a la matriz de incidencia de funciones de cruce del sistema, donde $n = \#\mathbf{d}$ y $m = \#ev\_zc$, es decir, las funciones de cruce que son influenciadas por la variable discreta d_i, definida de la siguiente manera:

$$MIFC(i,j) = \begin{cases} 1 & si\ d_i \in InfD(ev\_zc_j) \\ 0 & en\ caso\ contrario. \end{cases} \tag{4.11}$$

La formulación para el caso de sistemas híbridos es la siguiente:

Algoritmo 4.3: Sistemas híbridos

```
1  Engine(f, method, it, ft, x0, d0, dq, ev_zc, ev_handler)
2  begin
3    t := it;
4    x := x0;
5    q := x0;
6    d := d0;
7    tx := it;
8    te := it;
9    (t, i) := Proximo_Tiempo_Integracion(tx, te);
10   While (t <= ft )
11   begin
12     if (i ∈ StateVariables) then
13       x_i := Actualizar_Tiempo(x_i);
14       q_i := Actualizar_Variable_Cuantificada(method, x_i, q_i);
15       tx_i := Proximo_Tiempo(method, x_i, q_i, t, dq_i);
16       for j:=1 to #x do
```

```
17        begin
18          if (MI[i,j] = 1) then
19          begin
20            x_i := Actualizar_Tiempo(x_i);
21            x_i := Actualizar_Derivada(f,j,q,d,t);
22            tx_i := Proximo_Tiempo(method,x_j,q_j,t,dq_i);
23          end
24        end
25        for j:=1 to #MIE[i] do
26        begin
27          if (MIE[i,j] = 1) then
28          begin
29            ev_zc_j := Actualizar_Evento(ev_zc_j,q,d,t);
30            te_j := Proximo_Tiempo_Evento(ev_zc_j,q,d,t);
31          end
32        end
33      end
34      if (i ∈ Events) then
35      begin
36        ev_handler_i(x,d,t);
37        for j ∈ infD(ev_handler_i) do
38        begin
39          for k:=1 to #x do
40          begin
41            if (MIDE[j,k] = 1) then
42            begin
43              x_k := Actualizar_Tiempo(x_k);
44              x_k := Actualizar_Derivada(f,k,q,d,t);
45              tx_j := Proximo_Tiempo(method,x_k,q_k,t,dq_k);
46            end
47          end
48          for k := 1 to #ev_zc do
49          begin
50            if (MIFC[j,k] = 1) then
51            begin
52              ev_zc_k := Actualizar_Evento(ev_zc_k,q,d,t);
53              te_k := Proximo_Tiempo_Evento(ev_zc_k,q,d,t);
54            end
55          end
56        end
57      end
58      (t,i) := Proximo_Tiempo_Integracion(tx,te);
59    end
60  end
```

Donde, además de los parámetros definidos en el Algoritmo 4.1, tenemos:

- *ev_zc* Definición de las funciones de cruce por cero correspondientes a los eventos.
- *ev_handler* Definición de funciones de los handlers correspondientes a los eventos.

Para inicializar el motor, se debe inicializar las variables de estado, las variables cuantificadas, las variables discretas, obtener el tiempo del primer paso

de integración y determinar el tipo de transición, luego, en cada paso el motor tiene que hacer lo siguiente:

1. Controlar el tiempo de simulación.
2. Si el paso se debe a un cambio en una variable de estado:
   a) Actualizar el tiempo de la variable de estado x_i.
   b) Actualizar el valor de la variable cuantificada $q_i(t)$, asociada a la variable que da el paso, $x_i(t)$, a través de la función de cuantificación con histéresis correspondiente al método utilizado, como las definidas en la Sección 3.3.
   c) Calcular el próximo tiempo en el que la trayectoria de la variable de estado $x_i(t)$, cruza el nivel de discretización.
   d) $\forall x_j \in InfX(x_i)$ actualizar la derivada $\dot{x_j}$ y luego recalcular el tiempo en que la trayectoria de la variable de estado x_j, cruza el nivel de discretización, esto depende del método de integración utilizado.
   e) Para todos los eventos definidos en el sistema, si el evento es influenciado por x_i, tendremos que recalcular la función de cruce por cero correspondiente y actualizar el próximo tiempo para el evento correspondiente.
3. Si el paso se debe a un evento:
   a) Ejecutar el handler correspondiente al evento, $ev\_handler_i$.
   b) Luego, $\forall j \in InfD(ev\_handler_i)$
      - Para cada variable de estado, si es influenciada por d_j, tendremos que actualizar el tiempo de la variable de estado, recalcular el valor de la derivada y calcular el próximo tiempo de integración para esa variable.
      - Para cada función de cruce definida, si es influenciada por d_j, tendremos que recalcular el valor de la función de cruce correspondiente y actualizar el próximo tiempo para el evento.
4. Obtener el tiempo del próximo paso, es decir, el menor de los tiempos asociados a las variables de estado y a los eventos definidos en el sistema $(\mathbf{tx}, \mathbf{te})$.

Ejemplo

Utilizaremos el ejemplo de la pelotita rebotando definido en la Sección 2.5, agregando la definición del evento correspondiente, para mostrar la información necesaria para simular este tipo de sistemas.

Recordemos la definción del sistema:

$$\dot{x}_1(t) = x_2(t) \tag{4.12a}$$

$$\dot{x}_2(t) = -g - d_1 \cdot \frac{1}{m}(k \cdot x_1(t) + b \cdot x_2(t)) \tag{4.12b}$$

Para este ejemplo, definiremos el evento de estado s_w siendo su función de cruce por cero y handler:

$$s\_w\_zc = x_1(t) \tag{4.13a}$$

$$s\_w\_handler = \begin{cases} d_1 = 0 & \text{si } x_1(t) > 0 \\ d_1 = 1 & \text{en otro caso} \end{cases} \tag{4.13b}$$

En este modelo, se considera que cuando $x_1(t) > 0$ la pelotita está en el aire y responde a una ecuación de caída libre ($d_1 = 0$). Cuando la pelotita entra en contacto con el piso, en cambio, sigue un modelo *masa–resorte–amortiguador* ($d_1 = 1$), lo que produce el rebote.

A partir de las definiciones Ec. (4.12) y (4.13), podremos extraer la información que necesitamos para simular el sistema, a saber:

$$\begin{aligned} InfX(x_1) &= \{x_2\} \\ InfX(x_2) &= \{x_1, x_2\} \\ InfD(s\_w\_handler) &= \{d_1\} \\ MI &= \begin{pmatrix} & x_1 & x_2 \\ x_1 & 0 & 1 \\ x_2 & 1 & 1 \end{pmatrix} \\ MIE &= \begin{pmatrix} & s\_w\_zc \\ x_1 & 1 \\ x_2 & 0 \end{pmatrix} \\ MIDE &= \begin{pmatrix} & x_1 & x_2 \\ d_1 & 0 & 1 \end{pmatrix} \\ MIFC &= \begin{pmatrix} & s\_handler \\ d_1 & 0 \end{pmatrix} \end{aligned} \tag{4.14}$$

En el Modelo 4.4, se muestra una posible representación del sistema 4.12.

Modelo 4.4: Ecuaciones de estado y eventos

```
1  f(j, q, d, t)
2  begin
3     if (j = 1) then
4     begin
5        return x_2;
6     end
7     if (j = 2) then
8     begin
9        return -g - d_1 · (1/m)(k · x_1(t) + b · x_2(t));
10    end
11 end
12
13 ev_zc(q, d, t)
14 begin
15    return x_1;
16 end
17
```

```
18 ev_handler(x, d, t)
19 begin
20   if (x_1 > 0) then
21   begin
22     return 0;
23   else
24     return 1;
25   end
26 end
```

Con las matrices y vectores definidos en Ec. (4.14) y el Modelo 4.4 podremos simular el sistema utilizando el motor definido en esta sección.

4.2.3. Sistemas no autónomos

Con la formulación utilizada para los motores de las secciones anteriores, no podremos manejar sistemas no autónomos, es decir, sistemas que tienen ecuaciones de estado que dependen explícitamente del tiempo. Dado el sistema:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t) \tag{4.15}$$

En un motor de simulación para un método de integración clásico, todas las ecuaciones de estado son evaluadas, incluidas las no autónomas, en cada paso de integración y cuando tratamos con sistemas de eventos discretos, las ecuaciones de estado que dependan de $u_i(t)$ sólo se deben actualizar cuando se produce un cambio en su entrada.

Es necesario poder distinguir entre los distintos tipos de transiciones posibles:

1. El motor da un paso debido a una transición en una variable de estado.
2. El motor da un paso debido a un evento.
3. El motor da un paso debido a una entrada.

La formulación final que contempla el caso no autónomo es la siguiente:

Algoritmo 4.5: Formulación final

```
1  Engine(f, method, it, ft, x0, d0, dq, ev_zc, ev_handler)
2  begin
3    t := it;
4    x := x0;
5    q := x0;
6    d := d0;
7    tx := it;
8    te := it;
9    ti := it;
10   (t, i) := Proximo_Tiempo_Integracion(tx, te, ti);
11   While (t <= ft )
12   begin
13     if (i ∈ StateVariables) then
14       x_i := Actualizar_Tiempo(x_i);
15       q_i := Actualizar_Variable_Cuantificada(method, x_i, q_i);
16       tx_i := Proximo_Tiempo(method, x_i, q_i, t, dq_i);
17       for j:=1 to #x do
```

```
18        begin
19          if (MI[i,j] = 1) then
20          begin
21            x_i := Actualizar_Tiempo(x_i);
22            x_i := Actualizar_Derivada(f,j,q,d,t);
23            tx_i := Proximo_Tiempo(method,x_j,q_j,t,dq_i);
24          end
25        end
26        for j:=1 to #MIE[i] do
27        begin
28          if (MIE[i,j] = 1) then
29          begin
30            ev_zc_j := Actualizar_Evento(ev_zc_j,q,d,t);
31            te_j := Proximo_Tiempo_Evento(ev_zc_j,q,d,t);
32          end
33        end
34      end
35      if (i ∈ Events) then
36      begin
37        ev_handler_i(x,d,t);
38        for j ∈ infD(ev_handler_i) do
39        begin
40          for k:=1 to #x do
41          begin
42            if (MIDE[j,k] = 1) then
43            begin
44              x_k := Actualizar_Tiempo(x_k);
45              x_k := Actualizar_Derivada(f,k,q,d,t);
46              tx_k := Proximo_Tiempo(method,x_k,q_k,t,dq_k);
47            end
48          end
49          for k := 1 to #ev_zc do
50          begin
51          if (MIFC[j,k] = 1) then
52          begin
53            ev_zc_k := Actualizar_Evento(ev_zc_k,q,d,t);
54            te_k :=Proximo_Tiempo_Evento(ev_zc_k,q,d,t);
55          end
56        end
57      end
58      if (i ∈ Inputs) then
59        x_i := Actualizar_Derivada(f,i,q,d,t);
60        tx_i := Proximo_Tiempo(method,x_i,q_i,t,dq_i);
61        ti_i := Proximo_Tiempo_Entrada(f,i,x_i,q_i,t);
62      end
63      (t,i) := Proximo_Tiempo_Integracion(tx,te,ti);
64    end
65  end
```

La única diferencia con el motor planteado en la sección anterior, es que ahora se tiene en cuenta cuando una ecuación de estado, cambia su valor debido a un cambio en la entrada. En tal situación, se debe recalcular la derivada de la ecuación de estado correspondiente y recalcular el próximo tiempo para esa variable.

4.3. Solvers

En la Sección 3.3, se dio la definición de los distintos *solvers* QSS en el marco del formalismo DEVS, en esta sección daremos la definición de los *solvers* QSS para el motor de simulación formulado en el Algoritmo 4.5. Para esto, identificaremos primero cuáles son las partes del motor que dependen del solver utilizado.

En el Algoritmo 4.5, podemos distinguir dos tipos de llamadas a función:

1. Llamadas que dependen del orden del método.

2. Llamadas que dependen del método de integración utilizado.

En los algoritmos 4.1, 4.3 y 4.5, podemos ver que las únicas llamadas del motor que dependen del método de integración utilizado son : *Actualizar_Variable_Cuantificada*$(method, x_i, q_i)$ y *Proximo_Tiempo*$(method, x_i, q_i, t, dq_i)$. Por lo que, en principio, para escribir un solver QSS que funcione con el motor de simulación propuesto, será necesario definir solamente estas dos funciones para cada método.

Interfaz 4.6: Solver

```
Actualizar_Variable_Cuantificada(method, x, q);
Proximo_Tiempo(method, x, q, t, dq);
```

Además de la definición de los *solvers*, para el funcionamiento del motor, es necesario dar las definiciones de las llamadas a función que dependen del orden del método, *Actualizar_Tiempo(x)*, *Actualizar_Derivada*$(f, j, \mathbf{q}, \mathbf{d}, t)$, *Actualizar_Evento*$(ev\_zc, \mathbf{q}, \mathbf{d}, t)$, *Proximo_Tiempo_Evento*$(ev\_zc, \mathbf{q}, \mathbf{d}, t)$ y *Proximo_Tiempo_Entrada*(f, j, x, q, t). Lo que lleva a plantear la siguiente interfaz para el entorno del motor:

Interfaz 4.7: Entorno

```
Actualizar_Tiempo(x);
Actualizar_Derivada(f, j, q, d, t);
Actualizar_Evento(ev_zc, q, d, t);
Proximo_Tiempo_Evento(ev_zc, q, d, t);
Proximo_Tiempo_Entrada(f, j, x, q, t);
```

4.4. Conversión de modelos

En las secciones anteriores, vimos la fomulación del motor de simulación para métodos de cuantifiación de estados y la interfaz necesaria para los *solvers* QSS, separando las funciones que son propias de cada solver, de las funciones que dependen del orden del método. En esta sección daremos un marco sencillo en el cuál se puedan definir los modelos para este entorno de simulación.

Como mencionamos al comienzo de este capítulo y mostramos en el Algoritmo 4.5, el motor de simulación tiene que tener la posibilidad de evaluar individualmente las distintas derivadas, por lo que no es posible representar un modelo de igual manera que en un motor de simulación tradicional en el que se evalúan todas las derivadas al mismo tiempo, a su vez, este tipo de definiciones resultan

mucho más simples para el usuario. Por lo tanto, es importante contar con una herramienta que permita definir modelos de esta manera, y transforme estos modelos para poder ser simulados por el motor de simulación aquí propuesto.

En el entorno de simulación, un modelo está compuesto de las siguentes secciones:

1. *Constants* Donde se declaran las constantes del modelo.
2. *Parameters* Donde se declaran los parámetros del modelo.
3. *Initial Values* Donde se definen todos los valores iniciales del sistema: variables de estado, quántum y quántum relativo para cada variable de estado, variables discretas y parámetros.
4. *Equations* La definición de las ecuaciones de estado del sistema.
5. *Event* Declaración de evento, con su respectiva función de cruce por cero (*Fcross*) y handler (*Handler*).

Además de las secciones mencionadas anteriormente, para definir un modelo, se utiliza:

- $x[i]$ que representa a las variables de estado del sistema.
- $d[i]$ que representa a las variables discretas del sistema.
- $dqmin[i]$ que representa el valor ΔQ_i correspondiente a cada variable de estado.
- $dqrel[i]$ que representa el valor ΔQ_{rel_i} correspondiente a cada variable de estado.
- $der(x[i])$ que representa la derivada de $x[i]$.

Como ejemplo, veamos la definición de un modelo de la pelotita rebotando, planteado en Ec. (4.12).

Modelo 4.8: Modelo pelotita rebotando

```
1  Model
2  {
3    InitialValues
4    {
5      x[0] = 1;
6      x[1] = 0;
7      d[0] = 0;
8      dqmin[0] = 1e-6;
9      dqmin[1] = 1e-6;
10     dqrel[0] = 1e-3;
11     dqrel[1] = 1e-3;
12   }
13   Parameters
14   {
15     g = 9.8;
16     k = 1e6;
17     b = 30;
```

```
18      m = 1;
19    }
20    Equations
21    {
22      der(x[0]) = x[1];
23      der(x[1]) = -g - d[0] * (1/m) * (k * x[0] + b * x[1]);
24    }
25    Event
26    {
27      Fcross
28      {
29        x[0];
30      }
31      Handler
32      {
33        d[0] = 1 - d[0];
34      }
35    }
36    End
37  }
```

La principal función de la conversión de modelos, es la de llevar el Modelo 4.8 a la forma descripta en el Modelo 4.4, y además, extraer la información estructural necesaria para poder ejecutar la simulación.

