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

Tesis Doctoral

Simulación por Cuantificación de Sistemas Híbridos de Gran Escala.

Lic. Joaquín Francisco Fernández

Director: Dr. Ernesto Kofman

Tesis presentada en la Facultad de Ciencias Exactas, Ingeniería y Agrimensura, en cumplimiento parcial de los requisitos para optar al título de
Doctor en Informática

Marzo 2017

Certifico que el trabajo incluído en esta tesis es el resultado de tareas de investigación originales y que no ha sido presentado para optar a un título de posgrado en ninguna otra Universidad o Institución.

Joaquín Francisco Fernández

Resumen

En esta Tesis se presentan los fundamentos y la implementación de nuevas técnicas que permiten simular sistemas híbridos de gran escala de manera eficiente utilizando métodos de Quantized State System (QSS).

Los métodos QSS reemplazan la discretización temporal aplicada en los métodos clásicos de integración numérica por la cuantificación de las variables de estado. De esta manera se obtiene una aproximación por eventos discretos del sistema continuo original que presenta ciertas ventajas sobre los enfoques clásicos.

Debido a esta forma de discretización, la manera más sencilla de implementar estos métodos de aproximación numérica es mediante el uso de motores de simulación de eventos discretos. Sin embargo, el costo computacional adicional impuesto por el mecanismo de simulación de eventos discretos hace que estas implementaciones sean ineficientes. Los nuevos algoritmos de simulación para métodos QSS desarrollados fueron concebidos de manera tal que no es necesario utilizar un motor de simulación de eventos discretos y como consecuencia los algoritmos propuestos presentan una mejora notable en los tiempos de simulación con respecto a las implementaciones previas de este tipo de métodos.

Una característica adicional de los nuevos algoritmos de simulación es que permiten simular modelos complejos de gran escala que usualmente conllevan un costo computacional alto. Debido al desarrollo de procesadores multi–núcleo como así también de clusters de computadoras multi–nodo en los últimos años, la simulación en paralelo de sistemas de tiempo continuo representa la manera usual de reducir los costos de simulación. A partir de esta idea, se desarrollaron dos nuevas técnicas de paralelización específicamente diseñadas para los métodos de QSS.

Estás técnicas, basadas en la naturaleza asíncrona de los métodos QSS utilizan un mecanismo de sincronización no estricto entre los diferentes procesos lógicos que intervienen en la simulación.

Esta metodología de simulación en paralelo permite reducir de manera significativa los tiempos de simulación introduciendo un error numérico adicional que depende de ciertos parámetros dados. Un análisis teórico del error introducido nos permite demostrar que el mismo puede ser acotado bajo ciertos supuestos.

Finalmente, se desarrolló una herramienta que implementa los nuevos algoritmos propuestos y que permite analizar en profundidad su rendimiento en diferentes sistemas híbridos de gran escala. Adicionalmente, se desarrollaron herramientas que permiten la traducción automática de sistemas de ecuaciones diferenciales descriptos en diferentes lenguajes de modelado estandard (OpenModelica y SBML) posibilitando de esta manera la integración de las nuevas metodologías desarrolladas con diferentes entornos de modelado y simulación.

Abstract

This Thesis presents the implementation and fundamentals of new techniques that allow the simulation of large scale continuous and hybrid systems using Quantized State System (QSS) methods.

QSS methods replace the time discretization of classic numerical integration by the quantization of the state variables. These algorithms lead to discrete event approximations of the original continuous systems and show some advantages over classic numerical integration schemes.

For simplicity reasons, most implementations of QSS methods were confined to discrete event simulation engines. The problem is that they were not fully efficient as they wasted much of the computational load in the discrete event simulation mechanism. The new QSS algorithms developed overcome this problem and, as a consequence, the simulation times were noticeable improved compared to previous discrete event implementations.

A main feature of the new algorithms is that they allow the simulation of complex large scale hybrid systems that usually carry high computational costs. With the advent of multi–core processors and multi–node clusters of computers, the parallel simulation of continuous time systems became the usual way to reduce execution times of these simulations. Following this idea, two novel parallelization techniques for QSS methods were developed.

These techniques are based on the asynchronous nature of the QSS methods and they use a non–strict synchronization mechanism between the different logical processes involved in the simulation.

The novel methodologies are based on the use of non–strict synchronization between logical processes. The fact that the synchronization is not strict allows to achieve large speedups at the cost of introducing additional numerical errors that, under certain assumptions, are bounded depending on some given parameters. A

theoretical analysis of the aforementioned additional numerical error proves that it can be bounded under some given assumptions.

Finally, a software tool was developed that implements all the proposed algorithms in order to deeply evaluate the performance on different large scale hybrid systems. Additionaly, a set of software tools were developed that allows the automatic translation of systems of differential equations defined in standard modeling languages (OpenModelica and SBML) that integrates the new methodologies with different modeling and simulation tools.

Agradecimientos

Índice general

Capítulo 1

Introducción

Los sistemas continuos e híbridos a parámetros concentrados se representan generalmente mediante sistemas de ecuaciones diferenciales ordinarias. Por lo tanto, para simular estos sistemas, deben resolverse estos sistemas de ecuaciones lo que requiere normalmente el uso de métodos de integración numérica. Los métodos de integración numérica clásicos [12, 30, 63] están basados en la discretización de la variable independiente (que usualmente representa el tiempo).

Los métodos de integración numérica de QSS (Quantized State System) [12, 39] reemplazan la discretización temporal de los métodos clásicos por la cuantificación de las variables de estado. De esta manera, estos métodos resultan en una aproximación por eventos discretos del sistema original y tienen ciertas ventajas sobre los enfoques clásicos:

- Son muy eficientes para simular sistemas con discontinuidades frecuentes [37].
- Son muy eficientes para simular sistemas discontinuos de gran escala, dada su capacidad para explotar sistemas ralos [29].
- Pueden integrar sistemas stiff de manera eficiente, sin tener que invertir matrices o ejecutar iteraciones adicionales [45, 46].

La forma más sencilla de implementar los algoritmos QSS es mediante el uso de un motor de simulación DEVS (Discrete EVent System Specification) [64]. Por esta razón, muchas implementaciones de los métodos QSS son parte de herramientas de simulación DEVS.

Estas implementaciones, a pesar de ser simples, son ineficientes dado que tienen un costo computacional adicional alto impuesto por el mecanismo de sincronización y transmisión de eventos de los motores DEVS. Adicionalmente, los modelos deben ser descriptos como diagramas de bloques, lo que en algunas situaciones puede ser complejo.

Para poder resolver estos problemas, esta Tesis presenta el desarrollo de algoritmos y un motor de simulación autónomo para métodos de QSS, basado en las ideas de los motores de simulación para métodos de integración clásicos como DASSL [12, 53].

El motor de simulación autónomo QSS está compuesto por módulos escritos en lenguaje C que llevan a cabo las diferentes tareas necesarias para llevar a cabo la simulación. La familia entera de métodos de QSS está implementada en el motor y los modelos pueden contener discontinuidades temporales y de estado.

Una dificultad impuesta por los métodos QSS es que necesita información estructural del modelo a simular. Cada paso realizado con un método QSS involucra un cambio en una variable de estado y en las derivadas de estado que dependen de esta variable. Por lo tanto, el modelo debe permitir evaluar las derivadas de estado de manera individual y adicionalmente se deben definir matrices de incidencia para que el motor de simulación pueda evaluar las derivadas de estado que dependen explícitamente de la variable de estado que cambia en cada paso de simulación.

Dado que proporcionar la información estructural necesaria sobre el modelo puede resultar complejo para el usuario final, se desarrollaron herramientas que permiten obtener esta información de manera automática a partir de una descripción estandard del modelo a simular. Estas herramientas se implementaron en un entorno de modelado que permite definir modelos utilizando un subconjunto del lenguaje de simulación Modelica [25], y generan de manera automática el código C que describe el modelo y las matrices de incidencia necesarias.

El motor de simulación autónomo QSS fue diseñado para soportar la simulación de modelos complejos de gran escala desarrollados en diferentes áreas de la ingeniería y la comunidad científica en general. La simulación de este tipo de modelos tiene un costo computacional alto y dado el desarrollo de las computadoras multi–núcleo y los entornos de computación distribuida, su simulación en paralelo se impuso como la forma usual de reducir el costo computacional.

Los métodos de integración numérica clásicos requieren evaluar todas las funciones que definen el modelo en cada paso, lo que en el caso de modelos de gran escala (que pueden contener millones de variables) implica un costo computacional alto. Adicionalmente, si el modelo es stiff, se deben utilizar algoritmos implícitos que requieren invertir matrices de gran tamaño en una gran cantidad de pasos de simulación. Esto empeora si se tienen en cuenta modelos con discontinuidades frecuentes donde los algoritmos deben detectar cada discontinuidad y reiniciar la simulación luego de cada una de ellas.

Para poder reducir el costo computacional de este tipo de simulaciones, diferentes técnicas de paralelización para métodos de integración clásicos han sido propuestas [40, 41, 42, 52].

La naturaleza asíncrona de los métodos QSS implica que cada variable evoluciona de manera diferente. Por lo tanto, si un modelo de gran escala es dividido en dos submodelos o más, estos submodelos solamente necesitan interactuar en los pasos correspondientes a cambios en variables que son comunes a los diferentes submodelos. Este hecho permite ejecutar la simulación en diferentes unidades de cálculo de manera tal que cada unidad de cálculo integra un submodelo diferente.

Basados en esta idea, en [8] se desarrolló una implementación en paralelo de los métodos QSS para una arquitectura multi–núcleo. En ese trabajo se utilizó la herramienta de simulación de eventos discretos PowerDEVS [4], donde la sincronización entre los diferentes submodelos es llevada a cabo por un reloj de tiempo real. Una limitación impuesta por esta implementación es que se debe utilizar un sistema operativo de tiempo real, lo que restringe su uso general.

Utilizando la misma idea básica, desarrollamos dos nuevas técnicas para la simulación en paralelo de métodos QSS que no requieren el uso de un sistema operativo de tiempo real. En estas nuevas metodologías, la sincronización entre los diferentes submodelos es no estricta, permitiendo una diferencia acotada entre los tiempos de simulación lógicos. También demostramos formalmente que la sincronización no estricta de los nuevos algoritmos introducen un error numérico acotado adicional a la simulación que se suma al error introducido por los métodos QSS.

Todos los algoritmos desarrollados fueron implementados en el motor de simulación autónomo QSS y su rendimiento fue analizado en profundidad en diferentes

modelos de gran escala que nos permiten demostrar la potencialidad de el uso de las nuevas técnicas propuestas.

Como parte del análisis del rendimiento de las nuevas técnicas desarrolladas, se implementaron también herramientas que permiten la traducción automática de sistemas de ecuaciones diferenciales definidos en lenguajes de modelado estandard, utilizados en diferentes áreas de la comunidad científica, como OpenModelica y SBML lo que posibilita integrar el trabajo desarrollado con diferentes entornos de modelado y simulación.

1.1. Organización de la Tesis

La presente Tesis está organizada de la siguiente manera:

Este primer capítulo introductorio brinda una descripción de la Tesis completa relacionando los resultados obtenidos con el estado del arte actual.

En el segundo capítulo presentamos los conceptos preliminares necesarios que son utilizados a lo largo de la Tesis. Se introducen primero conceptos sobre métodos de integración numéricos clásicos y fundamentos de los métodos de intregración numérica QSS, como así también, conceptos básicos de lenguajes de modelado de sistemas contínuos.

También se introducen conceptos referentes a la simulación en paralelo de modelos de ecuaciones diferenciales ordinarias (EDOs) y la simulación en paralelo de sistemas de eventos discretos, para finalizar con una breve discusión sobre los resultados obtenidos hasta el momento en la simulación en paralelo de métodos QSS.

En el tercer capítulo se presentan los algoritmos y la implementación del nuevo entorno de simulación para métodos QSS desarrollado. Comenzamos describiendo la idea básica y la estructura del motor para luego pasar a describir en detalle cada uno de los componentes que lo integran. Finalmente, se analizan y discuten los resultados obtenidos con esta nueva herramienta.

El capítulo cuatro está dedicado a la integración del motor de simulación QSS con diferentes herramientas de modelado y simulación. Primero se presenta una extensión del compilador de la herramienta de modelado y simulación OpenModelica. Este trabajo está orientado a permitir simular modelos descriptos utilizando el lenguaje estandard Modelica utilizando el motor de simulación QSS. Luego se

presenta una herramienta de traducción de modelos descriptos en el lenguaje de modelado de sistemas biológicos SBML (System Biology Markup Language) y su integración con el motor de simulación QSS.

En el capítulo cinco se presentan resultados obtenidos al utilizar el motor de simulación QSS en modelos de diferentes campos de aplicación, donde se discuten y analizan en detalle los resultados obtenidos y se compara el rendimiento del nuevo motor de simulación con métodos clásicos de integración numérica.

En el capítulo seis, se presentan dos técnicas nuevas de simulación en paralelo para métodos QSS, presentando primero la idea básica y estructura del nuevo simulador en paralelo para luego discutir en detalle cada uno de los cambios introducidos al motor de simulación QSS secuencial. También en este capítulo se realiza un análisis formal del error introducido por la simulación en paralelo. Finalmente, se presentan y analizan en profundidad los resultados obtenidos en la simulación de cuatro modelos de gran escala.

Por último, en el capítulo siete, se presentan las conclusiones del trabajo realizado y los resultados obtenidos en esta Tesis.

1.2. Contribuciones Originales

La principal contribución de esta Tesis es el desarrollo de algoritmos que permiten simular modelos de gran escala utilizando métodos de QSS de manera eficiente. Adicionalmente, se desarrollaron herramientas que implementan los algoritmos presentados y permiten analizar en profundidad la aplicación de los mismos.

En la primer parte del trabajo, en el capítulo tres, se presentan nuevos algoritmos de simulación para métodos QSS y también un entorno de modelado que permite ejecutar las simulaciones.

En los capítulos cuatro y cinco se presentan contribuciones originales que muestran la integración de los algoritmos QSS con diferentes entornos de modelado y simulación.

Finalmente, en el capítulo seis, se presentan dos técnicas nuevas de para la simulación el paralelo de métodos QSS. Todo el trabajo de desarrollo presentado allí es original incluyendo también el desarrollo de una herramienta que permite evaluar las metodologías propuestas.

1.3. Publicaciones de Apoyo

La mayor parte de los resultados incluidos en esta Tesis ya fueron publicados en revistas o memorias de conferencias, mientras que el resto están en revisión.

Los primeros resultados relevantes de esta Tesis fueron el desarrollo de un motor de simulación autónomo para métodos de QSS que permite la simulación de sistemas híbridos de manera eficiente. Estos resultados fueron publicados en primer lugar en una conferencia local en dos partes [16, 18] y luego en una revista internacional [19].

Luego, se realizaron diferentes trabajos en los que permitieron comparar la eficiencia del nuevo motor de simulación y comparación con métodos de integración numérica clásicos. El estudio de la simulación con métodos QSS de ecuaciones de advección–difusión–reacción fue publicado primero en una conferencia nacional [3] y luego en una revista internacional [6].

Un estudio sobre la simulación eficiente de sistemas híbridos de generación de energía renovable fue publicado en una conferencia nacional [47].

También se realizó un estudio sobre la simulación de modelos de electrónica de conmutación que fue publicado en una revista internacional [44].

En este contexto, se realizó también un estudio sobre aplicación de los métodos QSS en modelos de toma de decisión celular en formación osea. El resultado de este trabajo fue publicado como capítulo del libro [59].

El trabajo correspondiente a la integración del motor de simulación QSS con el entorno de modelado y simulación OpenModelica fue publicado en una conferencia internacional en [7].

Por último, el trabajo correspondiente al desarrollo del motor de simulación en paralelo para métodos QSS fue enviado al la revista internacional y se encuentra en revisión.

Capítulo 2

Conceptos Preliminares

En este capítulo presentaremos los conceptos preliminares utilizados en el desarrollo de la presente Tesis.

Primero se introducen conceptos básicos de integración numérica (Sección 2.1) para luego describir los métodos de integración numérica QSS (Sección 2.2).

Luego se describe el formalismo de simulación de eventos discretos DEVS y la implementación de los métodos QSS utilizando este formalismo (Sección 2.2.4 y Sección 2.2.5).

A continuación, se presentan conceptos generales sobre la simulación en paralelo de Ecuaciones Diferenciales Ordinarias y simulación en paralelo de sistemas de eventos discretos, como así también la metodologías existentes para la simulación en paralelo de métodos QSS (Sección 2.3).

Finalmente, se presenta una breve descripción de las características principales del lenguaje Modelica (Sección 2.4).

2.1. Métodos Clásicos integración numérica

En las siguientes secciones se presentan características generales de los métodos de tiempo discreto.

2.1.1. Principios de integración numérica

Los métodos de integración numérica clásicos [12] surgen como una herramienta que permite la simulación de modelos de sistemas continuos dado que por lo general resulta muy difícil 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 numérica dado un 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 $f_i(\mathbf{x}, \mathbf{u}, t)$ o 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 siendo $t^* + h$ el instante de tiempo en el cual se quiere evaluar la aproximación, la trayectoria en dicho punto puede expresarse como sigue:

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

Luego, reemplazando $\frac{dx_i(t^*)}{dt}$ por la definición dada en la Ec. (2.1) en la serie (2.3) obtenemos:

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

Finalmente, a partir de la representación dada por la serie (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. Métodos 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:

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

El parámetro h, que define la distancia entre dos instantes de tiempo donde calculamos la solución, se denomina *paso de integración.*

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

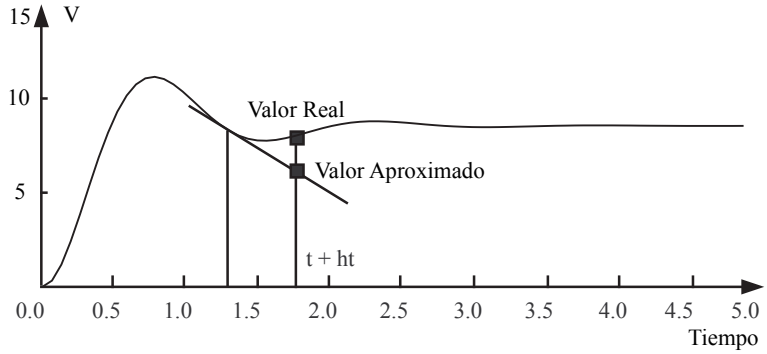


Figura 2.1: Integración numérica utilizando Forward Euler.

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

Otro método de integración numérica muy conocido, inspirado en el anterior y que recibe el nombre de *Método de Backward Euler*, reemplaza la fórmula de la

Ec. (2.5) por la siguiente:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\ \mathbf{f}(\mathbf{x}_{k+1}, t_k) \tag{2.6}$$

El método de Backward Euler tiene un pequeño inconveniente: de acuerdo a la Ec. (2.6) para calcular $\mathbf{x}_{k+1}$ necesitamos conocer $\mathbf{x}_{k+1}$.

Naturalmente, conociendo $\mathbf{x}_k$ podemos resolver de alguna forma la Ec. (2.6) y obtener de allí el valor de $\mathbf{x}_{k+1}$. Por este motivo, se dice que el método de Backward Euler es un *Método Implícito*, ya que para encontrar cada valor debemos resolver una ecuación.

Cuando la función $\mathbf{f}$ es lineal, la resolución de la ecuación implícita (2.6) es muy sencilla (si bien requiere invertir una matriz). Por el contrario, en el caso no lineal, en general necesitaremos utilizar algún algoritmo iterativo que encuentre la solución para cada instante de tiempo. Normalmente, se utiliza la *iteración de Newton*.

2.1.3. 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.

2.1.4. Estabilidad Numérica de un Método de Integración

Consideremos el sistema lineal y autónomo:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} \tag{2.7}$$

con condiciones iniciales como en la Ecuación (2.2). Su solución analítica es:

$$\mathbf{x}(t) = \exp(\mathbf{A} \cdot t) \cdot \mathbf{x_0} \tag{2.8}$$

la cual será *analíticamente estable* si todas las trayectorias permanecen acotadas cuando el tiempo tiende a infinito. El sistema (2.7) es analíticamente estable si y sólo si todos los autovalores de $\mathbf{A}$ tienen parte real negativa:

$$\mathbb{R}\mathrm{e}\{\mathrm{Eig}(\mathbf{A})\} = \mathbb{R}\mathrm{e}\{\lambda\} < 0{,}0 \tag{2.9}$$

Lugo, aplicando el algoritmo de FE definio por la Ec. (2.5) al sistema definido por la Ec. (2.7) se obtiene:

$$\mathbf{x}(t^* + h) = \mathbf{x}(t^*) + \mathbf{A} \cdot h \cdot \mathbf{x}(t^*) \tag{2.10}$$

que puede reescribirse en forma más compacta como:

$$\mathbf{x}(k+1) = [\mathbf{I^{(n)}} + \mathbf{A} \cdot h] \cdot \mathbf{x}(k) \tag{2.11}$$

donde $\mathbf{I^{(n)}}$ es una matriz identidad de la misma dimensión que $\mathbf{A}$, es decir, $n \times n$. En lugar de referirnos explícitamente al tiempo de simulación, lo que se hace es indexar el tiempo, es decir, k se refiere al k–ésimo paso de integración.

Lo que se hizo fue convertir el sistema continuo anterior en un sistema de *tiempo discreto* asociado:

$$\mathbf{x_{k+1}} = \mathbf{F} \cdot \mathbf{x_k} \tag{2.12}$$

donde la matriz de evolución discreta $\mathbf{F}$ puede calcularse a partir de la matriz de evolución continua $\mathbf{A}$ y del paso de integración h, como:

$$\mathbf{F} = \mathbf{I^{(n)}} + \mathbf{A} \cdot h \tag{2.13}$$

El sistema discreto de la Ec. (2.12) es analíticamente estable si y sólo si todos sus autovalores se encuentran dentro de un círculo de radio 1 alrededor del origen, llamado *círculo unitario*. Como consecuencia, podemos decir que todos los autovalores de Ec. (2.13) se puede concluir que **A** multiplicados por el paso de integración h deben estar contenidos en un círculo de radio 1 alrededor del punto $(-1,0)$.

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.

Por lo tanto, al utilizar Forward Euler, un sistema analíticamente estable puede dar un resultado numéricamente inestable si el paso de integración es demasiado grande.

El algoritmo de BE tiene la ventaja de que si el sistema es analíticamente estable, se garantizará la estabilidad numérica para cualquier paso de integración h. Este método es entonces mucho más apropiado que el de FE para resolver problemas con autovalores alejados sobre el eje real negativo del plano complejo. Esto es de crucial importancia en los sistemas *stiff* (ver Sección 2.1.9), es decir, sistemas con autovalores cuyas partes reales están alejadas entre sí a lo largo del eje real negativo.

A diferencia de FE, en BE el paso de integración deberá elegirse exclusivamente en función de los *requisitos de precisión*, sin importar el *dominio de estabilidad numérica*.

2.1.5. Métodos Monopaso

Los métodos de Forward y Backward Euler realizan sólo aproximaciones de primer orden. Debido a esto, para obtener una buena precisión en la simulación, se debe reducir excesivamente el paso de integración lo que implica una cantidad de pasos y de cálculos en general inaceptable.

Para obtener aproximaciones de orden mayor, es necesario utilizar más de una evaluación de la función $\mathbf{f}(\mathbf{x}, t)$ en cada paso. Cuando dichas evaluaciones se realizan de manera tal que para calcular $\mathbf{x}_{k+1}$ sólo se utiliza el valor de $\mathbf{x}_k$, se dice que el algoritmo es *monopaso*. Por el contrario, cuando se utilizan además valores anteriores de la solución ($\mathbf{x}_{k-1}$, $\mathbf{x}_{k-2}$, etc.), se dice que el algoritmo es *multipaso*.

Los métodos monopaso se suelen denominar también *Métodos de Runge–Kutta*, ya que el primero de estos métodos de orden alto fue formulado por Runge y Kutta a finales del siglo XIX.

Métodos de Runge–Kutta

Los métodos explícitos de Runge–Kutta realizan varias evaluaciones de la función $\mathbf{f}(\mathbf{x}, t)$ en cercanías del punto $(\mathbf{x}_k, t_k)$ y luego calculan $\mathbf{x}_{k+1}$ realizando una suma pesada de dichas evaluaciones.

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 fórmula 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 a partir de las ecuaciones:

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

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

Métodos Monopaso Implícitos

Como vimos antes, el método de Backward Euler es un algoritmo implícito cuya principal ventaja es preservar la estabilidad de la solución numérica para cualquier paso de integración. Sin embargo, al igual que Forward Euler, realiza sólo una aproximación de primer orden.

Hay diversos métodos implícitos monopaso de orden mayor que, al igual que Backward Euler, preservan la estabilidad.

Uno de los más utilizados es la *Regla Trapezoidal*. Este método implícito realiza una aproximación de segundo orden y tiene la propiedad (al menos en sistemas lineales y estacionarios) de que la solución numérica es estable si y sólo si la solución analítica es estable.

Si bien existen numerosos métodos implícitos monopaso, en la práctica no son tan utilizados ya que en general los métodos multipaso implícitos suelen ser más eficientes.

2.1.6. Algoritmos de Control de Pasos

Hasta aquí consideramos siempre el paso h como un parámetro fijo que debe elegirse previo a la simulación. Sin embargo, en muchos casos es posible implementar algoritmos que cambien el paso de integración de forma automática a medida que avanza la simulación.

Los algoritmos de *control de paso* tienen por propósito mantener el error de simulación acotado para lo cual ajustan automáticamente el paso en función del error estimado.

La idea es muy simple, en cada paso se hace lo siguiente:

1. Se da un paso con el método de integración elegido calculando $\mathbf{x}_{k+1}$ y cierto paso h.

2. Se estima el error cometido.

3. Si el error es mayor que la tolerancia, se disminuye el paso de integración h y se recalcula $\mathbf{x}_{k+1}$ volviendo al punto 1.

4. Si el error es menor que la tolerancia, se acepta el valor de $\mathbf{x}_{k+1}$ calculado, se incrementa el paso h y se vuelve al punto 1 para calcular $\mathbf{x}_{k+2}$.

La estima del error se realiza generalmente con dos métodos de orden distintos y suponiendo que el de orden mayor da una aproximación con menos error se hace la diferencia entre los dos métodos.

2.1.7. Métodos Multipaso

Los métodos monopaso obtienen aproximaciones de orden alto utilizando para esto varias evaluaciones de la función $\mathbf{f}$ en cada paso. Para evitar este costo computacional adicional, se han formulado diversos algoritmos que, en lugar de evaluar repetidamente la función $\mathbf{f}$ en cada paso, utilizan los valores evaluados en pasos anteriores.

Los métodos implícitos multipaso más utilizados en la práctica son los denominados *Backward Differentiation Formula* (BDF). Por ejemplo, el siguiente es el método de BDF de orden 3:

$$\mathbf{x}_{k+1} = \frac{18}{11}\mathbf{x}_k - \frac{9}{11}\mathbf{x}_{k-1} + \frac{2}{11}\mathbf{x}_{k-2} + \frac{6}{11} \cdot h \cdot \mathbf{f}_{k+1} \tag{2.14}$$

Este método tiene prácticamente el mismo costo computacional que Backward Euler, ya que la ecuación a resolver es muy similar. Sin embargo, BDF3 es de tercer orden.

En los métodos multipaso se puede también controlar el paso de integración de manera similar a la de los métodos monopaso. Sin embargo, las fórmulas de los métodos multipaso son sólo válidas asumiendo paso constante (ya que usan valores anteriores de la solución). Por lo tanto, para cambiar el paso en un método multipaso hay que interpolar los últimos valores de la solución a intervalos regulares correspondientes al nuevo paso de integración. En consecuencia, cambiar el paso tiene un costo adicional en este caso.

DASSL es un método de integración multi–paso muy popularizado basado en *Fórmulas de diferencia hacia atrás* o (Backward Differentiation Formula). Es el método utilizado por defecto en herramientas como OpenModelica y Dymola. Permite integrar también sistemas de ecuaciones diferenciales algebraicas (DAE).

2.1.8. Sistemas con Discontinuidades

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.

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 estará 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*.

Las discontinuidades pueden clasificarse dentro de dos grandes categorías: Eventos temporales y Eventos de estado.

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 cuándo ocurrirán, simplemente se le debe avisar al algoritmo de integración el tiempo de ocurrencia de los mismos. El algoritmo deberá entonces *programar* dichos eventos y cada vez que dé un paso deberá tener cuidado de no evitar ningún evento programado. Cada vez que el paso de integración h a utilizar sea mayor que el tiempo que falta para el siguiente evento, deberá utilizar un paso de integración que sea exactamente igual al tiempo para dicho evento.

De esta manera, una simulación de un modelo discontinuo puede interpretarse como una secuencia de varias simulaciones continuas separadas mediante transiciones discretas.

Eventos de Estado

Muy frecuentemente, el tiempo de ocurrencia de una discontinuidad no se conoce de antemano. Lo que se sabe es 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.

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 controladas. 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, se debe actuar 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.

2.1.9. Sistemas Rígidos (Stiff)

Un sistema lineal y estacionario se dice que es stiff cuando es estable y hay modos muy rápidos y modos muy lentos. El problema con los sistemas stiff es que la presencia de los modos rápidos obliga a utilizar un paso de integración muy pequeño para que la simulación no se vuelva inestable a causa del método de integración.

Formalmente, un sistema de Ecuaciones Diferenciales Ordinarias se dice stiff si, al integrarlo con un método de orden n y tolerancia de error local de 10^{-n}, el paso de integración del algoritmo debe hacerse más pequeño que el valor indicado por la estima del error local debido a las restricciones impuestas por la región de estabilidad numérica.

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. Los métodos implícitos que hemos visto poseen esta propiedad por lo cual son aptos para simular sistemas rígidos.

2.2. Métodos de Quantized State Systems

En esta sección presentamos los métodos de interacción numérica de Quantized State System (QSS), que reemplazan la discretización temporal que realizan los métodos de integración numérica clásicos por la cuantificación de las variables de estado. Comenzaremos por mostrar la idea básica detrás de estos métodos para luego dar una definición formal de los mismos.

2.2.1. Idea Básica

Consideremos el siguiente conjunto de EDOs

$$\begin{aligned} \dot{u}_1(t) &= 3 - u_1(t) \\ \dot{u}_2(t) &= u_1(t) - u_2(t) \\ \dot{u}_3(t) &= u_2(t) - u_3(t) \end{aligned} \tag{2.15}$$

con condiciones iniciales: $u_1(0) = 3$, $u_2(0) = u_3(0) = 0$. En este caso, las Ecs. (2.15), que pueden ser resueltas de manera analítica, pueden representar la aproximación por el método de líneas (MOL) de una ecuación de advección dada

$$\frac{\partial u(x,t)}{\partial t} = -a\frac{\partial u(x,t)}{\partial x}$$

para parámetros y condiciones iniciales dadas.