Cabe mencionar que esta conversión no es estrictamente necesaria desde el punto de vista funcional de la formulación autónoma aquí presentada, pero es una característica importante para los usuarios.

Capítulo 5

Implementación

En el Capítulo 4 dimos una descripción de cómo implementar un motor de simulación para métodos de integración por cuantificación de estados, en este capítulo veremos los detalles de implementación del mismo y además presentaremos el entorno de simulación desarrollado.

El desarrollo del entorno de simulación completo, se divide en tres módulos diferentes:

1. Motor de simulación.
2. Generador de modelos.
3. Interfaz gráfica de usuario.

Daremos aquí una breve descripción de estos módulos, y el las secciones siguientes, veremos los detalles de implementación.

En la Figura 5.1 se pueden ver cada una de las capaz que componen el motor de simulación, donde se distinguen cuatro estratos, uno de ellos transversal, donde:

- En el estrato transversal, se definen todas las estructuras de datos necesarias para el motor de simulación.
- En el estrato superior, contiene la implementación del motor definido en el Algoritmo 4.5.
- En el segundo estrato, se definen los distintos *solvers* utilizados y el entorno, definidos en las interfaces 4.6 y 4.7, así como también una interfaz para representar los modelos en el simulador.
- En el estrato inferior, se definen todas las utilidades extras que necesitan los estratos superiores, por ejemplo, los métodos de búsqueda, donde y cómo se guardan los datos de simulación, etc.

En la Figura 5.2, se ven los distintos estratos que componen el generador de código, que consiste de un parser, para los modelos definidos en la Sección 4.4 y un generador de código, que es el encargado de generar el archivo que implementa la interfaz del modelo en el simulador, y de obtener los datos que dependen de la formulación del modelo.

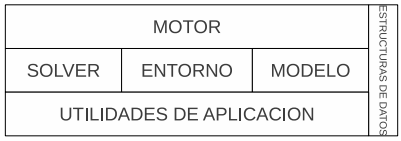

Figura 5.1: Arquitectura del motor de simulación

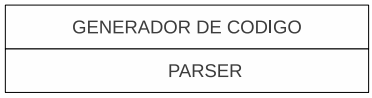

Figura 5.2: Arquitectura del generador de código

Por último, la Figura 5.3 muestra los distintos estratos que componen la interfaz de usuario del sistema. La función de este módulo, es permitir al usuario, definir los modelos de una forma más amigable y poder interactuar con el entorno de simulación.

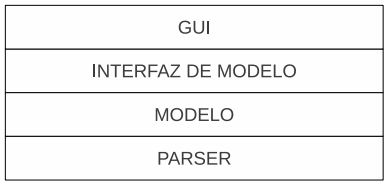

Figura 5.3: Arquitectura de la interfaz gráfica

Dentro del entorno de simulación, se encuentran librerias compiladas con las distintas configuraciones posibles para el motor de simulación, donde se distingue entre:

- Modelos de sistemas autónomos.
- Modelos de sistemas híbridos.

- Modelos con variables de salida definidas.
- El orden de los *solvers* utilizados.

Una vez que se genera el código del modelo a simular, al momento de compilar se linkea con la librería correspondiente. Esto produce el archivo ejecutable con el cuál se corre la simulación.

Para el desarrollo del motor de simulación, se utilizó el lenguaje de programación C, en tanto que el generador de código y la interfaz gráfica de usuario, se desarrollaron empleando el lenguaje de programación C++ y las librerias gráficas QT. Cabe mencionar que para el desarrollo se utilizaron herramientas de software libre, y que todo el software desarrollado está disponible libre bajo licencia GPL (General Pubic License).

En las siguientes secciones, daremos una descripción de los detalles de implementación de estos módulos.

5.1. Motor de simulación

En esta sección se explicarán los detalles de implementación del motor de simulación. Comenzaremos por las diferencias existentes con el motor presentado en el Algoritmo 4.5 y luego veremos las diferencias existentes en la implementación de los *solvers* presentados en la Interfaz 4.6 y en el entorno presentado en la Interfaz 4.7.

5.1.1. Motor

En la implementación del Algoritmo 4.5, debemos tener en cuenta:

- Para inicializar la simulación, debemos hacer lo siguiente:
  1. Obtener los valores iniciales de las derivadas de las variables de estado.
  2. Si el sistema es no autónomo, calcular el tiempo de la primera transición para todas las entradas.
  3. Si el sistema es híbrido, calcular el tiempo de la primera transición de todos los eventos.
  4. Inicializar los valores de salida de la simulación.
- Como vimos en la Sección 3.2.3, si se tiene un sistema general invariante en el tiempo:

$$\begin{array}{rcl} \dot{x}_1 & = & f_1(x_1, x_2, \cdots, x_n, u_1, \cdots, u_m) \\ \dot{x}_2 & = & f_2(x_1, x_2, \cdots, x_n, u_1, \cdots, u_m) \\ & \vdots & \\ \dot{x}_n & = & f_n(x_1, x_2, \cdots, x_n, u_1, \cdots, u_m) \end{array} \tag{5.1}$$

  puede transformarse en:

$$\begin{array}{rcl} \dot{x}_1 & = & f_1(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m) \\ \dot{x}_2 & = & f_2(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m) \\ & \vdots & \\ \dot{x}_n & = & f_n(q_1, q_2, \cdots, q_n, u_1, \cdots, u_m) \end{array} \tag{5.2}$$

donde $q_i(t)$ se relaciona con $x_i(t)$ mediante alguna función de cuantificación.

En el motor de simulación, por lo tanto, debemos mantener actualizados los valores de **x** y de **q** para poder simular el sistema. Internamente, cada componente de **x** y **q** se representan de la siguiente manera:

$$\begin{aligned} x_i &= a_0 + a_1 \cdot (t - t_{xi}) + a_2 \cdot (t - t_{xi})^2 + \cdots + a_n \cdot (t - t_{xi})^n \\ q_i &= b_0 + b_1 \cdot (t - t_{qi}) + b_2 \cdot (t - t_{qi})^2 + \cdots + b_{n-1} \cdot (t - t_{qi})^{n-1} \end{aligned} \tag{5.3}$$

donde $a_0, a_1, \cdots, a_n$ y $b_0, b_1, \cdots, b_{n-1}$ representan los coeficientes de la serie de Taylor. Es decir,

$$\begin{aligned} c_0 &= x_i \\ c_1 &= \frac{dx_i}{dt} \\ c_2 &= \frac{d^2 x_i}{dt^2 2!} \\ &\vdots \\ c_n &= \frac{d^n x_i}{dt^n n!} \end{aligned} \tag{5.4}$$

Por lo que para poder actualizar el valor tanto de **x** y **q** necesitaremos mantener t_{xi} y t_{qi} en el motor y actualizar sus valores cada vez que se modifique tanto el valor de x_i y q_i repectivamente (en los métodos de primer orden, la actualización de q_i no tiene ningún efecto, ya que las trayectorias son seccionalmente constantes).

- Como vimos en la Sección 3.3.4, para poder efectuar el control de error relativo, tenemos que calcular:

$$\Delta Q_i(t) = \max\left(E_{rel_i} \cdot |x_i(t)|, \Delta Q_{i_{min}}\right) \tag{5.5}$$

  cada vez que se actualiza q_i, para esto, además de los valores ΔQ_i, se deben definir E_{rel_i} para cada variable cuantificada.

- Cuando una transición se da debido a un cambio en un evento, se debe controlar el signo de la función de cruce por cero, en caso de que el signo de la función de cruce sea el mismo que en la última actualización, se debe recalcular el tiempo de cruce por cero del evento y no ejecutar el handler. Si el signo de la función cambia, se procede igual que en el Algoritmo 4.5.

- Internamente, para todas las matrices definidas en el sistema, solamente la infomación necesaria para simulación es almacenada, dando lugar a una representación no densa (*sparse*) más eficiente.

El código C del motor de simulación es:

Algoritmo 5.1: Motor de simulación

```
1 void simulate()
2 {
3   int i,j,stateVar = -1;
```

```
4   double elapsed,e;
5   double endTime, initTime = getTime();
6   while(t < simData->ft)
7   {
8    if(which == StateVar)
9    {
10    stateVar = index;
11    elapsed = t - simTime->tx[stateVar];
12    advanceTime(index,elapsed,simData->X,simData->order);
13    simTime->tx[stateVar] = t;
14    simData->lqu[stateVar] = updateQuantum();
15    updateQuantizedState(index,simData->Q,simData->X,simData->lqu);
16    simTime->tq[index] = t;
17    nextTime(stateVar,t,simTime->nextStepTime,simData->X,simData->lqu);
18    e = 0;
19    for(i = 0; i < simData->nI[stateVar]; i++)
20    {
21     j = simData->I[stateVar][i];
22     elapsed = t - simTime->tx[j];
23     if(elapsed > 0)
24     {
25      simData->X[0][j] = evaluatePoly(j,elapsed,simData->X,simData->order);
26     }
27     e += elapsed;
28     simTime->tx[j] = t;
29    }
30    e /= simData->nI[stateVar];
31    recomputeDerivatives(simModel,simData,simTime,e);
32    #ifdef HYBRID
33    for(i = 0; i < simData->nIE[stateVar]; i++)
34    {
35     j = simData->IE[stateVar][i];
36     elapsed = t - simTime->te[j];
37     nextEventTime(simModel,simData,simTime,elapsed,j);
38     simTime->te[j] = t;
39    }
40    #endif
41    recomputeNextTimes(simData->nI[index],simData->I[stateVar],t,simTime->
          nextStepTime,simData->X,simData->lqu,simData->Q);
42   }
43   #ifdef NON_AUTONOMOUS
44   else if(which == Input)
45   {
46    stateVar = simData->IT[index];
47    elapsed = t - simTime->tx[stateVar];
48    if(elapsed > 0)
49    {
50     simData->X[0][stateVar] = evaluatePoly(stateVar,elapsed,simData->X,simData->
          order);
51    }
52    recomputeDerivative(simModel,simData,simTime,elapsed,stateVar);
53    simTime->tx[stateVar] = t;
54    nextInputTime(simModel,simData,simTime,elapsed,stateVar,index);
55    recomputeNextTime(stateVar,t,simTime->nextStepTime,simData->X,simData->lqu,
          simData->Q);
56   }
57   #endif
58   #ifdef HYBRID
59   else if (which == Event)
60   {
61    double zc;
62    for(i = 0; i < simData->events[index].nIVars; i++)
63    {
64     j = simData->events[index].IVars[i];
65     simData->deltaValues[j] = evaluatePoly(j,t-simTime->tq[j],simData->Q,simData->
          order-1);
66    }
67    simModel->events->zeroCrossing(index,simData->deltaValues,simData->D,t,&zc,
          simData->Q[1]);
68    if(simData->events[index].zc_sign != _sign(zc) || simData->order == 1)
69    {
70     simModel->events->handler(index,simData->deltaValues,simData->D,t,simData->X[1])
          ;
71     simModel->events->zeroCrossing(index,simData->deltaValues,simData->D,t,&zc,
          simData->Q[1]);
72     simData->events[index].zc_sign = _sign(zc);
73     for(i = 0; i < simData->events[index].nFcrossDeps; i++)
74     {
75      j = simData->events[index].fcrossDeps[i];
76      elapsed = t - simTime->te[j];
77      nextEventTime(simModel,simData,simTime,elapsed,j);
78      simTime->te[j] = t;
79     }
80     for(i = 0; i < simData->events[index].nDependents; i++)
81     {
82      j = simData->events[index].dependents[i];
83      elapsed = t - simTime->tx[j];
84      if(elapsed > 0)
85      {
86       simData->X[0][j] = evaluatePoly(j,elapsed,simData->X,simData->order);
87      }
88      simTime->tx[j] = t;
89      recomputeDerivative(simModel,simData,simTime,elapsed,j);
90     }
91     recomputeNextTimes(simData->events[index].nDependents,simData->events[index].
          dependents,t,simTime->nextStepTime,simData->X,simData->lqu,simData->Q);
92    }
93    elapsed = t - simTime->te[index];
94    nextEventTime(simModel,simData,simTime,elapsed,index);
```

```
95      simTime->te[index] = t;
96     }
97     #endif
98     nextIntegrationTime();
99    }
100   endTime = getTime();
101   fprintf(logFile,"Simulation Time: %g\n",endTime-initTime);
102   #ifdef OUTPUT_VALUES
103   for(j = 0; j < simOutput->nOutVars; j++)
104   {
105    fprintf(logFile,"Variable x[%d] steps: %d\n",simOutput->outVars[j],getStepsOV(j));
106   }
107   #endif
108  }
```

5.1.2. Solvers

En la Sección 4.3 definimos la Interfaz 4.6 necesaria para implementar los métodos de integración por cuantificación en el contexto del motor de simulación presentado en el Algoritmo 4.5.

En la implementación de los diferentes *solvers*, las situaciones a tener en cuenta son:

- En el solver, antes de comenzar la simulación, se deben asignar los valores inciales de las variables de estado, las variables cuantificadas y en caso de estar definidas, las variables discretas.

- Cuando se calcula el próximo tiempo de transición de la variable de estado x_i, debemos calcular el tiempo en el que la variable de estado difiere de la variable de cuantificada q_i de acuerdo al control de error relativo definido en la el la sección anterior, es decir, calcular la diferencia entre los polinomios $|q_i - x_i| = \Delta Q_i$ definidos anteriormente.

  Cuando se debe actualizar la variable cuantificada q_i, debido a una transición en x_i, ambos polinómios difieren solamente en el último término, por lo tanto en ese caso el próximo tiempo de transición está definido por:

$$t^* = \frac{\Delta Q_i}{\frac{d^n x_i}{dt^n} n!} \tag{5.6}$$

  En otro caso, se debe calcular la próximo tiempo, obteniendo la mínima raíz positiva de $|q_i - x_i| = \Delta Q_i$, en la implementación de los *solvers*, estos casos están diferenciados, es decir, se agrega una función extra para calcular el próximo tiempo, cuando q_i y x_i son iguales y cuando difieren.

  Esta diferenciación se hace para evitar una llamada a función extra, mejorando la eficiencia de la simulación.

En este trabajo se implementaron siguientes *solvers*:

1. QSS
2. CQSS
3. LIQSS
4. QSS2
5. LIQSS2

6. QSS3

7. LIQSS3

Mostraremos aquí el código C del solver QSS, en el Apéndice A se encuentra el código del resto de los *solvers* implementados.

Algoritmo 5.2: QSS

```
1  #include <math.h>
2  #include "qss1.h"
3  #include "utility.h"
4  #include <stdio.h>
5
6  void initSolver_QSS1(SimulationData *simData, SimulationTime *simTime)
7  {
8   int i;
9   for(i = 0; i < simData->dim; i++)
10  {
11   simData->X[0][i] = simData->XInitValues[i];
12   simData->Q[0][i] = simData->X[0][i];
13   #ifdef HYBRYD
14   simData->deltaValues[i] = simData->XInitValues[i];
15   #endif
16  }
17  for(i = 0; i < simData->nDVars; i++)
18  {
19   simData->D[i] = simData->DInitValues[i];
20  }
21 }
22
23 void recomputeNextTime_QSS1(int var, double t,double *nTime, double **x, double *lqu
       ,double **q)
24 {
25  double coeff[2];
26  coeff[0] = q[0][var] - x[0][var] - lqu[var];
27  coeff[1] = -x[1][var];
28  nTime[var] = t + minPosRoot(coeff,1);
29  coeff[0] = q[0][var] - x[0][var] + lqu[var];
30  double timeaux = t + minPosRoot(coeff,1);
31  if(timeaux < nTime[var])
32  {
33   nTime[var] = timeaux;
34  }
35 }
36
37 void recomputeNextTimes_QSS1(int vars, int *inf, double t,double *nTime, double **x,
        double *lqu,double **q)
38 {
39  int i,j;
40  for(i = 0; i < vars; i++)
41  {
42   j = inf[i];
43   recomputeNextTime_QSS1(j,t,nTime,x,lqu,q);
44  }
45 }
46
47 void nextTime_QSS1(int var,double t,double *nTime, double **x, double *lqu)
48 {
49  if(x[1][var])
50  {
51   nTime[var]  = t + fabs(lqu[var]/x[1][var]);
52  }
53  else
54  {
55   nTime[var]  = INF;
56  }
57 }
58
59 void updateQuantizedState_QSS1(int i, double **q, double **x, double *lqu)
60 {
61  q[0][i] = x[0][i];
62 }
```

5.1.3. Entorno

En el caso de la implementación de la Interfaz 4.7 definida en la Sección 4.3, tenemos:

- En la implementación de $Proximo\_Tiempo\_Evento(ev\_zc, \mathbf{q}, \mathbf{d}, t)$, debemos calcular el tiempo del próximo cruce por cero, es decir, $ev\_zc(\mathbf{q}, \mathbf{d}, t) = 0$. En el motor de simulación, la representación de la función de cruce es la siguiente:

$$ev\_zc(\mathbf{q}, \mathbf{d}, t) = ev\_zc + \frac{dev\_zc}{dt} \cdot \Delta t + \frac{d^2 ev\_zc}{dt^2} \cdot \Delta t^2 + \cdots + \frac{d^{n-1} ev\_zc}{dt^{n-1}} \cdot \Delta t^{n-1} \tag{5.7}$$

Donde el parámetro Δ se define proporcional al tiempo final de simulación $\Delta = ft * \varepsilon$. Por lo tanto, para calcular el próximo tiempo de cruce, primero debemos obtener el valor actual de todas las variables de estado que influencian la función de cruce, utilizando la información dada en la definición 4.6 y evaluando los polinomios $q_i(t)$ en el tiempo actual de simulación. Luego, obtener t^*, calculando el valor de la mínima raiz positiva de la Ec. (5.7).

- Cuando tenemos una ecuación no autónoma, es decir,

$$\dot{\mathbf{x_i}}(t) = \mathbf{f}(\mathbf{x_i}(t), \mathbf{u_i}(t), t) \tag{5.8}$$

y se produce una transición debido a un cambio en $u_i(t)$, debemos estimar el tiempo en el que:

$|f(\mathbf{x_i}(t^*), u_i(t^*), t^*) - f(\mathbf{x_i}(t), u_i(t), t)| = \Delta Q_i$

debido a un cambio en $u_i(t)$, para esto, representaremos a $f(\mathbf{q}, \mathbf{d}, t)$ de la siguiente manera:

$$f(\mathbf{q}, \mathbf{d}, t) \quad = \quad f + \frac{df}{dt} \cdot \Delta t + \frac{d^2 f}{dt^2} \cdot \Delta t^2 + \cdots + \frac{d^{n-1} f}{dt^{n-1}} \cdot \Delta t^{n-1} + E_n \tag{5.9}$$

Donde

$$E_n \quad = \quad \frac{d^n f}{dt^n} \cdot \Delta t^n \tag{5.10}$$

Siendo n el orden del método utilizado y Δ esta definido como en el item anterior, por lo tanto, para calcular el próximo tiempo para la entrada debemos estimar el tiempo:

$$t^* \quad = \quad \frac{\frac{d^n f}{dt^n} \cdot \Delta t^n}{\Delta Q_i} \tag{5.11}$$

- Para estimar las derivadas de x_i debemos obtener el valor de cada q_i que influya en la definición de x_i, es decir, el valor actual de cada variable cuantificada definida en la ecuacion de estado $\dot{x_i}$, para esto, a partir de la matriz MI dada en la definición 4.2, se genera la matriz MI^T, donde quedan definidas las variables cuantificadas que influyen en cada ecuación de estado. Luego se debe estimar las derivadas calculando el cociente incremental, hasta el orden definido por el método utilizado y actualizar los coeficientes correspondientes del polinomio x_i.

- Para mejorar la efiencia de la simulación, cuando una transición se debe al cambio de una variable de estado o un evento, desde el motor se llama a una función que tiene la misma definición que en el item anterior, con la diferencia de que esto se hace para todas la ecuaciones influenciadas por la variable de estado x_i, por lo tanto, se evitan llamadas a funciones extra desde el motor.

Las funciones principales implementadas en el entorno son:

1. void recomputeDerivative(SimulationModel*,SimulationData*,SimulationTime*,double,int);
2. void recomputeDerivatives(SimulationModel*,SimulationData*,SimulationTime*,double);
3. void nextEventTime(SimulationModel*,SimulationData*,SimulationTime*,double,int);
4. void nextInputTime(SimulationModel*,SimulationData*,SimulationTime*,double,int,int);
5. void advanceTime(int,double,double**,int);
6. double evaluatePoly(int,double,double**,int);

En el Apéndice B se muestra el código C de parte de las principales funciones del entorno.

5.1.4. Modelo

En la Sección 4.4, vimos la definición necesaria para representar los modelos utilizados por el motor de simulación definido en el Algoritmo 4.5. En esta sección veremos la representación de un modelo dentro del marco de simulación desarrollado en este trabajo.

Para la implementación del modelo, se deben considerar dos situaciones:

- Se debe permitir evaluar de manera individual las ecuaciones de estado.
- Se debe permitir evaluar de manera individual las funciones de cruce por cero definidas.

En el Capítulo 6 veremos la definición de un modelo que representa una línea de transmisión (Ejemplo 6.2), a modo de ejemplo, en el Código 5.3 se muestra el código del archivo C que representa al sistema.

Donde podemos distinguir:

- La definición del modelo, líneas $1 - 14$
- La definición de la función de cruce, líneas $16 - 24$
- La definición del handler del evento, líneas $26 - 34$

Código 5.3: Línea de transmisión

```
1  void model(int i, double *x, double *d, double t, double *dx)
2  {
3   switch(i)
4   {
5    case 0:
6     *dx = d[0] - x[1];
7     return;
8    case 159:
9     *dx = x[158];
10    return;
11   default:
12    *dx = x[i-1] - x[i+1];
13  }
14 }
15
16 void model_zero_crossing(int i, double *x, double *d, double t, double *zc,double *
       dx)
17 {
18  switch(i)
19  {
20   case 0:
21    *zc = t - 1;
22    return;
23  }
24 }
25
26 void model_event_handler(int i, double *x, double *d, double t, double *dx)
27 {
28  switch(i)
```

```
29  {
30    case 0:
31      d[0] = 0;
32      return;
33    }
34  }
```

5.2. Generación de código

La función principal del módulo de generación de código, es la de generar un archivo en el lenguaje de programación C, que implemente la interfaz modelo definida en la Figura 5.1 que será utilizada en el motor de simulación, es decir, obtener a partir de la definición del modelo, toda la información necesaria para poder simular el sistema.

Un modelo es representado por un archivo *qsm*, en el cuál se pueden diferenciar dos secciones, en una sección se deben declarar las ecuaciones de estado, los eventos, parámetros y constantes del sistema de la misma manera que en la Sección 4.4. En una segunda sección se debe dar información que es necesaria para el entorno de simulación, y que no depende del modelo a simular, como por ejemplo, las variables de salida del sistema. Daremos ahora una breve descripción de los componentes de esta segunda sección del modelo:

- *Name* Nombre del modelo.
- *Description* Descripción del modelo.
- *Dimension* Dimension del modelo.
- *Solver* Solver utilizado en la simulación.
- *InitialTime* Tiempo inicial de simulación.
- *FinalTime* Tiempo final de simulación.
- *InitXCode* Define si el código de inicialización de las variables de estado, está en la definición del modelo.
- *SameStValue* Define si las variables de estado tienen el mismo valor inicial.
- *StateVarsInitialValue* Valores iniciales de las variables de estado.
- *InitDqminCode* Define si el código de inicialización del quántum mínimo de las variables de estado, está en la definición del modelo.
- *SameDqMinValue* Define si todas las variables de estado tienen el mismo valor de quántum mínimo.
- *DqMinValues* Define los valores de quántum mínimo para cada variable.
- *InitDqrelCode* Define si el código de inicialización del quántum relativo de las variables de estado, está en la definición del modelo.
- *SameDqRelValue* Define si todas las variables de estado tienen el mismo valor de quántum realtivo.
- *DqRelValues* Define los valores de quántum relativo para cada variable.

- *DiscreteVars* Define si las variables discretas tienen el mismo valor inicial.
- *InitDCode* Define si el código de inicialización de las variables discretas, está en la definición del modelo.
- *SameDscValue* Define si las variables de discretas tienen el mismo valor inicial.
- *DiscreteVarsInitialValue* Valores iniciales de las variables discretas.
- *CommInterval* Define, en caso de existir variables de salida en el sistema, cómo deben ser guardados estos valores.
- *SameOutputRate* Define, en el caso de que el tipo de salida seleccionado sea *Sampled*, el intervalo en el que se deben guardar los valores de la simulación.
- *SearchMethod* Define el tipo de búsqueda definido para el próximo tiempo de transición.
- *SaveValues* Define donde se guardan los datos de salida del sistema.
- *OutVars* Define la cantidad de variables de salida del sistema.
- *OutVarsNames* Define cuáles son las variables de estado de salida.
- *ModelDef* Utilizado para dar la definición del modelo propiamente dicha.

Explicaremos ahora en detalle algunos de los componentes del modelo:

- En el caso de que la simulación tenga valores de salida, en la definición del modelo, se puede distinguir entre tres tipos:
  - *Step* donde el valor de las variables de estado se guardan cada vez que se actualiza la variable cuantificada correspondiente, es decir, se guarda el valor de x_i, en todas las situaciones en que $|q_i - x_i| = 0$.
  - *Dense* donde cada vez que cambia el valor de $\dot{x_i}$ se guarda el valor de la variabe de estado x_i y todos los coeficientes del polinomio.
  - *Sampled* donde se guardan los mismos valores que en el caso anterior, pero solamente una vez en un intervalo de tiempo predefino, que también debe estar especificado en el modelo en el componente *SameOutputRate*.

  Por defecto, los datos de salida del sistema, se almacenan en memoria durante la simulación, y al finalizar se guardan en un archivo de texto que contiene los valores correspondientes al tipo de salida seleccionada. Existe también la posibilidad de guardar los datos de salida en un archivo de texto durante la simulación.
- El tipo de búsqueda para el próximo tiempo de transición, en este caso se puede distinguir entre:
  - *Lineal* donde se utiliza búsqueda lineal para calcular el próximo tiempo de transición.

- *Búsqueda binaria* donde se utiliza búsqueda binaria para el póximo tiempo de transición.

- En un modelo, la inicialización de las variables de estado, las variables discretas y los valores de quántum mínimo y relativo de las variables de estado, se pueden definir utilizando la sección *InitialValues* o dando su definición por separado. Esto se refleja en la interfaz de usuario, ambas definiciones son equivalentes, y su elección depende de cuánto ayudan a simplifcar la definición del modelo.

- Cuando se declaran ecuaciones de estado y eventos, se puede utilizar la sentencia $Case(a < i < b)$ para dar definiciones genéricas. Esta sentencia es transformada a un ciclo *for* por el generador de código, y es de gran ayuda para la definición de modelos grandes, ya que reduce la complejidad de su formulación y del código generado. En el Capítulo 6, veremos modelos que utilizan este tipo de definiciones.

En el entorno, existe un parser, que es utilizado por el generador de código y la interfaz gráfica de usuario, que es el encargado de transformar un archivo *qsm* en un objeto en el lenguaje C++, que luego es utilizado por estos módulos.

En el archivo C generado, queda declarada toda la información que depende de cada modelo a simular, las distintas operaciones a realizar son:

- Se debe reservar la memoria para todas la estructuras de datos que se utilizan en el sistema, así como también inicializar mismas.

- Definir e inicializar cada una de las matrices definidas en la Sección 4, a saber:

  - $MI^{n \cdot n}$ la matriz de incidencia.
  - $MIE^{n \cdot m}$ la matriz de incidencia de eventos.
  - $MIDE^{n \cdot m}$ la matriz de incidencia de variables discretas.
  - $MIFC^{n \cdot m}$ la matriz de incidencia de funciones de cruce.

- Definir los parámetros y constantes del modelo.

- Generar la función que define al modelo con esta función se pueden evaluar cada una de las ecuaciones de estado del sistema, como vimos en el Modelo 4.2.

- Generar la función para evaluar todas las dependencias de cada ecuación de estado, esta función es utilizada para mejorar la eficiencia, ya que de esta manera, desde el motor de simulación se debe producir una sola llamada a función para actualizar todas las ecuaciones dependientes de una variable de estado.

- Generar la definición de las funciones de cruce correspondientes a cada uno de los eventos del sistema, como vimos en el Modelo 4.4.

- Generar la definición de las funciones encargadas de manejar los cambios realizados por los eventos del sistema, como vimos en el Modelo 4.4.

- Por último, se debe liberar la memoria reservada.

Además, el generador de código, es el encargado de crear el archivo Makefile, utilizado para compilar el modelo, linkeando el archivo C generado con la libreria correspondiente.

En resumen, el generador de código transforma un modelo *qsm* en un archivo C, donde quedan difinidas las funciones que representan al modelo convencional, como el definido en el Modelo 4.4 y toda la información estructural necesaria para correr la simulación. En el Código 5.4 se muestra el código completo del archivo C para el modelo presentado en la Sección 5.1.4 que representa una línea de transmisión, donde podemos ver:

- En las líneas $12 - 25$, se define *model*, que representa el modelo.
- En las líneas $27 - 50$, se define *model_deps*, la función que evalua las ecuaciones dependientes de una variable de estado.
- En las líneas $52 - 60$, se define *model_zero_crossing*, la función de cruce por cero.
- En las líneas $62 - 70$, se define *model_event_handler*, la función handler para el evento.
- En las líneas $72 - 234$, se define *initializeDataStructs*, la función encargada de inicializar todas las estructuras de datos y matrices utilizadas en el modelo, entre ellas podemos distinguir la definición de las matrices utilizadas:
  - En las líneas $110 - 130$, se inicializan los vectores que definen la cantidad de variables influenciadas en las distintas matrices, $localData->nI$, $localData->nQDeps$ y $localData->nIE$ correspondientes a las matrices $MI^{n \cdot n}$, MI^T y $MIE^{n \cdot m}$ respectivamente.
  - En las líneas $131 - 157$, se inicializan las matrices $localData->I$, $localData->IQ$ y $localData->IE$ que se corresponden con las matrices señaladas en el punto anterior.
- En las líneas $236 - 300$,se define *deInitializeDataStructs*, la función que libera la memoria utilizada.

Código 5.4: lcline.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <math.h>
4  #include "model.h"
5
6  static SimulationData   *localData   = NULL;
7  static SimulationTime   *localTime   = NULL;
8  static SimulationOutput *localOutput = NULL;
9  static SimulationModel  *localModel  = NULL;
10
11
12 void model(int i, double *x, double *d, double t, double *dx)
13 {
14  switch(i)
15  {
16   case 0:
17    *dx = d[0] - x[1];
18    return;
19   case 159:
20    *dx = x[158];
21    return;
22   default:
23    *dx = x[i-1] - x[i+1];
24  }
```

```
25 }
26
27 void model_deps(int i, double *x, double *d, double t, double *dx)
28 {
29  int j;
30  switch(i)
31  {
32   case 0:
33    dx[1] = x[0] - x[2];
34    return;
35   case 159:
36    dx[158] = x[157] - x[159];
37    return;
38   case 1:
39    dx[2] =  x[1] - x[3];
40    dx[0] =  d[0] - x[1];
41    return;
42   case 158:
43    dx[159] =  x[158];
44    dx[157] =  x[156] - x[158];
45    return;
46   default:
47    dx[i+1] =  x[(i+1)-1] - x[(i+1)+1];
48    dx[i-1] =  x[(i-1)-1] - x[(i-1)+1];
49  }
50 }
51
52 void model_zero_crossing(int i, double *x, double *d, double t, double *zc,double *
       dx)
53 {
54  switch(i)
55  {
56   case 0:
57    *zc = t - 1;
58    return;
59  }
60 }
61
62 void model_event_handler(int i, double *x, double *d, double t, double *dx)
63 {
64  switch(i)
65  {
66   case 0:
67    d[0] = 0;
68    return;
69  }
70 }
71
72 void initializeDataStructs(simInitData initDataStructs)
73 {
74  int j,i;
75  localData = (SimulationData*)malloc(sizeof(SimulationData));
76  localData->XInitValues = (double *)calloc(160,sizeof(double));
77  localData->deltaValues = (double *)calloc(160,sizeof(double));
78  localData->dQRel = (double*)calloc(160,sizeof(double));
79  localData->dQMin = (double*)calloc(160,sizeof(double));
80  localData->lqu = (double*)calloc(160,sizeof(double));
81  localData->dxOld = (double*)calloc(160,sizeof(double));
82  localData->dxNew = (double*)calloc(160,sizeof(double));
83  localData->Q = (double**)calloc(3,sizeof(double*));
84  localData->X = (double**)calloc(4,sizeof(double*));
85  localData->DInitValues = (double*)calloc(1,sizeof(double));
86  localData->D = (double*)calloc(1,sizeof(double));
87  localData->nI = (int*)calloc(160,sizeof(int));
88  localData->nIE = (int*)calloc(160,sizeof(int));
89  localData->IT = NULL;
90  localData->nQDeps = (int*)calloc(160,sizeof(int));
91  localData->I = (int**)calloc(160,sizeof(int*));
92  localData->IE = (int**)calloc(160,sizeof(int*));
93  localData->IQ = (int**)calloc(160,sizeof(int*));
94  localData->dim = 160;
95  localData->nDVars =1;
96  localData->nEvents = 1;
97  localData->nInputs = 0;
98  localData->order = 3;
99  localData->it = 0;
100 localData->ft = 200;
101 localData->events = (EventData*)calloc(1,sizeof(EventData));
102 localData->solver = QSS3;
103
104 for(i = 0; i < 3;i++)
105 {
106  localData->Q[i] = (double*)calloc(160,sizeof(double));
107  localData->X[i] = (double*)calloc(160,sizeof(double));
108 }
109 localData->X[3] = (double*)calloc(160,sizeof(double));
110 for(i = 0; i < 160; i++)
111 {
112  localData->nI[i] = 0;
113  localData->nIE[i] = 0;
114  localData->nQDeps[i] = 0;
115 }
116
117 localData->nI[0] = 1;
118 localData->nI[1] = 2;
119 localData->nI[158] = 2;
120 localData->nI[159] = 1;
121 for(i = 2; i <= 157;i++)
122 {
```