Consideremos ahora que en lugar de resolver las Ecs.(2.15) utilizando un enfoque clásico de discretización temporal, modificamos el sistema sustituyendo $u_i(t)$ por su parte entera $q_i(t) \triangleq floor(u_i(t))$ en el lado derecho de cada ecuación:

$$\begin{aligned} \dot{u}_1(t) &= 3 - floor(u_1(t)) = 3 - q_1(t) \\ \dot{u}_2(t) &= floor(u_1(t)) - floor(u_2(t)) = q_1(t) - q_2(t) \\ \dot{u}_3(t) &= floor(u_2(t)) - floor(u_3(t)) = q_2(t) - q_3(t) \end{aligned} \tag{2.16}$$

Resolvamos ahora este conjunto de ecuaciones:

- En el tiempo $t_0 = 0$ tenemos que $q_1(t_0) = 3$, $q_2(t_0) = q_3(t_0) = 0$.
  - Inicialmente, de acuerdo con las Ecs.(2.16), tenemos que $\dot{u}_1(t_0) = \dot{u}_3(t_0) = 0$ y $\dot{u}_2(t_0) = 3$ y estas derivadas no serán modificadas hasta alguna de las variables $u_i(t)$ cambie su parte entera.
  - Como $\dot{u}_1(t_0) = \dot{u}_3(t_0) = 0$, tenemos que q_1 y q_3 no modificarán su valor en este momento.
  - Por otro lado, el próximo cambio en $q_2(t)$ ocurre cuando $u_2(t) = 1$, dado que $u_2(t_0) = 0$ y su derivada es $\dot{u}_2(t_0) = 3$, llegará a este valor en el tiempo $t_1 = 1/3$.
- Luego, en el tiempo $t_1 = 1/3$ resulta que $q_2(t_1) = u_2(t_1) = 1$.
  - De acuerdo a las Ecs.(2.16) tenemos que $\dot{u}_2(t_1) = 2$ y $\dot{u}_3(t_1) = 1$.
  - En este caso, el próximo cambio en $q_2(t)$ ocurre en el tiempo $t_2 = t_1 + 1/2$ mientras que el próximo cambio en q_3 ocurre en el tiempo $t_1 + 1/1$.

- De esta manera, en el tiempo $t_2 = t_1 + 1/2 = 5/6$ tenemos que $q_2(t_2) = u_2(t_2) = 2$, mientras que $u_3(t_2) = u_3(t_1) + (t_2 - t_1)\dot{u}_3(t_1) = 1/2$.
  - Nuevamente, de acuerdo a las Ecs.(2.16) las nuevas derivadas son $\dot{u}_2(t_2) = 1$ and $\dot{u}_3(t_2) = 2$.
  - Por lo tanto, el próximo cambio en $q_2(t)$ ocurre en el tiempo $t_2 + 1$ mientras que el próximo cambio en q_3 debe ser recalculado y ocurre en el tiempo $t_3 = t_2 + 0{,}5/2$.
- En el tiempo $t_3 = t_2 + 1/4 = 13/12$ resulta que $q_3(t_3) = u_3(t_3) = 1$.
  - De acuerdo a las Ecs.(2.16) tenemos ahora $\dot{u}_3(t_3) = 1$.
  - En consecuencia, el siguiente cambio en $q_3(t)$ ocurre en el tiempo $t_3 + 1$.
- En el tiempo $t_4 = t_2 + 1 = 11/6$ tenemos que $q_2(t_4) = u_2(t_4) = 3$ y $u_3(t_4) = u_3(t_3) + (t_4 - t_3)\dot{u}_3(t_3) = 7/4$.
  - De acuerdo a las Ecs.(2.16) los nuevos valores de las derivadas son $\dot{u}_2(t_2) = 0$ y $\dot{u}_3(t_2) = 2$.
  - Luego, $q_2(t)$ no va a cambiar y el próximo cambio en $q_3(t)$ se debe recalcular y ocurre en el tiempo $t_5 = t_4 + 0{,}25/2$.
- En el tiempo $t_5 = t_4 + 1/8 = 47/24$ tenemos que $q_3(t_5) = u_3(t_5) = 2$.
  - De acuerod a las Ecs.(2.16) el nuevo valor de la derivada es $\dot{u}_3(t_2) = 1$.
  - Luego, el siguiente cambio en q_3 ocurre en el tiempo $t_6 = t_5 + 1$.
- Finalmente, en el tiempo $t_6 = t_5 + 1 = 71/24$ tenemos que $q_3(t_6) = u_3(t_6)$.
  - En este momento todas las derivadas son iguales a 0 por los que luego del tiempo t_6 no se producen más cambios en el sistema.

Las trayectorias de esta solución se muestran en la Figura 2.2, donde no se muestran las variables $u_1(t)$ y $q_1(t)$ dado que no cambian para ningún valor de t.

Este ejemplo muestra que reemplazar una variable $u_i(t)$ por su parte entera $floor(u_i(t))$ en el lado derecho de una EDO parece proveer una forma de integrar la ecuación. Se puede notar que bajo este esquema, estamos reemplazando la

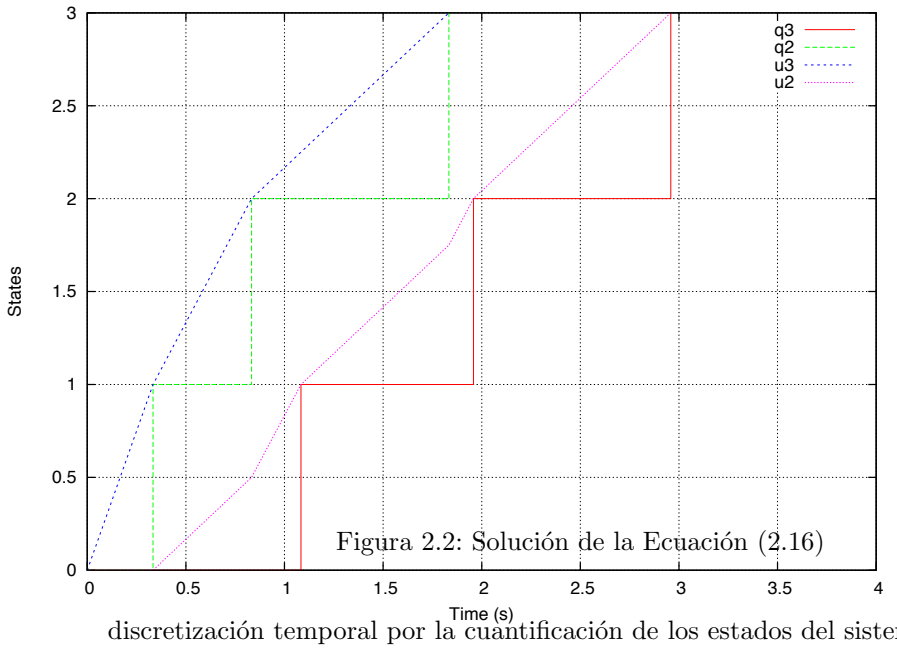


Figura 2.2: Solución de la Ecuación (2.16)

discretización temporal por la cuantificación de los estados del sistema, esta es la idea detrás de la familia de métodos de *Quantized State System*.

A partir del procedimiento descripto anteriormente, podemos destacar las siguientes observaciones:

- Luego de la inicialización, la simulación necesitó un total de 6 pasos.
- Cada paso fue local, relacionado a un cambio en la parte entera de un estado: En t_1, t_2 y t_4 el cambio ocurrió en la variable $q_2(t)$ mientras que en t_3, t_5 y t_6 el cambio ocurrió en $q_3(t)$, dado que $q_1(t)$ ya estaba en estado de equilibrio, su valor nunca cambio.
- Los cambios en $q_2(t)$ provocaron la evaluación de $\dot{u}_2$ y $\dot{u}_3$, mientras que los cambios en $q_3(t)$ provocaron la evaluación de $\dot{u}_3$ solamente. En consecuencia, luego de la inicialización, $\dot{u}_1$ nunca se evaluo, $\dot{u}_2$ fue evaluada 3 veces y $\dot{u}_3$ fue evaluada 6 veces.
- El análisis previo muestra que los cálculos son llevados a cabo solamente donde y cuando ocurre un cambio, lo que lleva a explotar de manera eficiente sistemas ralos.

- Los resultados presentados en la Figura 2.2 muestran pasos muy grandes, con saltos de 1 unidad entre los valores sucesivos de cada estado. Resultados más precisos se pueden obtener reemplazando la *función de cuantificación* $floor(u_i(t))$ por $\Delta Q \cdot floor(u_i(t)/\Delta Q)$, donde el parámetro ΔQ es llamado *quantum*.

- Si la primera linea de las Ecs.(2.15) se reemplaza por $\dot{u}_1(t) = 2{,}5 - u_1(t)$, entonces la primera linea de las Ecs.(2.16) se escribe como $\dot{u}_1(t) = 2{,}5 - q_1(t)$. En este caso el procedimiento falla, dado que inicialmente tenemos que $q_1(0) = 3$ y luego $\dot{u}_1(0) = -0{,}5$. En consecuencia tenemos que $u_1(0^+) < 3 \implies q_1(0^+) = 2$ de manera inmediata y luego $\dot{u}_1(0^+) = +0{,}5$, volviendo de esta forma a la situación inicial $u_1(0^{++}) = 3$. Este comportamiento cíclico provoca una oscilación infinita y la simulación no puede avanzar más alla del tiempo inicial.

  Esta dificultad puede ser resuelta mediante el uso de *histéresis* en la función de cuantificación, lo que lleva a la definición de los algoritmos de *Quantized State System*.

2.2.2. Definición de los Métodos QSS

A partir de la idea básica descripta en la sección anterior daremos ahora la definición de los métodos de Quantized State System (QSS).

Dada una ODE de la forma

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t) \tag{2.17}$$

el método Quantized State System de primer orden (QSS1) [39] aproxima (2.17) de la siguiente manera

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t), t) \tag{2.18}$$

Donde, $\mathbf{q}$ es el *vector de estados cuantificados*, y cada una de sus componentes está relacionada con la componente correspondiente del vector de estados $\mathbf{x}$ por la siguiente *función de cuantificación con histéresis*:

$$q_j(t) = \begin{cases} x_j(t) \text{ if } |x_j(t) - q_j(t^-)| \geq \Delta Q_j \\ q_j(t^-) \text{ otherwise} \end{cases} \tag{2.19}$$

donde ΔQ_j es llamado *quantum*.

Se puede ver fácilmente que $q_j(t)$ sigue una trayectoria seccionalmente constante que solamente cambia cuando la diferencia entre $q_j(t)$ y $x_j(t)$ es igual al quantum. De esta manera, luego de cada cambio en la variable cuantificada, tenemos que $q_j(t) = x_j(t)$.

Dada la particular forma de las trayectorias, la solución numérica de Eq. (2.18) es sencilla y a partir de esta solución podemos obtener un algoritmo de simulación que mostraremos a continuación.

Sea t_j el próximo tiempo en el cuál $|q_j(t) - x_j(t)| = \Delta Q_j$, para $j = 1, \ldots, n$ (i.e. el próximo tiempo de cambio de todas las variables cuantificadas), entonces, el algoritmo de simulación para QSS1 funciona de la siguiente manera:

Algoritmo 2.1: QSS1.

```
1  while(t < t_f) // simulate until final time tf
2    t = min(t_j) // adavance simulation time
3    i = argmin(t_j) // the i-th quantized state changes first
4    e_xi = t - t_i^x // elapsed time since last xi update
5    x_i = x_i + ẋ_i · e_xi // update i-th state value
6    q_i = x_i // update i-th quantized state
7    t_i = min(τ > t) subject to |q_i - x_i(τ)| = ΔQ_i // compute next i-th
         quantized state change
8    for each j ∈ [1,n] such that ẋ_j depends on q_i
9      e_xj = t - t_j^x // elapsed time since last xj update
10     x_j = x_j + ẋ_j · e_xj // update j-th state value
11     t_j^x = t // last xj update
12     ẋ_j = f_j(q,t) // recompute j-th state derivative
13     t_j = min(τ > t) subject to |q_j - x_j(τ)| = ΔQ_j // recompute j-th
           quantized state changing time
14   end for
15   t_i^x = t // last xi update
16 end while
```

De la línea 8 podemos ver que el algoritmo requiere solamente la evaluación de las derivadas de estado que dependen de cada variable de estado, i.e., la implementación de un integrador para el método QSS1 necesita información estructural del modelo.

El método QSS1 tiene las siguientes características:

- Las variables de estado cuantificadas $q_j(t)$ siguen trayectorias seccionalmente constantes, y las variables de estado $x_j(t)$ siguen trayectorias seccionalmente lineales.

- Las variables de estado y cuantificadas nunca difieren en más de un quantum

ΔQ_j. Este hecho permite definir propiedades como estabilidad y cotas de error globales para el método [39].

- El quantum ΔQ_j de cada variable de estado se puede seleccionar proporcional a la magnitud del estado, lo que permite un control intrínseco del error relativo [38].

- Cada paso en la simulación es local a la variable de estado x_j, y solamente provoca la evaluación de las derivadas de estado que dependen explícitamente de ella.

- Dado que las variables de estado siguen trayectorias seccionalmente lineales, es muy sencillo detectar discontinuidades. Asimismo, cuando se detecta una discontinuidad, su tratamiento no difiere al de un paso normal de simulación. Este hecho hace que el método QSS1 sea muy eficiente para simular sistemas con discontinuidades [37].

Sin embargo, el método QSS1 tiene algunas limitaciones dado que lleva a cabo una aproximación de primer orden, y no es adecuado para simular sistemas *stiff*.

La primer limitación se resolvió con la introducción de métodos de QSS de orden superior como el método de segundo order QSS2 [36], donde las variables cuantificadas siguen trayectorias seccionalmente lineales y el método de tercer orden QSS3, donde las variables cuantificadas siguen trayectorias seccionalmente parabólicas.

El algoritmo de simulación para el método QSS2 es similar al del método QSS1, con la diferencia de que también se deben calcular las pendientes de los estados cuantificados $\dot{q}_i(t)$ y las segundas derivadas de estado $\ddot{x}_i(t)$, como se muestra en el Algoritmo 2.2.

Cada paso dado utilizando el método QSS2 es más costoso que un paso dado con el método QSS1. En particular, se deben evaluar dos funciones escalares $\dot{x}_j$ y $\ddot{x}_j$ (lineas 16–17) y el cálculo de el próximo tiempo de cambio de q_j en la linea 18 implica resolver una ecuación cuadrática. Este costo adicional es compensado por el hecho de que el método QSS2 puede ejecutar pasos más grandes obteniendo mejores cotas de error.

Algoritmo 2.2: QSS2.

```
while(t < t_f) // simulate until final time tf
  t = min(t_j) // adavance simulation time
  i = argmin(t_j) // the i-th quantized state changes first
  e_xi = t - t_i^x // elapsed time since last xi update
  // update i-th state value and its derivative
  x_i = x_i + ẋ_i · e_xi + 0,5 · ẍ_i · e_xi^2
  ẋ_i = ẋ_i + ẍ_i · e_xi
  // update i-th quantized state
  q_i = x_i
  q̇_i = ẋ_i
  t_i = min(τ > t) subject to |q_i(τ) - x_i(τ)| = ΔQ_i // compute next i-th
      quantized state change
  for each j ∈ [1, n] such that ẋ_j depends on q_i
    e_xj = t - t_j^x // elapsed time since last xj update
    // update j-th state value and its derivatives
    x_j = x_j + ẋ_i · e_xj + 0,5 · ẍ_j · e_xj^2
    ẋ_j = f_j(q(t), t) // recompute state derivative
    ẍ_j = f_j(q(t), t) // recompute state second derivative
    t_j = min(τ > t) subject to |q_j(τ) - x_j(τ)| = ΔQ_j // compute next j-th
        quantized state change
    t_j^x = t // last xj update
  end for
  t_i^x = t // last xi update
end while
```

Los métodos QSS2 y QSS3 comparten las mismas ventajas y propiedades que QSS1 mencionadas anteriormente y son muy eficientes para simular sistemas discontinuos.

2.2.3. Métodos QSS Linealmente Implícitos

A pesar de las ventajas que presentan los métodos QSS1, QSS2 y QSS3, estos métodos son ineficientes para simular sistemas *stiff*. En estos casos, los métodos introducen oscilaciones de alta frecuencia espurias que provocan una gran cantidad de pasos de simulación y su consequente costo computacional [12].

Para poder resolver este problema, se extendió la familia de métodos QSS con nuevos algortimos llamados métodos QSS Linealmente Implícitos (LIQSS) que son adecuados para simular cierto tipo de sistemas *stiff* [45]. Los métodos LIQSS combinan los principios de los métodos QSS con los de los métodos linealmente implícitos clásicos. Existen algoritmos LIQSS que efectúan aproximaciones de primer, segundo y tercer orden llamados: LIQSS1, LIQSS2 y LIQSS3 respectivamente.

La idea principal detrás de los métodos LIQSS está inspirada en los métodos implícitos clásicos que evalúan las derivadas de estado en instantes de tiempo

futuro. En estos métodos clásicos, estas evaluaciones requieren iteraciones y/o inversiones de matrices para resolver las ecuaciones implícitas resultantes. Sin embargo, teniendo en cuenta que los métodos QSS conocen el valor futuro de los estados cuantificados (es $q_i(t) \pm \Delta Q_i$), la implementación de los algoritmos LIQSS es explícita y no requiere iteraciones o inversiones de matrices.

Los métodos LIQSS comparten con los métodos QSS la definición de la Ec. (2.18), pero los estados cuantificados son calculados de manera diferente, teniendo en cuenta el signo de las derivadas de estado.

En el método LIQSS1 la idea es que a $q_i(t)$ se le asigna el valor $x_i(t) + \Delta Q_i(t)$ cuando la derivada de estado futura $\dot{x}_i(t^+)$ es positiva. En otro caso, cuando la derivada de estado futura es negativa a $q_i(t)$ se le asigna el valor $x_i(t) - \Delta Q_i(t)$. Luego, cuando x_i alcanza a q_i, se da un nuevo paso de simulación. De esta manera, el estado cuantificado es un valor futuro de la variable de estado y las derivadas de la Ec. (2.18) son calculadas utilizando el valor futuro de las variables de estado, de igual manera que en los algoritmos implícitos clásicos.

Para poder predecir el signo de la derivada de estado futura se utiliza la siguiente aproximación lineal de la dinámica de la i–th variable de estado:

$$\dot{x}_i(t) = A_{i,i} \cdot q_i(t) + u_{i,i}(t) \tag{2.20}$$

donde $A_{i,i} = \frac{\partial f_i}{\partial x_i}$ es la i-th entrada de la diagonal principal de la matriz Jacobiana y $u_{i,i}(t) = f_i(\mathbf{q}(t), t) - A_{i,i} \cdot q_i(t)$ es un coeficiente afín.

Puede suceder que $A_{i,i} \cdot (x_i(t) + \Delta Q_i) + u_{i,i}(t) < 0$, i.e., cuando utilizamos $q_i(t) = x_i(t) + \Delta Q_i$ la derivada $\dot{x}_i(t^+)$ se vuelve negativa. Como también puede suceder que $A_{i,i} \cdot (x_i(t) - \Delta Q_i) + u_{i,i}(t) > 0$. En consecuencia, en este caso $q_i(t)$ no puede ser seleccionada como valor futuro de $x_i(t)$. Sin embargo, podemos seleccionar q_i de manera tal que $\dot{x}_i(t) = 0$. Este valor de equilibrio para q_i puede ser calculado a partir de la Ec. (2.20) como:

$$q_i = -\frac{u_{i,i}}{A_{i,i}} \tag{2.21}$$

Luego, el algoritmo de simulación LIQSS1 puede ser descripto de la siguiente manera:

Podemos ver que un paso dado por el método LIQSS1 agrega unos pocos cálculos adicionales con respecto a un paso dado con el método QSS1. En particular,

Algoritmo 2.3: LIQSS1.

```
1  while(t < t_f) // simulate until final time tf
2    t = min(t_j) // adavance simulation time
3    i = argmin(t_j) // the i-th quantized state changes first
4    e_xi = t − t_i^x // elapsed time since last xi update
5    x_i = x_i + ẋ_i · e_xi // update i-th state value
6    q_i^− = q_i //store previous value of qi
7    ẋ_i^− = ẋ_i //store previous value of dxi/dt
8    ẋ_i^+ = A_i,i · (x_i + sign(ẋ_i) · ΔQ_i) + u_i,i // future state derivative estimation
9    if (ẋ_i · ẋ_i^+ >0) //the state derivative keeps its sign
10     q_i = x_i + sign(ẋ_i) · ΔQ_i
11   else //the state changes its direction
12     q_i = −u_i,i/A_i,i // choose qi such that dxi/dt = 0
13   end if
14   t_i = min(τ > t) subject to x_i(τ) = q_i // compute next i-th quantized
         state change
15   for each j ∈ [1, n] such that ẋ_j depends on q_i
16     e_xj = t − t_j^x // elapsed time since last xj update
17     x_j = x_j + ẋ_j · e_xj // update j-th state value
18     t_j^x = t// last xj update
19     ẋ_j = f_j(q, t) // recompute j-th state derivative
20     t_j = min(τ > t) subject to x_j(τ) = q_j or |q_j − x_j(τ)| = 2ΔQ_j // recompute
           next j-th quantized state change
21   end for
22   // update linear approximation coefficients
23   A_i,i = (ẋ_i − ẋ_i^−)/(q_i − q_i^−) // Jacobian diagonal entry
24   u_i,i = ẋ_i − A_i,i · q_i // affine coefficient
25   t_i^x = t // last xi update
26 end while
```

el método LIQSS1 estima derivada de estado futura utilizando un modelo lineal (linea 8) y estima la entrada $A_{i,i}$ de la diagonal principal de la matriz Jacobiana y el coeficiente afín (lineas 23–24).

Cabe mencionar que en la linea 20 el algoritmo controla la condición adicional $|q_j - x_j(\tau)| = 2\Delta Q_j$, dado que un cambio en la variable q_i puede cambiar el signo de la derivada de estado $\dot{x}_j(t)$ y de esta manera x_j deja de aproximarse a q_j. En este caso, todavía se puede asegurar que la diferencia entre x_j y q_j es acotada (por $2\Delta Q_j$).

El método LIQSS1 comparte las mismas ventajas que el método QSS1 y puede integrar de manera eficiente sistemas *stiff* siempre que esta característica se manifieste mediante valores grandes en las entradas de la diagonal principal de la matriz Jacobiana. De la misma manera que el método QSS1, el método LIQSS1 realiza una aproximación de primer orden lo que implica que no es apto para simulaciones que requieran una precisión alta. Como consecuencia de esta limitación

se desarrollaron métodos LIQSS de mayor orden.

El método LIQSS2 combina las ideas de el método QSS2 y el método LIQSS1. En este caso, como en el método QSS2, el estado cuantificado sigue trayectorias seccionalmente lineales.

En este caso, q_i y $\dot{q}_i$ son calculadas de manera tal que se verifican las siguientes ecuaciones:

$$\begin{aligned} \dot{q}_i &= \dot{x}_i + h \cdot \ddot{x}_i = A_{i,i} \cdot q_i + u_{i,i} + h \cdot (A_{i,i} \cdot \dot{q}_i + \dot{u}_{i,i}) \\ q_i + h \cdot \dot{q}_i &= x_i + h \cdot \dot{x}_i + \frac{h^2}{2} \cdot \ddot{x}_i = \\ &= x_i + h \cdot (A_{i,i} \cdot q_i + u_{i,i}) + \frac{h^2}{2} \cdot (A_{i,i} \cdot \dot{q}_i + \dot{u}_{i,i}) \end{aligned} \tag{2.22}$$

donde $\dot{u}_{i,i}$ es el coeficiente afín de la pendiente. Se puede notar que la primer ecuación dice que la pendiente del estado cuantificado $\dot{q}_i$ es igual a la derivada de estado $\dot{x}_i$ en el tiempo $t + h$. La segunda ecuación dice que el estado cuantificado q_i es igual al estado x_i en el tiempo $t + h$.

Aquí, h se calcula como el tamaño máximo del paso de la Ec. (2.22) tal que $|q_i - x_i| \leq \Delta Q_i$.

Finalmente, el algoritmo de simulación para el método LIQSS2 funciona de la siguiente manera:

Podemos notar que un paso utilizando el método LIQSS2 agrega pocos cálculos con respecto al método QSS2. Se debe calcular el tamaño de paso máximo de segundo orden h (linea 12) y este valor es utilizado para calcular el estado cuantificado y su derivada (linea 13). También se debe calcular la entrada $A_{i,i}$ de la diagonal principal del Jacobiano y los coeficientes afines $u_{i,i}$ y $\dot{u}_{i,i}$ (lineas 26–27).

Al igual que en el método LIQSS1, el algoritmo debe controlar una condición que asegura que la diferencia entre x_j y q_j sea acotada (por $2\Delta Q_j$) si un cambio en el valor del estado cuantificado q_j causa que x_j deje de aproximarse a q_j (linea 22).

Siguiendo el mismo razonamiento, se puede desarrollar el método LIQSS3 combinando las ideas de el método QSS3 y el método LIQSS2.

Una característica fundamental de la familia de métodos QSS descripta en estas secciones es su naturaleza asíncrona, lo que lleva a una aproximación por

Algoritmo 2.4: LIQSS2.

```
while(t < t_f) // simulate until final time tf
  t = min(t_j) // adavance simulation time
  i = argmin(t_j) // the i-th quantized state changes first
  e_xi = t − t_i^x // elapsed time since last xi update
  // update i-th state value and its derivative
  x_i = x_i + ẋ_i · e_xi + 0,5 · ẍ_i · e_xi^2
  ẋ_i = ẋ_i + ẍ_i · e_xi
  ẋ_i^− = ẋ_i //store previous value of dxi/dt
  u_ii = u_ii + e_xi · u̇_ii // affine coefficient projection
  e_qi = t − t_i^q // elapsed time since last qi update
  q_i^− = q_i + e_qi · q̇_i //store previous value of qi projected
  h = MAX_2ND_ORDER_STEP_SIZE(x_i)
  [q_i, q̇_i] = 2ND_ORDER_step(x_i, h)
  t_i^q = t // last qi update
  t_i = min(τ > t) subject to x_i(τ) = q_i(τ)// compute next i-th quantized
      state change
  for each j ∈ [1, n] such that ẋ_j depends on q_i
    e_xj = t − t_j^x // elapsed time since last xj update
    // update j-th state value and its derivatives
    x_j = x_j + ẋ_j · e_xj + 0,5 · ẍ_j · e_xj^2
    ẋ_j = f_j(q(t), t) // recompute state derivative
    ẍ_j = ḟ_j(q(t), t) // recompute state second derivative
    t_j = min(τ > t) subject to x_j(τ) = q_j(τ) or |q_j(τ) − x_j(τ)| = 2ΔQ_j //
        compute next j-th quantized state change
    t_j^x = t // last xj update
  end for
  // update linear approximation coefficients
  A_i,i = (ẋ_i − ẋ_i^−)/(q_i − q_i^−) // Jacobian diagonal entry
  u_i,i = ẋ_i − A_i,i · q_i // affine coefficient
  t_i^x = t // last xi update
end while
```

eventos discretos del sistema original a diferencia de los métodos clásicos de integración numérica que realizan una discretización del tiempo lo que lleva a una aproximación por ecuaciones en diferencias del sistema original. Debido a esto, la forma natural de representar este tipo de métodos es utilizando un formalismo para especificar sistemas de eventos discretos que describiremos a continuación.

2.2.4. Sistemas de Eventos Discretos y DEVS

El formalismo DEVS (*Discrete EVent System specification*) permite representar todos los sistemas cuyo comportamiento entrada/salida pueda describirse mediante secuencias de eventos. Dentro de este formalismo, se pueden construir de modelos jerárquicos de una manera simple.

Un modelo DEVS está compuesto por submodelos atómicos que pueden ser combinados en modelos acoplados. Formalmente un modelo DEVS atómico se define mediante la siguiente estructura:

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

donde X es el conjunto de los valores de los eventos de entrada, i.e., el conjunto de todos los valores que un evento de entrada puede tomar, S es el conjunto de los valores del estado, Y es el conjunto de valores de los eventos de salida y λ es la función de salida. Para cada modelo DEVS atómico, la función de transición interna δ_{int}, la función de transición externa δ_{ext} y la función de avance de tiempo ta definen la dinámica del sistema que puede ser descripta de la siguiente manera:

- 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.

- La función de avance de tiempo retorna un número real no negativo que indica cuánto tiempo el sistema debe permanecer en un estado dado en ausencia de eventos de entrada.

- Cuando el tiempo indicado por la función de avance de tiempo es alcanzado, se ejecuta la función de salida y se produce un eveto de salida que depende del estado actual del modelo. Adicionalmente, se ejecuta la función de transición interna que puede cambiar el estado del modelo. El nuevo valor depende del estado actual.

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 teora 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. Los modelos DEVS acoplados son generalmente representados mediante el uso de puertos de entrada y salida.

2.2.5. Implementación de los Métodos de QSS

Como mencionamos anteriormente, la forma más sencilla de implementar los métodos de integración numérica QSS es mediante el uso de un motor de simulación de eventos discretos DEVS (Discrete EVent System Specification). En esta implementación, la aproximación QSS1 de cada componente del vector de estados dada por la Ec. (2.18) puede pensarse como el acoplamiento de dos subsistemas elementales: uno estático,

$$\dot{x}_j(t) = f_j(q_1, \cdots, q_n, t), \tag{2.23}$$

y uno dinámico

$$q_j(t) = Q_j(x_j(\cdot)) = Q_j(\int \dot{x}_j(\tau)d\tau) \tag{2.24}$$

donde Q_j es la función de cuantificación con histéresis definida por la Ec. (2.19) (notar que no es una función del valor instantáneo de $x_j(t)$, sino que es una función de la trayectoria $x_j(\cdot)$).

Teniendo en cuenta que las variables cuantificadas $q_j(t)$ siguen trayectorias seccionalmente constantes, y asumiendo que $f_j(\cdot)$ depende de t mediante una aproximación seccionalmente constante, resulta que ambos subsistemas, Ec. (2.23) y Ec. (2.24), reciben entradas seccionalmente constantes y calculan trayectorias de salida seccionalmente constantes. Estas trayectorias seccionalmente constantes pueden ser representadas por secuencias de eventos de manera sencilla.