```
123   localData->nI[i] = 2;
124  }
125  localData->nQDeps[0] = 1;
126  localData->nQDeps[159] = 1;
127  for(i = 1; i <= 158;i++)
128  {
129   localData->nQDeps[i] = 2;
130  }
131  for(i = 0; i < 160;i++)
132  {
133   localData->I[i] = (int*)calloc(localData->nI[i],sizeof(int));
134   localData->IQ[i] = (int*)calloc(localData->nQDeps[i],sizeof(int));
135   localData->IE[i] = (int*)calloc(localData->nIE[i],sizeof(int));
136  }
137
138  localData->I[0][0] = 1;
139  localData->I[1][0] = 2;
140  localData->I[1][1] = 0;
141  for(i = 2; i <= 157;i++)
142  {
143   localData->I[i][0] = i+1;
144   localData->I[i][1] = i-1;
145  }
146  localData->I[158][0] = 159;
147  localData->I[158][1] = 157;
148  localData->I[159][0] = 158;
149
150
151  localData->IQ[0][0] = 1;
152  localData->IQ[159][0] = 158;
153  for(i = 1; i <= 158; i++)
154  {
155   localData->IQ[i][0] = i+1;
156   localData->IQ[i][1] = i-1;
157  }
158
159  for(i = 0; i < 1;i++)
160  {
161   localData->DInitValues[i] = 1;
162  }
163
164  for(i =0; i <= 159; i++)
165  {
166   localData->XInitValues[i] = 0;
167   localData->dQMin[i ] = 1e-6;
168   localData->dQRel[i] = 1e-3;
169  }
170
171  localData->events[0].nDependents = 1;
172  localData->events[0].dependents = (int*)calloc(localData->events[0].nDependents,
       sizeof(int));
173  localData->events[0].dependents[0] = 0;
174  localData->events[0].nIVars = 0;
175  localData->events[0].IVars = NULL;
176  localData->events[0].nFcrossDeps = 0;
177  localData->events[0].fcrossDeps = NULL;
178  localData->events[0].zc_sign = 0;
179  localData->events[0].zc_h = 0;
180
181  for(i = 0; i < 160;i++)
182  {
183   localData->lqu[i] = 0;
184  }
185
186  localTime = (SimulationTime*)malloc(sizeof(SimulationTime));
187  localTime->tsMode = BinarySearch;
188  localTime->which = -1;
189  localTime->minIndex = -1;
190  localTime->minValue = INF;
191  localTime->time = 0;
192  localTime->nextStepTime = (double*)calloc(160,sizeof(double));
193  localTime->tx = (double*)calloc(160,sizeof(double));
194  localTime->tq = (double *)calloc(160,sizeof(double));
195  for(i = 0; i < 160; i++){
196   localTime->nextStepTime[i] = 0;
197   localTime->tx[i] = 0;
198   localTime->tq[i] = 0;
199  }
200  localTime->dEvents = (double*)calloc(1,sizeof(double));
201  localTime->te = (double*)calloc(1,sizeof(double));
202  for(i = 0; i < 1; i++)
203  {
204   localTime->dEvents[i] = 0;
205   localTime->te[i] = 0;
206  }
207  localTime->iEvents = NULL;
208
209  localOutput = (SimulationOutput*)malloc(sizeof(SimulationOutput));
210  localOutput->modelName = "lc_line";
211  localOutput->nOutVars = 2;
212  localOutput->outVars = (int*)calloc(2,sizeof(double));
213  localOutput->varNames = (char**)calloc(2,sizeof(char*));
214  localOutput->outVars[0] = 158;
215  localOutput->outVars[1] = 159;
216  for(i = 0; i < localOutput->nOutVars; i++)
217  {
218   localOutput->varNames[i] = (char*)calloc(128,sizeof(char));
219  }
220  localOutput->varNames[0] = "x158";
```

```
221   localOutput->varNames[1] = "x159";
222   localOutput->method = StepOutput;
223   localOutput->output = Memory;
224   localOutput->ftOut = NULL;
225
226   localModel = (SimulationModel*) malloc(sizeof(SimulationModel));
227   localModel->f = model;
228   localModel->deps = model_deps;
229   localModel->events = (EventDef*)malloc(sizeof(EventDef));
230   localModel->events->zeroCrossing = model_zero_crossing;
231   localModel->events->handler = model_event_handler;
232
233   initDataStructs(localData,localTime,localOutput,localModel);
234 }
235
236 void deInitializeDataStructs()
237 {
238   int j;
239   free(localData->XInitValues);
240   free(localData->dQRel);
241   free(localData->dQMin);
242   free(localData->lqu);
243   for(j = 0; j < localData->order;j++)
244   {
245     free(localData->Q[j]);
246     free(localData->X[j]);
247   }
248   free(localData->X[localData->order]);
249   free(localData->Q);
250   free(localData->X);
251   free(localData->nI);
252   for(j = 0; j < localData->dim;j++)
253   {
254     free(localData->I[j]);
255   }
256   free(localData->I);
257   free(localData->deltaValues);
258   free(localData->nQDeps);
259   for(j = 0; j < localData->dim;j++)
260   {
261     free(localData->IQ[j]);
262   }
263   free(localData->IQ);
264   free(localData->dxNew);
265   free(localData->dxOld);
266   free(localData->DInitValues);
267   free(localData->D);
268   free(localData->nIE);
269   for(j = 0; j < localData->dim;j++)
270   {
271     free(localData->IE[j]);
272   }
273   free(localData->IE);
274   for(j = 0; j < localData->nEvents;j++)
275   {
276     if(localData->events[j].nIVars)
277     {
278       free(localData->events[j].IVars);
279     }
280     if(localData->events[j].nDependents)
281     {
282       free(localData->events[j].dependents);
283     }
284     if(localData->events[j].nFcrossDeps)
285     {
286       free(localData->events[j].fcrossDeps);
287     }
288   }
289   free(localData->events);
290   free(localData);
291   free(localTime->nextStepTime);
292   free(localTime->tx);
293   free(localTime->tq);
294   free(localTime->te);
295   free(localTime->dEvents);
296   free(localTime);
297   free(localOutput->outVars);
298   free(localOutput->varNames);
299   free(localOutput);
300 }
```

En la Sección 3.6, hicimos referencia a la eficiencia de la implementación de los métodos QSS basados en el formalismo DEVS, donde indicamos que actualizar el valor de las variables de estado tiene un costo computacional adicional debido a la transmisión de eventos entre submodelos del sistema. Utilizando la implementación del motor presentado en el Algoritmo 5.1, y modelos definidos como el el ejemplo anterior, vemos que para actualizar el valor de las ecuaciones de estado, solamente es necesario llamar a la función que contiene la definición del modelo, indicando cuál es la ecuación a evaluar, evitando de esta manera el costo adicional que implica la transmisión de información.

5.3. Interfaz de usuario

La función principal de la interfaz de usuario, es dar al usuario la posibilidad de definir modelos e interactuar con el entorno de simulación de una manera más amigable. Esta interfaz gráfica, se encuentra dividida en dos partes, una sección para el ingreso de datos del sistema y una segunda sección, que consta de un editor de texto, que permite ingresar la definición del modelo.

En la Figura 5.4 se muestra la pantalla de ingreso de datos del sistema, donde se puede definir:

- La dimensión del sistema.
- La cantidad de variables discretas.
- El tiempo inicial y final de simulación.
- El solver utilizado en la simulación.
- Las variables de salida del sistema.
- El tipo de salida del sistema.
- El tipo de búsqueda para el tiempo de simulación.
- La descripción del sistema.
- Los valores iniciales de las variables de estado.
- Los iniciales de las variables discretas.
- El quántum para cada una de las variables de estado.
- El quántum relativo para cada una de las variables de estado.

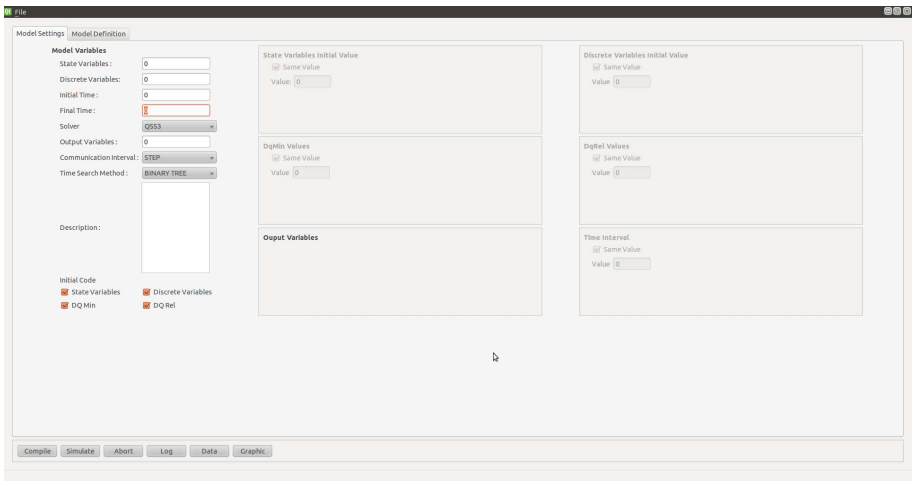

Figura 5.4: Variables del modelo

En la Figura 5.5 se muestra el editor de texto para definir los modelos.

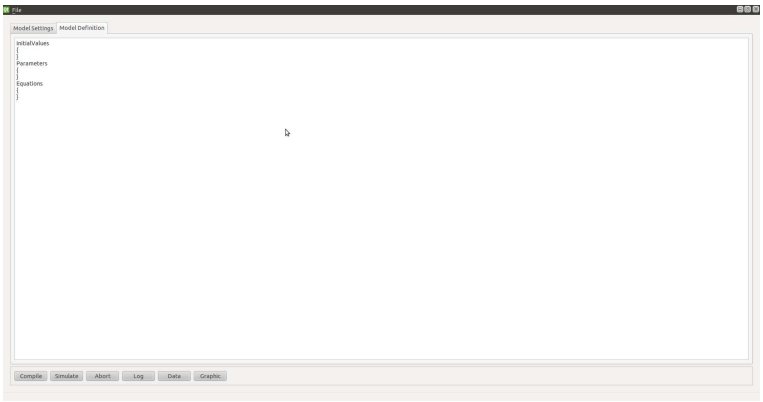

Figura 5.5: Editor de texto.

Por último, desde esta interfaz, se puede:

- Compilar el modelo.
- Correr la simulación.
- Ver el archivo de log de la simulación.
- Ver los archivos de datos de salida de la simulación.
- Ver las gráficas correspondientes a los archivos de salida.

Capítulo 6

Ejemplos

En este capítulo, veremos diferentes ejemplos, donde se presentará tanto la formulación de distinos modelos dentro del entorno de simulación desarrollado, así como también la comparación de los tiempos de simulación con el entorno de simulación PowerDEVS, ya que es el único donde se encuentran implementados todos los diferentes métodos de integración de estados cuantificados y es la implementación más eficiente. La comparación siempre es realizada teniendo en cuenta las condiciones óptimas para simular los diferentes modelos en los dos entornos, dado que no siempre se pueden definir las mismas condiciones. Los modelos *qsm* presentados en esta sección fueron generados utilizando la interfaz gráfica del entorno de simulación.

6.1. Pelota picando en escalera

El siguiente ejemplo representa una pelota picando en una escalera.

$$\dot{x} = v_x \tag{6.1a}$$

$$\dot{v}_x = -\frac{b_a}{m} \cdot v_x \tag{6.1b}$$

$$\dot{y} = v_y \tag{6.1c}$$

$$\dot{v}_y = -g - \frac{b_a}{m} \cdot v_y - s_w \cdot [\frac{b}{m} \cdot v_y + \frac{k}{m}(y - int(h + 1 - x))] \tag{6.1d}$$

donde

$$s_w = \begin{cases} 0 & \text{si } x(t) - h > 0 \\ 1 & \text{en otro caso} \end{cases} \tag{6.2}$$

Los valores de los parámetros son $g = 9{,}8$, $b = 30$, $b_a = 0{,}1$, $k = 1e6$ y $m = 1$ y condiciones iniciales $x = 0{,}575$, $v_x = 0{,}5$, $y = 10{,}5$ y $v_y = 0$. El sistema consta de 4 variables de estado, 2 variables discretas y 2 eventos.

En el Modelo 6.1, vemos la representación del sistema en el entorno de simulación QSS. Para simular el sistema se utilizó el método QSS3 con precisión $\Delta Q_{min} = 10^{-6}$ y $\Delta Q_{rel} = 0$ y se simuló el sistema durante 30 segundos.

En la Figura 6.1 se muestra la representación en el entorno PowerDEVS y en la Figura 6.2 vemos la salida del sistema.

Modelo 6.1: bball.qsm

```
Model
{
   Name=bball_downstairs;
   Description=Description;
   Dimension=4;
   Solver=QSS3;
   InitialTime=0;
   FinalTime=30;
   InitXCode=0;
   SameStValue=0;
   StateVarsInitialValue={10.5,0,0.575,0.5};
   InitDqminCode=0;
   SameDqMinValue=1;
   DqMinValues={1e-6};
   InitDqrelCode=0;
   SameDqRelValue=1;
   DqRelValues={0};
   DiscreteVars=2;
   InitDCode=0;
   SameDscValue=0;
   DiscreteVarsInitialValue={0,10};
   CommInterval=StepOutput;
   SameOutputRate=0;
   SearchMethod=BinarySearch;
   SaveValues=Memory;
   OutVars=1;
   OutVarsNames={0};
   ModelDef=
   Parameters
   {
      g = 9.8;
      b = 30;
      ba = 0.1;
      k = 1e6;
      m = 1;
   }
   Equations
   {
      der(x[0]) = x[1];
      der(x[1]) = -g - (ba/m) * x[1] - d[0] * ((x[0] - d[1]) * (k/m) + x[1] * (b/m));
      der(x[2]) = x[3];
      der(x[3]) = -(ba/m) * x[3];
   }
   Event
   {
      Fcross
      {
         x[0] - d[1];
      }
      Handler
      {
         d[0] = 1 -d[0];
      }
   }
   Event
   {
      Fcross
      {
         x[2] - 11 + d[1];
      }
      Handler
      {
         d[1] = d[1] - 1;
      }
   }
   End
}
```

Tabla 6.1: Tiempo de simulación

	Tiempo (milisegundos)
Motor Autónomo QSS	20 ms
PowerDEVS	413 ms

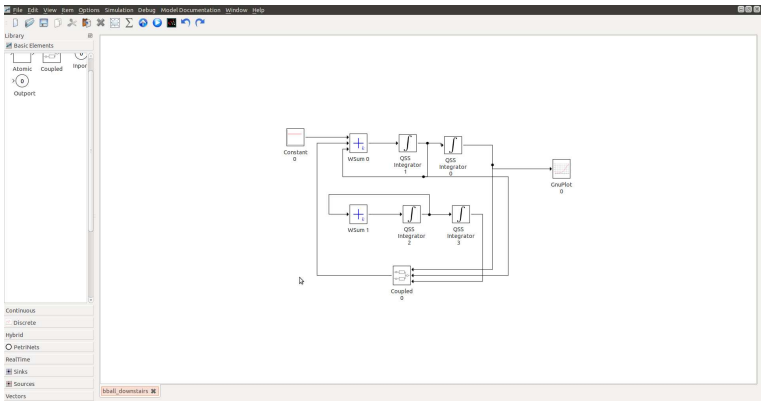

Figura 6.1: Modelo PowerDEVS para el Ejemplo 6.1

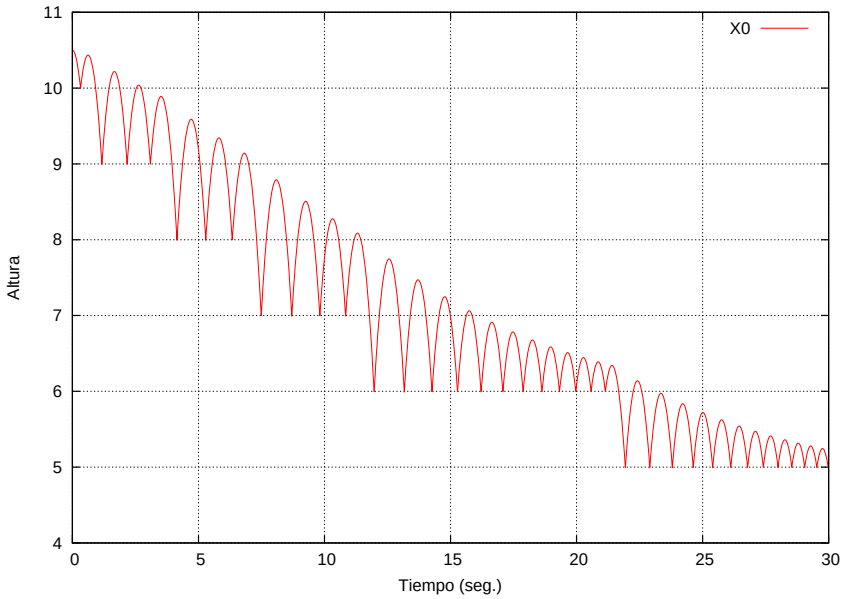

Figura 6.2: Salida del sistema para el Ejemplo 6.1

En la Tabla 6.1 comparamos el tiempo de simulación de ambos entornos, en este caso, el motor autónomo resulta ser más de 20 veces más rápido que PowerDEVS.

6.2. Linea de transmisión

El siguiente sistema representa una línea de transmisión conformada por n secciones de circuitos LC con parámetros $L = C = 1$.

$$
\begin{aligned}
\dot{u}_0(t) &= \begin{cases} 1 - u(t) & si\ t \leq 1 \\ -u(t) & en\ caso\ contrario. \end{cases} \\
\dot{\phi}_1(t) &= u_0(t) - u_1(t) \\
\dot{u}_1(t) &= \phi_1(t) - \phi_2(t) \\
&\vdots \\
\dot{\phi}_j(t) &= u_{j-1}(t) - u_j(t) \\
\dot{u}_j(t) &= \phi_j(t) - \phi_{j+1}(t) \\
&\vdots \\
\dot{\phi}_n(t) &= u_{n-1}(t) - u_n(t) \\
\dot{u}_n(t) &= -\phi_n(t)
\end{aligned}
\tag{6.3}
$$

Con las condiciones iniciales $u_i = \phi_i = 0,\ i = 1, \ldots, n$.

En el Modelo 6.2, vemos la representación del sistema en el entorno de simulación QSS, en este modelo se definen la constante $\#N$ que indica la cantidad de variables de estado. La ventaja de utilizar constantes, en este caso para definir la dimensión del sistema, es que permite modelar sistemas en los que la única diferencia se encuentra en la cantidad de variables definidas en el sistema y no en su comportamiento, que queda aislado en la declaración del modelo. Para simular el sistema se consideraron 160 variables de estado y 1 variable discreta, además de 1 evento temporal definido en el sistema, utilizando el método QSS3 con precisión $\Delta Q_{min} = 10^{-6}$ y $\Delta Q_{rel} = 10^{-3}$ y se simuló el sistema durante 200 segundos.

En la Figura 6.3 se muestra la representación en el entorno PowerDEVS y en la Figura 6.4 vemos la salida del sistema.

Modelo 6.2: lcline.qsm

```
1  Model
2  {
3     Name=lc_line;
4     Description=LC Line.;
5     Dimension=#N;
6     Solver=QSS3;
7     InitialTime=0;
8     FinalTime=200;
9     InitXCode=1;
10    SameStValue=1;
11    StateVarsInitialValue={0};
12    InitDqminCode=1;
13    SameDqMinValue=1;
14    DqMinValues={0};
15    InitDqrelCode=1;
16    SameDqRelValue=1;
17    DqRelValues={0};
18    DiscreteVars=1;
19    InitDCode=0;
20    SameDscValue=1;
21    DiscreteVarsInitialValue={1};
22    CommInterval=StepOutput;
23    SameOutputRate=0;
24    SearchMethod=BinarySearch;
25    SaveValues=Memory;
26    OutVars=2;
27    OutVarsNames={#N-2,#N-1};
28    ModelDef=
29    Constants
30    {
```

```
31      #N = 160;
32    }
33    InitialValues
34    {
35      Case(0 <= i < #N)
36      {
37        x[i] = 0;
38        dqmin[i ] = 1e-6;
39        dqrel[i] = 1e-3;
40      }
41    }
42    Equations
43    {
44      der(x[0]) = d[0] - x[1];
45      der(x[#N-1]) = x[#N-2];
46      Case(1 <= i <= #N-2)
47      {
48        der(x[i]) = x[i-1] - x[i+1];
49      }
50    }
51    Event
52    {
53      Fcross
54      {
55        t - 1;
56      }
57      Handler
58      {
59        d[0] = 0;
60      }
61    }
62    End
63  }
```

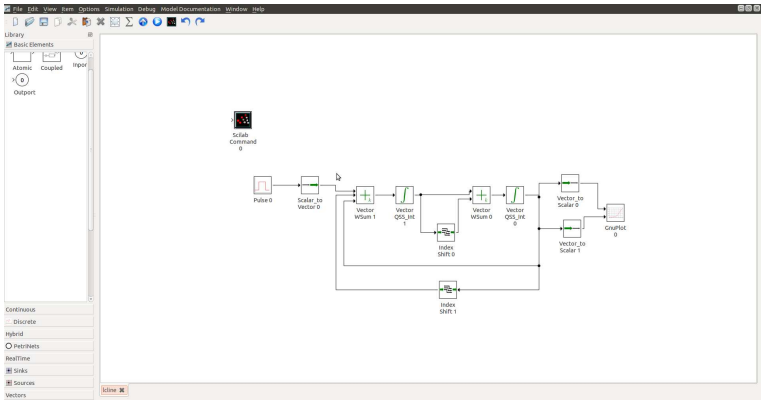

Figura 6.3: Modelo PowerDEVS para el Ejemplo 6.2

Tabla 6.2: Tiempo de simulación

	Tiempo (milisegundos)
Motor Autónomo QSS	560 ms
PowerDEVS	3766 ms

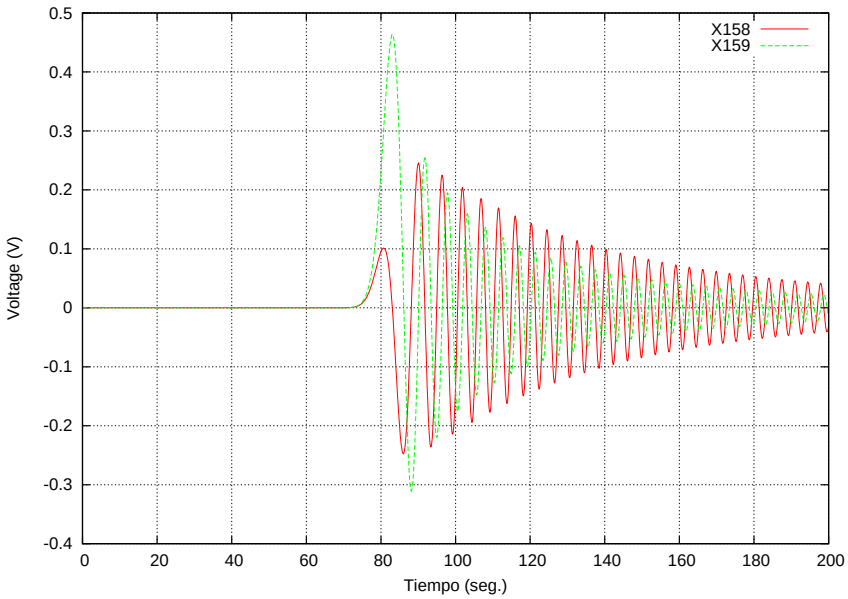

Figura 6.4: Salida del sistema para el Ejemplo 6.2

En la Tabla 6.2 se ve la diferencia en cuanto a tiempos de simulación, para este ejemplo, debido a la simplicidad en la definición de las ecuaciones, el motor autónomo es aproximadamente 7 veces más rápido que PowerDEVS.

6.3. Aires acondicionados

En este ejemplo consideramos un conjunto de aires acondicionados utilizados para controlar la temperatura de diferentes habitaciones. La temperatura de la habitación i $\theta_i(t)$ está dada por:

$$\frac{d\theta_i(t)}{dt} = -\frac{1}{C_i \cdot R_i}[\theta_i(t) - \theta_a + R_i \cdot P_i \cdot m_i(t) + w_i(t)], \tag{6.4}$$

donde R_i y C_i son parámetros que representan la resistencia térmica y la capacidad de la habitación i, respectivamente. P_i es la potencia del aire acondicionado i cuando se encuentra encendido. θ_a es la temperatura ambiente y $w_i(t)$ es el ruido que representa la perturbación térmica.

La variable $m_i(t)$ representa el estado del aire acondicionado i, su valor es 1

cuando el aire acondicionado está encendido o 0 en caso contrario.

$$m_i(t^+) = \begin{cases} 0 & \text{if } \theta_i(t) \leq \theta_r(t) - 0{,}5 \text{ and } m_i(t) = 1 \\ 1 & \text{if } \theta_i(t) \geq \theta_r(t) + 0{,}5 \text{ and } m_i(t) = 0 \\ m_i(t) & \text{en otro caso} \end{cases} \tag{6.5}$$

donde $\theta_r(t)$ es la temperatura de referencia.

El consumo del conjunto de aires acondicionados se calcula de la siguiente manera

$$P(t) = \sum_{i=1}^{N} m_i(t) \cdot P_i$$

y es regulado por un sistema de control global. La temperatura de referencia se controla con las ecuaciones

$$\theta_r(t) = K_P \cdot [P_r(t) - P(t)] + K_I \cdot \int_{\tau=0}^{t} [P_r(\tau) - P(\tau)] \mathrm{d}\tau$$

En el Modelo 6.3, vemos la representación del sistema en el entorno de simulación QSS. En este modelo, se definen las constantes $\#D$ y $\#N$ que indican la cantidad de variables discretas y de estado respectivamente. Para simular el sistema se consideraron 2001 variables de estado y 2004 valiables discretas, lo que implica 2003 eventos definidos, utilizando el método QSS3 con precisión $\Delta Q_{min} = 10^{-6}$ y $\Delta Q_{rel} = 10^{-3}$ y se simuló el sistema durante 3000 segundos.

En la Figura 6.5 se muestra la representación en el entorno PowerDEVS y en la Figura 6.6 vemos la salida del sistema.

Modelo 6.3: airs.qsm

```
1  Model
2  {
3     Name=airs;
4     Description=Description;
5     Dimension=#N;
6     Solver=QSS3;
7     InitialTime=0;
8     FinalTime=3000;
9     InitXCode=1;
10    SameStValue=1;
11    StateVarsInitialValue={TH0[i]};
12    InitDqminCode=1;
13    SameDqMinValue=1;
14    DqMinValues={1e-6};
15    InitDqrelCode=1;
16    SameDqRelValue=1;
17    DqRelValues={1e-3};
18    DiscreteVars=#D;
19    InitDCode=1;
20    SameDscValue=1;
21    DiscreteVarsInitialValue={1};
22    CommInterval=StepOutput;
23    SameOutputRate=0;
24    SearchMethod=BinarySearch;
25    SaveValues=Memory;
26    OutVars=1;
27    OutVarsNames={#N-1};
28    ModelDef=
29    Constants
30    {
31       #N = 2001;
32       #D = 2004;
33    }
34    InitialValues
35    {
36       d[#D-1] = 0;
37       d[#D-2] = 0;
38       d[#D-3] = 20;
39       d[#D-4] = 0;
40       Case(0<=i<#N-1)
41       {
42          x[i] = TH0[i];
43          if(x[i] > d[#D-3]-0.5)
44          {
45             d[i] = 1;
```

```
46            d[#D-4] = d[#D-4] + POT[i];
47          }
48          else
49          {
50            d[i] = 0;
51          }
52          dqmin[i] = 1e-6;
53          dqrel[i] = 1e-3;
54        }
55        dqmin[#N-1] = 1e-6;
56        dqrel[#N-1] = 1e-3;
57        x[#N-1] = d[#D-4];
58      }
59      Parameters
60      {
61        CAP[#N-1];
62        TH0[#N-1];
63        RES[#N-1];
64        POT[#N-1];
65        THA = 32;
66        Case(0<= i <#N-1)
67        {
68          TH0[i] = 4.0*rand()/RAND_MAX + 18;
69          CAP[i] = 100.0*rand()/RAND_MAX + 550;
70          RES[i] = 0.4*rand()/RAND_MAX + 1.8;
71          POT[i] = 2.0*rand()/RAND_MAX + 13;
72        }
73      }
74      Equations
75      {
76        Case(0<= i <#N-1)
77        {
78          der(x[i]) = (THA/RES[i]-POT[i]*d[i]-x[i]/RES[i]+d[#D-1]/RES[i])/CAP[i];
79        }
80        der(x[#N-1]) = d[#N-1] - x[#N-1];
81      }
82      Event
83      {
84        Fcross
85        {
86          t - 1000;
87        }
88        Handler
89        {
90          d[#D-3] = 20.5;
91        }
92      }
93      Event
94      {
95        Fcross
96        {
97          t - 2000;
98        }
99        Handler
100       {
101         d[#D-3] = 20;
102       }
103     }
104     Event
105     {
106       Case(0<=i<#N-1)
107       {
108         Fcross
109         {
110           x[i] - d[#D-3] + d[i] - 0.5;
111         }
112         Handler
113         {
114           d[i] = 1 - d[i];
115           d[#N-1] = d[#N-1] + (2 * d[i] -1) * POT[i];
116         }
117       }
118     }
119     Event
120     {
121       Fcross
122       {
123         t - d[#D-2];
124       }
125       Handler
126       {
127         d[#D-2] = d[#D-2]+1;
128         d[#D-1] = 2.0*(rand() / RAND_MAX) -1;
129       }
130     }
131     End
132   }
```

Tabla 6.3: Tiempo de simulación

	Tiempo (milisegundos)
Motor Autónomo QSS	390 ms
PowerDEVS	53460 ms

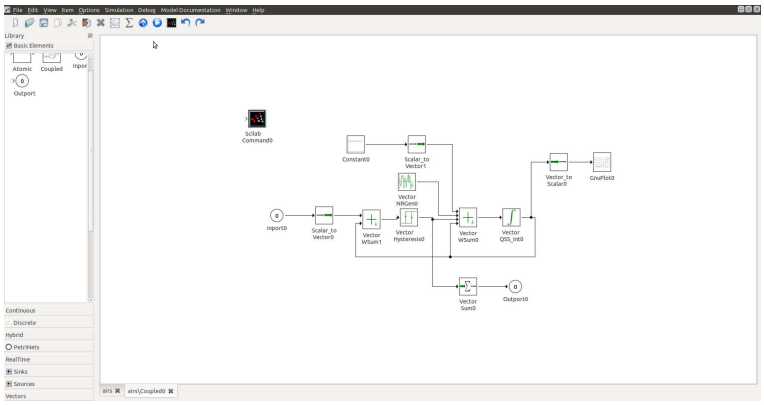

Figura 6.5: Modelo PowerDEVS para el Ejemplo 6.3

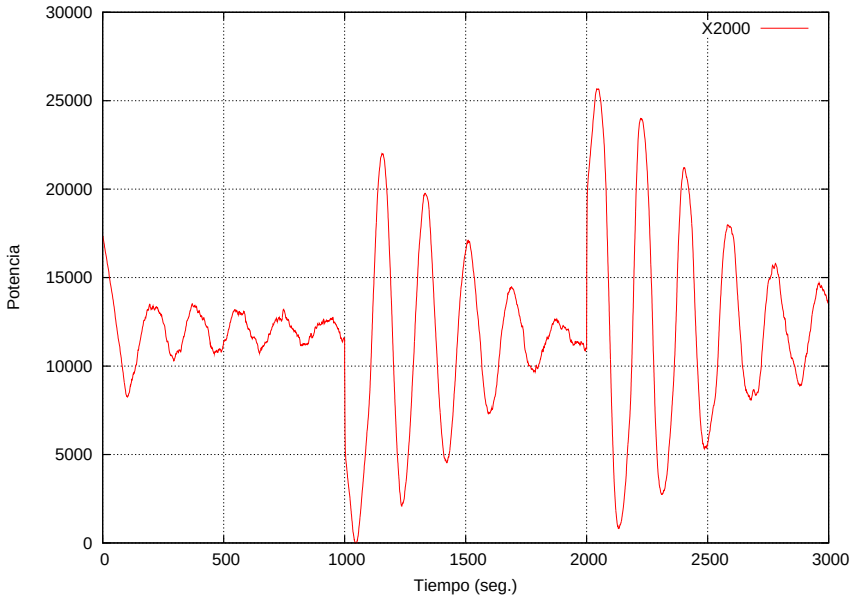

Figura 6.6: Salida del sistema para el Ejemplo 6.3

En la Tabla 6.3 se muestra la comparación de los tiempos de simulación en ambos entornos, dada la complejidad de este sistema, se logró una mejora significativa en la eficiencia, de dos órdenes de magnitud.

6.4. Inversores digitales

Un inversor lógico realiza un operación lógica sobre una señal. Cuando la señal está en un nivel alto, la señal de salida es baja y viceversa.

Una cadena de inversores, es la concatenación de una serie de inversores, donde la salida de cada inversor es la entrada del inversor siguiente en la serie.

El siguiente modelo representa una cadena de m inversores digitales:

$$\begin{cases} \dot{\omega}_1(t) & = U_{op} - \omega_1(t) - \Upsilon g\left(u_{in}(t), \omega_1(t)\right) \\ \dot{\omega}_j(t) & = U_{op} - \omega_j(t) - \Upsilon g\left(\omega_{j-1}(t), \omega_j(t)\right) \quad j = 2, 3, .., m \end{cases} \tag{6.6}$$

donde

$$g(u, v) = \left(\text{máx}(u - U_{thres}, 0)\right)^2 - \left(\text{máx}\left(u - v - U_{thres}, 0\right)\right)^2 \tag{6.7}$$

Los valores de los parámetros utilizados en la simulación son: $\Upsilon = 100$, $U_{thres} = 1$ y $U_{op} = 5$.

Con condiciones iniciales $\omega_j = 6{,}247 \cdot 10^{-3}$ para los valores impares de j y $\omega_j = 5$ para los valores pares de j.

En el Modelo 6.4, vemos la representación del sistema en el entorno de simulación QSS. En este modelo, se define la constante $\#N$ que indica la cantidad de variables de estado y la cantidad de variables discretas es $(2 * \#N) + 2$. Para simular el sistema se consideraron 100 variables de estado, 202 variables discretas y 101 eventos definidos, utilizando el método LIQSS2, dado que el sistema es stiff, con precisión $\Delta Q_{min} = 10^{-3}$ y $\Delta Q_{rel} = 10^{-2}$, para $j = 0$, $\Delta Q_{min} = 10^{-3}$ y $\Delta Q_{rel} = 10^{-3}$, para $j = 2, \ldots, m$ y se simuló el sistema durante 100 segundos.

En la Figura 6.7 se muestra la representación en el entorno PowerDEVS y en la Figura 6.8 vemos la salida del sistema.

Modelo 6.4: inverters.qsm