La relación entre las secuencias de eventos de entrada y salida de estos subsistemas puede ser expresada por modelos DEVS simples. Las representaciones DEVS de la Ec. (2.23) son llamadas *funciones estáticas* y las representaciones DEVS de la Ec. (2.24) son llamadas *integradores cuantificados* [12].

Como consecuencia, la aproximación QSS de la Ec. (2.18) puede ser simulada por un modelo DEVS formado por el acoplamiento de n integradores cuantificados y n funciones estáticas (con la eventual adición de fuentes). El modelo DEVS obtenido de esta manera es idéntico a la representación por diagramas de bloques del sistema original definido por Ec. (2.17).

Los métodos QSS de orden superior se implementan de la misma manera. En este caso, los eventos representan cambios en trayectorias seccionalmente lineales o parabólicas y las funciones estáticas y los integradores cuantificados toman

en cuenta el valor de las variables y las pendientes y segundas derivadas de las trayectorias que reciben y envían.

Basado en estas ideas, la familia entera de métodos QSS fueron implementadas en PowerDEVS [4], una plataforma de simulación basada en el formalismo DEVS especialmente diseñada y adaptada para simular sistemas híbridos utilizando métodos QSS. Adicionalmente, los métodos explícitos de primer a tercer orden fueron implementados en una librería DEVS para Modelica [2] y el método de primer orden QSS1 se implementó en CD++ [13] y VLE [54].

Las implementaciones de métodos QSS basadas en el formalismo DEVS son simples pero no son eficientes. El siguiente ejemplo muestra este hecho.

Consideremos el siguiente sistema de segundo orden

$$\begin{aligned} \dot{x}_1(t) &= 2 \cdot x_2 \\ \dot{x}_2(t) &= -\sin(x_1) - 3 \cdot x_2 \end{aligned}$$

y su aproximación QSS

$$\begin{aligned} \dot{x}_1(t) &= 2 \cdot q_2 \\ \dot{x}_2(t) &= -\sin(q_1) - 3 \cdot q_2 \end{aligned} \tag{2.25}$$

Esta aproximación puede ser simulada en PowerDEVS por el modelo definido en la Figura 2.3.

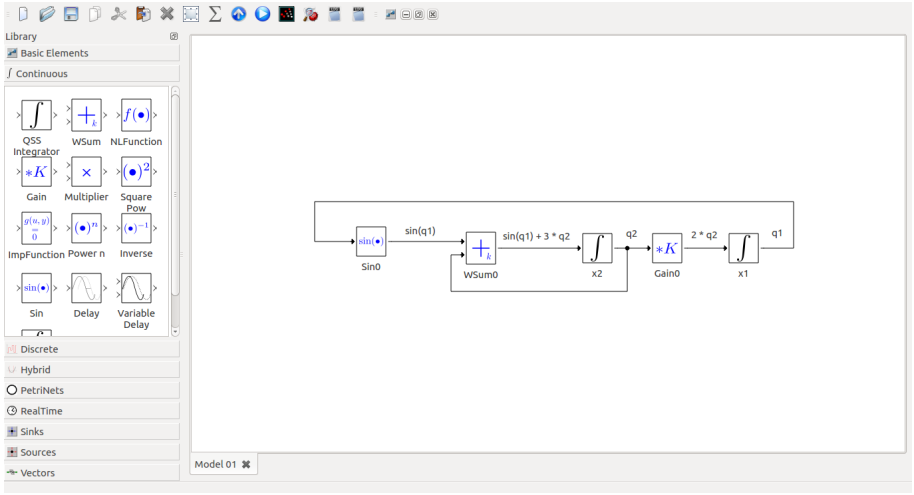


Figura 2.3: Modelo PowerDEVS de la Ec. (2.25)

Asumamos que el primer paso de la simulación corresponde a un cambio en la variable q_2. Este caso representa una transición interna en el integrador cuantifi-

cado `x2` en el modelo DEVS.

Luego, el motor de simulación DEVS procede de la siguiente manera:

1. El motor de simulación avanza el tiempo al tiempo correspondiente al próximo evento (i.e., el tiempo de cambio en q_2).

2. El integrador cuantificado `x2` calcula el nuevo valor de q_2 y envía la salida correspondiente a los bloques `WSum0` y `Gain0` (1 llamada a función).

3. Las funciones estáticas `Wsum0` y `Gain0` ejecutan sus funciones de transición internas al recibir el nuevo valor de q_2 y actualizan el valor de la variable de *avance de tiempo* a $\sigma = 0$ (4 llamadas a función).

4. El integrador cuantificado `x2` ejecuta su *Función de Transición Interna* y calcula el tiempo para el próximo evento de salida (2 llamadas a función).

5. El motor debe calcular cuál de los 5 bloques ejecuta el próximo evento. En este caso los bloques `WSum0` y `Gain0` deben ejecutarse de manera inmediata y debe seleccionarse el que tenga mayor prioridad. Asumamos que selecciona el bloque `Gain0`.

6. La función estática `Gain0` calcula $2 \cdot q_2$ y envía el evento de salida correspondiente al bloque `x1` (1 llamada a función).

7. El integrador cuantificado `x1` ejecuta su función de transición interna al recibir $2 \cdot q_2$, recalcula x_1 y el tiempo restante para su próximo evento de salida (i.e., el tiempo para el próximo cambio en q_1) (2 llamadas a función).

8. La función estática `Gain0` ejecuta su *Función de Transición Interna* y actualiza el valor de la variable de *avance de tiempo* a $\sigma = \infty$ (2 llamadas a función).

9. La función estática `Wsum0` calcula $\sin(q_1) + 3 \cdot q_2$ y envía el evento de salida correspondiente al bloque `x2` (1 llamada a función).

10. El integrador cuantificado `x2` ejecuta su función de transición externa al recibir $\sin(q_1) + 3 \cdot x_2$, recalcula x_2 y el tiempo para su próximo evento de salida (i.e., el tiempo para el próximo cambio en q_2) (2 llamadas a función).

11. La función estática `Wsum0` ejecuta si *Función de Transición Interna* y actualiza el valor de la variable de avance de tiempo a $\sigma = \infty$ (2 llamadas a función).

12. El motor calcula cuál de los 5 bloques debe ejecutar el próximo evento. En este caso, debe seleccionar un integrador cuantificado (`x1` o `x2`).

Por lo tanto, cada cambio en una variable cuantificada implica una serie de acciones que son llevadas a cabo por los diferentes bloques que integran el modelo y por el motor de simulación DEVS.

Como muestra el ejemplo, durante un paso de simulación se debe ejecutar un mínimo de 17 llamadas a función. En este caso consideramos llamadas a las funciones externas, internas, de salida y de avance de tiempo de los 5 bloques DEVS. Adicionalmente, el motor debe ejecutar 3 búsquedas del mínimo entre los 5 bloques definidos.

Cabe mencionar que estas acciones son independientes de la implementación del motor de simulación DEVS. En este ejemplo solamente tomamos en cuenta la definición del formalismo DEVS.

Sin embargo, dado un cambio en q_2 las únicas acciones necesarias son

1. Avanzar el tiempo al tiempo de cambio de la variable q_2.

2. Calcular el nuevo valor de q_2.

3. Calcular las nuevas derivadas $\dot{x}_1(t) = 2 \cdot q_2$ y $\dot{x}_2 = -\sin(q_1) - 3 \cdot q_2$.

4. Recalcular el tiempo de cambio para las variables q_1 y q_2.

5. Seleccionar cuál de las dos variables ejecuta el próximo evento.

De este análisis podemos concluir que dividir el modelo original en integradores cuantificados y funciones estáticas para construir un modelo DEVS equivalente es ineficiente dado que para poder ejecutar un paso de simulación, el mecanismo de simulación DEVS debe calcular y propagar varios eventos.

Una implementación DEVS más eficiente debería consistir solamente en 2 bloques atómicos *super–bloques*: el primero que calcule q_1 a partir de q_2 y el segundo que calcule q_2 a partir de q_1. Sin embargo, el comportamiento de estos

super–bloques sería más complejo que utilizar integradores cuantificados y funciones estáticas. Un modelo compuesto de *super–bloques* depende de las funciones $f_i(\mathbf{q}, t)$ y los usuarios deberían definir un modelo atómico DEVS para cada estado, lo cuál no es práctico o incluso imposible para modelos complejos y de gran escala.

Debemos mencionar también que una implementación basada en *super–bloques* ejecutaría pasos de simulación innecesarios, que involucran transmitir los valores de las variables cuantificadas q_i entre los diferentes bloques. Podemos concluir que es más eficiente compartir los valores de las variables en un arreglo común.

El análisis anterior motivó el desarrollo de un motor de simulación autónomo para métodos de integración QSS. A pesar de que el concepto de *motor de simulación autónomo* implica que la simulación no es llevada a cabo por un motor de simulación DEVS, veremos que los algoritmos presentados contienen rutinas que pueden ser pensadas como un motor de simulación DEVS *ad–hoc*.

Un primer intento de implementar los métodos de QSS de primer a tercer orden de manera autónoma fue presentado en la herramienta de simulación basada en Java *Open Source Physics* [14], pero esta implementación resultó ser más ineficiente que PowerDEVS y adicionalmente, la información estructural sobre el modelo necesaria para la simulación con métodos QSS debía ser proporcionada de manera manual.

2.3. Simulación en Paralelo

La simulación de modelos complejos y de gran escala usualmente conlleva un costo computacional alto. Debido al desarrollo de procesadores multi–núcleo como así también de clusters de computadoras multi–nodo en los últimos años, la simulación en paralelo de sistemas de tiempo continuo representa la manera usual de reducir los costos de simulación.

La simulación en paralelo utilizando métodos de QSS se puede enmarcar dentro del campo de la simulación en paralelo de EDOs en general y en el campo de la simulación en paralelo de sistemas de eventos discretos. Diferentes estrategias en ambos campos han sido propuestas a lo largo de los años y en las siguientes secciones presentaremos un breve resumen de ambos.

2.3.1. Simulación en Paralelo de EDOs

En las últimas décadas, se han propuesto diferentes estrategias para la simulación en paralelo de EDOs utilizando métodos de integración numérica clásicos. Estas estrategias son caracterizadas de acuerdo a los cálculos que realizan en paralelo. Las principales categorias son:

- *Paralelismo sobre el modelo:* Esta técnica esta basada en el división del modelo matemático representado por la Ec. (2.17) en submodelos que pueden ser ejecutados en paralelo.
- *Paralelismo sobre el método:* Dentro de este enfoque, los cálculos intrínsecos del método de integración numérica utilizado son ejecutados en paralelo. Esta técnica es usualmente integrada al paralelismo sobre el modelo.
- *Paralelismo sobre los pasos de simulación:* Aquí, los sucesivos pasos de integración ejecutados por la simulación son calculados en paralelo.

Un entorno de simulación experimental que permite comparar los diferentes enfoques propuestos es presentado en [52], en donde se presentan resultados de simulaciones realizadas en sistemas relativamente pequeños obteniendo una aceleración de aproximadamente 2 veces para un máximo de 10 núcleos.

En [35, 40, 41] se presenta una implementación que permite el eficiente aprovechamiento de accesos a memoria utilizando métodos Iterativos de Runge Kutta en arquitecturas de memoria compartida. En estos trabajos, combinan estrategias de paralelismo sobre el método, modelo y sobre los pasos de simulación para obtener una aceleración de hasta 300 veces utilizando 480 núcleos en una supercomputadora, simulando un modelo de hasta 8.000.000 de variables de estado. A su vez, diferentes aplicaciones de la strategia de paralelismo sobre los pasos de simulación son presentados y estudiados en [1, 15, 42].

2.3.2. Simulación en Paralelo de Sistemas de Eventos Discretos

La simulación en paralelo de Sistemas de Eventos Discretos ha sido estudiada en los últimos años [27]. La idea básica es dividir el modelo original en submodelos y simular cada submodelo de manera concurrente utilizando diferentes *procesos lógicos* (PLs), donde cada uno de estos procesos tiene su tiempo de simulación

lógico. Dado que los submodelos son usualmente interdependientes, el tiempo lógico de cada submodelo debe ser sincronizado para poder satisfacer la *restricción de causalidad*.

En la literatura, las diferentes estrategias existentes son divididas en tres categorías:

- Algoritmos *Conservadores*, como CMB [48], donde existe un mecanismo de sincronización que fuerza a cada PL a esperar hasta que es seguro que no va a recibir eventos del "pasado". Las estrategias conservadoras usualmente permiten una pequeña aceleración de los tiempos de simulación y tienen ciertos inconvenientes (como posibles *deadlocks*). Estos algoritmos pueden ser mejorados mediante la introducción de estrategias de *LookAhead* [33].

- Algoritmos *Opatimistas*, donde para PL avanza su tiempo lógico tanto como puede y utiliza un mecanismo que permite retrotraer la simulación cuando se detecta una inconsistencia entre los mensajes intercambiados. Comparadas con las técnicas conservadoras, las estrategias optimistas permiten ejecutar una mayor cantidad de cálculos en paralelo con el costo adicional de tener que guardar los estados intermedios de simulación y la introducción del mecanismo de que permite retrotraer la simulación. Algoritmos como TimeWarp [34, 55], son ejemplos de este enfoque.

- Finalmente, en [56], se estudia la idea de evitar la sincronización entre PL por completo, i.e. cada PL avanza su tiempo lógico tan rápido como puede sin tener en cuenta el tiempo lógico del resto de los PLs. Los resultados muestran que esta estrategia permite una mayor aceleración en los tiempos de simulación que los enfoques previamente nombrados con el costo de introducir errores en la simulación debido a la violación de la restricción de causalidad.

2.3.3. Simulación en Paralelo con Métodos QSS

La naturaleza asíncrona de los métodos QSS simplifica la paralelización de los cálculos realizados. Como mencionamos anteriormente, los métodos QSS pueden ser representados como sistemas de eventos discretos, por lo tanto, se podrían

utilizar en principio las estrategias de paralelización presentadas en la sección anterior.

Sin embargo, ninguna de estas estrategias se adapta de manera satisfactoria a los algoritmos de simulación QSS. Al aplicar la sincronización estricta propuesta en las estrategias conservadoras, no se permite la ejecución de cálculos concurrentes como se muestra en [8]. Por otro lado, aplicar estrategias optimistas requiere el uso de una gran cantidad de memoria para implementar el mecanismo que permite retrotraer la simulación. Finalmente, aplicar la técnica que evita la sincronización entre diferentes PLs podría introducir un error inaceptable en la simulación.

Un estudio inicial sobre la aplicación de algoritmos optimistas para la simulación en paralelo de métodos QSS en un cluster utilizando memoria distribuida fue realizado en [50] donde se obtuvo una aceleración de 2 veces para simulaciones que utilizaron 4 procesadores. Luego, en [43] los autores presentan una implementación de los métodos QSS para una arquitectura GPU de memoria compartida. Los resultados preliminares muestran que se puede obtener una aceleración de hasta 8 veces para modelos de 64 variables de estado y utilizando 64 procesadores. Estos resultados no fueron extendidos a modelos de mayor escala debido a las limitaciones impuestas por la arquitectura GPU.

Otro trabajo relacionado fue presentado en [4] donde se proponen dos técnicas de paralelización, SRTS y ASRTS para una arquitectura multinúcleo de memoria compartida. En este trabajo, la sincronización entre los diferentes submodelos es llevada a cabo por un reloj de tiempo real. Los resultados obtenidos en modelos híbridos de hasta 5000 variables muestran que utilizando estas técnicas se puede obtener una aceleración de hasta 9 veces utilizando 12 procesos lógicos. Adicionalmente, en este trabajo se analiza el error introducido por las técnicas de paralelización mostrando que si la diferencia entre los tiempos lógicos de los distintos PLs se mantiene acotada, la paralelización introduce un error numérico adicional acotado. Cabe mencionar que estas técnicas fueron implementadas en PowerDEVS, una herramienta basada en el formalismo DEVS, por lo que presenta las limitaciones mencionadas anteriormente.

En conclusión, previo a este trabajo, la simulación en paralelo utilizando métodos QSS estaba limitada a problemas relativamente pequeños (varios miles de va-

riables de estado, como máximo) y su implementación en diferentes arquitecturas utilizando un número pequeño de procesadores.

2.4. Modelica

La implementación de los métodos QSS basadas en el formalismo DEVS obligan a dar una descripción de diagramas de bloques del modelo. Sin embargo, existen lenguajes de modelado de alto nivel que permiten otro tipo de representaciones y que son utilizados por gran parte de la comunidad de modelado y simulación. Parte del objetivo del trabajo es permitir la integración de los métodos QSS con este tipo de representaciones.

Las primeras herramientas de modelado y simulación desarrollados requerían que los modelos sean representados en el mismo lenguaje de programación que el motor de simulación, usualmente C o Fortran. De esta manera, el modelo a simular y el motor de simulación compartían la misma representación. Este modo de representar modelos no es adecuado dado dificulta la reutilización y la codificación de modelos complejos.

En la década del 70 se comenzaron a desarrollar lenguajes de modelado específicos que permitían describir el modelo a simular de una manera estructura e independiente del motor de simulación utilizado. Producto de este esfuerzo, en las últimas décadas, se desarrolló un estandard llamado Modelica [25] que fue adoptado ampliamente por la comunidad de modelado y simulación.

Modelica es un lenguaje de modelado de alto nivel orientado a objetos que permite describir sistemas complejos y heterogéneos de gran escala permitiendo además reutilizar y componer modelos de manera sencilla.

Los modelos en este lenguaje son definidos por ecuaciones diferenciales, algebraicas y discretas. Diferentes submodelos pueden ser interconectados para crear nuevos modelos y existen numerosas herramientas de software que permiten describir estos de modelos.

Existen también compiladores comerciales y libres que convierten modelos Modelica en modelos ejecutables. Dentro de los más reconocidos podemos mencionar a Dymola [10] y OpenModelica [26].

Capítulo 3

Implementación de un Simulador Autónomo de QSS

Como vimos en la Sección 2.2.5, la implementación de los métodos de integración numérica QSS mediante el uso de un motor de simulación de eventos discretos DEVS es sencilla pero ineficiente debido a que la sobrecarga impuesta por el mecanismo de sincronización y transmisión de eventos propio del motor generan un costo computacional adicional. Otra limitación es que, como mencionamos en la Sección 2.4, utilizar este tipo de representaciones implica que los modelos deben ser definidos como diagramas de bloques.

Debido a estas restricciones, se desarrolló un motor de simulación autónomo para métodos QSS que permite simular modelos definidos sin utilizar diagramas de bloques, utilizando una representación estandard compartida con lo métodos de integración clásicos. En este capítulo, describimos las nuevas metodologías y su implementación.

3.1. Estructura Básica

Como explicamos anteriormente, los métodos de integración numérica QSS calculan

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{q}, t) \tag{3.1}$$

donde cada componente de $\mathbf{q}(t)$ es una aproximación seccionalmente polinomial del estado $\mathbf{x}(t)$ correspondiente. Los diferentes métodos de QSS difieren en la forma en que realizan esta aproximación.

Esto tipo representa el caso puramente continuo. El motor de simulación autónomo para métodos QSS permite simular modelos que contengan discontinuidades. Por lo tanto en el caso general, los modelos son representados de la siguiente manera:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}, \mathbf{d}, t) \tag{3.2}$$

donde $\mathbf{d}$ es un vector de variables discretas que pueden cambiar de valor solamente cuándo se cumple la condición

$$ZC_i(\mathbf{x}, \mathbf{d}, t) = 0 \tag{3.3}$$

para algún $i \in \{1, \cdots, z\}$. Las componentes ZC_i forman un vector de funciones de cruce por cero $\mathbf{ZC}(\mathbf{x}, \mathbf{d}, t)$. Cuando una condición de cruce por cero dada por la Ec. Eq. (3.3) se cumple, las variables discretas pueden cambiar de valor de acuerdo con el handler del evento correspondiente:

$$(\mathbf{x}(t), \mathbf{d}(t)) = H_i(\mathbf{x}(t^-), \mathbf{d}(t^-), t) \tag{3.4}$$

El motor de simulación presentado está estructurado de manera tal que existen tres módulos que interactúan en tiempo de ejecución y son los encargados de llevar adelante la simulación:

1. El **Integrador**, que integra la Ec. (3.2) asumiendo conocidas las trayectorias seccionalmente polinomiales de las variables de estado cuantificadas $\mathbf{q}(t)$.

2. El **Cuantificador**, que calcula $\mathbf{q}(t)$ a partir de $\mathbf{x}(t)$ de acuerdo al método de QSS utilizado y las tolerancias especificadas (existe una instancia de este módulo para cada método de QSS implementado). Este módulo es el encargado de calcular los coeficientes del polinomio de aproximación de cada variable de estado cuantificada $q_i(t)$ y los respectivos tiempos en los que debe comenzar una nueva sección polinomial (i.e. cuando se cumple la condición $|q_i(t) - x_i(t)| = \Delta Q_i$)

3. El **Modelo**, que calcula las derivadas de estado escalares $\dot{x}_i = f_i(\mathbf{q}, \mathbf{d}, t)$, las funciones de cruce $ZC_i(\mathbf{x}, \mathbf{d}, t)$, y sus handlers correspondientes $H_i(\mathbf{q}, \mathbf{d}, t)$. Además de proveer información estructural sobre el modelo requerida por los algoritmos.

La información estructural necesaria para los métodos QSS es extraída de manera automática en tiempo de compilación por un cuarto módulo llamada **Generador de Modelos**. Este módulo es el encargado de obtener la información estructural a partir de modelos descriptos en un subconjunto del lenguaje de modelado Modelica [24] y de generar un instancia del módulo **Modelo** para un modelo dado incluyendo la posibilidad de evaluar las derivadas de estado escalares de manera separada.

La Figura 3.1 muestra el esquema de interacción básica entre los módulos mencionados anteriormente.

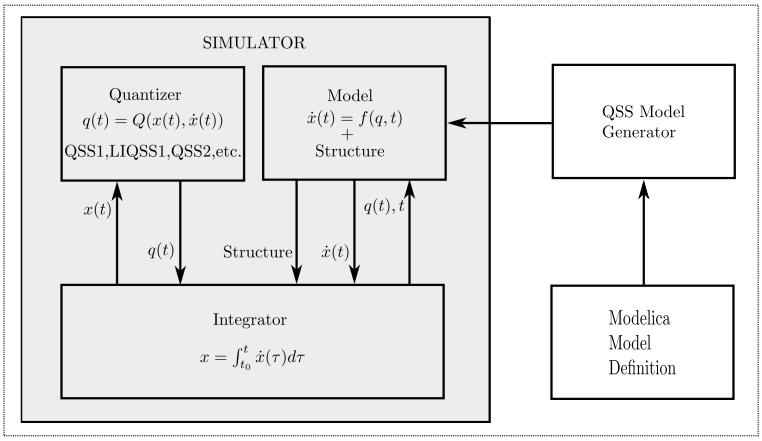


Figura 3.1: Stand Alone QSS Solver – Basic Interaction Scheme

Cabe mencionar que los métodos de integración numérica clásicos evalúan el lado derecho completo de la Ec. (2.17) en cada paso de simulación. En consecuencia, los modelos solamente contienen el código correspondiente que permite evaluar $\mathbf{f}(\mathbf{x}, t)$.

Una característica distintiva los métodos QSS es que diferentes variables de estado son actualizadas en tiempos de simulación diferentes. Como consecuencia, los modelos deben permitir la evaluación de las componentes individuales de la función $\mathbf{f}$ para que luego de un cambio en una variable de estado cuantificada q_i solamente se evalúen las componentes de $\mathbf{f}$ que dependen explícitamente de q_i.

La información estructural necesaria para la simulación es representada por cuatro matrices de incidencia:

- SD (estados a derivadas) donde $SD_{i,j} = 1$ indica que la variable de estado x_i es necesaria para calcular derivada de estado $\dot{x}_j$.
- SZ (estados a funciones de cruce por cero) donde $SZ_{i,j} = 1$ indica que la variable de estado x_i es necesaria para calcular la función de cruce por cero ZC_j.
- HD (handlers a derivadas de estado) donde $HD_{i,j} = 1$ indica que la ejecución del handler H_i cambia el valor de algunas variables de estado o discretas que son necesarias para calcular $\dot{x}_j$.
- HZ (handlers a funciones de cruce por cero) donde $HZ_{i,j} = 1$ indica que la ejecución del handler H_i cambia el valor de algunas variables de estado o discretas que son necesarias para calcular ZC_j.

3.2. Implementación de los Módulos

En esta sección describiremos en detalle el funcionamiento de los módulos que componen el motor de simulación autónomo para métodos QSS.

3.2.1. Módulo Integrador

El módulo **Integrador** es el encargado de avanzar el tiempo de simulación y calcular la representación polinomial de las componentes $x_i(t)$ del vector de estados $\mathbf{x}(t)$:

$$x_i(t) = \sum_{k=0}^{n} x_{i,k} \cdot (t - t_i^x)^k \tag{3.5}$$

utilizando una aproximación dada por las componentes $q_i(t)$ del vector de estados cuantificados $\mathbf{q}(t)$:

$$q_i(t) = \sum_{k=0}^{n-1} q_{i,k} \cdot (t - t_i^q)^k \tag{3.6}$$

donde n es el orden del método. Con este fin, integra la Ec. (3.2) evaluando las componentes de $\mathbf{f}$ correspondientes y, en presencia de discontinuidades, las correspondientes funciones de cruce por cero.

Cada paso en la simulación corresponde a un cambio en una variable de estado cuantificada q_i o a la ejecución de un handler H_i debido a que se cumple la condición de cruce por cero $ZC_i(t) = 0$. El integrador guarda los valores de las variables de estado, los estados cuantificados y las variables discretas x_i, q_i, y d_i,

respectivamente. Adicionalmente, también guarda el valor del tiempo en el cual se produce el próximo cambio para cada variable de estado cuantificada tx_i y el tiempo en el cual se produce el próximo cruce por cero para cada función de cruce por cero tz_i.

Teniendo en cuenta estas consideraciones, la rutina de simulación principal llevada a cabo por el módulo **Integrador** se muestra en el Algoritmo 3.1.

Algoritmo 3.1: Módulo Integrador

```
1   while t < t_f //while simulation time t is less than the final time tf
2     t_x = min(tx_j) // time of the next change in a quantized variable
3     t_z = min(tz_j) // time of the next zero--crossing time
4     t=min(t_x,t_z) //advance simulation time
5     if t = t_x then //quantized state change
6       i = argmin(tx_j) // the i-th quantized state changes first
7       Quantized_State_Step(i) //execute a quantized state change on
            variable i
8     else //zero--crossing
9       i = argmin(tz_j) // the i-th event handler is executed first
10      Event_Handler_Step(i) //execute the procedure for the i-th
            event handler
11    end if
12  end while
```

Cuando el próximo paso de simulación es debido a un cambio en una variable cuantificada q_i en el tiempo t, el **Integrador** procede de la siguiente manera:

Algoritmo 3.2: QSS Integrator Module - Quantized State Change

```
1  Quantized_State_Step(i)
2  {
3    integrateState(x_i, ẋ_i, t) // integrate i-th state up to time t.
4    Quantizer.update(x_i, q_i) // update i-th quantized state q_i
5    tx_i = Quantizer.nextTime(x_i, q_i) //compute next i-th quantized state
         change time
6    for each j such that SD_{i,j} = 1
7      integrateState(x_j, ẋ_j, t) // integrate j-th state up to time t.
8      ẋ_j = Model.f_j(q(t), d(t), t) // recompute j-th state derivative
9      tx_j = Quantizer.nextTime(x_j(t), q_j(t)) //recompute next j-th
           quantized state change time
10   end for
11   for each j such that SZ_{i,j} = 1
12     zc_j = Model.zc_j(q, d(t), t) // recompute j-th zero--crossing function
13     tz_j = nextEventTime(zc_j) //recompute next j-th zero--crossing time
14   end for
15 }
```

De manera similar, cuando el próximo paso de simulación es debido a la ejecución del handler H_i en el tiempo t, el **Integrador** procede de la siguiente manera:

Algoritmo 3.3: QSS Integrator Module - Event Handler Execution

```
1  Event_Handler_Step(i)
2  {
3    Model.H_i(q(t), d(t), t) //execute i-th event handler
4    for each j such that HD_{i,j} = 1
5      integrateState(x_j, ẋ_j, t) // integrate j-th state up to time t.
6      ẋ_j = Model.f_j(q(t), d(t), t) // recompute j-th state derivative
7      tx_j = Quantizer.nextTime(x_j(t), q_j(t)) //recompute next j-th
           quantized state change time
8    end for
9    for each j such that HZ_{i,j} = 1
10     zc_j = Model.zc_j(q, d(t), t) // recompute j-th zero--crossing function
11     tz_j = nextEventTime(zc_j) //recompute next j-th zero--crossing time
12   end for
13 }
```

3.2.2. Módulo Cuantificador

Como describimos anteriormente, el módulo *Integrador* utiliza la interfaz definida en el módulo **Cuantificador** para obtener las trayectorias de estado cuantificadas $q_i(t)$ como función de las trayectorias de estado $x_i(t)$. El módulo **Cuantificador** es el encargado de calcular los estados cuantificados de acuerdo al método QSS especificado (QSS1, QSS2, QSS3, LIQSS1, LIQSS2, etc) y la tolerancia seleccionada.

Las trayectorias de estado cuantificadas pueden ser caracterizadas por los coeficientes del polinomio de aproximación ($q_{i,k}$) y los instantes de cambio (t_i^q) como puede verse en la Ec. (3.6). De esta forma, la interfaz del módulo **Cuantificador** puede resumirse en las siguientes funciones:

- *Update Quantized State*: Que calcula los coeficientes $q_{i,k}$ del polinomio de aproximación a partir de $x_{i,k}$.
- *Compute Next Time*: Que calcula el próximo tiempo de cambio para la variable cuantificada $q_i(t)$ luego del comienzo de una nueva sección polinomial.
- *Recompute Next Time*: Que calcula el próximo tiempo de cambio para la variable cuantificada $q_j(t)$ luego de un cambio en la derivada $\dot{x}_j(t)$.

Estas tres funciones dependen del método QSS especificado y la tolerancia seleccionada. La tolerancia es caracterizada por dos parámetros: ΔQ_{min} y ΔQ_{rel}, de manera tal que es posible utilizar cuantificación logarítmica. Estos parámetros

pueden ser diferentes en cada variable de estado, y el quantum se calcula de la siguiente manera:

$$\Delta Q_i = \text{máx}(\Delta Q_{i,rel} \cdot |x_{i,0}|, \Delta Q_{i,min})$$

Por ejemplo, en el método QSS1 de primer orden las funciones de interfaz del módulo **Cuantificador** calculan:

- *Update Quantized State*: Asigna el coeficiente del polinomio de aproximación q_i:

$$q_{i,0} = x_{i,0}.$$

- *Compute Next Time*: Calcula el próximo tiempo de cambio de acuerdo a:

$$t_i^{q^+} = t + \frac{\Delta Q_i}{x_{i,1}}$$

- *Recompute Next Time*: Recalcula el tiempo de próximo cambio como el mínimo t que cumpla con:

$$|x_i(t) - q_i(t)| = \Delta Q_i$$

donde $x_i(t)$ y $q_i(t)$ se obtienen de las Ecs. (3.5)–(3.6).