```
Model
{
   Name=inverters;
   Description=Description;
   Dimension=#N+1;
   Solver=LIQSS3;
   InitialTime=0;
   FinalTime=100;
   InitXCode=1;
   SameStValue=1;
   StateVarsInitialValue={0};
   InitDqminCode=1;
   SameDqMinValue=1;
   DqMinValues={0};
   InitDqrelCode=1;
   SameDqRelValue=1;
   DqRelValues={0};
   DiscreteVars=(2*#N)+2;
   InitDCode=1;
   SameDscValue=1;
   DiscreteVarsInitialValue={0};
   CommInterval=StepOutput;
   SameOutputRate=0;
   SearchMethod=BinarySearch;
   SaveValues=Memory;
   OutVars=2;
   OutVarsNames={1,100};
   ModelDef=
   Constants
   {
   #N=100;
   }
   InitialValues
   {
      x[0]=0;
      dqmin[0]=1e-4;
      dqrel[0]=1e-2;
      d[0]=0;
      d[1]=5;
      Case (0<=i<#N/2)
```

```
41      {
42         x[2*i+1]=5;
43         d[4*i+2]=0;
44         d[4*i+3]=0;
45         x[2*i+2]=6.247e-3;
46         d[4*i+4]=1;
47         d[4*i+5]=1;
48         dqmin[2*i+1]=1e-3;
49         dqrel[2*i+1]=1e-3;
50         dqmin[2*i+2]=1e-3;
51         dqrel[2*i+2]=1e-3;
52      }
53   }
54   Parameters
55   {
56      UOP=5;
57      G=100;
58      UTH=1;
59   }
60   Equations
61   {
62      der(x[0])=d[0];
63      Case(1<=i<=#N)
64      {
65         der(x[i])=UOP-x[i]-G*(d[i*2]*pow(x[i-1]-UTH,2)-d[(i*2)+1]*pow(x[i-1]-x[i]-UTH
                ,2));
66      }
67   }
68   Event
69   {
70      Case(1<=i<=#N)
71      {
72         Fcross
73         {
74            x[i-1]-UTH;
75         }
76         Handler
77         {
78            d[i*2]=1-d[i*2];
79         }
80      }
81   }
82   Event
83   {
84      Case(1<=i<=#N)
85      {
86         Fcross
87         {
88            x[i-1]-x[i]-UTH;
89         }
90         Handler
91         {
92            d[i*2+1]=1-d[i*2+1];
93         }
94      }
95   }
96   Event
97   {
98      Fcross
99      {
100        t-d[1];
101     }
102     Handler
103     {
104        if (d[1]==5)
105        {
106           d[0]=1;
107           d[1]=10;
108        }
109        else if(d[1]==10)
110        {
111           d[0]=0;
112           d[1]=15;
113        }
114        else if(d[1]==15)
115        {
116           d[0]=-2.5;
117           d[1]=17;
118        }
119        else
120        {
121           d[0]=0;
122           d[1]=1e10;
123        }
124     }
125  }
126  End
127 }
```

Tabla 6.4: Tiempo de simulación

	Tiempo (milisegundos)
Motor Autónomo QSS	20 ms
PowerDEVS	2670 ms

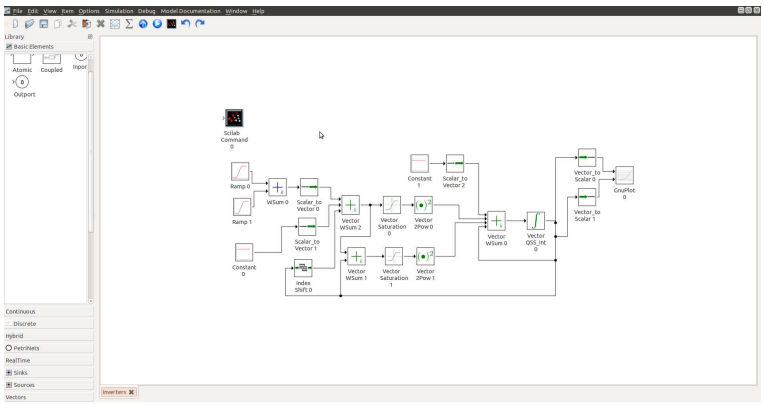

Figura 6.7: Modelo PowerDEVS para el Ejemplo 6.4

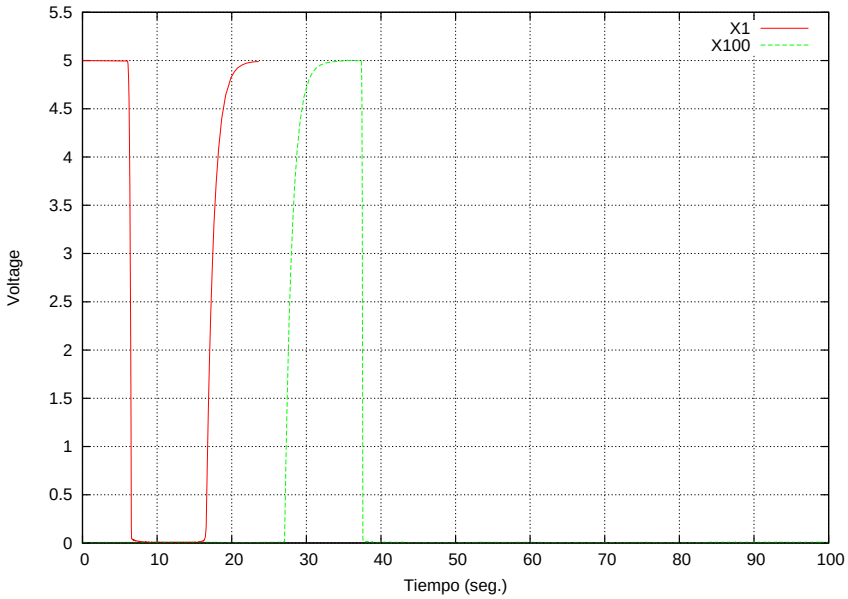

Figura 6.8: Salida del sistema para el Ejemplo 6.4

En la Tabla 6.4 se muestra la comparación de los tiempos de simulación en ambos entornos, en este caso, al igual que en el ejemplo anterior, se logró una mejora significativa en la eficiencia, de dos órdenes de magnitud utilizando el motor autónomo.

Capítulo 7

Conclusiones y Trabajo futuro

En este trabajo, hemos desarrollado un entorno de simulación completo para métodos de integración por cuantificación de estado QSS.

Formulamos un motor de simulación autónomo, marcando las diferencias existentes tanto con los motores de simulación para métodos tradicionales, así cómo también con las implementaciones de los métodos de QSS basadas en el formalismo DEVS. Vimos cómo definir modelos dentro de este entorno, desarrollando un módulo capaz de obtener la información estructural necesaria para la simulación de modelos en el motor presentado. También se implementó una interfaz gráfica de usuario, que permite definir modelos de manera simple, dando la posibilidad de interactuar con el entorno de simulación completo, esto incluye, la compilación y simulación de los modelos, ver los datos de salida de simulación y las gráficas de los mismos.

Por último, comparamos su rendimiento con PowerDEVS, que es el entorno de simulación para métodos QSS más eficiente en la actualidad. Logrando mejoras significativas en cuanto a tiempos de simulación en los ejemplos presentados.

Enumeraremos algunas posibles modificaciones y trabajos futuros a considerar:

1. Extender la definción de los modelos, agregando la capacidad de declarar *variables algebraicas*, esta modificación hace posible mejorar la eficiencia de la simulación, ya que permite abstraer definiciones comunes a más de una ecuación de estado, en una variable compartida por ambas ecuaciones, evitando que tengan que ser reevaluadas.

2. Agregar un módulo al entorno de simulación, que permita utilizar derivación simbólica para obtener derivadas de orden superior, a partir de la definición de las ecuaciones de estado.

3. Agregar la posibilidad de resetar variables de estado en los handlers de los eventos definidos en un sistema.

4. Modificar el generador de modelos, mejorando la eficiencia del código generado a partir de los modelos.

5. Implementar una interfaz capaz de traducir modelos definidos en PowerDEVS a modelos adecuados para ser simulados por el motor autónomo para métodos QSS.

6. Implementar una interfaz capaz de traducir modelos definidos en OpenModelica a modelos adecuados para ser simulados por el motor autónomo para métodos QSS.

Bibliografía

[1] T. Beltrame. Design and Development of a Dymola/Modelica Library for Discrete Event-oriented Systems Using DEVS Methodology. Master's thesis, ETH Zurich, Zurich, Switzerland, 2006.

[2] F. Bergero and E. Kofman. PowerDEVS. A Tool for Hybrid System Modeling and Real Time Simulation. *Simulation: Transactions of the Society for Modeling and Simulation International*, 87(1–2):113–132, 2011.

[3] F.E. Cellier and E. Kofman. *Continuous System Simulation*. Springer, New York, 2006.

[4] M. D'Abreu and G. Wainer. M/CD++: Modeling Continuous Systems Using Modelica and DEVS. In *Proc. MASCOTS 2005*, pages 229–238, Atlanta, GA, 2005.

[5] F. Esquembre. Easy Java Simulations: A software tool to create scientific simulations in Java. *Computer Physics Communications*, 156(1):199–204, 2004.

[6] E. Kofman. A second order approximation for devs simulation of continuous systems. *Simulation*, 78(2):76–89, 2002.

[7] E. Kofman. Discrete Event Simulation of Hybrid Systems. *SIAM Journal on Scientific Computing*, 25(5):1771–1797, 2004.

[8] E Kofman. A third order discrete event method for continuous system simulation. *Latin American Applied Research*, 2006.

[9] E. Kofman. Relative error control in quantization based integration. *Proceedings of RPIC'07*, 2007.

[10] E. Kofman. Relative Error Control in Quantization Based Integration. *Latin American Applied Research*, 39(3):231–238, 2009.

[11] E. Kofman and S. Junco. Quantized State Systems. A DEVS Approach for Continuous System Simulation. *Transactions of SCS*, 18(3):123–132, 2001.

[12] G. Migoni. *Simulación por Cuantifiación de Sistemas Stiff*. PhD thesis, Universidad Nacional de Rosario, Rosario, Argentina, 2010.

[13] G. Migoni, E. Kofman, and F. Cellier. Quantization-Based New Integration Methods for Stiff ODEs. *Simulation: Transactions of the Society for Modeling and Simulation International*, 2011. in Press.

[14] J. Nutaro and B. Zeigler. On the Stability and Performance of Discrete Event Methods for Simulating Continuous Systems. *J. Computational Physics*, 227(1):797–819, 2007.

[15] G. Quesnel, R. Duboz, E. Ramat, and M. Traoré. VLE: a Multi-modeling and Simulation Environment. In *Proc. 2007 Summer Computer Simulation Conference*, pages 367–374, San Diego, California, 2007.

Apéndice A

Código C de los solvers QSS implementados

A.1. CQSS

```
1  #include <math.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include "cqss1.h"
5  #include "utility.h"
6
7  double *qsup;
8  double *qinf;
9  double *lt;
10 double *simt;
11 int slope_change = 0;
12
13 void initSolver_CQSS1(SimulationData *simData, SimulationTime *simTime)
14 {
15  int i;
16  qsup = (double *)calloc(simData->dim,sizeof(double));
17  qinf = (double *)calloc(simData->dim,sizeof(double));
18  lt = (double *)calloc(simData->dim,sizeof(double));
19  for(i = 0; i < simData->dim; i++)
20  {
21   simData->X[0][i] = simData->XInitValues[i];
22   #ifdef HYBRYD
23   simData->deltaValues[i] = simData->XInitValues[i];
24   #endif
25   qsup[i] = simData->Q[0][i] + simData->dQMin[i];
26   qinf[i] = simData->Q[0][i] - simData->dQMin[i];
27   lt[i] = simData->it;
28  }
29  for(i = 0; i < simData->nDVars; i++)
30  {
31   simData->D[i] = simData->DInitValues[i];
32  }
33  simt = &simTime->time;
34 }
35
36 void recomputeNextTimes_CQSS1(int vars, int *inf, double t,double *nTime, double **x
       , double *lqu,double **q)
37 {
38  int i,j;
39  for(i = 0; i < vars; i++)
40  {
41   j = inf[i];
42   recomputeNextTime_CQSS1(j,t,nTime,x,lqu,q);
43  }
44 }
45
46 void recomputeNextTime_CQSS1(int var, double t,double *nTime, double **x, double *
       lqu,double **q)
47 {
48  if(lt[var] == t)
49  {
50   if((x[1][var] * (q[0][var] - x[0][var])) < 0 && (fabs(x[1][var]) > EPSILON))
51   {
52    slope_change = 1;
53    qinf[var] += lqu[var];
54    qsup[var] -= lqu[var];
55   }
56  }
57  if(x[1][var] > 0)
```