El el método de segundo orden QSS2, las funciones de la interfaz se definen de la siguiente manera:

- *Update Quantized State*: Asigna los coeficientes del polinomio de aproximación q_i:

$$q_{i,0} = x_{i,0}, \quad q_{i,1} = x_{i,1}$$

- *Compute Next Time*: Calcula el próximo tiempo de cambio de acuerdo a:

$$t_i^{q^+} = t + \sqrt{\frac{\Delta Q_i}{|x_{i,2}|}}$$

- *Recompute Next Time*: Recalcula el tiempo de próximo cambio como el mínimo $t^* > t$ que cumpla con:

$$|x_i(t^*) - q_i(t^*)| = \Delta Q_i$$

donde $x_i(t^*)$ y $q_i(t^*)$ se obtienen de las Ecs. (3.5)–(3.6). Para calcular t^*, el **Cuantificador** busca las raíces de dos polinomios de segundo orden.

Las interfaz del **Cuantificador** para QSS3 puede definirse de manera similar.

Para el caso del método LIQSS1, las funciones del **Cuantificador** hacen lo siguiente:

- *Update Quantized State*: Asigna los coeficientes del polinomio de aproximación q_i:

  $$q_{i,0} = x_{i,0} + \delta q_i.$$

  donde

  $$\delta q_i = \begin{cases} \Delta Q_i & \text{if } x_{i,1} > 0 \text{ and } \tilde{f}_i(x_{i,0} + \Delta Q_i) > 0 \\ -\Delta Q_i & \text{if } x_{i,1} < 0 \text{ and } \tilde{f}_i(x_{i,0} - \Delta Q_i) < 0 \\ \tilde{q}_i - x_{i,0} & \text{en otro caso} \end{cases}$$

  donde $\tilde{f}_i(q_i) = a_i \cdot q_i + u_i$ es una aproximación lineal de la derivada de estado $\dot{x}_i$ y $\tilde{q}_i = -u_i/a_i$ es el valor en el cual la aproximación es cero.

- *Compute Next Time*: Calcula el próximo tiempo de cambio de acuerdo a:

  $$t_i^{q^+} = t + \frac{\Delta Q_i}{x_{i,1}}$$

- *Recompute Next Time*: Recalcula el tiempo de próximo cambio como el mínimo t que cumpla con:

  $$|x_i(t) - q_i(t) - \delta q_i| = \Delta Q_i$$

  donde $x_i(t)$ y $q_i(t)$ se obtienen de las Ecs. (3.5)–(3.6).

  También actualiza el parámetro u_i de la aproximación lineal:

  $$u_i = x_{i,1} - a_i \cdot q_{i,0}$$

  En caso de que la función recalcule el tiempo de cambio de la variable q_i debido a un cambio en q_i, se debe actualizar primero el parámetro a_i de la aproximación lineal:

  $$a_i = \frac{x_{i,1} - \text{old}(x_{i,1})}{q_{i,0} - \text{old}(q_{i,0})}$$

  donde $\text{old}(x_{i,1})$ es el valor previo de la derivada de estado $x_{i,1}$ y $\text{old}(q_{i,0})$ es el valor previo del estado cuantificado$q_{i,0}$.

Los métodos de cuantificación LIQSS2 y LIQSS3 combinan esta implementación con la implementación de los métodos QSS2 y QSS3.

3.2.3. Módulo Modelo

El módulo **Integrador** calcula la Ec. (3.2), a partir de $\mathbf{q}(t)$ utilizando el módulo **Cuantificador** y la evaluación de las derivadas de estado $\dot{x}_i = f_i(\mathbf{q}, \mathbf{d}, t)$ y las funciones de cruce por cero $ZC_i(\mathbf{q}, \mathbf{d}, t)$ definidas en la instancia del módulo **Modelo** generada. Además de evaluar estas funciones, el **Modelo** provee información la estructural del sistema mencionada anteriormente.

Las principales funciones del módulo **Modelo** permiten:

- Evaluar una **derivada de estado escalar** $\dot{x}_i = f_i(\mathbf{q}, \mathbf{d}, t)$.
- Evaluar una **función de cruce por cero** $ZC_i(\mathbf{q}, \mathbf{d}, t)$.
- Ejecutar un **handler** $H_i(\mathbf{q}, \mathbf{d}, t)$.
- Evaluar las cuatro matrices de incidencia que expresan influencias directas desde variables de estado y handlers a derivadas de estado y funciones de cruce por cero.

Adicionalmente, por razones de eficiencia, en el módulo **Modelo** existen rutinas que permiten:

- Evaluar **todas las derivadas de estado** que dependen una variable de estado x_j. De esta manera, el **Integrador** puede reevaluar todas las derivadas de estado que cambian luego de un paso en una sola llamada a función.
- Evaluar las **derivadas de orden superior** de las variables de estado y las funciones de cruce por cero. Las mismas son requeridas por los métodos de QSS de orden superior. En caso de no estar disponibles en el modelo, son calculadas numéricamente lo cual implica más llamadas a función y cálculos adicionales.

Como veremos en las siguientes secciones, las matrices de incidencia y las rutinas que calculan las derivadas de orden superior como así también las funciones de la interfaz para este módulo son generadas por una interfaz de modelado que permite procesar los modelos descriptos utilizando un lenguaje de modelado estandard por el usuario final.

Una vez generada la instancia del **Modelo**, la misma es compilada en conjunto con el **Integrador** y el **Cuantificador** para obtener un modelo ejecutable. Cabe mencionar que todos los módulos descriptos en esta sección y las anteriores fueron implementados en el lenguaje de programación C.

3.2.4. Análisis de Eficiencia

En la Sección 2.2.5 analizamos las implementaciones de los métodos QSS basadas en el formalismo DEVS mediante un ejemplo de simulación sencillo representado por la Ec. (2.25). Es ese caso pudimos ver que para procesar un cambio en una variable cuantificada $q_2(t)$, el motor de simulación DEVS necesitó varios pasos de simulación.

Ahora vamos a analizar el mismo cambio en la variable cuantificada $q_2(t)$ en el mismo modelo con el motor de simulación autónoma de QSS:

1. El **Integrador** avanza el tiempo de simulación al tiempo de cambio de q_2.
2. El **Integrador** obtiene del **Cuantificador** el nuevo valor de q_2 (1 llamada a función).
3. El **Integrador** obtiene del **Modelo** los nuevos valores de las derivadas $\dot{x}_1$ y $\dot{x}_2$ (1 llamada a función).
4. El **Integrador** obtiene del **Cuantificador** los nuevos tiempos de cambio para las variables cuantificadas q_1 y q_2 (2 llamadas a función).
5. El **Integrador** busca cuál de las 2 variables ejecuta el próximo cambio.

Durante este proceso, el motor de simulación ejecuta 4 llamadas a función solamente y una búsqueda del mínimo entre 2 valores.

En comparación con las 17 llamadas a función y 3 búsquedas del mínimo entre 5 valores realizadas por el mecanismo de simulación DEVS, podemos esperar que la implementación autónoma de el motor de simulación QSS sea más eficiente que las implementaciones basadas en DEVS.

El análisis realizado a partir de este ejemplo pueden ser extendido fácilmente a sistemas generales con conclusiones similares.

3.3. Especificación de Modelos

Comparado con los motores de simulación para métodos de integración numérica clásicos, el motor de simulación QSS autónomo tiene la desventaja de que se debe proporcionar información estructural adicional sobre el modelo para poder evaluar las derivadas de estado $\dot{x}_i = f_i(\mathbf{q}, \mathbf{d}, t)$ y las funciones de cruce por cero $ZC_i(\mathbf{q}, \mathbf{d}, t)$ solamente cuando es necesario. Esta información estructural es representada como matrices de incidencia.

Para superar esta dificultad, desarrollamos un entorno de modelado que permite al usuario describir modelos en una manera estandard, utilizando un subconjunto del lenguaje de modelado Modelica llamado μ–Modelica, y que a partir de esta descripción extrae automáticamente toda la información estructural que necesita el motor de simulación y genera el código C correspondiente.

La transformación del modelo original descripto en μ–Modelica al código C final es realizada en diferentes etapas por los módulo descriptos a continuación:

1. El módulo **Parser μ-Modelica**, transforma el modelo descripto en μ-Modelica para obtener una nueva representación estructurada del modelo.

2. El módulo de **Representación Intermedia (RI)**, que se encarga de obtener información con respecto a todas las variables de estado, algebraicas y discretas definidas en las ecuaciones y los eventos del modelo.

3. El módulo **Generador de Modelos**, que construye las matrices de incidencia del sistema y genera la instancia del módulo **Modelo** requerida por el motor.

Para completar el entorno de modelado, desarrollamos un cuarto módulo que implementa una interfaz gráfica de usuario (GUI) simple que permite crear y editar modelos además de interactuar con el entorno de simulación desarrollado. El esquema básico de interacción entre los diferentes módulo mencionados se muestra en la Figura 3.2.

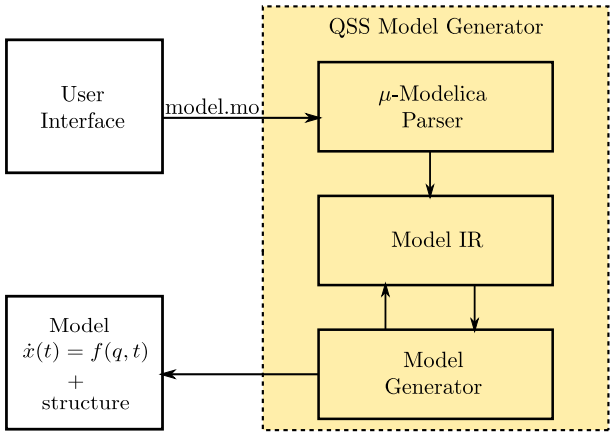


Figura 3.2: Entorno de Modelado - Esquema de Interacción Básico.

3.3.1. Parser μ–Modelica

El módulo **Parser μ-Modelica** transforma el modelo descripto utilizando el lenguaje de modelado de alto nivel en una representación estructurada, AST (*Abstract Syntax Tree*), que es utilizada en las siguientes etapas.

Con este fin, definimos un lenguaje llamado μ-Modelica que consiste en un subconjunto del lenguaje de modelado Modelica. μ-Modelica fue concebido de manera tal que contiene solo las sentencias y estructuras de datos necesarias para definir modelos híbridos representados por las Ecs. (3.2)-(3.4).

Por ejemplo, el siguiente código representa una pelota rebotando en el suelo en μ-Modelica:

```
model bball
  Real y(start = 10),vy(start = 0), F;
  parameter Real m = 1, b = 30, g = 9.8, k = 1e6;
  discrete Real contact(start = 0);
equation
  F = k*y+b*vy;
  der(y) = vy;
  der(vy) = -g - (contact * F)/m;
algorithm
  when y < 0 then
    contact := 1;
```

```
  elsewhen y > 0 then
    contact := 0;
  end when;
end bball;
```

El motor de simulación QSS fue concebido para soportar la simulación de modelos de gran escalar. Por lo tanto, se permite la definición de arreglos y ciclos `for` que son procesados de manera eficiente. El siguiente ejemplo muestra esta característica del lenguaje en un modelo de Advección–Reacción:

```
model advection
  parameter Real alpha=0.5,mu=1000;
  constant Integer N = 500, T = 0.3*N;
  Real u[N];
initial algorithm
  for i in 1:T loop
    u[i]:=1;
  end for;
equation
  der(u[1])=(-u[1]+1)*N-mu*u[1]*(u[1]-alpha)*(u[1]-1);
  for i in 2:N loop
    der(u[i])=(-u[i]+u[i-1])*N-mu*u[i]*(u[i]-alpha)*(u[i]-1);
  end for;
end advection;
```

El lenguaje μ–Modelica tiene las siguientes restricciones con respecto a Modelica:

- Los modelos deben estar aplanados, i.e. no se permite la definición de clases.
- Todas las variables definidas pertenecen a los tipos predefinidos `Real` y sólo pueden pertenecer a tres categorías: **estados continuos**, **estados discretos** y **variables algebraicas**. Por ejemplo, en el modelo de la pelota rebotando en el suelo, y y vy son estados continuos, F es una variable algebraica y *contact* es una variable discreta.

- Los parámetros definidos también son del tipo `Real`. En el modelo de Advección–Reacción, *alpha* y *mu* son parámetros.
- Se permite la definición de arreglos. Los índices en los ciclos `for` son restringidos a expresiones de la siguiente forma:

$$\alpha \cdot i + \beta \tag{3.7}$$

  donde α y β son expresiones enteras y i es el índice de iteración.
- La sección `equation` está compuesta por:
  - Definiciones de **derivadas de estado**: $\texttt{der}(x) = f(\mathbf{x}(t), \mathbf{d}, \mathbf{a}(t), t)$; en forma de EDO explícita.
  - Definiciones de **variables algebraicas**:

$$(a_1, \cdots, a_n) = \mathbf{g}(\mathbf{x}(t), \mathbf{d}, \mathbf{a}(t), t); \tag{3.8}$$

    con la restricción de que cada variable algebraica puede depender solamente de estados y variables algebraicas definidas previamente.
- Las discontinuidades son expresadas solamente mediante las sentencias `when` y `elsewhen` dentro de la sección`algorithm`. Las condiciones de cruce en ambas sentencias están restringidas a relaciones $(<, \leq, >, \geq)$ y, en las definiciones de los handlers, se permiten solamente asignaciones de variables discretas y `reinit` de estados continuos.

La especificación completa del lenguaje μ–Modelica se puede encontrar en [17].

Dado un modelo definido en este lenguaje, el **Parser μ-Modelica** produce la primera transformación, generando el AST correspondiente al modelo.

3.3.2. Representación Intermedia (RI)

La siguiente transformación es llevada a cabo por el módulo de **Representación Intermedia (RI)**. El objetivo de esta transformación es extraer información estructural del AST generado en el paso anterior.

Adicionalmente, en presencia de discontinuidades, esta etapa está a cargo de construir las funciones de cruce por cero a partir de las condiciones dadas. Para esto, una condición de cruce por cero

$$f_1(\mathbf{x}, \mathbf{d}, \mathbf{a}, t) < f_2(\mathbf{x}, \mathbf{d}, \mathbf{a}, t)$$

es transformada en la función de cruce por cero

$$zc(\mathbf{x}, \mathbf{d}, \mathbf{a}, t) = f_1(\mathbf{x}, \mathbf{d}, \mathbf{a}, t) - f_2(\mathbf{x}, \mathbf{d}, \mathbf{a}, t)$$

El módulo **RI** analiza todas las ecuaciones y sentencias definidas en el modelo para obtener las listas de variables involucradas en cada expresión.

Para esto, dada una expresión de la forma $e = f(\mathbf{x}(t), \mathbf{d}(t), \mathbf{a}(t), t)$, el módulo **RI** debe construir listas que contengan:

- Las variables de estado x_i involucradas en el cálculo de e.
- Las variables discretas d_i involucradas en el cálculo de e.
- Las variables algebraicas a_i involucradas en el cálculo de e.

Aquí, decimos que una variable v está involucrada en el cálculo de una expresión e si:

- v aparece en e, o
- v está involucrada en el cálculo de una variable algebraica que a su vez esta involucrada en el cálculo de la expresión e.

Esta información es utilizada para construir las matrices de incidencia SD (de estados a derivadas de estado), SZ (de estados a funciones de cruce por cero), HZ (de handlers a funciones de cruce por cero) y HD (de handlers a derivadas de estado).

Por ejemplo, dadas las siguientes ecuaciones

```
a1=f1(x2,x3);
der(x1)=f2(a1,x4);
```

el módulo **RI** encuentra que las variables `x2`, `x3` y `x4` están involucradas en el cálculo de `der(x1)`. Luego esta información es utilizada para construir la matriz de incidencia SD:

- Se incrementa el número de derivadas influencias por `x2`, $NSD_2 = NSD_2 + 1$.
- Se agrega la entrada $SD_{2,NSD_2} = 1$ en la matriz SD, indicando que `x2` está involucrada en el cáculo de`der(x1)`.

- TSe incrementa el número de derivadas influencias por `x3`, $NSD_3 = NSD_3 + 1$.
- Se agrega la entrada $SD_{3,NSD_3} = 1$ en la matriz SD, indicando que `x3` está involucrada en el cálculo de `der(x1)`.
- Se procede de la misma manera para la variable `x4`.

El módulo también encuentra que la variable algebraica `a1` está involucrada en el cálculo de `der(x1)`. Esta información es utilizada en las próximas etapas por el módulo **Generador de Modelos**.

Supongamos ahora que tenemos el siguiente código μ-Modelica:

```
equation
 der(x1)=d1;
algorithm
 when x1>2 then
    d1=-1;
 end when;
```

En este caso, el módulo **RI** debe construir primero las funciones de cruce por cero $ZC_1 = x_1 - 2$. Luego encuentra que el handler H_1 (correspondiente a la función de cruce por cero ZC_1) influencia el cálculo de `der(x1)` (por medio de la variable discreta `d1`) y que la variable de estado `x1` influencia la función de cruce por cero ZC_1.

Con esta información, las matrices de incidencia correspondientes se construyen de la siguiente manera:

- Se incremente el número de derivadas influenciadas por H_1, $NHD_1 = NHD_1 + 1$.
- Se agrega la entrada $HD_{1,NHD_1} = 1$ en la matrix HD, indicando que la ejecución del handler H_1 modifica una variable involucrada en el cálculo de `der(x1)`.
- Se incrementa el número de funciones de cruce por cero influenciadas por `x1`, $NSZ_1 = NSZ_1 + 1$.

- Se agrega la entrada $SZ_{1,NSZ_1} = 1$ a la matriz SZ, indicando que la variable `x1` está involucrada en el cálculo de la función de cruce por cero ZC_1 .

Para poder procesar de manera eficiente modelos de gran escala, las expresiones definidas dentro de ciclos `for` son tratadas de manera genérica sin necesidad de expandir el ciclo. En este caso, la lista de variables asociada a la expresión incluyen información adicional sobre el rango en el cuál están definidas.

Por ejemplo, dado el siguiente código

```
for i in 2:100 loop
   der(x[i])=x[i-1]+x[i];
 end for;
```

el módulo encuentra que

- la variable `x[i]` influencia `der(x[i])` en el rango $2 \leq i \leq 100$.
- la variable `x[i]` influencia `der(x[i+1])` en el rango $1 \leq i \leq 99$.

Esta información es utilizada para construir la matriz SD de la siguiente manera:

- Para $i \in [2, 100]$ se incrementa $NSD_i = NSD_i + 1$ y se asigna $SD_{i,NSD_i} = i$.
- Para $i \in [1, 99]$ se incrementa $NSD_i = NSD_i + 1$ y se asigna $SD_{i,NSD_i} = i + 1$.

Esta manera de tratar los ciclos `for` permite reducir de manera significativa el tamaño del código C generado para las matrices de incidencia, lo que reduce a su vez los tiempo de compilación en modelos de gran escala.

3.3.3. Generador de Código

Este módulo está a cargo de generar el código C correspondiente a la instancia del módulo **Modelo** que necesita el motor de simulación autónomo QSS. Basado en la información generada por el módulo **RI**, el módulo **Generador de Código** genera código C para las siguientes funciones:

- Código de inicialización, que ejecuta las siguientes acciones:
  - Inicialización de las variables y parámetros del modelo.

  - Inicialización y cálculo de las matrices estructurales.
  - Inicialización de los eventos del modelo.
  - Obtener el tiempo de próximo cambio inicial para cada variable de estado y eventos definidos en el modelo.

- Código para calcular las derivadas de estado, que involucra
  - Calcular derivadas de estado individuales $\dot{x}_i = f_i(\mathbf{x}(t), \mathbf{d}(t), \mathbf{a}(t), t)$
  - Calcular todas las derivadas de estados que dependen de una variable de estado dada en una sola llamada a función.
  - En ambos casos, también se puede incluir el código para el cálculo de las derivadas de orden superior ($\ddot{x}_i$, $\dddot{x}_i$). Las expresiones correspondientes se obtienen utilizando la librería *GiNaC*.
- Código para la evaluación de las funciones de cruce por cero zc_i (que también puede incluir código para el cálculo de las derivadas de orden superior de zc_i).
- Código para los handlers de los eventos definidos.

Por ejemplo, el siguiente código C muestra parte de la instancia del módulo **Modelo** generada por el módulo **Generador de Código** para el modelo de la pelota rebotando en el suelo presentado anteriormente.

```
void
MOD_definition(int i, double *x, double *d, double *alg, double t, double *dx)
{
 switch(i)
 {
  case 0:
   dx[1] = x[4];
   return;
  case 1:
   alg[0] = k*x[0]+b*x[4];
   dx[1] = -g-(d[0]*(alg[0]))/m;
   return;
 }
}

void
MOD_zeroCrossing(int i, double *x, double *d, double *alg, double t, double *zc)
{
 switch(i)
 {
  case 0:
   zc[0] = x[0]-(0);
   return;
 }
}

void
MOD_handlerPos(int i, double *x, double *d, double *alg, double t)
{
 switch(i)
```

```
 {
  case 0:
   d[0] = 0;
   return;
 }
}

void
MOD_handlerNeg(int i, double *x, double *d, double *alg, double t)
{
 switch(i)
 {
  case 0:
   d[0] = 1;
   return;
 }
}

void
QSS_initializeDataStructs(QSS_simulator simulator)
{
 ...
 //incidence matrix from states to derivatives
 modelData->SD[1][states[1]++] = 0;
 modelData->SD[1][states[1]++] = 1;
 modelData->SD[0][states[0]++] = 1;
 //incidence matrix from states to zero--crossings
 modelData->SZ[0][events[0]++] = 0;
...
 //incidence matrix from handlers to derivatives
 modelData->HD[0][events[0]++] = 1;
}...
```

3.4. Interfaz de Usuario

El entorno de modelado fue complementado con el desarrollo de una interfaz gráfica de usuario (GUI) que unifica y simplifica el uso del los diferentes componentes del motor de simulación.

La GUI tiene las siguientes características:

- Un editor de texto, donde se pueden definir y modificar modelos μ-Modelica.
- Permite invocar las herramientas correspondientes para compilar y correr las simulaciones.
- Permite generar información de depuración de errores en la compilación del modelo como así también de información detallada de la simulación.
- Permite invocar la herramienta de generación de gráficos *GNUPlot* para ver las trayectorias de salida.
- Muestra estadísticas de la simulación (número de pasos de simulación, tiempo de simulación, etc.)

La Figura 3.3 muestra un ejemplo de uso de la GUI con el modelo de Advección–Reacción presentado anteriormente. En el panel del lado izquierdo podemos ver la interfaz de *GNUPlot* que permite mostrar los resultados de la simulación.

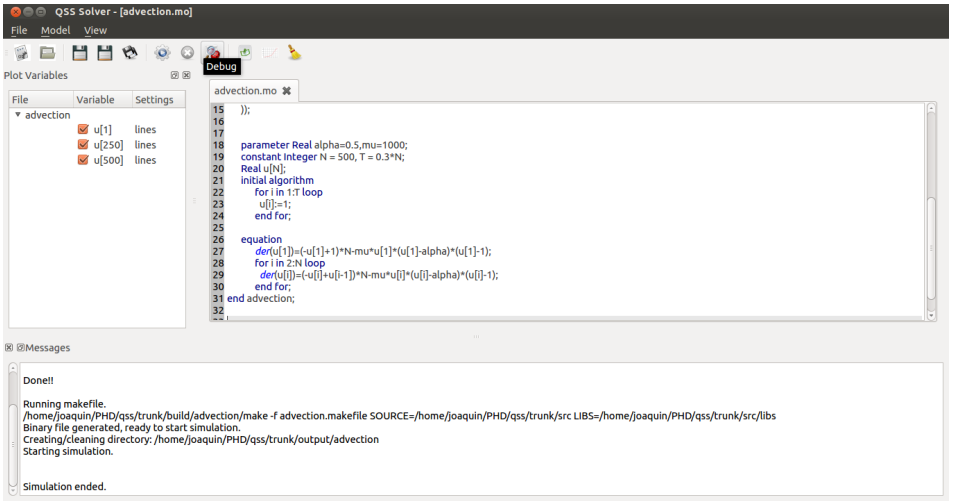


Figura 3.3: Interfaz Gráfica de Usuario (GUI)

3.5. Resultados y Comparaciones

Esta sección estudia el rendimiento del nuevo motor de simulación evaluando cuatro ejemplos, con este fin, comparamos los resultados obtenidos con el motor de simulación con los resultados utilizando los mismos algoritmos en PowerDEVS y también comparamos los resultados con la herramienta OpenModelica utilizando métodos de integración clásicos como DASSL y Runge–Kutta.

Los ejemplos analizados cubren diferentes características de sistemas en los que los métdos QSS son eficientes. El primer ejemplo simula el consumo de energía de un conjunto de aires acondicionados, este es un modelo que presenta discontinuidades frecuentes. El segundo ejemplo corresponde a un sistema ralo que además es stiff obtenido al discretizar una ecuación de Advección–Reacción en una dimensión aplicando el método de líneas. El tercer ejemplo es un modelo stiff con discontinuidades frecuentes correspondiente a un convertidor de energía de potencia y el último ejemplo es un modelo stiff y discontinuo que representa una cadena de inversores lógicos.

En todos los casos, utilizamos métodos QSS configurados con diferentes tolerancias. En cada modelo, los errores reportados corresponden al error medio cuadrado, calculado contra una trayectoria de referencia obtenida utilizando DASSL con una tolerancia de 10^{-10}.

Todas las simulaciones fueron corridas en la misma computadora con un procesador i7 y un sistema operativo Linux. Se obtuvieron también resultados utilizando Dymola para el método DASSL, pero no son reportados en la mayoría de los ejemplos dado que no difieren con los resultados obtenidos con OpenModelica de manera significativa y en este caso se requiere utilizar otro sistema operativo (Windows XP). Solamente son analizados en los casos en que no pudimos correr las simulaciones correspondientes en OpenModelica.

3.5.1. Conjunto de Aires Acondicionados

Este primer ejemplo, presentado en [51], es un modelo propuesto para estudiar el consumo de energía de un conjunto de aires acondicionados (AAs). Cada AA mantiene la temperatura de una habitación cerca de la temperatura de referencia $\theta_{\text{ref}}(t)$, enciendo o apagando el sistema de refrigeración. La evolución de la temperatura de la i–ésima habitación es descripta por la ecuación diferencial:

$$\frac{d\theta_i(t)}{dt} = -\frac{1}{C_i \cdot R_i}[\theta_i(t) - \theta_a + R_i \cdot P_i \cdot m_i(t)], \tag{3.9}$$

donde R_i y C_i son la resistencia y la capacidad térmica respectivamente. P_i representa el consumo de energía de una unidad de AA cuando está encendido y θ_a es la temperatura externa.

El término $m_i(t)$ representa el control de encendido–apagado del AA, que sigue la ley:

$$m_i(t) = \begin{cases} 1 & \text{if } m_i(t^-) = 0 \text{ and } \theta_i(t) > \theta_{\text{ref}}(t) + 0{,}5 \\ 0 & \text{if } m_i(t^-) = 1 \text{ and } \theta_i(t) < \theta_{\text{ref}}(t) - 0{,}5 \\ m_i(t^-) & \text{otherwise} \end{cases} \tag{3.10}$$

Simulamos este sistema para $N = 100$ habitaciones, considerando los siguiente cambios en la temperatura de referencia:

$$\theta_{\text{ref}}(t) = \begin{cases} 20 & \text{if } t < 1000 \text{ or } t > 2000 \\ 20{,}5 & \text{otherwise} \end{cases}$$

Utilizamos los métodos QSS2 y QSS3 con PowerDEVS y con el nuevo motor de simulación QSS. También simulamos el sistema utilizando DASSL y Runge–

Kutta en la herramienta OpenModelica. La Figura 3.4 muestra el consumo de energía total del sistema y la Tabla 3.1 reporta los tiempos de simulación y errores obtenidos.

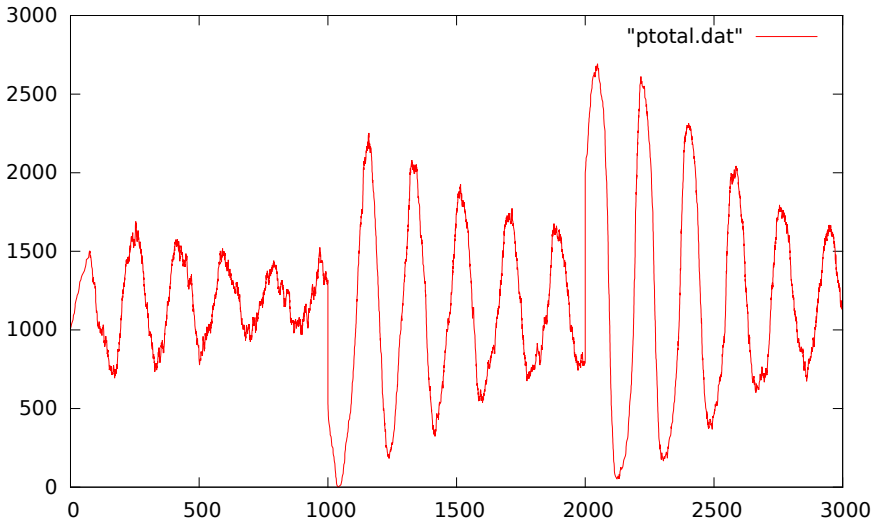


Figura 3.4: Consumo Total de Energía.

Tabla 3.1: Conjunto de Aires Acondicionados - Resultados

	Método	Tolerancia	Tiempo CPU (ms)	Error
QSS Solver	**QSS2**	10^{-3}	5	4.16E-03
	QSS2	10^{-7}	123	7.01E-07
	QSS3	10^{-3}	11	2.84E-03
	QSS3	10^{-7}	28	4.29E-07
	LIQSS3	10^{-3}	12	2.48E-02
	LIQSS3	10^{-7}	30	2.12E-06
OpenModelica	**RUNGE KUTTA**	10^{-3}	1260	1.13E-02
	RUNGE KUTTA	10^{-7}	1290	1.40E-02
	DASSL	10^{-3}	25056	2.56E-02
	DASSL	10^{-7}	28280	1.42E-02
PowerDEVS	**QSS2**	10^{-3}	50	4.64E-03
	QSS2	10^{-7}	1180	1.04E-06
	QSS3	10^{-3}	60	1.26E-02
	QSS3	10^{-7}	140	3.88E-04

Los resultados muestran que el nuevo motor de simulación QSS es entre 5 y 10 veces más rápido que PowerDEVS utilizando la misma configuración para las

tolerancias. Sin embargo, los errores obtenidos por el motor de simulación QSS son mejores que los obtenidos con PowerDEVS, particularmente para tolerancias bajas. Esto es debido al hecho de que el motor de simulación QSS tiene en cuenta la configuración global de las tolerancias en las rutinas de detección de eventos.

Los resultados del motor de simulación QSS son también un orden de magnitud más rápidos que Runge–Kutta y más de tres órdenes de magnitud comparado con el algoritmo DASSL que es apto para sistemas stiff. De todas formas, este modelo no es stiff, utilizar DASSL en este ejemplo no es necesario.

Vemos también que para tolerancias bajas, el método de segundo orden QSS2 es más rápido que el método de tercer orden QSS3. Para tolerancias más altas, sin embargo, QSS3 es más eficiente.

Con respecto a los algritmos Runge–Kutta y DASSL, el uso de diferentes tolerancias no afecta demasiado los tiempos de simulación. En este sistema las discontinuidades son tan frecuentes que el tamaño del paso de simulación no puede ser incrementado aún cuando la tolerancia es baja. Por esta razón, el error no cambia demasiado con las diferentes tolerancias, dado que depende más de la precisión con la que estos algoritmos detectan las discontinuidades.

En este caso, también simulamos el sistema con el método apto para sistemas stiff LIQSS3, donde observamos un rendimiento idéntico al método no stiff QSS3. Esto indica que los métodos LIQSS pueden ser utilizados como algoritmos por defecto sin incurrir en un costo computacional alto. Cabe mencionar que en los algoritmos clásicos esto no es posible, como se puede ver en los tiempos de simulación obtenidos para los métodos Runge–Kutta y DASSL.

3.5.2. Advección–Reacción

Este ejemplo representa la discretización mediante el Método de Líneas de un modelo de Advección–Reacción, esta discretización lleva al siguiente conjunto de EDOs:

$$\dot{u_i} = (-u_i + u_{i-1}) \cdot N - \mu \cdot u_i \cdot (u_i - \alpha) \cdot (u_i - 1);$$

para $i = 1, \ldots, N$. utilizamos los parámetros $u_0 = 1$, $\alpha = 0{,}5$ y $\mu = 1000$ con condiciones iniciales $u_i(0) = 1$ for $i < 0{,}3 \cdot N$ y $u_i(0) = 0$ en otro caso.

Este es un sistema ralo y stiff debido a la presencia de el término de reacción $\mu \cdot u_i \cdot (u_i - \alpha) \cdot (u_i - 1)$.

Simulamos el modelo para $N = 500$, obteniendo las trayectorias de salida que se muestran en la Figura 3.5 y los tiempos de CPU y errores reportados en la Tabla 3.2. Teniendo en cuanta que es un sistema stiff, solamente fue simulado utilizando los métodos aptos para este tipo de problemas (LIQSS y DASSL).

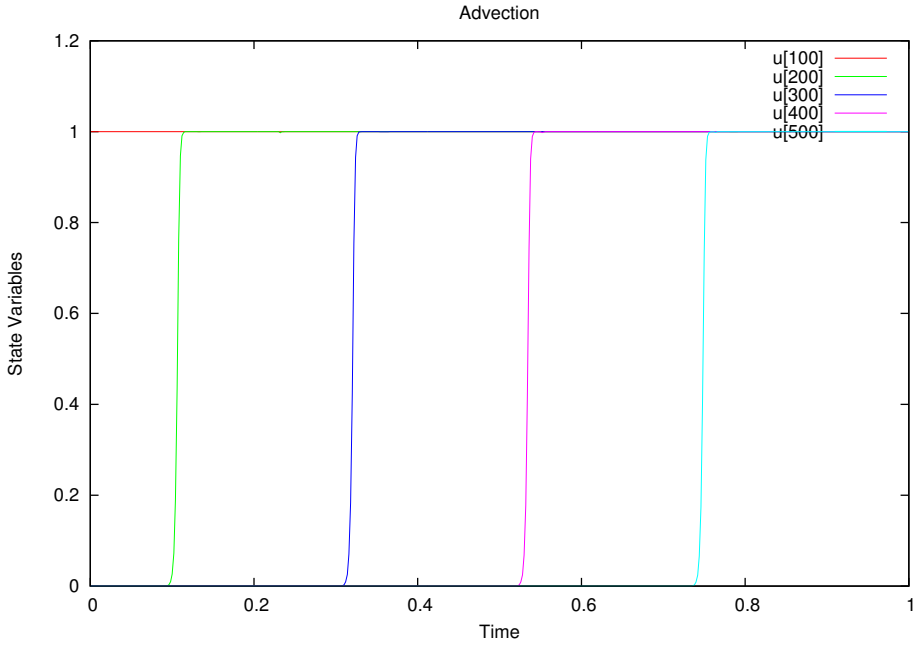


Figura 3.5: Advección–Reacción - Trayectorias de Salida.

Tabla 3.2: Advección–Difusión–Reacción - Resultados

	Método	Tolerancia	Tiempo CPU (ms)	Error
QSS Solver	**LIQSS2**	10^{-3}	8	1.59E-03
	LIQSS2	10^{-7}	600	2.60E-11
	LIQSS3	10^{-3}	64	1.04E-03
	LIQSS3	10^{-7}	217	4.21E-12
OpenModelica	**DASSL**	10^{-3}	789	4.01E-03
	DASSL	10^{-7}	3260	3.45E-11
PowerDEVS	**LIQSS2**	10^{-3}	250	3.40E-03
	LIQSS2	10^{-7}	17000	7.30E-07
	LIQSS3	10^{-3}	950	8.32E-03
	LIQSS3	10^{-7}	1870	6.87E-07

Los resultados muestran una aceleración de hasta un orden de magnitud con respecto a PowerDEVS y de hasta dos ordenes de magnitud con respecto a DASSL.

Con respecto a los errores, son similares en todos los métodos para las misma configuración de las tolerancias, excepto PowerDEVS que para tolerancias bajas no experimenta la reducción en el error que si muestran las otras implementaciones.

Como ocurrió con los métodos QSS2 y QSS3 en el ejemplo previo, el método de segundo orden LIQSS2 es más eficiente para tolerancias bajas y a medida que se aumentan las tolerancias el método LIQSS3 se vuelve más eficiente.

3.5.3. Convertidor Buck

Consideramos en este caso un convertidor Buck intercalado con 4 ramas, como el que se muestra en la Figura 3.6, con los parámetros $C = 10^{-4}$ para el capacitor, $L = 10^{-4}$ para la inductancia, $R = 10$ para la resistencia, y $V_{cc} = 24$ para el voltaje de entrada. Consideramos también que el switch tienen un período de $T = 10^{-4}$, un duty cycle de $DC = 0{,}5/4$, y asumimos que el switch y el diodo tienen una resistencia de $R_{\text{on}} = 10^{-5}$ cuando están en el estado *encendido* y $R_{\text{off}} = 10^{5}$ en el estado *apagado*.

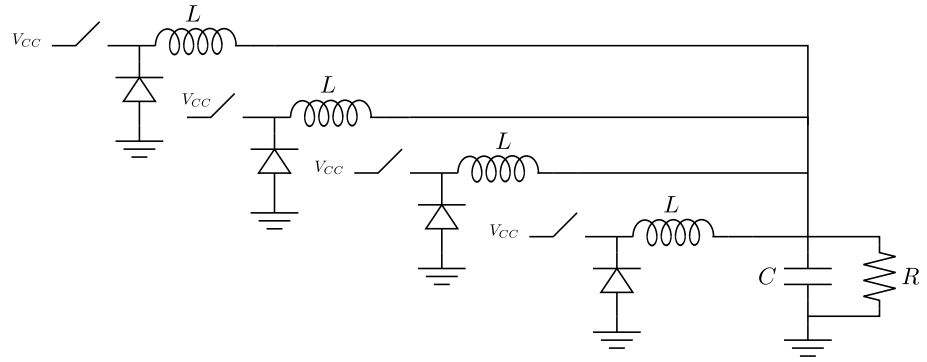


Figura 3.6: Convertidor Buck

Este es un modelo stiff con discontinuidades frecuentes. Por este hecho, solamente fue simulado con los métodos LIQSS y DASSL. La Figura 3.7 muestra las trayectorias de salida para el modelo.

Los resultados reportados en la Tabla 3.3, muestran relaciones similares entre el nuevo motor de simulación QSS y PowerDEVS a las del ejemplo previo

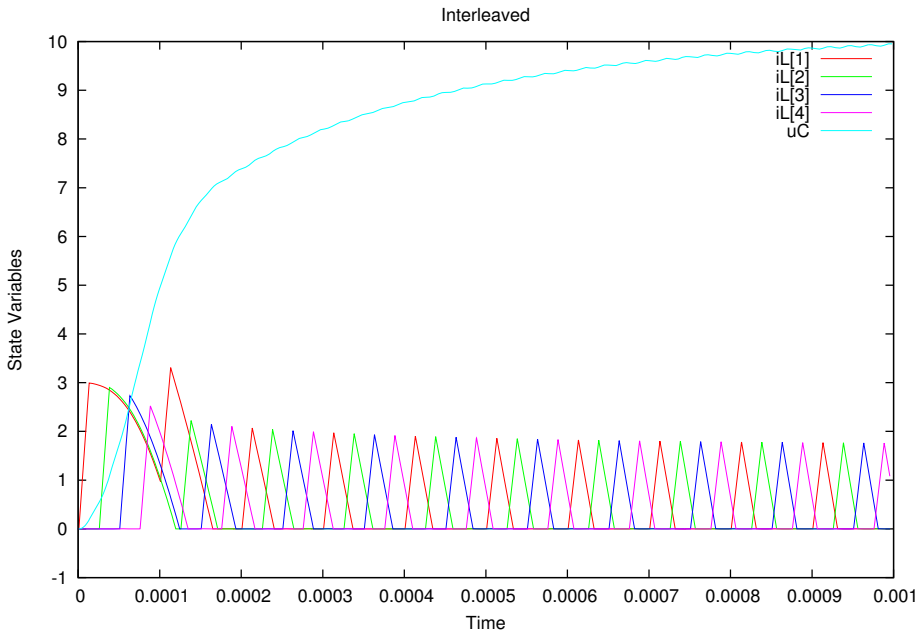


Figura 3.7: Convertidor Buck - Trayectorias de Salida.

(i.e., el nuevo motor de simulación es 10 veces más rápido que PowerDEVS). Las comparaciones con DASSL también muestran una aceleración similar.

Tabla 3.3: Convertidor Buck - Resultados.

	Método	**Tolerancia**	**Tiempo CPU (ms)**	**Error**
QSS Solver	**LIQSS2**	10^{-3}	7	1.50E-04
	LIQSS2	10^{-7}	186	7.13E-10
	LIQSS3	10^{-3}	25	1.67E-04
	LIQSS3	10^{-7}	59	7.29E-10
OpenModelica	**DASSL**	10^{-3}	157	3.16E-04
	DASSL	10^{-7}	264	6.76E-10
PowerDEVS	**LIQSS2**	10^{-3}	300	1.55E-06
	LIQSS2	10^{-7}	10200	5.13E-07
	LIQSS3	10^{-3}	780	1.72E-04
	LIQSS3	10^{-7}	1640	5.12E-07

3.5.4. Cadena de Inversores Lógicos

El último modelo, presentado en [57], representa una cadena de m inversores lógicos

$$\dot{\omega}_j(t) = U_{op} - \omega_j(t) - \Upsilon g\left(\omega_{j-1}(t), \omega_j(t)\right) \tag{3.11}$$

para $j = 1, \ldots, m$ donde

$$g(u, v) = \left(\text{máx}(u - U_{th}, 0)\right)^2 - \left(\text{máx}\left(u - v - U_{th}, 0\right)\right)^2 \tag{3.12}$$

En este ejemplo, utilizamos los parámetros y condiciones iniciales dadas es [57]: $\Upsilon = 100$, $U_{th} = 1$ y $U_{op} = 5$, $\omega_j(0) = 6{,}247 \cdot 10^{-3}$ para los valores impares of j y $\omega_j = 5$ para los pares j. La entrada u_0 es una señal trapezoidal que cambia de 0 a 5 desde el tiempo 5 al tiempo 10 y luego se mantiene en ese nivel, volviendo a 0 desde el tiempo 15 al tiempo 17.

Consideramos un sistema de $m = 100$ inversores, por lo que en este caso tenemos un conjunto de 100 ecuaciones diferenciales con 200 discontinuidades debido a las funciones 'máx' en la Ec. (3.12).

La Figura 3.8 muestra algunas trayectorias de salida obtenidas utilizando LIQSS2 para este sistema.

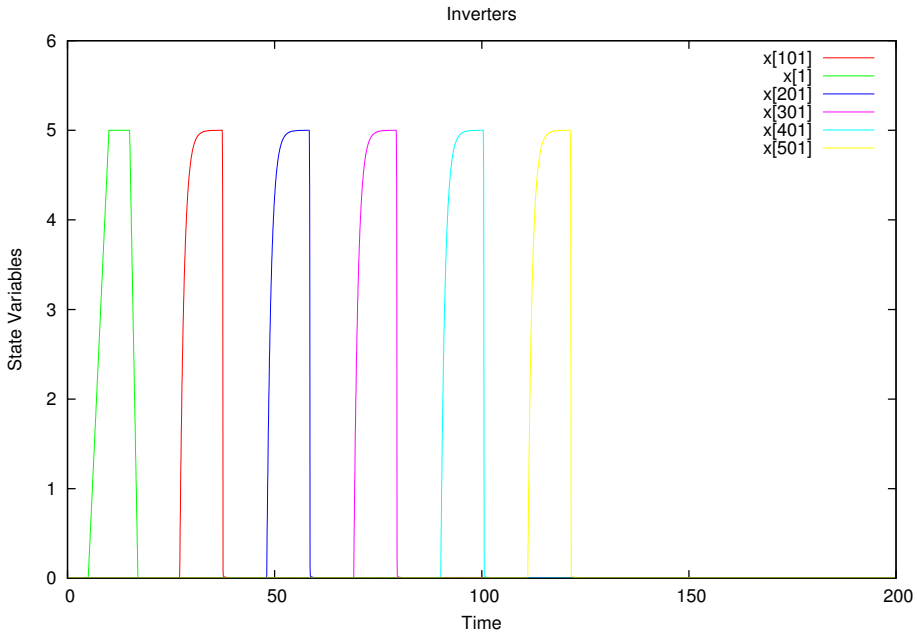


Figura 3.8: Cadena de Inversores Lógicos - Trayectorias de Salida.

La Tabla 3.4 reporta los tiempos de simulación y errores para diferentes métodos y tolerancias.

Tabla 3.4: Cadena de Inversores Lógicos - Resultados.

	Método	Tolerancia	Tiempo CPU (ms)	Error
QSS Solver	**LIQSS2**	10^{-3}	28	3.90E-03
	LIQSS2	10^{-7}	996	6.38E-09
	LIQSS3	10^{-3}	44	1.30E-06
	LIQSS3	10^{-7}	210	4.16E-11
OpenModelica	**DASSL**	10^{-3}	4405	8.43E-03
	DASSL	10^{-7}	8734	9.01E-08
PowerDEVS	**LIQSS2**	10^{-3}	300	2.66E-05
	LIQSS2	10^{-7}	10200	3.98E-06
	LIQSS3	10^{-3}	780	6.06E-02
	LIQSS3	10^{-7}	1640	3.74E-06

Al igual que en los ejemplos anteriores, el motor de simulación autónomo para métodos QSS es más rápido que PowerDEVS mostrando también un aceleración considerablemente mayor a DASSL.

En este último ejemplo, dejamos la tolerancia fija en 10^{-3} y realizamos simulaciones variando el número de inversores, utilizando LIQSS2 para el motor QSS y PowerDEVS, y DASSL en OpenModelica y Dymola [1]. La Figura 3.9 muestra los tiempos de CPU obtenidos para este experimento.

Podemos ver que los tiempos de simulación crecen linealmente con respecto al número de inversores en los métodos LIQSS y que el motor de simulación QSS mantiene una ventaja constante de más de un orden de magnitud con respecto a PowerDEVS. Sin embargo, para el método DASSL los tiempos crecen casi de manera cúbica. En consecuencia, para mil inversores, LIQSS2 toma menos de 200 milisegundos contra los 7000 segundos que necesita DASSL.

Para simular 500 inversores, LIQSS2 tarda aproximadamente 110 milisegundos. En este caso, el uso de algoritmos especializados reportados en [57] tardan aproximadamente 6 segundos. Por lo tanto, la motor de simulación QSS supera en más de 50 veces a estos algoritmos especialmente diseñados para este tipo de problemas.

[1] A partir de 500 inversores,no pudimos simular el sistema con OpenModelica por lo que en este caso, analizamos los resultados obtenidos con Dymola.

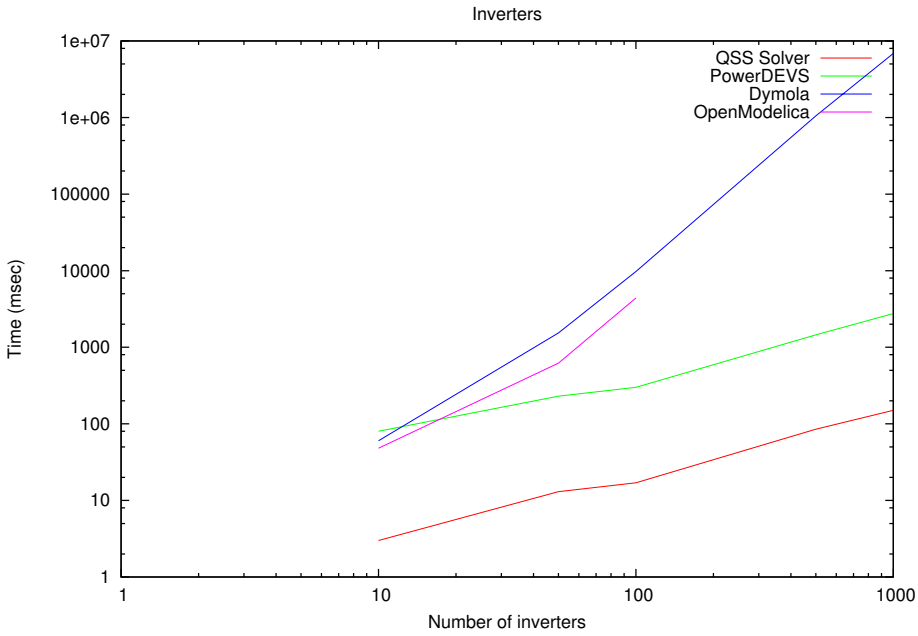


Figura 3.9: Tiempo de CPU vs. Número de Inversores.

Capítulo 4

Integración del Simulador Autónomo con otras Herramientas

Como vimos en el capítulo anterior, el simulador autónomo para métodos QSS permite definir modelos híbridos de gran escala utilizando un subconjunto del lenguaje Modelica denominado μ–Modelica. Cabe destacar que esta representación fue diseñada para contener los componentes mínimos de Modelica que permiten extraer la información estructural que necesita el motor de simulación, por lo que definir modelos utilizando μ–Modelica puede resultar complejo para el usuario final.

Es por esto que como parte del desarrollo de la presente Tesis, el simulador autónomo para métodos QSS fue integrado con diferentes herramientas de modelado y simulación que permiten ampliar el campo de aplicación de los métodos QSS y a su vez nos permite evaluar su rendimiento. En este capítulo presentaremos brevemente estos trabajos.

4.1. Integración con OpenModelica

En [7] se presentó una extensión introducida al compilador del entorno de modelado y simulación OpenModelica que traduce modelos Modelica regulares al lenguaje μ–Modelica lo que permite la integración con el motor de simulación QSS.

OpenModelica es un entorno de modelado y simulación de código abierto que soporta la especificación del lenguaje de modelado Modelica. Esta herramienta toma como entrada un modelo Modelica y realiza una serie de transformaciones sobre el modelo original que involucran: aplanado del modelo, reducción de índice,

ordenar las ecuaciones del modelo, resolver lazos algebraicos, como así también la optimización del código final. Como resultado de este proceso, se obtiene una presentación optimizada en forma de EDO del modelo original que permite generar código C++ eficiente para poder ejecutar la simulación.

La simulación de modelos Modelica genéricos mediante el uso de algoritmos QSS se estudió en [21, 22] donde se presenta una interfaz entre los entornos de simulación PowerDEVS y OpenModelica (OMPD). Esta interfaz permite la traducción automática de modelos Modelica de gran escala al formalismo DEVS permitiendo de esta manera simular el modelo original utilizando la herramienta PowerDEVS. Sin embargo, como mencionamos anteriormente 2.2.5, la implementación de los algoritmos QSS mediante motores de simulación DEVS no es la manera más eficiente.

Como consecuencia, se implementó una extensión del Compilador OpenModelica (OMC) que permite traducir de manera automática modelos Modelica genéricos al lenguaje μ–Modelica que es aceptado por el motor de simulación autónomo QSS. De esta manera, permitimos que los usuarios del entorno OpenModelica puedan utilizar los algoritmos de simulación QSS de la misma manera que utilizan cualquier otro algoritmo de simulación tradicional, sin necesidad de adquirir ningún conocimiento adicional.

La Figura 4.1 muestra las diferentes etapas por las que debe pasar el modelo Modelica original hasta la obtención del modelo ejecutable.

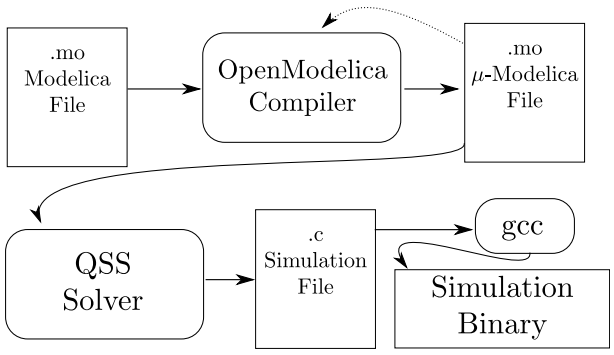


Figura 4.1: Integración OMC - Motor de Simulación QSS. Etapas

Finalmente, realizamos un estudio comparativo del rendimiento de los algoritmos QSS y la implementación DASSL nativa de OpanModelica en 2 modelos

discontinuos, lo cuál nos permite ver las ventajas de utilizar algoritmos QSS en este tipo de modelos.

4.1.1. Extensión del Compilador OMC

Para poder simular modelos Modelica con el motor de simulación autónomo QSS, agregamos primero una nueva salida al compilador OMC que genera el modelo μ–Modelica correspondiente.

Luego, se procesa la representación optimizada final del modelo original y las estructuras generadas por el compilador OMC para generar el código μ–Modelica que necesita el motor de simulación QSS. Los pasos a seguir en la transformación son los siguientes:

1. Encontrar las variables de estado continuas (i.e. las variables donde se utiliza el operador `der`), variables algebraicas, variables discreta y variables booleanas definidas en el modelo. Las variables booleanas son reemplazadas por variables de tipo `Real` que asumen valores 1 o 0.

2. Cada · que aparece en un identificador de variable del modelo es reemplazado por _.

3. Se genera código para definir e inicializar todas las variables de estado, algebraicas o discretas definidas en el modelo.

4. Si la ecuación es parte de un lazo algebraico, se genera una función externa que resuelve el lazo y se también se genera la correspondiente llamada a función en el código μ–Modelica.

5. Convertir las condiciones booleanas generales permitidas en Modelica a las relaciones permitidas en μ–Modelica ($<, \leq, >, \geq$). Luego, para cada nueva condición, se generan las sentencias `when` y `elsewhen` correspondientes.

6. Se expande el operador `sample` utilizando una variable discreta adicional.

Por ejemplo, el siguiente modelo que representa una pelota rebotando en el suelo definido en Modelica:

```
model bball
  Real y(start = 1),v,a;
  Boolean flying(start=true);
  parameter Real m = 1;
  parameter Real g = 9.8;
  parameter Real k = 10000;
  parameter Real b = 10;
equation
  der(y) = v;
  der(v) = a;
  a = if flying then -g else -g - (b * v + k * y)/m;
end bball;
```

se traduce al siguiente código μ–Modelica:

```
model bball
 constant Integer N = 2;
 Real x[N](start = xinit());
 discrete Real d[1](start = dinit());
 Real a[1];
 parameter Real m = 1;
 parameter Real g = 9.8;
 parameter Real k = 10000;
 parameter Real b = 10;
 function xinit
   output Real x[N];
   algorithm
     x[2]:= 1.0 /* y */;
     x[1]:= 0.0 /* v */;
 end xinit;
 function dinit
   output Real d[1];
   algorithm
```

```
    d[1]:=(1.0) /* flying*/;
  end dinit;
  /* Equations */
  equation
    der(x[2]) = x[1];
    a[1] = -d[1] * g + (1.0 - d[1]) *
    (((-b) * x[1] + (-k) * x[2]) / m - g);
    der(x[1]) = a[1];
  algorithm
  /* Discontinuities */
  when x[2] > 0.0 then
    d[1] := 1.0;
  elsewhen x[2] < 0.0 then
    d[1] := 0.0;
  end when;
end bball1;
```

Podemos ver que este modelo tiene dos estados continuos, una variable algebraica y una variable discreta junto con una discontinuidad que depende del estado `x[2]` que actualiza la variable discreta `d[1]`.

Cuando el modelo Modelica original contiene lazos algebraicos, el compilador OMC se encarga de detectar esta situación y en este caso el modelo μ–Modelica generado incluye el código correspondiente para resolver el lazo, por ejemplo:

```
...
function fsolve15
  input Real i0;
  input Real i1;
  output Real o0;
  output Real o1;
  output Real o2;
  external "C" fsolve15(i0,i1,o0,o1,o2);
end fsolve15;
```

```
...
equation
...
(a[1],a[2],a[3])=fsolve15(x[2],d[1])
```

En este caso, además del código μ–Modelica también se genera una función C externa que contiene el código necesario para resolver el lazo. Para esto se utiliza la GNU Scientific Library (GSL).

El código de ejemplo mostrado anteriormente indica que para poder calcular los valores de las variables algebraicas `a[1:3]` se debe llamar a la función externa `fsolve15`. El compilador μ–Modelica reconoce esta situación y adicionalmente obtiene las dependencias correspondientes en este caso basado en los argumentos que toma la función, a saber, `x[2]` y `d[1]`.

Cabe mencionar que en la función externa generada se optimizó el código generado por el compilador OMC teniendo en cuenta una característica de los lazos algebraicos lineales. En general, una ecuación algebraica lineal tiene la forma $A\dot{z} = b$ (donde z es un valor desconocido) y donde A depende solamente de variables discretas. Por lo tanto, cuando un cambio en la variable de estado afecta solamente al término b no es necesario invertir la matriz A en ese paso.

4.1.2. Resultados

En esta sección presentamos los resultados obtenidos al utilizar las herramientas presentadas en este Capítulo en 2 modelos que presentan discontinuidades frecuentes, un convertidor Buck y un circuito DC-DC intercalado. Ambos modelos fueron construidos utilizando la librería Modelica Standard Library 3.1 y se encuentran disponibles en [5].

En todos los experimentos utilizamos la versión modificada de OMC (revisión 11645) para generar el modelo μ-Modelica correspondiente y luego utilizamos el motor de simulación autónomo QSS para simular. En cada caso, comparamos los tiempos de CPU y la precisión de los algoritmos QSS contra el algoritmos DASSL de OpenModelica.

Los experimentos fueron realizados en una máquina Dell de 32 bits con un procesador quad–core y 4 GB de memoria RAM. Para estimar la exactitud de

las simulaciones dada una configuración, se calcularon trayectorias de referencia ($\mathbf{t^{ref}}, \mathbf{y^{ref}}$) utilizando el método LIQSS2 con una precisión de 10^{-7}.

Los errores reportados fueron obtenidos comparando las trayectorias simuladas contra las referencias. Para esto, las trayectorias de salida de todos los algoritmos generan la misma cantidad de puntos de salida equidistantes ($\mathbf{t^{ref}}, \mathbf{y^{sim}}$) sin cambiar el paso de integración. Luego, el error se calcula de la siguiente manera:

$$error = \frac{mean(|y^{sim} - y^{ref}|)}{mean(|y^{ref}|)} \tag{4.1}$$

4.1.3. Convertidor Buck

En la Figura 4.2 se muestra un circuito Buck. El circuito tiene 2 variables de estado, la corriente del inductor L1 y el voltaje del capacitor C1. La presencia del interruptor introduce el comportamiento híbrido en el sistema. Para calcular el error medimos la variable de estado C1.V. El modelo fue simulado por 0.01 segundos y la trayectoria de referencia se puede ver en la Figura 4.3

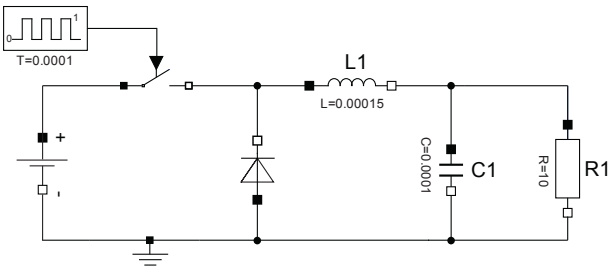


Figura 4.2: Circuito Buck

Inicialmente, simulamos el modelo con OMC utilizando el número de puntos de salida por defecto (500) y observamos que el algoritmo DASSL no detectaba correctamente los eventos. Por otro lado, seleccionar un mayor número de puntos de salida, el error cometido por DASSL disminuye ya que la evaluación requerida para calcular los mismos hace que DASSL reevalue las funciones de cruce por cero y detecte los eventos correspondientes. Por esta razon, comparamos el algoritmo DASSL de OMC para diferentes cantidades de puntos de salida contra los algoritmos LIQSS2 y LIQSS3. Los resultados obtenidos se muestran en la Tabla 4.1.

De hecho, se puede observar que para 500 puntos de salida, DASSL no reduce el error cometido por la simulación al aumentar los requerimientos de tolerancia.