```
58  {
59   nTime[var] = t + fabs((qsup[var] - x[0][var]) / x[1][var]);
60  }
61  else if(x[1][var] < 0)
62  {
63   nTime[var] = t + fabs((qinf[var] - x[0][var]) / x[1][var]);
64  }
65  else
66  {
67   nTime[var] =  INF;
68  }
69 }
70
71 void nextTime_CQSS1(int var,double t,double *nTime, double **x, double *lqu)
72 {
73  if(x[1][var] > 0)
74  {
75   nTime[var] = t + fabs((qsup[var] - x[0][var]) / x[1][var]);
76  }
77  else if(x[1][var] < 0)
78  {
79   nTime[var] = t + fabs((qinf[var] - x[0][var]) / x[1][var]);
80  }
81  else
82  {
83   nTime[var] =  INF;
84  }
85 }
86
87 void updateQuantizedState_CQSS1(int var, double **q, double **x, double *lqu)
88 {
89  lt[var] = *simt;
90  if(x[1][var] > 0)
91  {
92   qsup[var] = qsup[var] + lqu[var];
93   q[0][var] = qsup[var] - lqu[var]/2;
94   qinf[var] = qsup[var] - 2*lqu[var];
95  }
96  else if(x[1][var] < 0)
97  {
98   qinf[var] = qinf[var] - lqu[var];
99   q[0][var] = qinf[var] + lqu[var]/2;
100  qsup[var] = qinf[var] + 2*lqu[var];
101 }
102 if(slope_change)
103 {
104  q[0][var] = x[0][var];
105  qsup[var] = q[0][var] + lqu[var];
106  qinf[var] = q[0][var] - lqu[var];
107  slope_change = 0;
108 }
109 }
```

A.2. LIQSS

```
1  #include <math.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include "liqss1.h"
5  #include "utility.h"
6
7  double *dq;
8  double *a;
9  double *old_dx;
10 double *q_aux;
11 double *u;
12 double *lt;
13 double *simt;
14
15 void initSolver_LIQSS1(SimulationData *simData, SimulationTime *simTime)
16 {
17  dq = (double *)calloc(simData->dim,sizeof(double));
18  old_dx = (double *)calloc(simData->dim,sizeof(double));
19  q_aux = (double *)calloc(simData->dim,sizeof(double));
20  a = (double *)calloc(simData->dim,sizeof(double));
21  u = (double *)calloc(simData->dim,sizeof(double));
22  lt = (double *)calloc(simData->dim,sizeof(double));
23  int i;
24  for(i = 0; i < simData->dim; i++)
25  {
26   simData->X[0][i] = simData->XInitValues[i];
27   #ifdef HYBRYD
28   simData->deltaValues[i] = simData->XInitValues[i];
29   #endif
30   simData->Q[0][i] = simData->X[0][i];
31   old_dx[i] = 0;
32   a[i] = 0;
33   u[i] = 0;
34   dq[i] = 0;
35   lt[i] = simData->it;
36   q_aux[i] = simData->X[0][i];
37  }
```

```
38   for(i = 0; i < simData->nDVars; i++)
39   {
40    simData->D[i] = simData->DInitValues[i];
41   }
42   simt = &simTime->time;
43  }
44
45  void recomputeNextTimes_LIQSS1(int vars, int *inf, double t,double *nTime, double **
         x, double *lqu,double **q)
46  {
47   int i,j;
48   for(i = 0; i < vars; i++)
49   {
50    j = inf[i];
51    recomputeNextTime_LIQSS1(j,t,nTime,x,lqu,q);
52   }
53  }
54
55  void recomputeNextTime_LIQSS1(int var, double t,double *nTime, double **x, double *
         lqu,double **q)
56  {
57   if(lt[var] == t)
58   {
59    double diffQ;
60    diffQ = q[0][var] - q_aux[var];
61    if(diffQ)
62    {
63     a[var] = (x[1][var] - old_dx[var])/diffQ;
64    }
65    else
66    {
67     a[var] = 0;
68    }
69   }
70   u[var] = x[1][var] - q[0][var] * a[var];
71   if(x[1][var] > 0)
72   {
73    nTime[var] = t + fabs(( lqu[var] + q[0][var]-dq[var] - x[0][var])/x[1][var]);
74   }
75   else if(x[1][var] < 0)
76   {
77       nTime[var] = t + fabs((q[0][var]-dq[var] - lqu[var] - x[0][var])/x[1][var]);
78   }
79   else
80   {
81    nTime[var] =  INF;
82   }
83  }
84
85  void nextTime_LIQSS1(int var,double t,double *nTime, double **x, double *lqu)
86  {
87   if(x[1][var] == 0)
88   {
89    nTime[var]  = INF;
90   }
91   else
92   {
93    nTime[var]  = t + fabs(lqu[var]/x[1][var]);
94   }
95  }
96
97  void updateQuantizedState_LIQSS1(int var, double **q, double **x, double *lqu)
98  {
99   double dx;
100  q_aux[var] = q[0][var];
101  old_dx[var] = x[1][var];
102  lt[var] = *simt;
103  q[0][var] = x[0][var];
104  if(x[1][var] > 0)
105  {
106   dx = u[var] + (q[0][var]+lqu[var])*a[var];
107   if(dx > 0)
108   {
109    dq[var] = lqu[var];
110   }
111   else
112   {
113    dq[var] = -u[var]/a[var]-q[0][var];
114   }
115  }
116  else
117  {
118   dx = u[var] +(q[0][var]-lqu[var])*a[var];
119   if(dx < 0)
120   {
121    dq[var] = -lqu[var];
122   }
123   else
124   {
125    dq[var] = -u[var]/a[var]-q[0][var];
126   }
127  }
128  q[0][var] += dq[var];
129 }
```

A.3. QSS2

```
#include <math.h>
#include "qss2.h"
#include "utility.h"

void initSolver_QSS2(SimulationData *simData, SimulationTime *simTime)
{
 int i;
 for(i = 0; i < simData->dim; i++)
 {
  simData->X[0][i] = simData->XInitValues[i];
  simData->X[2][i] = 0;
  simData->Q[0][i] = simData->X[0][i];
  simData->Q[1][i] = 0;
  simData->deltaValues[i] = simData->XInitValues[i];
 }
 for(i = 0; i < simData->nDVars; i++)
 {
  simData->D[i] = simData->DInitValues[i];
 }
}

void recomputeNextTimes_QSS2(int vars, int *inf, double t,double *nTime, double **x,
      double *lqu,double **q)
{
 int i,j;
 for(i = 0; i < vars; i++)
 {
  j = inf[i];
  recomputeNextTime_QSS2(j,t,nTime,x,lqu,q);
 }
}

void recomputeNextTime_QSS2(int var, double t,double *nTime, double **x, double *lqu
     ,double **q)
{
 if(fabs(q[0][var] - x[0][var]) >= lqu[var]*0.999999999)
 {
  nTime[var] = t;
 }
 else
 {
  double coeff[3];
  coeff[0] = q[0][var] + lqu[var] - x[0][var];
  coeff[1] = q[1][var] - x[1][var];
  coeff[2] = -x[2][var];
  nTime[var] = t + minPosRoot(coeff,2);
  coeff[0] = q[0][var] - lqu[var] - x[0][var];
  double timeaux = t + minPosRoot(coeff,2);
  if(timeaux < nTime[var])
  {
   nTime[var] = timeaux;
  }
 }
}

void nextTime_QSS2(int var,double t,double *nTime, double **x, double *lqu)
{
 if(x[2][var])
 {
  nTime[var] = t + sqrt(lqu[var]/fabs(x[2][var]));
 }
 else
 {
  nTime[var] = INF;
 }
}

void updateQuantizedState_QSS2(int i, double **q, double **x, double *lqu)
{
 q[0][i] = x[0][i];
 q[1][i] = x[1][i];
}
```

A.4. LIQSS2

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include "liqss2.h"
#include "utility.h"

double *dq;
double *a;
double *old_dx;
double *q_aux;
int *flag2;
int *flag3;
```

```
13 int *flag4;
14 double finTime;
15 static double *u0;
16 static double *u1;
17 static double *lt;
18 static double *ltq;
19 static double *lquOld;
20 SimulationTime *lSimTime;
21
22 void initSolver_LIQSS2(SimulationData *simData, SimulationTime *simTime)
23 {
24  dq = (double *)calloc(simData->dim,sizeof(double));
25  old_dx = (double *)calloc(simData->dim,sizeof(double));
26  q_aux = (double *)calloc(simData->dim,sizeof(double));
27  a = (double *)calloc(simData->dim,sizeof(double));
28  u0 = (double *)calloc(simData->dim,sizeof(double));
29  u1 = (double *)calloc(simData->dim,sizeof(double));
30  lt = (double *)calloc(simData->dim,sizeof(double));
31  ltq = (double *)calloc(simData->dim,sizeof(double));
32  lquOld = (double *)calloc(simData->dim,sizeof(double));
33  flag2 = (int *)calloc(simData->dim,sizeof(int));
34  flag3 = (int *)calloc(simData->dim,sizeof(int));
35  flag4 = (int *)calloc(simData->dim,sizeof(int));
36  finTime = simData->ft;
37  int i;
38  for(i = 0; i < simData->dim; i++)
39  {
40   simData->X[0][i] = simData->XInitValues[i];
41   simData->X[2][i] = 0;
42   simData->Q[0][i] = simData->X[0][i];
43   simData->Q[1][i] = 0;
44   simData->deltaValues[i] = simData->XInitValues[i];
45   old_dx[i] = 0;
46   a[i] = 0;
47   u0[i] = 0;
48   u1[i] = 0;
49   dq[i] = 0;
50   lt[i] = simData->it;
51   ltq[i] = simData->it;
52   q_aux[i] = simData->X[0][i];
53   lquOld[i]= simData->lqu[i];
54   flag2[i] = 0; //this flag becomes true when a future situation ddx=0 is detected.
55   flag3[i] = 0; //this flag becomes true after trying to provoke ddx=0.
56   flag4[i] = 0; //this flag becomes true after detecting a sign change in ddx.
57  }
58  for(i = 0; i < simData->nDVars; i++)
59     {
60   simData->D[i] = simData->DInitValues[i];
61  }
62  lSimTime=simTime;
63 }
64
65 void recomputeNextTime_LIQSS2(int var, double t,double *nTime, double **x, double *
       lqu,double **q)
66 {
67   if(ltq[var] == t)
68   {
69   double diffQ;
70   diffQ = q[0][var] - q_aux[var];
71   if(fabs(diffQ)>lqu[var]*1e-6)
72   {
73    a[var] = (x[1][var] - old_dx[var])/diffQ;
74    if (a[var]>0)
75    {
76     a[var]=0;
77    }
78   }
79  }
80  else
81  {
82   flag3[var]=0;
83  }
84  u0[var] = x[1][var] - q[0][var] * a[var];
85  u1[var] = 2*x[2][var] - q[1][var] * a[var];
86  lt[var]=t;
87  if(flag4[var])
88  {
89   nTime[var] = t;
90  }
91  else
92  {
93   double diffxq[3];
94   diffxq[0] = q[0][var] - dq[var] + lqu[var] - x[0][var];
95   diffxq[1] = q[1][var] - x[1][var];
96   diffxq[2] = -x[2][var];
97   nTime[var] = t + minPosRoot(diffxq,2);
98   diffxq[0] = q[0][var] - dq[var] - lqu[var] - x[0][var];
99   double timeaux = t + minPosRoot(diffxq,2);
100  if(timeaux < nTime[var])
101  {
102   nTime[var] = timeaux;
103  }
104  else
105  {
106   diffxq[0] = q[0][var] - dq[var] + lqu[var] - x[0][var];
107  }
108  if (a[var]!=0&&(fabs(x[2][var])>1e-10)&&!flag3[var]&&!flag2[var])
109  {
110   double coeff[2];
```

```
111     coeff[0]=a[var]*a[var]*q[0][var]+a[var]*u0[var]+u1[var];
112     coeff[1]=a[var]*a[var]*q[1][var]+a[var]*u1[var];
113     timeaux=t + minPosRoot(coeff,1);
114     if (timeaux<nTime[var])
115     {
116      flag2[var]=1;
117      nTime[var] = timeaux;
118      lquOld[var]=lqu[var];
119     }
120    }
121    else
122    {
123     flag2[var]=0;
124    }
125    if (nTime[var]>finTime)
126    {
127     nTime[var]=finTime;
128    }
129    double err1=q[0][var]-x[0][var]+diffxq[1]*(nTime[var]-t)/2+diffxq[2]*pow((nTime[
           var]-t)/2,2);
130    if (fabs(err1)>3*fabs(lqu[var]))
131    {
132     nTime[var]=t+finTime*1e-14;
133    }
134   }
135  }
136
137  void recomputeNextTimes_LIQSS2(int vars, int *inf, double t,double *nTime, double **
         x, double *lqu,double **q)
138  {
139   int i,j;
140   for(i = 0; i < vars; i++)
141   {
142    j = inf[i];
143    recomputeNextTime_LIQSS2(j,t,nTime,x,lqu,q);
144   }
145  }
146
147  void nextTime_LIQSS2(int var,double t,double *nTime, double **x, double *lqu)
148  {
149   if(x[2][var] == 0)
150   {
151    nTime[var]  = INF;
152   }
153   else
154   {
155    nTime[var]  = t + sqrt(fabs(lqu[var]/x[2][var]));
156   }
157  }
158
159  void updateQuantizedState_LIQSS2(int var, double **q, double **x, double *lqu)
160  {
161   double dx,elapsed;
162   flag3[var]=0;
163   elapsed = lSimTime->time - lSimTime->tq[var];
164   q_aux[var] = q[0][var]+elapsed*q[1][var];
165   old_dx[var] = x[1][var];
166   elapsed = lSimTime->time - lt[var];
167   ltq[var]=lSimTime->time;
168   u0[var] = u0[var]+elapsed*u1[var];
169   if (flag2[var])
170   {
171    lqu[var]=lquOld[var];
172    flag2[var]=0;
173    q[0][var]=q_aux[var];
174   }
175   else
176   {
177    q[0][var]=x[0][var];
178   }
179   if (a[var]<-1e-30)
180   {
181    if(x[2][var] < 0)
182    {
183     dx = a[var]*a[var]*(q[0][var] + lqu[var])+a[var]*u0[var]+u1[var];
184     if(dx <= 0)
185     {
186      dq[var] = lqu[var];
187     }
188     else
189     {
190      dq[var] = (-u1[var]/a[var]/a[var])-(u0[var]/a[var])-q[0][var];
191      flag3[var]=1;
192      if (fabs(dq[var])>lqu[var]) dq[var]=lqu[var];
193     }
194    }
195    else
196    {
197     dx = a[var]*a[var]*(q[0][var] - lqu[var]) + a[var]*u0[var] + u1[var];
198     if(dx >= 0)
199     {
200      dq[var] = -lqu[var];
201     }
202     else
203     {
204      dq[var] = -u1[var]/a[var]/a[var]-u0[var]/a[var]-q[0][var];
205      flag3[var]=1;
206      if (fabs(dq[var])>lqu[var])
207      {
```

```
208       dq[var]=-lqu[var];
209      }
210     }
211    }
212    if (q[1][var]*x[1][var]<0&&!flag4[var]&&!flag2[var]&&!flag3[var])
213    {
214     if (q[1][var]<0)
215     {
216      dq[var]=q_aux[var]-q[0][var]-fabs(lquOld[var])*0.1;
217     }
218     else
219     {
220      dq[var]=q_aux[var]-q[0][var]+fabs(lquOld[var])*0.1;
221     }
222     flag4[var]=1;
223    }
224    else if (flag4[var])
225    {
226     flag4[var]=0;
227     if (fabs(-u1[var]/a[var]/a[var]-u0[var]/a[var]-q[0][var])<3*lqu[var])
228     {
229      dq[var] = -u1[var]/a[var]/a[var]-u0[var]/a[var]-q[0][var];
230      flag3[var]=1;
231     }
232    }
233   }
234   else
235   {
236    flag4[var]=0;
237    if (x[2][var]<0)
238    {
239     dq[var]=-lqu[var];
240    }
241    else
242    {
243     dq[var]=lqu[var];
244    }
245   }
246   q[0][var] = q[0][var]+dq[var];
247   if (flag3[var])
248   {
249    q[1][var] = a[var]*q[0][var]+u0[var];
250   }
251   else
252   {
253    q[1][var]=x[1][var];
254   }
255  }
```