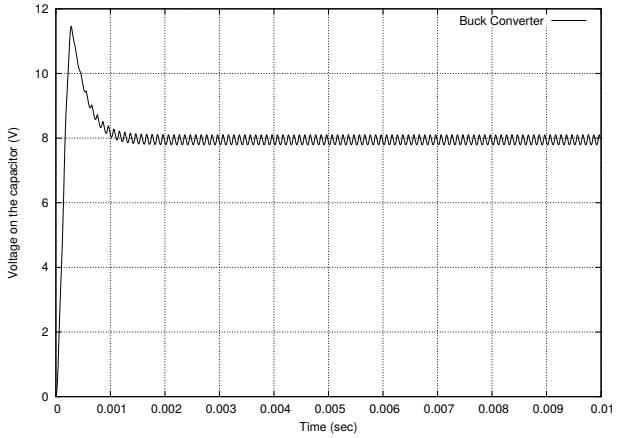


Figura 4.3: Cirtuito Buck - Simulación

Por otro lado, al seleccionar 10000 puntos de salida los resultados para DASSL el comportamiento del algoritmo es el esperado.

Debido a esto, tiene sentido comparar los tiempos de simulación para 10000 putnos de salida donde podemos ver claramente que los métodos QSS son más eficientes que DASSL. Para una tolerancia seleccionada de 10^{-5} podemos ver que el algoritmo LIQSS3 demora 12 ms mientras que DASSL demora 74 ms en simular. Por lo tanto, se acelera la simulación por un factor superior a 6. La reducción en tiempos de simulación y en el error cometido se muestra en la Figura 4.4.

Realizando una comparación entre los métodos QSS, vemos que el método de tercer orden LIQSS3 es más eficiente que el método LIQSS2 en este caso, más aún cuando la tolerancia requerida disminuye. Esto es esperable, dado que LIQSS2 debe dar pasos más pequeños comparado con LIQSS3. Por ejemplo, para un error de 10^{-6} LIQSS2 necesita dar 53391 pasos mientras que LIQSS3 sólo necesita 11314 pasos para terminar la simulación.

Cabe mencionar que los métodos QSS proveen salidas densas, por lo que la cantidad de puntos de salida seleccionados no afecta los tiempos de simulación.

4.1.4. Convertidor Buck Intercalado

La Figura 4.5 muestra el modelo de un convertidor Buck intercalado. Este circuito es similar al convertidor Buck analizado previamente pero contiene varios

Tabla 4.1: En esta tabla podemos ver los resultados de simulación obtenidos con diferentes algoritmos para el Convertidor Buck con un tiempo final de simulación de 0,01 segundos. En la comparación se muestra el tiempo de CPU, número de pasos, y el error relativo a la trayectoria de referencia obtenida con LIQSS2

			500 puntos			10000 puntos		
			Tiempo (msec)	Pasos	Error	Tiempo (msec)	Pasos	Error
QSS	**LIQSS3**	10^{-2}	4	3351	5.84E-03	16	3351	5.83E-03
	LIQSS3	10^{-3}	8	4163	7.31E-04	20	4163	7.32E-04
	LIQSS3	10^{-4}	12	6804	4.60E-05	24	6804	4.61E-05
	LIQSS3	10^{-5}	20	11314	1.07E-06	28	11314	1.08E-06
	LIQSS2	10^{-2}	4	3863	7.83E-03	16	3863	7.84E-03
	LIQSS2	10^{-3}	8	6715	1.32E-03	16	6715	1.32E-03
	LIQSS2	10^{-4}	12	18519	1.15E-04	24	18519	1.15E-04
	LIQSS2	10^{-5}	32	53391	6.42E-06	40	53391	6.42E-06
OpenModelica	**DASSL**	10^{-3}	22	4273	3.56E-03	70	5249	2.66E-04
	DASSL	10^{-4}	28	5636	3.17E-03	72	5955	1.75E-04
	DASSL	10^{-5}	32	7781	3.28E-03	74	7623	2.40E-05

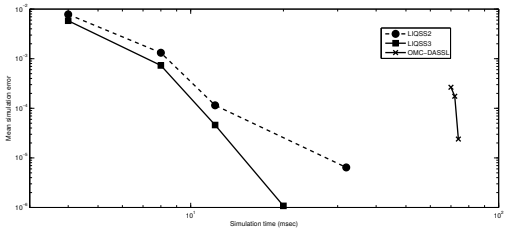


Figura 4.4: Tiempo de CPU vs Error para el modelo de convertidor buck (10000 puntos de salida)

interruptores que son activados en tiempos diferentes. En este caso, consideramos un circuito con cuatro ramas.

Todos los componentes utilizados para construir este modelo fueron tomados de la MSL 3.1, excepto por el componente `booleanDelay` que implementa un retardo booleano que emite el valor booleano recibido como entrada luego de un período de tiempo fijo `T`. El retardo no tiene memoria, i.e. cuando se recibe una entrada, cualquier salida agendada es cancelada y se guarda la nueva entrada.

Simulamos este modelo por 0.01 segundos y tomamos como salida el voltaje del capacitor, la Figura 4.6 muestra la trayectoria obtenida. Realizamos los mismos experimentos que en el ejemplo anterior y los resultados se muestran en la Tabla 4.2.

En la Figura 4.7 se puede ver que para un error de 10^{-3} DASSL demora 488 ms mientras que LIQSS2 demora 12 ms y LIQSS3 demora 60 ms en terminar la simulación. Por lo que en este caso los incrementos en velocidad de simulación varian entre 8 y 40 veces. La diferencia entre los tiempos de simulación entre

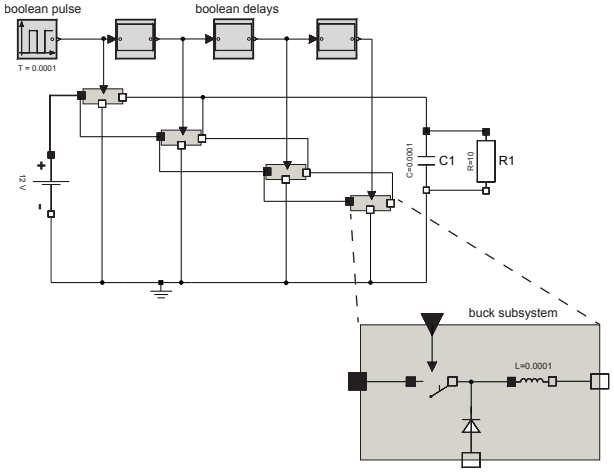


Figura 4.5: Circuito DC-DC intercalado.

LIQSS2 y LIQSS3 se dan debido a que la implementación de LIQSS3 no está optimizada en comparación a LIQSS2.

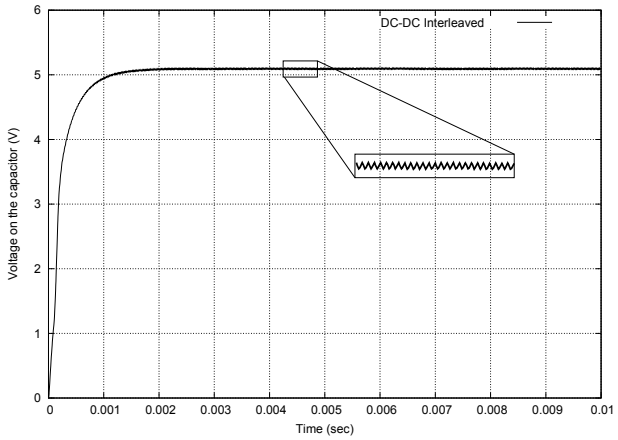


Figura 4.6: Circuito DC-DC Intercalado - Simulación

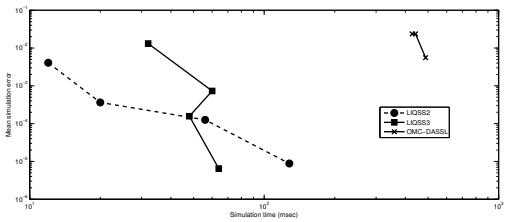


Figura 4.7: Tiempo de CPU vs Error para el modelo del ciruito DC-DC intercalado (10000 puntos de salida)

Tabla 4.2: En esta tabla podemos ver los resultados de simulación obtenidos par diferentes algoritmos para el circuito DC-DC con un tiempo de simulación final de 0.01 segundos. En la comparación se muestra el tiempo de CPU (msec), el número de pasos y el error relativo a la trayectoria de referencia obtenida con LIQSS2.

			500 puntos			10000 puntos		
			Tiempo (msec)	Pasos	Error	Tiempo (msec)	Pasos	Error
QSS	LIQSS3	10^{-2}	32	18396	1.32E-02	44	18396	1.32E-02
	LIQSS3	10^{-3}	60	33426	7.31E-04	72	33426	7.31E-04
	LIQSS3	10^{-4}	48	29408	1.57E-04	60	29408	1.57E-04
	LIQSS3	10^{-5}	64	39951	6.48E-06	76	39951	6.48E-06
	LIQSS2	10^{-2}	12	10715	4.08E-03	20	10715	4.08E-03
	LIQSS2	10^{-3}	20	29082	3.63E-04	36	29082	3.63E-04
	LIQSS2	10^{-4}	56	73218	1.26E-04	68	73218	1.26E-04
	LIQSS2	10^{-5}	128	198001	8.80E-06	140	198001	8.80E-06
OpenModelica	DASSL	10^{-3}	310	14421	4.96E-02	428	17571	2.37E-02
	DASSL	10^{-4}	363	22375	5.03E-02	442	18574	2.37E-02
	DASSL	10^{-5}	496	31387	5.41E-02	488	23625	5.57E-03

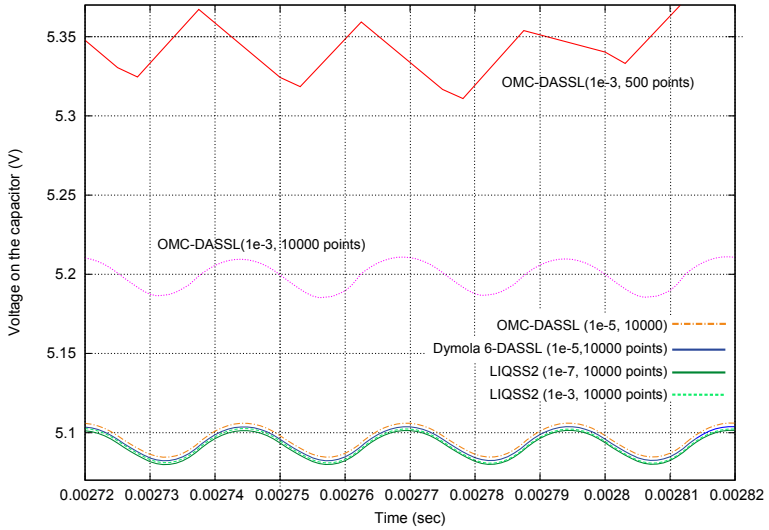


Figura 4.8: Comparación del estado final para diferentes configuraciones.

La Figura 4.8 muestra los resultados obtenidos para diferentes configuraciones. Donde podemos ver que la detección de discontinuidades de OMC es afectada por el número de puntos de salida. En esta caso también se incluyen los resultados obtenidos utilizando Dymola 6.0 que es utilizado para obtener trayectorias de referencia.

La integración del motor de simulación QSS a la herramienta de modelado y simulación OpenModelica permite acceder a una gran cantidad de modelos previamente definidos que son utilizados en aplicaciones de ingeniería y otras áreas en

las que el lenguaje de simulación Modelica es tomado como un estandard de modelado obteniendo las ventajas ya mencionadas de los métodos QSS para sistemas con discontinuidades frecuentes.

4.2. Integración con SBML

Como parte de la integración del motor de simulación QSS con diferentes lenguajes de modelado utilizados por la comunidad científica, se desarrolló una herramienta que permite traducir y simular modelos SBML (System Biology Markup Language)[20, 31].

SBML es un lenguaje de especificación de sistemas biológicos que permite definir la dinámica de entidades biológicas interactuando a través de diferentes procesos que evolucionan en el tiempo. Este lenguaje fue diseñado para ser interpretado por herramientas de software, no para el usuario final, permitiendo definir modelos de complejidad arbitraria mediante un formato de especificación unificado que permita compartir y reutilizar modelos de una manera flexible. Existen numerosas herramientas [58] que permiten crear, editar y simular modelos SBML.

En este contexto, muchos de estos modelos tienen características que pueden ser aprovechadas por los métodos QSS (modelos stiff, discontinuos, etc.) y de esta manera obtener simulaciones más eficientes en términos de tiempos de simulación. En las siguientes secciones daremos una breve descripción de la estructura de los modelos SBML, la traducción de estos modelos a μ–Modelica, como así también veremos algunos ejemplos de uso.

4.2.1. Representación de Modelos SBML

Un modelo SBML es representado mediante la descripción de *procesos* que tienen las siguientes características:

- Se representan mediante un conjunto de reacciones.
- Las diferentes entidades que intervienen en una reacción son denominadas especies.
- Las diferentes especies que intervienen en los procesos son ubicadas en contenedores.

- Las especies que aparecen en el lado izquierdo de la reacción de denominan reactantes y las especies que aparecen del lado derecho se denominan productos.

Teniendo en cuenta estas especificaciones, la forma general de un *proceso* es la siguiente:

$$\begin{aligned} A + B &\xrightarrow{f_1(\cdots)} C \\ A + D &\xrightarrow{f_2(\cdots)} E \\ C &\xrightarrow{f_3(\cdots)} F + G \\ &\vdots \end{aligned}$$

Donde a partir de esta definición se pueden derivar las tasas de cambio de las diferentes especies que intervienen en el proceso.

Adicionalmente, los modelos pueden incluir construcciones adicionales como constantes y variables que no intervengan en los procesos biológicos, eventos, anotaciones, reglas, etc.

En particular, las reglas se utilizan en un modelo SBML para describir relaciones matemáticas que no pueden ser deducidas del sistema de reacciones y se puede distinguir entre reglas algebraicas $0 = f(\mathbf{W})$ y derivadas $dx = f(\mathbf{W})$.

De esta manera, el modelo completo de ecuaciones SBML puede ser caracterizado por:

- Ecuaciones derivadas del sistema de reacciones

$$\begin{aligned} \frac{dS_1}{dt} &= r_1 + r_2 + r_3 + \cdots \\ \frac{dS_2}{dt} &= r_1 + r_2 + r_3 + \cdots \\ &\cdots \end{aligned}$$

- Ecuaciones algebraicas

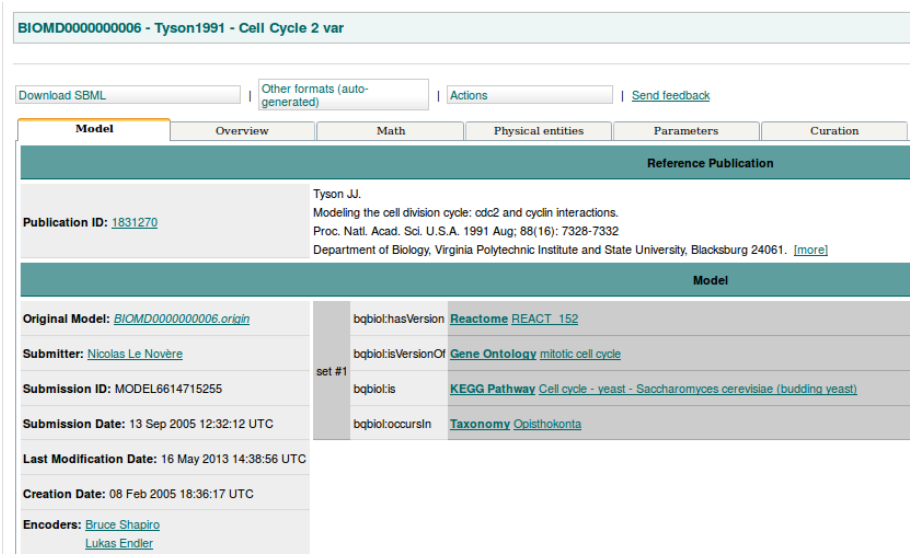
$$\begin{aligned} x &= g_1(\mathbf{W}) \\ y &= g_2(\mathbf{W}) \\ &\cdots \end{aligned}$$

- Derivadas

$$\frac{dm}{dt} = g_1(\mathbf{W})$$
$$\frac{dq}{dt} = g_2(\mathbf{W})$$
$$\dots$$

En la Figura 4.9 podemos ver un ejemplo de un modelo simple presentado en [60] tomado de base de datos pública de modelos SBML BioModels.

Figura 4.9: Modelo SBML Tyson 1991 - Ciclo celular de 2 variables



El conjunto de reacciones que definen este modelo son las siguientes:

$$\begin{aligned} \emptyset &\xrightarrow{kappa} z \\ u &\xrightarrow{k6 \times u} \emptyset \\ z &\xrightarrow{k4 \times z \times (\frac{k4prime}{k4} + u^2)} u \end{aligned} \tag{4.2}$$

En la siguiente sección veremos cómo a partir de esta definición se puede obtener un modelo ejecutable que perita simular el sistema.

4.2.2. Traducción de Modelos SBML a μ–Modelica

A partir de este modelo de ecuaciones descripto en la sección anterior, se desarrollo una herramienta de traducción de modelos SBML, utilizando la librería C++ de SBML. Esta librería permite acceder a los modelos SBML a través de

una interfaz de aplicación mediante la cuál se pueden generar las transformaciones necesarias para obtener el código μ–Modelica necesario.

El traductor implementado aplica las reglas de conversión que permiten generar código μ–Modelica equivalente a cada una de las construcciones definidas en la especificación del lenguaje a partir de un modelo SBML dado. A continuación se muestran algunas de las reglas de traducción implementadas:

- Reglas: Existen tres tipos de reglas diferentes en la especificación SBML, todas las reglas definidas son traducidas de acuerdo al tipo correspondiente dentro de la sección `equation` del modelo μ–Modelica correspondiente:
  - Las reglas *Algebraicas* y de *Asignación* son traducidas de la siguiente manera:

    ```
    variable_id = eq_exp;
    ```

  - Las reglas que definen la *Tasa de Cambio* de una variable generan el siguiente código:

    ```
    der(variable_id) = eq_exp;
    ```

- Derivadas: Para poder construir un sistema de ecuaciones diferenciales que represente las tasas de cambio de las especies definidas en un modelo SBML, se tiene que procesar todas las reacciones definidas en el modelo donde aparecen las diferentes especies. Teniendo esto en cuenta, la expresión final para la derivada de una especie se construye sumando todas las leyes correspondientes a las reacciones donde la especie aparece como producto y restando todas las leyes de las reacciones donde la especie aparece como reactante. Luego de procesar todas las reacciones del sistema, la derivada de una especie en un modelo SBML tiene la siguiente forma en μ–Modelica:

  ```
  der(species_id) = klp_1 + ... + klp_n - klr_1 - ... - klr_m;
  ```

  where

  ```
  [klp_1,...,klp_n]
  ```

  representa todas las reacciones donde la especie aparece como producto y

`[klr_1,...,klr_m]`

son las reacciones donde la especie aparece como reactante.

- Eventos: En este caso, se deben traducir las condiciones booleanas que disparan el evento a las relaciones aceptadas por el lenguaje μ–Modelica para poder ser luego interpretadas como funciones de cruce por cero en el motor de simulación QSS. Una vez obtenida esta expresión para esto, se deben transformar las asignaciones correspondientes al handler del evento. En el caso general, la expresión genera el siguiente código μ–Modelica: generates the following μ–Modelica code:

```
when cond then
  assignments;
end when;
```

donde `cond` es la traducción de la condición SBML original y `assignments` representa una lista de una o más asignaciones que se ejecutan cuando la condición del evento se cumple. Adicionalmente, para cada evento definido, se debe tener en cuenta la prioridad definida, los posibles retrasos que pueda tener la ejecución del handler entre otras particularidades, el código anterior es solamente un ejemplo del caso general.

A modo de ejemplo, en la Figura 4.10 se puede ver el modelo μ-Modelica generado automáticamente por el traductor a partir del sistema de reacciones definido en (4.2) y los resultados de la simulación del modelo se muestran en la Figura 4.11

En conclusión, la herramienta de traducción de modelos SBML permite utilizar el motor de simulación QSS en una gran variedad de modelos biológicos que son de interés en la comunidad científica. En particular, nos permite simular modelos en los que se pueden apreciar las ventajas proporcionadas por los métodos QSS.

Figura 4.10: Modelo μ–Modelica Generado.

BIOMD0000000003.mo

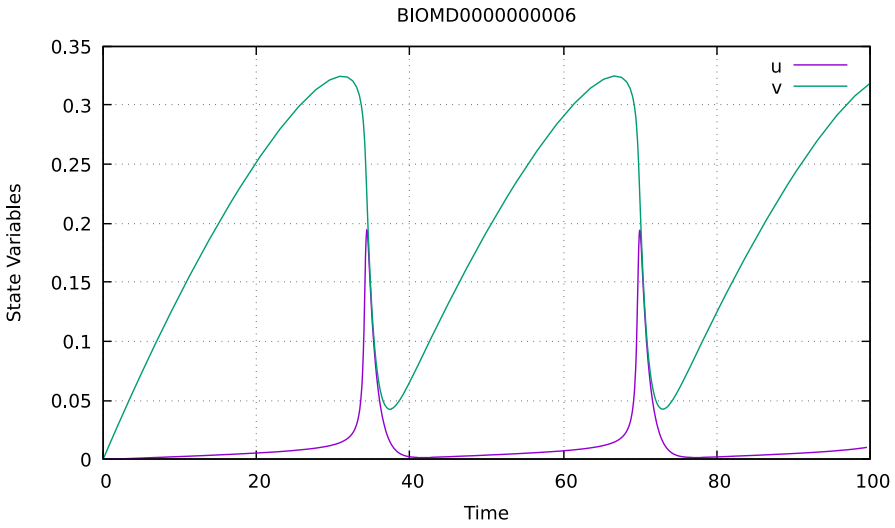
```
model BIOMD0000000003

 Real C(start = 0.01);
 parameter Real K1 = 0.005;  parameter Real K2 = 0.005;
 parameter Real K3 = 0.005;  parameter Real K4 = 0.005;
 parameter Real Kc = 0.5;      parameter Real Kd = 0.02;
 Real M(start = 0.01);            Real V1;
 Real V3;                            Real X(start = 0.01);
 parameter Real V2 = 1.5;     parameter Real V4 = 0.5;
 parameter Real VM1 = 3;     parameter Real VM3 = 1;
 parameter Real cell = 1;      parameter Real kd = 0.01;
 parameter Real vd = 0.25;    parameter Real vi = 0.025;

equation

 V1 = (C * VM1) * ((C + Kc) ^ ((-1)));
 V3 = M * VM3;
 der(C) = ((cell * vi) + ((-1) * ((C * cell) * kd))) + ((-1) * ((((C * cell) * vd) * X) * ((C + Kd) ^ ((-1)))));
 der(M) = (((cell * (1 + ((-1) * M))) * V1) * (((K1 + ((-1) * M)) + 1) ^ ((-1)))) + ((-1) * (((cell * M) * V2) * ((K2 + M) ^ ((-1)))));
 der(X) = (((cell * V3) * (1 + ((-1) * X))) * (((K3 + ((-1) * X)) + 1) ^ ((-1)))) + ((-1) * (((cell * V4) * X) * ((K4 + X) ^ ((-1)))));
```

Figura 4.11: Modelo Tyson - Resultados de simulación.



Capítulo 5

Aplicaciones y Resultados

En los capítulos anteriores se presentó el simulador autónomo QSS y su integración con distintos entornos de modelado y simulación. Adicionalmente, otro objetivo de la presente Tesis es poder estudiar en profundidad el desempeño del simulador en comparación con diferentes métodos de integración numérica clásicos, analizando las ventajas que presenta su aplicación a modelos provenientes de diferentes campos de las ciencia y la ingeniería. Los resultados obtenidos del trabajo realizado son presentados en este capítulo.

5.1. Aplicación en Modelos de Electrónica de Conmutación

En el trabajo presentado en [44] se estudió el rendimiento de los métodos QSS en la simulación de modelos de electrónica de conmutación (Switched Mode Power Suplies SMPS). Desde un enfoque de modelado realista, estos modelos son stiff y presentan discontinuidades frecuentes lo que dificulta su simulación utilizando métodos de integración numérica clásicos. El objetivo del trabajo es aplicar métodos QSS linealmente implícitos (LIQSS) que fueron desarrollados para simular este tipo de modelos de manera eficiente. Con este fin, se construyeron modelos que describen las diferentes topologías posibles de SMPS. Luego se realizó un análisis de los resultados obtenidos utilizando el método de integración numérica clásico DASSL y métodos LIQSS de segundo y tercer orden. Llegando a la conclusión que los métodos LIQSS son entre 3 y 200 veces más rápidos y más precisos que los métodos tradicionales.

A continuación presentaremos los modelos presentados en este trabajo y los resultados obtenidos.

5.1.1. Modelos SMPS

En esta sección describiremos los modelos correspondientes a las diferentes topologías de SMPSs. Para esto, primero describimos los modelos matemáticos para los componentes de conmutación (llaves y diodos) y luego derivamos las ecuaciones correspondientes al circuito.

Modelo de Llave Controlada

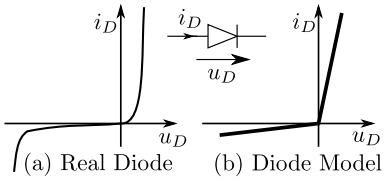
Una llave controlada es un elemento que actúa como un circuito abierto o cerrado de acuerdo al estado de una señal de control. Este elemento puede ser modelado como una resistencia R_{s} con un valor alto o bajo de acuerdo a una señal de control. Este comportamiento puede se puede describir mediante la siguiente ecuación.

$$R_{\mathrm{s}} = \begin{cases} R_{\mathrm{On}} & \text{if } \mathit{control} = 1 \\ R_{\mathrm{Off}} & \text{if } \mathit{control} = 0 \end{cases} \tag{5.1}$$

donde R_{On} y R_{Off} son valores de resistencia muy bajos o muy altos, respectivamente.

Modelado del Diodo

La Figura 5.1 muestra la caracterización corriente–voltaje de un diodo real en el lado izquierdo y su aproximación seccionalmente lineal en el lado derecho. La aproximación es el resultado de representar el estado *OFF* con una resistencia grande y el estado *ON* con una resistencia pequeña.



(a) Real Diode (b) Diode Model

Figura 5.1: Caracterización real y aproximada de un diodo.

De acuerdo a esta figura, el valor del diodo R_D sigue la siguiente ley

$$R_{\mathrm{s}} = \begin{cases} R_{\mathrm{On}} & \text{if } u_D > 0 \\ R_{\mathrm{Off}} & \text{if } u_D \leq 0 \end{cases}$$

5.1.2. Modelos para las Diferentes Topologías

Existen tres tipos de topologías básicas para los modelos SMPS. En las próximas secciones derivaremos las ecuaciones correspondientes a cada una de ellas.

Convertidor Buck

El convertidor Buck es un convertidor que genera un voltage de salida menor al voltage de entrada. La llave de dos puntos de este convertidor es implementada en aplicaciones reales por un transistor y diodo como se muestra en la Figura 5.2.

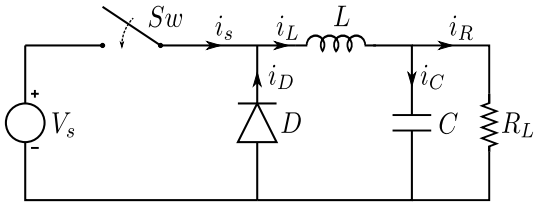


Figura 5.2: Convertidor Buck

Representando la llave y el diodo por resistencias R_s y R_D como discutimos anteriormente, se puede obtener la siguiente representación espacio-estado de este circuito:

$$\begin{aligned} \frac{di_L}{dt} &= \frac{-i_D R_D - u_C}{L} \\ \frac{du_C}{dt} &= \frac{i_L - u_C/R}{C} \end{aligned} \tag{5.2}$$

donde

$$i_D = \frac{i_L R_s - U}{R_s + R_D} \tag{5.3}$$

En este último modelo, incluimos la generación de la señal de control para la llave controlada. Esta señal de control asume el valor 1 (el estado ON) cada T unidades de tiempo y cambia a 0 luego de D_C unidades de tiempo.

El uso de esta señal de control corresponde a una regulación de voltage de *lazo abierto*. En muchas aplicaciones, es preferible utilizar una estrategia de *lazo cerrado*, donde la señal de control se calcula comparando el voltage de salida con una referencia.

El uso de estrategias de lazo abierto o cerrado no introduce ninguna diferencia significativa desde un punto de vista numérico, por lo que aquí trabajaremos con un esquema de lazo abierto.

Convertidor Boost

El circuito Boost, mostrado en la Figura 5.3, es un convertidor que genera voltages de salida mayores a los voltages de entrada.

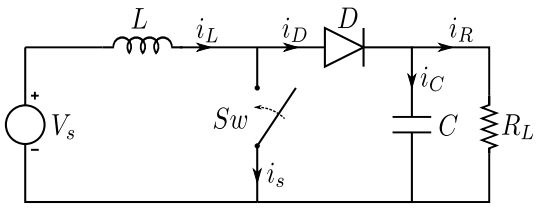


Figura 5.3: Circuito Boost

Procediendo de igual manera que en el ejemplo anterior, se obtienen las siguientes ecuaciones de estado:

$$\begin{aligned} \frac{di_L}{dt} &= \frac{-R_s i_L + R_s i_D + U}{L} \\ \frac{du_C}{dt} &= \frac{i_D}{C} - \frac{u_C}{R_L C} \end{aligned} \tag{5.4}$$

donde

$$i_D = \frac{R_s i_L - u_C}{R_D + R_s} \tag{5.5}$$

Convertidor Buck-Boost

La Figura 5.4 muestra el circuito Buck-Boost. En este convertidor, la magnitud del voltaje de salida puede ser mayor o menor que el voltaje de entrada de acuerdo al ciclo de trabajo. La polaridad del voltaje de salida es siempre la opuesta al voltaje de entrada.

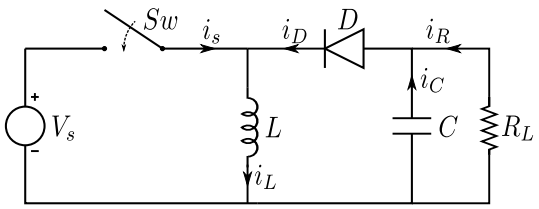


Figura 5.4: Circuito Buck-Boost

Procediendo como antes, se obtiene:

$$\begin{aligned} \frac{di_L}{dt} &= \frac{-u_C - R_D i_D}{L} \\ \frac{du_C}{dt} &= \frac{i_D}{C} - \frac{u_C}{R_L C} \end{aligned} \tag{5.6}$$

donde

$$i_D = \frac{R_s i_L - u_C - U}{R_D + R_s} \tag{5.7}$$

Convertidor Cuk

El circuito Cuk es una variante del convertidor Buck–Boost donde también la magnitud del voltaje de salida puede ser mayor o menor que el voltaje de entrada pero en este caso con la polaridad opuesta. El circuito correspondiente al convertidor Cuk se muestra en la Figura 5.5.

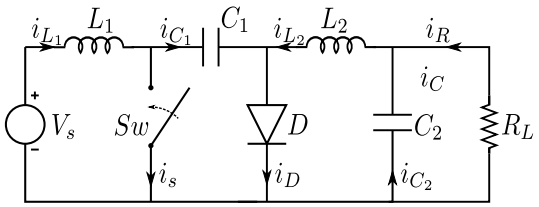


Figura 5.5: Convertidor Cuk

Las ecuaciones de estado para este circuito son:

$$\begin{aligned} \frac{di_{L_1}}{dt} &= \frac{U - u_{C_1} - R_D i_D}{L_1} \\ \frac{du_{C_1}}{dt} &= \frac{i_D - i_{L_2}}{C_1} \\ \frac{di_{L_2}}{dt} &= \frac{-u_{C_2} - R_D i_D}{L_2} \\ \frac{du_{C_2}}{dt} &= \frac{R_L i_{L_2} - u_{C_2}}{R_L C_2} \end{aligned} \tag{5.8}$$

donde

$$i_D = \frac{R_s(i_{L_2} + i_{L_1}) - u_{C_1}}{R_D + R_s} \tag{5.9}$$

Convertidores Intercalados

La Figura 5.6 muestra el circuito correspondiente a un convertidor Buck intercalado de cuatro etapas. En este circuito, cada período es dividido por cuatro, durante cada sub–período sólo una etapa puede de cambiar el estado *ON* y *OFF*

para poder alimentar la carga mientras que las otras etapas se mantienen en el estado *OFF*. De esta manera, la frecuencia de cambio del circuito entero es cuatro veces más rápida que la de las etapas individuales.

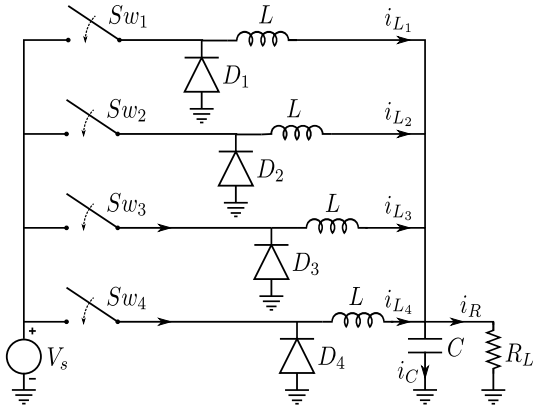


Figura 5.6: Convertidor Buck intercalado de cuatro etapas

Las ecuaciones de estado para un circuito intercalado Buck de N etapas se pueden escribir de la siguiente manera:

$$\begin{aligned} \frac{di_{L_j}}{dt} &= \frac{-i_{D_j}R_{D_j} - u_C}{L} \quad \text{para } j = 1 \cdots, N \\ \frac{du_C}{dt} &= \frac{\sum_{j=1}^{N} i_{L_j}}{C} - \frac{u_C}{R_L C} \end{aligned} \tag{5.10}$$

con

$$i_{D_j} = \frac{i_{L_j}R_{s_j} - U}{R_{s_j} + R_{D_j}} \tag{5.11}$$

5.1.3. Resultados

Esta sección muestra los resultados obtenidos, comparando el rendimiento de los métodos LIQSS con el método clásico DASSL en la simulación de los cinco modelos SMPS presentados anteriormente.

Para realizar esta comparación, ejecutamos un conjunto de experimentos de acuerdo que comparten la siguiente configuración:

- Simulamos todos los modelos con dos configuraciones de tolerancia diferentes: rel.tol = abs.tol = 10^{-3} y rel.tol = abs.tol = 10^{-5}.

- En todas las simulaciones el tiempo final fue de $t_f = 0{,}01 sec.$.

- Las simulaciones fueron ejecutadas utilizando un procesador Intel i7-3770@3,40GHz con un sistema operativo Ubuntu.
- Los resultados reportados para el método LIQSS fueron obtenidos utilizando el motor de simulación autónoma QSS.
- Los resultados reportados para el método DASSL fueron obtenidos utilizando el código DASSRT, mediante una interfaz provista por el motor de simulación autónomo QSS, por lo que los modelos simulados por LIQSS y DASSL fueron exactamente los mismos.
- Los sistemas también fueron simulados utilizando la implementación del método DASSL de OpenModelica y Dymola. Sin embargo, el uso directo del método DASSRT reportó mejores resultados que el mismo método en las herramientas mencionadas, por lo que se reportan solamente los resultados obtenidos para DASSRT.
- En todos los casos, medimos el tiempo de CPU utilizado, el número de evaluaciones de funciones escalares, el número de cálculos del Jacobiano y el error relativo, calculado como:

$$e_{rr} = \sqrt{\frac{\sum (u_C[k] - u_{C_{REF}}[k])^2}{\sum u_{C_{REF}}[k]^2}} \tag{5.12}$$

  donde la solución de referencia $u_{C_{REF}}[k]$ fue obtenida utilizando DASSL con una tolerancia muy pequeña (10^{-9}).
- El tiempo de CPU fue medido como la media de 10 simulaciones.

Convertidor Buck

Este SMPS, cuyo modelo fue descripto es la Sección 5.1.2 fue simulado con los siguientes parámetros:

- Voltaje de entrada: $V_s = 24V$,
- Salida: $C = 10^{-4}F$,
- Inductancia: $L = 10^{-4}H$,
- Resistencia de carga: $R_L = 10\Omega$,

- Resistencia la llave y el diodo en el estado ON: $R_{\mathrm{On}} = 10^{-5}\Omega$,
- Resistencia la llave y el diodo en el estado OFF: $R_{\mathrm{Off}} = 10^{5}\Omega$,
- Período de la señal de control la llave: $T = 10^{-4} sec.$,
- Ciclo de trabajo del control la llave: $DC = 0{,}5$.

La parte transitoria de los resultados se muestra en la Figura 5.7. Como es esperado en esta topología, el voltaje de salida $u_C(t)$ tiene un valor promedio menor que el voltaje de entrada V_s y exhibe una pequeña onda en la frecuencia de cambio. El comportamiento discontinuo de este modelo se puede observar claramente el la trayectoria $i_L(t)$ correspondiente a la corriente.

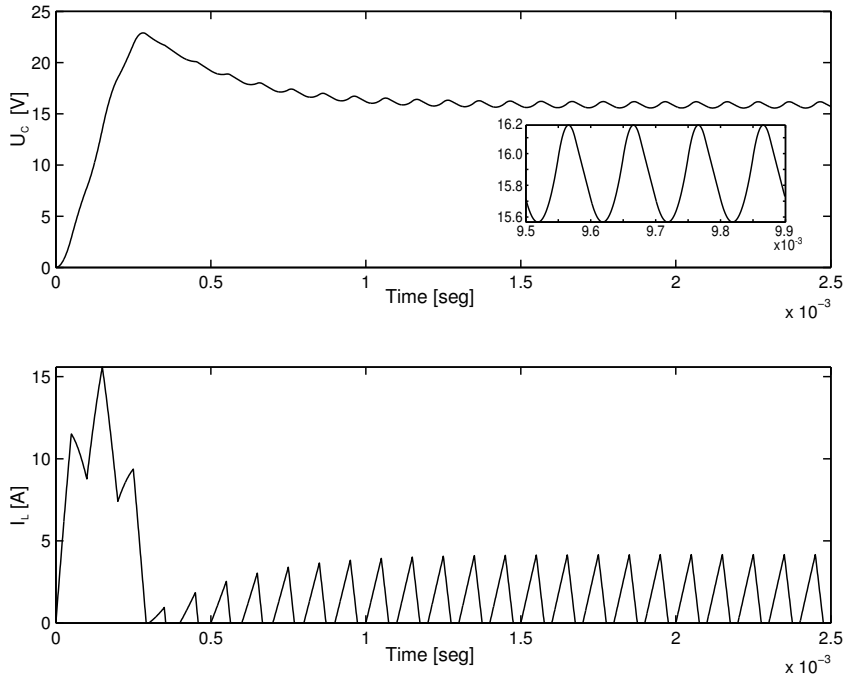


Figura 5.7: Trayectorias de Salida Convertidor Buck

La Tabla 5.1 compara los tiempos de CPU, el número de evaluaciones y los errores obtenidos con los distintos métodos.

En esta tabla puede verse que todos los métodos cumplen con la tolerancia de error requerida. Con respecto a los tiempos de simulación, al exigir un error relativo pequeño (10^{-5}), los tres algoritmos consumen tiempos de CPU similares aún cuando el número de evaluaciones de funciones realizadas por el método LIQSS2

Tabla 5.1: Convertidor Buck - Resultados.

Método	**Tolerancia**	**Error Relativo**	**Eval. Jacobiano**	**Eval. f_i**	**Tiempo CPU (ms)**
DASSL	err.tol=$1 \cdot 10^{-3}$	$2{,}28 \cdot 10^{-}3$	3079	26670	6,58589
	err.tol.=$1 \cdot 10^{-5}$	$9{,}63 \cdot 10^{-}6$	4474	44772	11,6278
LIQSS2	$\Delta Q_i = 1 \cdot 10^{-3}$	$1{,}31 \cdot 10^{-}3$	–	13286	2,26316
	$\Delta Q_i = 1 \cdot 10^{-5}$	$1{,}06 \cdot 10^{-}5$	–	117198	11,3644
LIQSS3	$\Delta Q_i = 1 \cdot 10^{-3}$	$1{,}09 \cdot 10^{-}3$	–	11355	3,43807
	$\Delta Q_i = 1 \cdot 10^{-5}$	$1{,}04 \cdot 10^{-}5$	–	35283	11,2723

fueron aproximadamente tres veces más que las correspondientes a los métodos LIQSS3 y DASSL. Esto se puede explicar teniendo en cuenta que detectar discontinuidades en LIQSS2 es menos costoso que en LIQSS3 y DASSL (para detectar una discontinuidad LIQSS2 solamente resuelve una ecuación lineal escalar) y que LIQSS2 no requiere calcular el Jacobiano. Además, los pasos continuos de LIQSS2 son menos costosos que los pasos continuos de LIQSS3 y DASSL.

Para una tolerancia mayor (10^{-3}), que es la opción usual para este tipo de circuitos, la simulación utilizando el método LIQSS2 fue 3 veces más rápida que DASSL y 1.5 veces más rápida que LIQSS3.

Este hecho no es sorprendente dado que los métodos de menor orden son generalmente más eficiente para simular sistemas con requerimientos de precisión menores.

Convertidor Boost

Para este circuito, cuyo modelo fue descripto en la Sección 5.1.2, utilizamos el los mismos parámetros que para el modelo anterior. La Tabla 5.2 compara el rendimiento exhibido por los diferentes métodos.

Los resultados son muy similares a los obtenidos para el convertidor Buck. Sin embargo, para una tolerancia de 10^{-5} DASSL muestra un error muy superior a la tolerancia y que los métodos LIQSS.

Esta diferencia es debido a que en los métodos LIQSS las discontinuidades son detectadas de manera exacta, mientras que en DASSL pueden tener un cierto error

Tabla 5.2: Convertidor Boost - Resultados.

Método	Tolerancia	Error Relativo	Eval. Jacobiano	Eval. f_i	Tiempo CPU (ms)
DASSL	err.tol=$1 \cdot 10^{-3}$	$1{,}30 \cdot 10^{-}3$	2215	18778	4,94262
	err.tol.=$1 \cdot 10^{-5}$	$5{,}34 \cdot 10^{-}5$	3192	28834	7,2436
LIQSS2	$\Delta Q_i = 1 \cdot 10^{-3}$	$1{,}52 \cdot 10^{-}3$	–	10476	1,54468
	$\Delta Q_i = 1 \cdot 10^{-5}$	$1{,}96 \cdot 10^{-}5$	–	70628	8,25393
LIQSS3	$\Delta Q_i = 1 \cdot 10^{-3}$	$1{,}11 \cdot 10^{-}3$	–	9648	4,36562
	$\Delta Q_i = 1 \cdot 10^{-5}$	$1{,}26 \cdot 10^{-}5$	–	21420	8,18659

debido a los procesos de iteración lo que incremente el error global de simulación.

Convertidor Buck-Boost

Para este circuito, descripto en la Sección 5.1.2, utilizamos la misma configuración de parámetros que para el convertidor Buck excepto por el ciclo de trabajo, que ahora es $DC = 0{,}25$. La comparación del rendimiento de los diferentes métodos es reportada en la Tabla 5.3.

Tabla 5.3: Convertidor Buck-Boost - Resultados.

Método	Tolerancia	Error Relativo	Eval. Jacobiano	Eval. f_i	Tiempo CPU (ms)
DASSL	err.tol=$1 \cdot 10^{-3}$	$5{,}12 \cdot 10^{-}3$	3689	29240	6,88186
	err.tol=$1 \cdot 10^{-5}$	$3{,}04 \cdot 10^{-}4$	5138	46532	11,8803
LIQSS2	$\Delta Q_i = 1 \cdot 10^{-3}$	$1{,}39 \cdot 10^{-}3$	–	14632	2,74414
	$\Delta Q_i = 1 \cdot 10^{-5}$	$1{,}47 \cdot 10^{-}5$	–	84476	10,1215
LIQSS3	$\Delta Q_i = 1 \cdot 10^{-3}$	$5{,}23 \cdot 10^{-}4$	–	12618	5,28576
	$\Delta Q_i = 1 \cdot 10^{-5}$	$1{,}34 \cdot 10^{-}5$	–	27912	8,23346

Con respecto a los tiempos de CPU y evaluaciones de funciones, los resultados son similares a los de los convertidores Buck y Boost. Sin embargo, al comparar el error relativo reportado, los resultados obtenidos con DASSL son aún peores que los reportados para el convertidor Boost. El error relativo es entre 5 y 30 veces

mayor que la tolerancia mientras que los métodos LIQSS cumplen con la tolerancia predefinida.

Convertidor Cuk

En el caso del convertidor Cuk, se utilizó el modelo presentado en la Sección 5.1.2, configurado con los mismos parámetros que para el convertidor Buck–Boost, con los siguientes parámetros adicionales $C_1 = C_2 = 10^{-4}F$ y $L_1 = L_2 = 10^{-4}H$.

Los resultados obtenidos son reportados en la Tabla 5.4, que muestra la comparación de rendimiento para los diferentes algoritmos.

Tabla 5.4: Convertidor Cuk - Resultados.

Método	**Tolerancia**	**Error Relativo**	**Eval. Jacobiano**	**Eval. f_i**	**Tiempo CPU (ms)**
DASSL	err.tol=$1 \cdot 10^{-3}$	$1{,}75 \cdot 10^{-}2$	2858	77016	10,7689
	err.tol=$1 \cdot 10^{-5}$	$1{,}97 \cdot 10^{-}4$	4270	123200	17,3262
LIQSS2	$\Delta Q_i = 1 \cdot 10^{-3}$	$7{,}40 \cdot 10^{-}3$	–	73082	9,6474
	$\Delta Q_i = 1 \cdot 10^{-5}$	$8{,}71 \cdot 10^{-}5$	–	448310	30,3638
LIQSS3	$\Delta Q_i = 1 \cdot 10^{-3}$	$6{,}07 \cdot 10^{-}3$	–	53547	14,0629
	$\Delta Q_i = 1 \cdot 10^{-5}$	$5{,}02 \cdot 10^{-}5$	–	97509	23,9215

Con respecto a los tiempos de simulación, para una tolerancia de error pequeña (10^{-5}) DASSL es ahora más rápido que ambos métodos LIQSS, mientras que para tolerancias mayores (10^{-3}), los tiempos de simulación son similares para los tres métodos. Sin embargo, el error reportado por DASSL es casi 20 veces mayor que la tolerancia permitida mientras que los métodos LIQSS son claramente más precisos.

Convertidor Buck Intercalado

El último circuito simulado, descripto en la Sección 5.1.3, representa un convertidor Buck Intercalado. En este caso, los parámetros utilizados para este modelo son los mismos que para el convertidor Buck y adicionalmente se definen $L_1 = L_2 = \cdots = L_N = 10^{-4}H$.

La Figura 5.8 muestra los voltajes de salida $u_C(t)$ y la corriente de inductancia $i_{L_k}(t)$ $(k = 1 \cdots 4)$ para un convertidor Buck intercalado de cuatro etapas. Comparando estas trayectorias con las del convertidor Buck de la Figura 5.7, se puede ver que, aún cuando la señal de control de ambos modelos fue la misma, la amplitud de onda en el voltaje de salida es sensiblemente menor en el modelo intercalado. Las trayectorias de las corrientes muestran el comportamiento *intercalado* de este circuito.

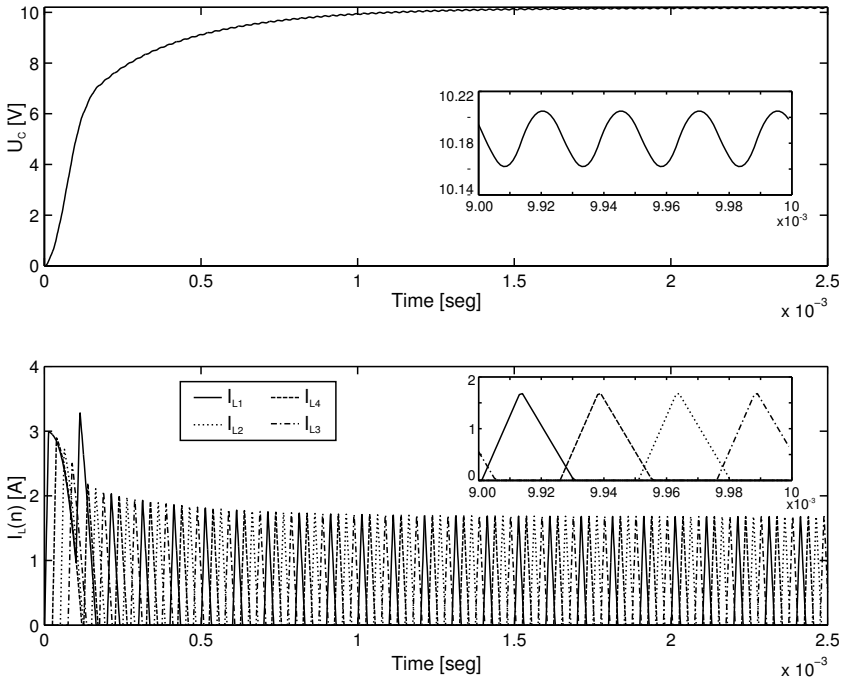


Figura 5.8: Trayectorias de Salida para un Convertidor Buck Intercalado de 4 etapas

La comparación de rendimiento para los distintos métodos es reportada en la Tabla 5.5.

En este caso, los métodos LIQSS fueron más rápidos que DASSL para todas las configuraciones, mostrando aún más diferencias que las observadas en los convertidores Buck, Boost, y Buck–Boost.

Como siempre, los métodos LIQSSS cumplen con los requerimientos de tolerancia de error. Sin embargo, el método DASSL presenta errores inaceptables en este caso. Son 25 veces mayores que la tolerancia definida de 10^{-3} y $14,000$ veces

Tabla 5.5: Convertidor Buck Intercalado de 4–Etapas - Resultados.

Método	**Tolerancia**	**Error Relativo**	**Eval. Jacobiano**	**Eval. f_i**	**Tiempo CPU (ms)**
DASSL	err.tol=$1 \cdot 10^{-3}$	$2{,}50 \cdot 10^{-}2$	12538	435224	29,7604
	err.tol=$1 \cdot 10^{-5}$	$1{,}40 \cdot 10^{-}2$	16433	620754	42,1447
LIQSS2	$\Delta Q_i = 1 \cdot 10^{-3}$	$1{,}26 \cdot 10^{-}3$	–	61870	8,82444
	$\Delta Q_i = 1 \cdot 10^{-5}$	$1{,}62 \cdot 10^{-}5$	–	463170	35,0696
LIQSS3	$\Delta Q_i = 1 \cdot 10^{-3}$	$8{,}04 \cdot 10^{-}4$	–	67425	17,0736
	$\Delta Q_i = 1 \cdot 10^{-5}$	$9{,}89 \cdot 10^{-}6$	–	122958	25,9182

mayores que la tolerancia 10^{-5}. Por lo tanto, los últimos resultados son inválidos para poder compararlos.

La gran magnitud de estos errores es debido a que cada switch permanece en el estado ONpor un período de tiempo muy corto. Debido a esto, un error pequeño en la detección de una discontinuidad puede resultar en un error grande en el voltaje de salida.

Adicionalmente, este modelo es ralo, lo que representa una ventaja para los métodos LIQSS que se ve reflejada en un número sensiblemente menor de evaluaciones de funciones con respecto a DASSL.

Para verificar este hecho, simulamos el modelo variando el número de etapas de 4 a 32. En cada uno de estos experimentos, configuramos la tolerancia para cada método de manera tal que el error medido resulte equivalente. De esta manera, podemos comparar el tiempo de CPU consumido por cada método para simular los modelos y obtener errores del mismo orden.

El tiempo de CPU consumido por cada método para simular un convertidor Buck intercalado de N etapas es ilustrado en la Figura 5.9 (para un error de 10^{-3}) y en la Figura 5.10 (para un error de 10^{-5}).

La Figura 5.9 muestra que, cuando se simula el sistema con un error relativo de 10^{-3}, LIQSS2 muestra el mejor rendimiento, seguido por el método LIQSS3 y DASSL. El método LIQSS2 es 3 veces más rápido que DASSL para 4 etapas y casi 200 veces más rápido que DASSL para 32 etapas.

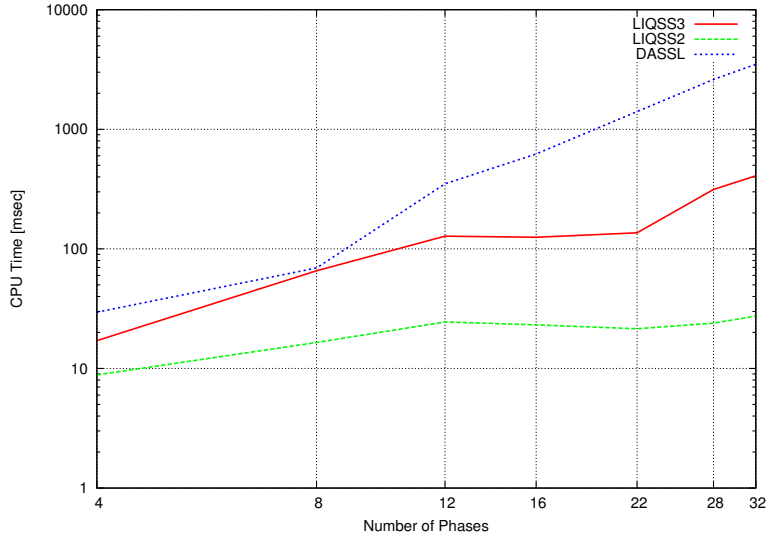


Figura 5.9: Convertidor Buck Intercalado: Tiempo de CPU vs. Nro. Etapas (err=$1 \cdot 10^{-3}$)

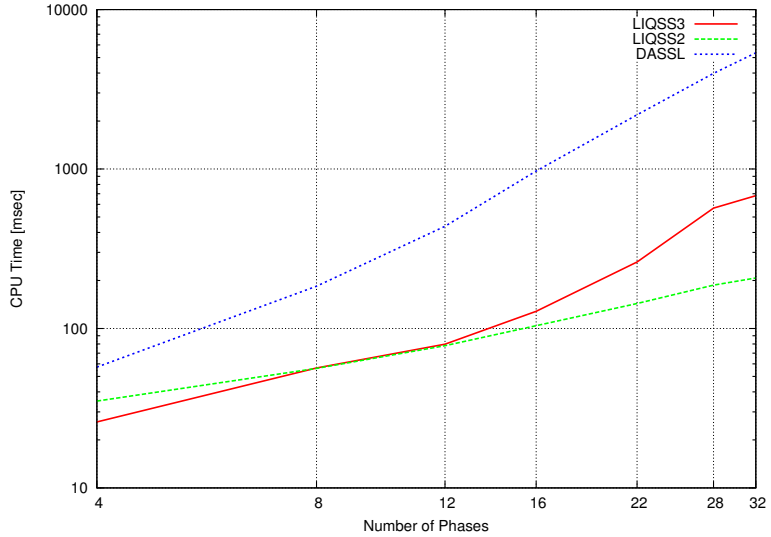


Figura 5.10: Convertidor Buck Intercalado: Tiempo de CPU vs. Nro. Etapas (err=$1 \cdot 10^{-5}$)

El incremento de tiempos de CPU en DASSL puede ser explicado fácilmente. En principio, el tasa de ocurrencia de las discontinuidades crece de manera lineal con el número de etapas y por lo tanto, el tamaño máximo del paso de simulación se debe reducir. En segundo lugar, la dimensión de la EDO crece linealmente con el número de etapas y en consecuencia cada evaluación completa de las funciones

que defines la EDO requiere más cálculos. De esta manera, al utilizar tamaños menores en los pasos de simulación y tener un costo mayor para evaluar la EDO, el costo computacional crece casi cuadráticamente con el número de etapas.

Sin embargo, en los métodos LIQSS, cada paso o discontinuidad provoca solamente cálculos locales lo que resulta en un crecimiento casi lineal de los costos de simulación con respecto al número de etapas.

Los resultados obtenidos para una tolerancia de 10^{-5} son similares, con la excepción de que LIQSS3 es más eficiente que LIQSS2 para un número pequeño de etapas. Para esta configuración del error LIQSS3 puede realizar pasos de simulación más grandes que LIQSS2 lo que reduce considerablemente el número de evaluaciones de función. Sin embargo, cuando el número de etapas crece, las discontinuidades son tan frecuentes que no es posible dar pasos grandes y en consecuencia LIQSS2 es más eficiente que LIQSS3.

De todas maneras, ambos métodos LIQSS son significativamente más eficientes que DASSL para un número de etapas grande.

Los resultados reportados en esta sección muestran que el método de segundo orden LIQSS2 resulta aproximadamente 3 veces más rápidos que DASSL para una tolerancia de 10^{-3} en circuitos de una etapa. Para resultados más precisos (error relativo de 10^{-5}), LIQSS2, LIQSS3 y DASSL consumen tiempos de CPU similares. Sin embargo, en todos los casos los métodos LIQSS cumplen con la tolerancia especificada mientras que DASSL resulta hasta 20 veces más grande. Por lo tanto, los resultados para los métodos LIQSS no solo son más rápidos sino que también son más robustos.

El manejo y detección eficiente las discontinuidades del modelo y el hecho de que los métodos LIQSS no necesitan calcular e invertir matrices Jacobianas para integrar sistemas stiff explican estas ventajas.

El análisis de los convertidores Buck nos permite concluir que ambas ventajas (la velocidad y el error) se vuelven más significativas a medida que se incrementa el tamaño del circuito. Para un convertidor intercalado de 32 etapas, LIQSS2 es aproximadamente 200 veces más rápido que DASSL.

En estos casos, el aprovechamiento de la naturaleza rala del modelo que efectúan los métodos QSS da una ventaja adicional a las ventajas ya mencionadas para circuitos de una etapa.

5.2. Aplicación en Modelos de Reproducción Celular

En esta sección presentamos la aplicación de los métodos QSS a un modelo matemático que representa la evolución temporal de una población de células de levadura de fisión, incluyendo el efecto y la proliferación de marcas epigenéticas a lo largo del proceso. El modelo también contempla la distribución de las marcas epigenéticas entre las células. Con este fin, consideramos el proceso de crecimiento y división de cada célula como así también el la herencia epigenética y el efecto sobre el entorno de esta dinámica. Las condiciones ambientales que incluimos son el *espacio de crecimiento* (que trabaja como una restricción) y la concentración de glucosa cuyo incremento estimula la proliferación de la las células actuando como marca epigénetica. La dinámica de la población de células está definida a partir de las variables N y E que representan el número total de células y el número de células con marcas epigenéticas respectivamente. La evolución del número de células en relación al nacimiento está dada por el modelo desarrollado por Tyson [61], que controla el ciclo de vida de cada célula. Adicionalmente, este modelo se modificó para incluir fallas aleatorias durante la división de las células, herencia epigenética asimétrica y considerando también en los períodos de división el número de células activas y el estímulo de proliferación de las marcas epigenéticas.

La evolución del número de células marcadas epigenéticamente está dada por la probabilidad de herencia, que es modelada como un proceso aleatorio y el efecto de la concentración de glucosa en la proliferación de este tipo de células.

En consecuencia, el sistema que modelamos combina dinámicas diferentes, por un lado el crecimiento de las células es continuo mientras que la herencia, división y las señales de estímulo incorporan elementos discretos y estocásticos. Más aún, consideramos diferentes niveles de jerarquía al modelar la dinámica de cada célula y de manera separada las reglas de interacción entre la población y los efectos de los cambios en el espacio disponible para crecer y las restricciones calóricas.

En las siguientes secciones presentaremos una breve descripción del modelo de Tyson, para luego señalar las modificaciones introducidas y finalmente se muestran y discuten los resultados obtenidos.

5.2.1. Modelo de Tyson

Como mencionamos anteriormente, la evolución del crecimiento de las células es representada mediante el modelo de Tyson que controla el ciclo de reproducción de las mismas.

El ciclo está compuesto por 4 fases consecutivas: La fase de crecimiento (Gap 1), la síntesis, la interfaz (Gap2) y la mitosis. Durante el Gap1 y el Gap2, se utilizan mecanismos de verificación que aseguran que la célula está en condiciones de ingresar en la siguiente etapa. La síntesis involucra todos los procesos que llevan a la replicación del ADN y la mitosis involucra todos los procesos de división necesarios en la célula para generar 2 células hijas. Si la división no ocurre luego de estas etapas, se puede considerar que la célula entra en un estado de reposo durante un periodo de tiempo.

Dado que en este trabajo nos restringimos a células de levadura de fisión, utilizamos el modelo de Tyson para regular la división celular, que es un modelo simple y comúnmente utilizado para organismos pequeños.

En el modelo de Tyson, la división celular se lleva a cabo cuando una célula alcanza una concentración suficientemente alta de MPF (factor de promoción de maduración). El modelo de Tyson es un conjunto de ecuaciones diferenciales (EDOs) que describen la interacción entre las concentraciones de proteína quinasa ($C2$) y un *cyclin* adicional (Y) que forman el MPF. De acuerdo a este modelo, las nuevas subunidades de *cyclin* sintetizadas (yP) se combinan con subunidades preexistentes de $C2$ para formar un complejo inactivo de MPF ($pMPF$).