A.5. QSS3

```
1  #include <math.h>
2  #include "qss3.h"
3  #include "utility.h"
4
5  void initSolver_QSS3(SimulationData *simData, SimulationTime *simTime)
6  {
7   int i;
8   for(i = 0; i < simData->dim; i++)
9   {
10   simData->X[0][i] = simData->XInitValues[i];
11   simData->X[2][i] = 0;
12   simData->X[3][i] = 0;
13   simData->Q[0][i] = simData->X[0][i];
14   simData->Q[1][i] = 0;
15   simData->Q[2][i] = 0;
16   simData->deltaValues[i] = simData->XInitValues[i];
17  }
18  for(i = 0; i < simData->nDVars; i++)
19  {
20   simData->D[i] = simData->DInitValues[i];
21  }
22 }
23
24 void recomputeNextTimes_QSS3(int vars, int *inf, double t,double *nTime, double **x,
       double *lqu,double **q)
25 {
26  int i,j;
27  for(i = 0; i < vars; i++)
28  {
29   j = inf[i];
30   recomputeNextTime_QSS3(j,t,nTime,x,lqu,q);
31  }
32 }
33
34 void recomputeNextTime_QSS3(int var, double t,double *nTime, double **x, double *lqu
      ,double **q)
35 {
36  if(fabs(q[0][var] - x[0][var]) >= lqu[var]*0.999999999)
37  {
38   nTime[var] = t;
39  }
```

```
40   else
41   {
42    double coeff[4];
43    coeff[0] = q[0][var] + lqu[var] - x[0][var];
44    coeff[1] = q[1][var] - x[1][var];
45    coeff[2] = q[2][var] -x[2][var];
46    coeff[3] = -x[3][var];
47    nTime[var] = t + minPosRoot(coeff,3);
48    coeff[0] = q[0][var] - lqu[var] - x[0][var];
49    double timeaux = t + minPosRoot(coeff,3);
50    if(timeaux < nTime[var])
51    {
52     nTime[var] = timeaux;
53    }
54   }
55  }
56
57  void nextTime_QSS3(int var,double t,double *nTime, double **x, double *lqu)
58  {
59   if(x[3][var])
60   {
61    nTime[var] = t + pow(lqu[var]/fabs(x[3][var]),1.0/3);
62   }
63   else
64   {
65    nTime[var] = INF;
66   }
67  }
68
69  void updateQuantizedState_QSS3(int i, double **q, double **x, double *lqu)
70  {
71   q[0][i] = x[0][i];
72   q[1][i] = x[1][i];
73   q[2][i] = x[2][i];
74  }
```

A.6. LIQSS3

```
1  #include <math.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include "liqss3.h"
5  #include "utility.h"
6
7  double *dq;
8  double *a;
9  double *old_dx;
10 double *q_aux;
11 int *flag2;
12 int *flag3;
13 int *flag4;
14 double finTime;
15 static double *u0;
16 static double *u1;
17 static double *u2;
18 static double *lt;
19 static double *ltq;
20 static double *lquOld;
21 SimulationTime *lSimTime;
22
23 void initSolver_LIQSS3(SimulationData *simData, SimulationTime *simTime)
24 {
25  dq = (double *)calloc(simData->dim,sizeof(double));
26  old_dx = (double *)calloc(simData->dim,sizeof(double));
27  q_aux = (double *)calloc(simData->dim,sizeof(double));
28  a = (double *)calloc(simData->dim,sizeof(double));
29  u0 = (double *)calloc(simData->dim,sizeof(double));
30  u1 = (double *)calloc(simData->dim,sizeof(double));
31  u2 = (double *)calloc(simData->dim,sizeof(double));
32  lt = (double *)calloc(simData->dim,sizeof(double));
33  ltq = (double *)calloc(simData->dim,sizeof(double));
34  lquOld = (double *)calloc(simData->dim,sizeof(double));
35  flag2 = (int *)calloc(simData->dim,sizeof(int));
36  flag3 = (int *)calloc(simData->dim,sizeof(int));
37  flag4 = (int *)calloc(simData->dim,sizeof(int));
38  finTime = simData->ft;
39  int i;
40  for(i = 0; i < simData->dim; i++)
41  {
42   simData->X[0][i] = simData->XInitValues[i];
43   simData->X[2][i] = 0;
44   simData->X[3][i] = 0;
45   simData->Q[0][i] = simData->X[0][i];
46   simData->Q[1][i] = 0;
47   simData->Q[2][i] = 0;
48   simData->deltaValues[i] = simData->XInitValues[i];
49   old_dx[i] = 0;
50   a[i] = 0;
51   u0[i] = 0;
52   u1[i] = 0;
53   u2[i] = 0;
54   dq[i] = 0;
```

```
55    lt[i] = simData->it;
56    ltq[i] = simData->it;
57    q_aux[i] = simData->X[0][i];
58    lquOld[i]= simData->lqu[i];
59    flag2[i] = 0; //this flag becomes true when a future situation dddx=0 is detected.
60    flag3[i] = 0; //this flag becomes true after trying to provoke dddx=0.
61    flag4[i] = 0; //this flag becomes true after detecting a sign change in ddx.
62   }
63   for(i = 0; i < simData->nDVars; i++)
64   {
65    simData->D[i] = simData->DInitValues[i];
66   }
67   lSimTime=simTime;
68  }
69
70  void recomputeNextTime_LIQSS3(int var, double t,double *nTime, double **x, double *
        lqu,double **q)
71  {
72   if(ltq[var] == t)
73   {
74    double diffQ;
75    diffQ = q[0][var] - q_aux[var];
76    if(fabs(diffQ)>lqu[var]*1e-6)
77    {
78     a[var] = (x[1][var] - old_dx[var])/diffQ;
79     if (a[var]>0)
80     {
81      a[var]=0;
82     }
83    }
84   }
85   else
86   {
87    flag3[var]=0;
88   }
89   u0[var] = x[1][var] - q[0][var] * a[var];
90   u1[var] = 2*x[2][var] - q[1][var] * a[var];
91   u2[var] = 3*x[3][var] - q[2][var] * a[var];
92   lt[var]=t;
93   if(flag4[var])
94   {
95    nTime[var] = t;
96   }
97   else
98   {
99    double diffxq[4];
100   diffxq[0] = q[0][var] - dq[var] + lqu[var] - x[0][var];
101   diffxq[1] = q[1][var] - x[1][var];
102   diffxq[2] = q[2][var] - x[2][var];
103   diffxq[3] = -x[3][var];
104   nTime[var] = t + minPosRoot(diffxq,3);
105   diffxq[0] = q[0][var] - dq[var] - lqu[var] - x[0][var];
106   double timeaux = t + minPosRoot(diffxq,3);
107   if(timeaux < nTime[var])
108   {
109    nTime[var] = timeaux;
110   }
111   else
112   {
113    diffxq[0] = q[0][var] - dq[var] + lqu[var] - x[0][var];
114   }
115   if (a[var]!=0&&(fabs(x[3][var])>1e-10)&&!flag3[var]&&!flag2[var])
116   {
117    double coeff[3];
118    coeff[0]=a[var]*a[var]*a[var]*q[0][var]+a[var]*a[var]*u0[var]+a[var]*u1[var]+2*u2
          [var];
119    coeff[1]=a[var]*a[var]*a[var]*q[1][var]+a[var]*a[var]*u1[var]+a[var]*2*u2[var];
120    coeff[2]=a[var]*a[var]*a[var]*q[2][var]+a[var]*a[var]*u2[var];
121    timeaux=t + minPosRoot(coeff,2);
122    if (timeaux<nTime[var])
123    {
124     flag2[var]=1;
125     nTime[var] = timeaux;
126     lquOld[var]=lqu[var];
127    }
128   }
129   else
130   {
131    flag2[var]=0;
132   }
133   if (nTime[var]>finTime)
134   {
135    nTime[var]=finTime;
136   }
137   double err1=q[0][var]-x[0][var]+diffxq[1]*(nTime[var]-t)/2+diffxq[2]*pow((nTime[
        var]-t)/2,2)+diffxq[3]*pow((nTime[var]-t)/2,3);
138   if (fabs(err1)>3*fabs(lqu[var]))
139   {
140    nTime[var]=t+finTime*1e-14;
141   }
142  }
143 }
144
145 void recomputeNextTimes_LIQSS3(int vars, int *inf, double t,double *nTime, double **
      x, double *lqu,double **q)
146 {
147  int i,j;
148  for(i = 0; i < vars; i++)
149  {
```

```
150     j = inf[i];
151     recomputeNextTime_LIQSS3(j,t,nTime,x,lqu,q);
152   }
153 }
154
155 void nextTime_LIQSS3(int var,double t,double *nTime, double **x, double *lqu)
156 {
157   if(x[3][var] == 0)
158   {
159    nTime[var]  = INF;
160   }
161   else
162   {
163    nTime[var]  = t + pow(fabs(lqu[var]/x[3][var]),1.0/3);
164   }
165 }
166
167 void updateQuantizedState_LIQSS3(int var, double **q, double **x, double *lqu)
168 {
169   double dx,elapsed;
170   flag3[var]=0;
171   elapsed = lSimTime->time - lSimTime->tq[var];
172   q_aux[var] = q[0][var] + elapsed * q[1][var] +elapsed * elapsed *q[2][var];
173   old_dx[var] = x[1][var];
174   ltq[var] = lSimTime->time;
175   elapsed = lSimTime->time - lt[var];
176   u0[var] = u0[var] + elapsed * u1[var] + elapsed * elapsed * u2[var];
177   u1[var] = u1[var]+2*elapsed * u2[var];
178   lt[var] = lSimTime->time;
179   if (flag2[var])
180   {
181    lqu[var]=lquOld[var];
182    flag2[var]=0;
183    q[0][var]=q_aux[var];
184   }
185   else
186   {
187    q[0][var]=x[0][var];
188   }
189   if (a[var]<-1e-30)
190   {
191    if(x[3][var] > 0)
192    {
193     dx = a[var]*a[var]*a[var]*(q[0][var] + lqu[var])+a[var]*a[var]*u0[var]+a[var]*u1[
           var]+2*u2[var];
194     if(dx >= 0)
195     {
196      dq[var] = lqu[var];
197     }
198     else
199     {
200      dq[var] =  -2*u2[var]/a[var]/a[var]/a[var]-u1[var]/a[var]/a[var]-u0[var]/a[var]-
             q[0][var];
201      flag3[var] = 1;
202      if (fabs(dq[var])>lqu[var])
203      {
204       dq[var]=lqu[var];
205      }
206     }
207    }
208    else
209    {
210     dx = a[var]*a[var]*a[var]*(q[0][var] - lqu[var])+a[var]*a[var]*u0[var]+a[var]*u1[
           var]+2*u2[var];
211     if(dx <= 0)
212     {
213      dq[var] = -lqu[var];
214     }
215     else
216     {
217      dq[var] =  -2*u2[var]/a[var]/a[var]/a[var]-u1[var]/a[var]/a[var]-u0[var]/a[var]-
             q[0][var];
218      flag3[var]=1;
219      if (fabs(dq[var])>lqu[var])
220      {
221       dq[var]=-lqu[var];
222      }
223     }
224       }
225    if (q[1][var]*x[1][var]<0&&!flag4[var]&&!flag2[var]&&!flag3[var])
226    {
227     if (q[1][var]<0)
228     {
229      dq[var]=q_aux[var]-q[0][var]-fabs(lquOld[var])*0.1;
230     }
231     else
232     {
233      dq[var]=q_aux[var]-q[0][var]+fabs(lquOld[var])*0.1;
234     }
235     flag4[var]=1;
236    }
237    else if (flag4[var])
238    {
239     flag4[var]=0;
240     if (fabs(-2*u2[var]/a[var]/a[var]/a[var]-u1[var]/a[var]/a[var]-u0[var]/a[var]-q
           [0][var])<3*lqu[var])
241     {
242      dq[var] =  -2*u2[var]/a[var]/a[var]/a[var]-u1[var]/a[var]/a[var]-u0[var]/a[var]-
             q[0][var];
```

```
243       flag3[var]=1;
244      }
245     }
246    }
247    else
248    {
249     flag4[var]=0;
250     if (x[3][var]>0)
251     {
252      dq[var]=-lqu[var];
253     }
254     else
255     {
256      dq[var]=lqu[var];
257     }
258    }
259    q[0][var] = q[0][var]+dq[var];
260    if (flag3[var])
261    {
262     q[1][var] = a[var]*q[0][var]+u0[var];
263     q[2][var] = a[var]*q[1][var]/2+u1[var]/2;
264    }
265    else
266    {
267     q[1][var]=x[1][var];
268     q[2][var]=x[2][var];
269    }
270   }
```

Apéndice B

Código C del entorno del motor simulación

```
1  void recomputeDerivative(SimulationModel *simModel, SimulationData *simData,
       SimulationTime *simTime, double elapsed,int stateVar)
2  {
3   int k,j;
4   double e;
5   for(k = 0; k < simData->nQDeps[stateVar]; k++)
6   {
7    j = simData->IQ[stateVar][k];
8    e = simTime->time - simTime->tq[j];
9    if(e > 0)
10   {
11    advanceTime(j,e,simData->Q,simData->order-1);
12   }
13   simTime->tq[j] = simTime->time;
14   simData->deltaValues[j] = evaluatePoly(j,delta,simData->Q,simData->order-1);
15  }
16  simModel->f(stateVar,simData->Q[0],simData->D,simTime->time,&simData->X[1][stateVar
       ]);
17  simModel->f(stateVar,simData->deltaValues,simData->D,simTime->time+delta,&simData->
       dxNew[stateVar]);
18
19  if(delta > 0)
20  {
21   simData->X[2][stateVar] = (simData->dxNew[stateVar] - simData->X[1][stateVar])/(
         delta*2);
22  }
23  for(k = 0; k < simData->nQDeps[stateVar]; k++)
24  {
25   j = simData->IQ[stateVar][k];
26   simData->deltaValues[j] = evaluatePoly(j,-delta,simData->Q,simData->order-1);
27  }
28  simModel->f(stateVar,simData->deltaValues,simData->D,simTime->time-delta,&simData->
       dxOld[stateVar]);
29
30  dxn_old = simData->X[3][stateVar];
31  if(delta > 0)
32  {
33   simData->X[3][stateVar] = (simData->dxNew[stateVar] - 2*simData->X[1][stateVar] +
       simData->dxOld[stateVar])/(delta*delta*6);
34  }
35 }
36
37 void nextEventTime(SimulationModel *simModel, SimulationData *simData,
       SimulationTime *simTime, double elapsed, int index)
38 {
39  int i,j,s;
40  double coeff[5], t = simTime->time;
41  double d_new,d_old;
42  int order = simData->order;
43  for(i = 0; i < simData->events[index].nIVars; i++)
44  {
45   j = simData->events[index].IVars[i];
46   simData->deltaValues[j] = evaluatePoly(j,t-simTime->tq[j],simData->Q,order-1);
47  }
48  simModel->events->zeroCrossing(index,simData->deltaValues,simData->D,t,&coeff[0],
       simData->X[1]);
49  s = _sign(coeff[0]);
50  if(simData->events[index].zc_sign != s)
51  {
52   simTime->dEvents[index] = t;
53  }
54  else
55  {
56   for(i = 0; i < simData->events[index].nIVars; i++)
```

```
57    {
58     j = simData->events[index].IVars[i];
59     simData->deltaValues[j] = evaluatePoly(j,t-simTime->tq[j]+delta_h,simData->Q,
           order-1);
60    }
61    simModel->events->zeroCrossing(index,simData->deltaValues,simData->D,t+delta_h,&
          d_new,simData->X[1]);
62    coeff[1] = (d_new - coeff[0])/(delta_h);
63    for(i = 0; i < simData->events[index].nIVars; i++)
64    {
65     j = simData->events[index].IVars[i];
66     simData->deltaValues[j] = evaluatePoly(j,t-simTime->tq[j]-delta_h,simData->Q,
           order-1);
67    }
68    simModel->events->zeroCrossing(index,simData->deltaValues,simData->D,t-delta_h,&
          d_old,simData->X[1]);
69    coeff[2] = (d_new - 2 * coeff[0] + d_old)/(delta_h*delta_h*2);
70    coeff[0] += simData->events[index].zc_sign * simData->events[index].zc_h;
71    simTime->dEvents[index] = t + minPosRoot(coeff,2);
72   }
73  }
74
75  void nextInputTime(SimulationModel *model, SimulationData *data, SimulationTime *
        time, double elapsed, int var, int index)
76  {
77   double df;
78   if(elapsed > 0)
79   {
80    df = 6 * (data->X[3][var] - dxn_old)/elapsed;
81   }
82   else
83   {
84    df = 6*data->lqu[var]*1e18;
85   }
86   if(df != 0)
87   {
88    time->iEvents[index] = time->time + pow(6*data->lqu[var]/fabs(df),1.0/3);
89   }
90   else
91   {
92    time->iEvents[index] = INF;
93   }
94  }
```