Las ecuaciones que definen este modelo son las siguientes:

$$aMPF = pMPF \times (0{,}018 + 180 \times (\frac{MPF}{C2 + CP + pMPF + MPF})^2) \quad (5.13)$$

$$aCPY = 200 \times CP \times Y \quad (5.14)$$

$$k6 = 2{,}5 \times exp(-0{,}693 \times mod(time, TD)) \quad (5.15)$$

$$\dot{MPF} = aMPF \quad (5.16)$$

$$\dot{C2} = k6 \times MPF - 10^6 \times C2 + 10^3 \times CP \quad (5.17)$$

$$\dot{CP} = 10^6 \times C2 - 10^3 \times CP - aCPY \quad (5.18)$$

$$\dot{pMPF} = 200 \times CP \times Y - aMPF \quad (5.19)$$

$$\dot{Y} = 0{,}015 - aCPY \quad (5.20)$$

$$\dot{yP} = k6 \times MPF - 0{,}6 \times yP \quad (5.21)$$

Cabe mencionar que el este modelo incluye la posibilidad de seleccionar la fase de crecimiento de las células en relación a el desarrollo del organismo o en relación al avance del cultivo celular. En particular, en la etapa de crecimiento controlado, donde la división de las células son reguladas por el tiempo de duplicación de masa, se introduce el parámetro $k6$. Este parámetro representa la tasa de caída del complejo activo MPF como función del tiempo. La definición del parámetro $k6$ incluye un nuevo parámetro TD, que es fijo, y no existen indicaciones sobre cómo seleccionar el valor de dicho parámetro.

5.2.2. Modificaciones Introducidas

En esta sección presentamos las modificaciones realizadas al modelo de Tyson, un resumen de las diferencias existentes como así también los valores iniciales de los parámetros del modelo se muestran en la Tabla 5.6. En este caso, consideramos que una versión del modelo de Tyson donde se controla el crecimiento y fallas en el proceso de división que incluyen la posibilidad de no pasar el Gap1, ingresando de esta manera en un estado de reposo y luego de un periodo de tiempo, volver al ciclo original nuevamente. Adicionalmente, consideramos que la célula puede morir en cualquier momento del ciclo. Esta dinámica se muestra en la Figura 5.11 Cabe mencionar que consideramos que el tiempo de duplicación TD varia en el tiempo dependiendo del espacio para crecer del cultivo y de la proliferación del estímulo a las células con marcas epigenéticas.

Cada céclula es caracterizada en el tiempo por la variable continua MPF, y las variables discretas *epig* y *active* que describen si la célula está marcada epigenéticamente y si está activa respectivamente. En nuestro modelo, la relación entre una célula y el entorno está dado por el proceso de división, que produce 2 nuevas células. Inicialmente, el sistema comienza con pequeño grupo de células vivas y activas. Cuando una célula se divide, 2 células hijas son creadas en lugar de la madre, donde para las células hijas, la dinámica de la variable *epig* depende de la madre, que transmite la marca epigenética de manera aleatoria. Si la marca está ausente en la madre, la herencia no es posible, pero si la célula madre tiene la marca epigenética, cada hija hereda la marca con probabilidad $Pepig$. Generando de esta manera la herencia asimétrica de la marca epigenética.

Cuando nace una célula, se encuentra activa ($active = 1$) y comienza el el ciclo de vida de la misma, luego, el crecimiento de la célula es modelado por la dinámica de la variable MPF de acuerdo con el modelo de Tyson y cuando el valor de MPF supera el umbral $maxPMF$ (que es un parámetro del modelo) la célula intenta dividirse con probabilidad pS o entra en un estado de reposo con probabilidad $1 - pS$. Cuando la división falla, la variable MPF toma el valor 0 y la variable $k6$ toma el valor 2,5, finalmente, luego de un periodo de tiempo fijo, se reinicia el ciclo de la célula. Cabe mencionar que en el caso de las células marcadas epigenéticamente, la probabilidad de reproducción es 0,1 más baja que en las células no marcadas.

Finalmente, consideramos que las células pueden morir en cualquier momento del ciclo de vida, pero la probabilidad aumenta a medida que la maduración de la célula aumenta. Por lo tanto, para cada célula, modelamos el tiempo de vida como una variable aleatoria distribuida uniformemente tomando valores en el intervalo $[LT - sd, LT + sd]$.

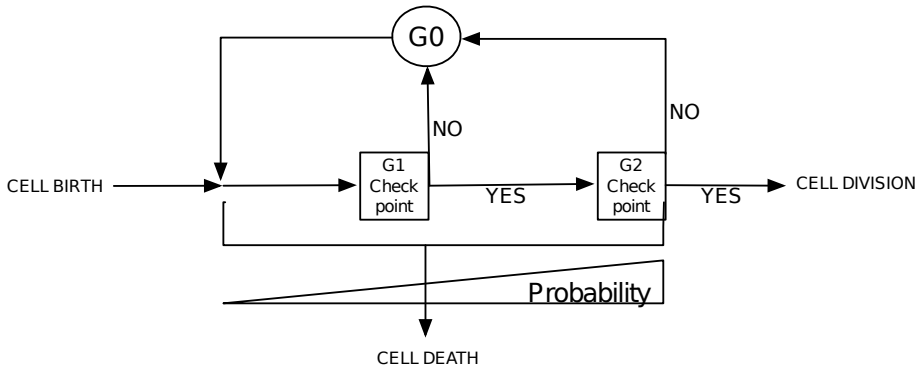


Figura 5.11: Modelado a Nivel Celular

Tabla 5.6: Especificación del Modelo

Notación	Descripción	Original	Modificado
MPF	Nivel de crecimiento de una célula: Concentración del complejo activo MPF	0	0
$C2$	Concentración de la proteina quinasa cdc2	0,75	$0{,}75 + rand[-0{,}02, 0{,}02]$
CP	Concentración de la proteina quinasa fosforilada cdc2	0	$0{,}75 + rand[-0{,}02, 0{,}02]$
$pMPF$	Concentración del factor de promoción de premaduración	0,25	$0{,}25 + rand[-0{,}02, 0{,}02]$
y	Concentración de cyclin	0	0
yP	Concentración de cyclin fosforilado	0	0
maxMPF	Umbral del nivel de crecimiento para la división celular	0,19	0,19
$k6$	Tasa de caida para MPF	$k6 = 2{,}5 \times exp(-0{,}693 \times mod(time, TD)))$	2 o 2,5 generado aleatoriamente en cada célula
LT	Tiempo de vida promedio de una célula	Ausente	150
sd	Desviación estandard de LT	Ausente	100
$maxN$	Umbral para las células activas, constante que limita el espacio para el crecimiento	Ausente	500
$room$	Espacio para el crecimiento	Ausente	$maxN - 64$,$maxN - N$
$room_2$	Espacio para duplicar TD	Ausente	64
TD	Tiempo para duplicar TD	57	$TD = ROOM\_EFFECT \times MARK\_PROLIF$ $ROOM\_EFFECT = 57 \times \frac{room_2 + room}{room}$ $MARK\_PROLIF = (epig = 1 \wedge time >= st\_time) \times st\_coeff$
G	Glucosa disponible	Ausente	$G = (time < st\_time) \times 2 + (time >= st\_time) \times 0{,}5$
$epig$	Marca epigenética	Ausente	1 si la marca está presente 0 en caso contrario
st_time	Tiempo de estimulo para la proliferación de las marcas epigenéticas	Ausente	500
st_coeff	Coeficiente de estimulo de las marcas epigenéticas	Ausente	0,2
$active$	Estado de una célula	Ausente	1 si la célula está activa 0 en caso contrario
PS	Estado de una célula	Ausente	0,7
sP	Estado de una célula	Ausente	$pS = PS - 0{,}1 \times epig$
$Pepig$	Estado de una célula	Ausente	0,8
N	Estado de una célula	Ausente	64
E	Estado de una célula	Ausente	32

5.2.3. Resultados

El modelo presentado tiene las siguientes características:

- cada célula es representada por una ODE stiff no lineal de orden 6
- cada célula contiene 2 discontinuidades que involucran las variables $k6$ y MPF
- para poder realizar experimentos con resultados significativos, el sistema debería estar compuesto por cientos de células como mínimo, lo que lleva a un sistema de alto orden con una estructura rala

Estas características imponen limitaciones a los métodos de integración numérica clásicos (que deben ser implícitos ya que el sistema es stiff), dado que si tomamos un conjunto de 1000 células por ejemplo, el sistema contiene 6000 variables de estado y debe invertir una matriz de orden 6000×6000 en cada paso. En principio, podemos decir que la simular un modelo de estas características tiene un costo computacional alto. En las siguientes secciones presentaremos resultados obtenidos al comparar los métodos QSS con los métodos de integración clásicos, y luego veremos los resultados obtenidos al simular el modelo con diferentes configuraciones de los parámetros.

Comparación con Métodos de Integración Clásicos

Para corroborar las hipótesis mencionadas anteriormente, realizamos un primer experimento que compara el el rendimiento de los métodos QSS, utilizando LIQSS2, contra el algoritmo implícito DASSL.

Todas las simulaciones que realizadas fueron ejecutadas en una PC con un procesador Intel(R) Core(TM) i7-2600 @ 3.4GHz con 4GB de memoria RAM. Con una tolerancia de $1e-4$ para DASSL y una tolerancia relativa de $1e-3$ y una tolerancia absoluta de $1e-5$ para LIQSS2 y un tiempo final de simulación de 2000 minutos.[1]

Los resultados obtenidos para sistemas de diferente tamaño se reportan en la Tabla 5.7, donde podemos ver que el tiempo de ejecución requerido por el método LIQSS2 es significativamente menor que el requerido por DASSL. Aún

[1] Utilizamos la misma configuración para el método LIQSS2 para todas las simulaciones presentadas en esta sección.

en sistemas pequeños de 100 células, DASSL necesita 139 segundos contra 1,45 segundos requeridos por LIQSS2, mientras que para sistemas de 500 células o más la simulación con DASSL falló. A partir de estos experimentos podemos ver las ventajas que presentan los métodos QSS para simular este tipo de sistemas.

Tabla 5.7: Tiempo de CPU DASSL vs LIQSS2

Número de Células	Tiempo CPU DASSL	Tiempo CPS (seg) LIQSS2
1	0,079	0,016
5	0,255	0,057
10	0,718	0,0139
50	25,8	0,789
100	139	1,445
500	-	9,114
1000	-	26,995

Análisis del Modelo de Reproducción Celular

En esta sección se presentan y analizan de manera general los resultados obtenidos al simular el modelo, los valores iniciales utilizados y los parámetros de referencia son los presentados en la Tabla 5.6. En todos los experimentos, se utilizó el método LIQSS2 con la misma configuración para las tolerancias que en la sección anterior. Para cada una de las configuraciones analizadas se realizaron 50 simulaciones y los resultados obtenidos se muestran en las Figuras 5.12 y 5.13 donde se indica la media y el desvío estandard para las variables N y E.

La probabilidad de éxito de las células marcadas epigenéticamente fue seleccionada de manera tal que menor que las células no marcadas, por lo que en general tienen un tiempo de vida menor.

En la Figura 5.12 podemos ver que están presentes las fases de crecimiento del cultivo indicadas en [49], de hecho, para el número total de células N, la fase de retraso inicial es seguida por una fase exponencial y luego por una fase estacionaria, dado que la tasa de crecimiento disminuye a medida que el número de células total aumenta. Adicionalmente, cuando el número de células es lo suficientemente grande, el tiempo requerido para la división celular se vuelve mayor al tiempo de vida promedio de una célula y como consecuencia, las células comienzan a morir sin reproducirse. Luego de esto, la tasa de crecimiento se vuelve lo suficientemente

grande como para incrementar el número de células vivas nuevamente oscilando cerca del umbral $maxN$. Un comportamiento similar se puede observar en el número de células marcadas epigenéticamente E, pero en este caso la tasa de crecimiento es menor dado que este tipo de células se incrementa solamente cuando la célula madre tiene la marca y ambas células hijas heredan la marca. La fase estacionaria es seguida por una fase donde las células comienzan a morir (luego de alrededor de 200 minutos) debido a que al llegar al tiempo de vida promedio para una célula, la tasa de crecimiento de estas células no es lo suficientemente grande.

En las siguientes secciones analizaremos los resultados obtenidos para diferentes configuraciones de los parámetros del modelo.

Control de las Células Marcadas Epigenéticamente

En esta sección analizaremos los resultados obtenidos al modificar las condiciones en las cuales las células marcadas epigenéticamente se reproducen. Los resultados obtenidos para las simulaciones realizadas utilizando los parámetros definidos en la Tabla 5.6 se muestran en la Figura 5.12, donde la proliferación de las células marcadas epigenéticamente es activada en el tiempo $tiem = 500$ con un coeficiente igual a $st\_coeff = 0{,}2$. La probabilidad de éxito de la división celular pS, la cantidad inicial de células marcadas y la probabilidad de herencia para las células marcadas $Pepig$ juegan un rol importante en la dinámica de la población de este tipo de células. No se observan cambios significativos en la cantidad total de células N al modificar estos parámetros.

Podemos ver que un incremento en pS de 0,7 a 0,9 (Figuras 5.12a,5.12b,5.12c) tiene un efecto significativo en la recuperación final de E al llegar al tiempo final de simulación. El efecto del parámetro $Pepig$ sobre la dinámica de la población puede verse en las Figura 5.12a y 5.12d, donde al incrementar su valor a 0,9 la recuperación final aumenta de manera significativa. Bajo las mismas condiciones E es también mayor antes del tiempo de estimulo $time = 500$.

Al incrementar el número inicial de células marcadas (%75 en la Figura 5.12e y %100 en la Figura 5.12f), la recuperación final de E no se modifica de manera significativa, pero si lo hace el valor previo al estímulo.

Al combinar los 3 parámetros, podemos ver que los mejores resultados para la recuperación de las células marcadas epigenéticamente se obtienen con los

parámetros $st\_coeff = 0{,}2$ en el tiempo $time = 500$, $pS = 0{,}7$, $Pepig = 0{,}9$ y un porcentaje de células marcadas inicialmente de %75.

Adicionalmente, realizamos simulaciones donde todas las células son marcadas epigenéticamente al inicio, con $Pepig = 1$ y $pS = 1$, los resultados se muestran en la Figura 5.12g. Los resultados obtenidos en este caso fueron mejores, como se podía esperar, pero el modelo no es realista dado que la herencia epigenética y división celular perfecta no es posible, principalmente debido a la complejidad del control enzimático de la herencia epigenética.

Estímulo a las Células Marcadas Epigenéticamente

En esta sección analizamos los resultados obtenidos al modificar el estímulo a la proliferación de las células marcadas epigenéticamente, los resultados obtenidos se muestran al la Figura 5.13. En estos experimentos, junto a la probabilidad de exito de la división celular pS, la cantidad de células marcadas inicialmente y la probabilidad de herencia epigenética, modificamos el coeficiente de estímulo a las células marcadas y el tiempo en el que se activa. Como se podía esperar, el estímulo a la proliferación de las células marcadas nos permite incrementar el número de este tipo de células.

Este efecto no cambia significativamente si vemos la recuperación final de E (aproximadamente %20 de N) al variar el tiempo en el que se activa el estímulo. Los resultados para valores de $st\_coeff = 100, 250, 500, 750$ pueden verse en las Figuras 5.13a-5.13d, al modificar el coeficiente de estímulo a 0,1 (Figura 5.13e) el valor de E decae a lo largo del tiempo llegando a un valor final cercano a 0.

El efecto de la proliferación del estímulo se percibe si el valor del estímulo se modifica a 0,3, en las Figuras 5.13f y 5.13h se pueden ver los resultados para los valores de $st\_time = 500$ y $st\_time = 100$. Bajo estas condiciones E lleva a recuperar el %60 de N.

Por otro lado, si asignamos $Pepig = 0{,}9$ y la proliferación es estimulada en $st\_time - 100$ con $st\_coeff = 0{,}2$ (Figura 5.13g) la recuperación final de E es significativamente mayor como podía esperarse. Por último, una configuración de parámetros $pS = 0{,}9$, $Pepig = 0{,}8$, $st\_coeff - 0{,}2$ y $st\_time = 100$ también observamos que la recuperación final de E mejora de manera significativa.

Estos resultados remarcan la importancia que tiene el momento en el que se estimula la proliferación de las células.

Como conclusión, este trabajo se implementó un modelo dinámico que muestra las ventajas de utilizar los métodos QSS en sistemas híbridos y stiff que reducen significativamente los tiempos de simulación con respecto a los métodos de integración clásicos permitiendo de esta manera realizar experimentos nuevos y poder analizar los resultados en profundidad en diferentes escenarios.

Los resultados obtenidos concuerdan con la literatura, desde las curvas de crecimiento, los rangos en los cultivos y el porcentaje funcional de células. Adicionalmente, se obtuvieron predicciones sobre el efecto de la proliferación del estímulo a las células marcadas para distintos tiempo.

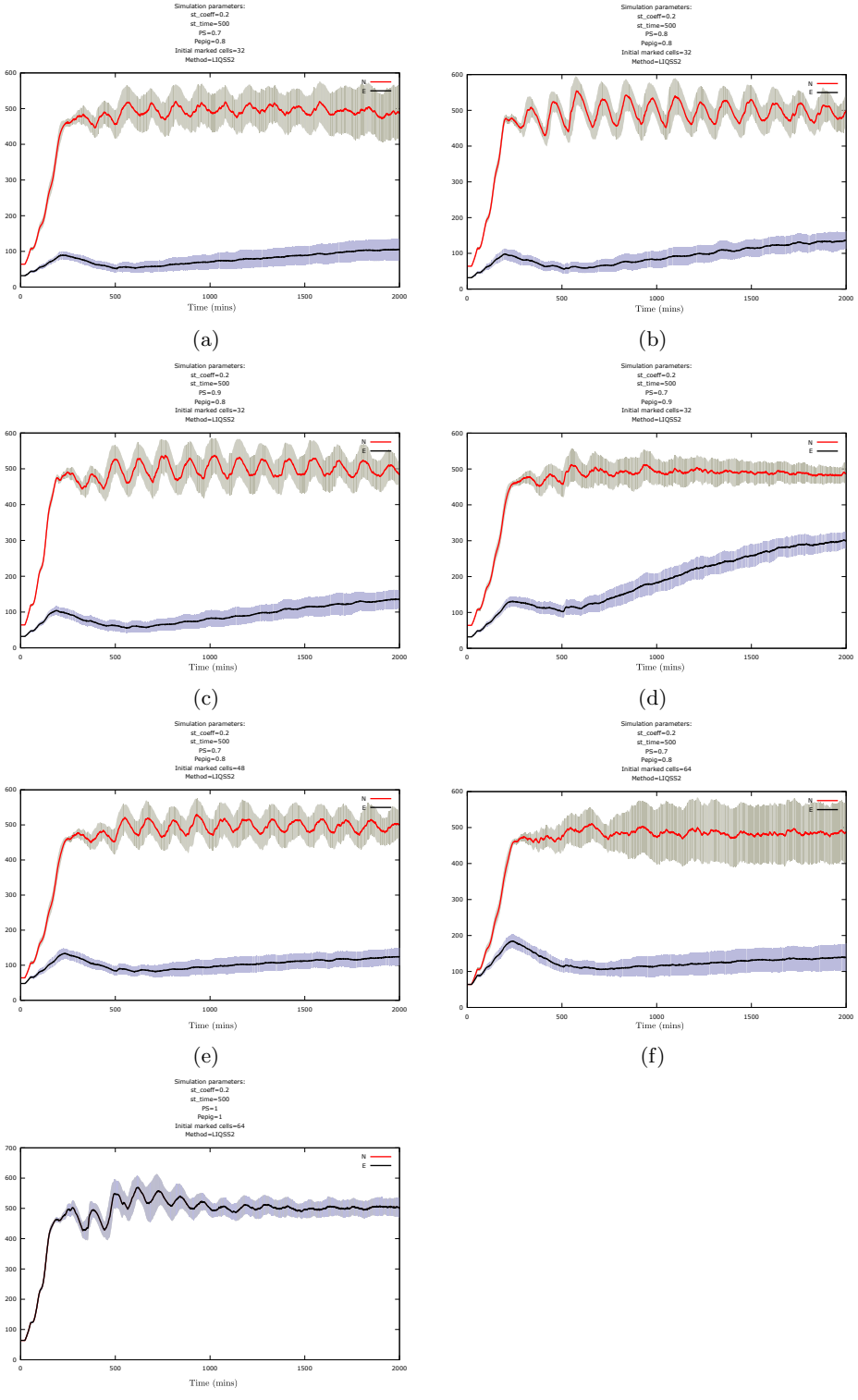
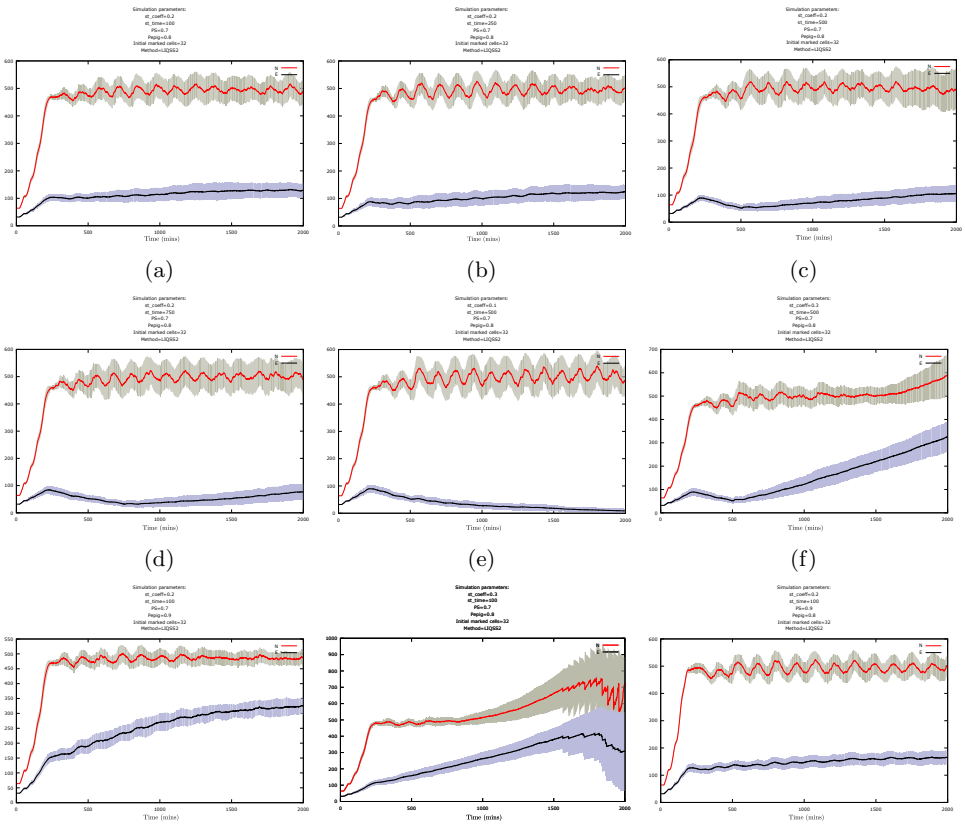


Figura 5.12: Modelo de Reproducción Celular - Resultados



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Figura 5.13: Modelo de Reproducción Celular - Resultados

5.3. Aplicación en Modelos de Advección–Difusión–Reacción

En el trabajo presentado en [6] se muestran los resultados obtenidos al simular modelos de Advección–Difusión–Reacción utilizando el motor de simulación autónomo QSS. En este contexto se estudia los efectos de reemplazar la discretización temporal por la cuantificación espacial en un modelos de Advección–Difusión–Reacción (ADR) de una dimensión (1D). Con este fin, la ecuación ADR 1D se discretiza primero en el espacio utilizando una grilla regular, para obtener un conjunto de ecuaciones diferenciales ordinarias (EDOs). Luego comparamos la eficiencia de la simulación de este modelo utilizando algoritmos de tiempo discreto clásicos y métodos QSS.

El análisis del rendimiento es realizado para diferentes parámetros de difusión y reacción y también variando el refinamiento de la discretización espacial.

5.3.1. Modelo de Advección-Difusión-Reacción

En esta sección describimos el modelo de advección–difusión–reacción utilizado para las simulaciones. Supongamos que $u(x,t)$ representa la concentración de alguna especie en la coordinada espacial x en el tiempo t. Luego, el proceso de advección y difusión 1D [32] puede ser descripto por la siguiente ecuación diferencial parcial (EDP):

$$\frac{\partial u(x,t)}{\partial t} + a\frac{\partial u(x,t)}{\partial x} = d\frac{\partial^2 u(x,t)}{\partial x^2} \tag{5.22}$$

Tomando en cuenta que la especie sufre una reacción química, incluimos un término de reacción *no lineal* siguiendo la ecuación de Zeldovich [28] de la siguiente manera:

$$\frac{\partial u(x,t)}{\partial t} + a\frac{\partial u(x,t)}{\partial x} = d\frac{\partial^2 u(x,t)}{\partial x^2} + r(u(x,t)^2 - u(x,t)^3) \tag{5.23}$$

Este es el modelo que utilizaremos en este trabajo. Aquí a, d y r son parámetros que expresan los coeficientes de advección, difusión y reacción respectivamente.

Consideraremos que el dominio espacial está limitado al intervalo $0 \leq x \leq 10$ y que las condiciones de frontera son:

$$u(x=0,t) = 1; \quad \frac{\partial u(x=10,t)}{\partial x} = 0; \tag{5.24}$$

En las simulaciones, trabajaremos con las siguientes iniciales:

$$u(x, t = 0) = \begin{cases} 1 & \text{if } x < 2 \\ 0 & \text{otherwise} \end{cases} \tag{5.25}$$

5.3.2. Discretización MOL del Modelo ADR

Para poder discretizar el problema utilizando el Método de Líneas, utilizaremos una grilla ancho regular

$$\Delta x = \frac{10}{N} \tag{5.26}$$

donde N es el número de puntos de la grilla.

El término de advección de la Ec. (5.23) $\frac{\partial u(x,t)}{\partial x}$ se reemplaza de la siguiente manera:

$$\frac{\partial u}{\partial x}(x = x_i, t) \approx \frac{u_i - u_{i-1}}{\Delta x} \tag{5.27}$$

para $i = 1, \cdots, N$, donde

$$u_i(t) \approx u(x_i, t) \tag{5.28}$$

es la i–ésima variable de estado de la EDO resultante y

$$x_i = i \cdot \Delta x \tag{5.29}$$

es el i–ésimo punto de la grilla.

Tomando en cuenta las condiciones de frontera de la Ec. (5.24) en el punto $x = 0$, tenemos que $u_0 = 1$.

Discretizaremos el término de difusión reemplazando la expresión $\frac{\partial^2 u}{\partial x^2}$ por:

$$\frac{\partial^2 u}{\partial x^2}(x = x_i, t) \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} \tag{5.30}$$

para $i = 1, \cdots, N - 1$.

Para el último punto de la grilla, teniendo en cuenta la condición de borde simétrica de la Ec. (5.24) para $x = 10$, podemos reemplazar:

$$\frac{\partial^2 u}{\partial x^2}(x = x_N, t) \approx \frac{u_{N-1} - 2u_N + u_{N-1}}{\Delta x^2} \tag{5.31}$$

Reemplazando las Ecs. (5.27)–(5.31) en la Ec. (5.23) se obtiene el siguiente conjunto de EDOs:

$$\dot{u_i} = -a\frac{(u_i - u_{i-1})}{\Delta x} + d\frac{(u_{i+1} - 2u_i + u_{i-1})}{\Delta x^2} + r(u_i^2 - u_i^3) \tag{5.32}$$

para $i = 1, \cdots, N-1$ y

$$\dot{u_N} = -a\frac{(u_N - u_{N-1})}{\Delta x} + d\frac{(2u_{N-1} - 2u_N)}{\Delta x^2} + r(u_N^2 - u_N^3) \tag{5.33}$$

5.3.3. Resultados

En esta sección comparamos el rendimiento de los distintos métodos de integración numéricos aplicados al problema ADR semi–discretizado mediante el método de líneas. Con este fin, el modelo resultante de la Ec. (5.32) es simulado con configuraciones diferentes para los parámetros utilizando LIQSS2, DASSL, Radau5 and DOPRI.

- Los resultados para DASSL fueron obtenidos utilizando el código Fortran DASPK [9].
- Los resultados para DOPRIy Radau5 fueron obtenidos utilizando la implementación C++ disponible en `http://www.unige.ch/~hairer/software.html`, escrito por Blake Ashby.
- Los resultados para LIQSS2 fueron obtenidos utilizando el motor de simulación autónomo QSS.
- Para todas las simulaciones se utilizó una computadora con un procesador Intel i7-3770@3.40GHz con un sistema operativo Linux (Ubuntu).
- Los errores en todos los casos fueron calculados contra una solución de referencia obtenida utilizando DOPRI con una tolerancia de $(1 \cdot 10^{-10})$. Consideramos también el error en el último de la grilla $u_N(t)$ dado que este punto acumula el error de todos los puntos previos. El error promedio fue calculado en 5000 puntos equidistantes por $\sum_{i=1}^{5000} |u_{N_{ref}}(t_i) - u_{N_{sim}}(t_i)|/5000$ mientras que el error máximo se define como $\text{máx}_i(\{|u_{N_{ref}}(t_i) - u_{N_{sim}}(t_i)|\})$ donde $u_{N_{ref}}(t)$ es la referencia y $u_{N_{sim}}(t)$ es la trayectoria simulada.
- No se calcularon errores de consistencia debido a errores en la discretización espacial. Solamente estamos interesados en el error de la integración de la EDO.
- En todos los escenarios utilizamos tolerancia relativa de $1 \cdot 10^{-3}$ y una tolerancia absoluta de $1 \cdot 10^{-4}$.

- El tiempo final de simulación es de $t = 10$ segundo.
- En todos los casos, se reporta el número de evaluaciones de funciones correspondientes a los componentes escalares.

Variación en el Tamaño de la grilla Δx

En este escenario estudiamos el costo computacional y el error introducido por los distintos algoritmos para diferentes cantidades de puntos (N) en la grilla. El resto de los parámetros se mantiene fijo, $a = 1, d = 1 \cdot 10^{-4}, r = 1000$. El número Péclet resultante es $a/d = 10000$.

El objetivo de este experimento es establecer que tan eficientes son los métodos para manejar modelos resultantes de grillas más refinadas, que son utilizados frecuentemente para reducir los errores de consistencia introducidos por el método de lineas.

La Figura 5.14 compara los tiempos de CPU consumidos por DASSL, DOPRI, Radau5 y LIQSS2 a medida que crece N. La Tabla 5.8 se reportan los resultados obtenidos junto con el número de evaluaciones de funciones de cada método.

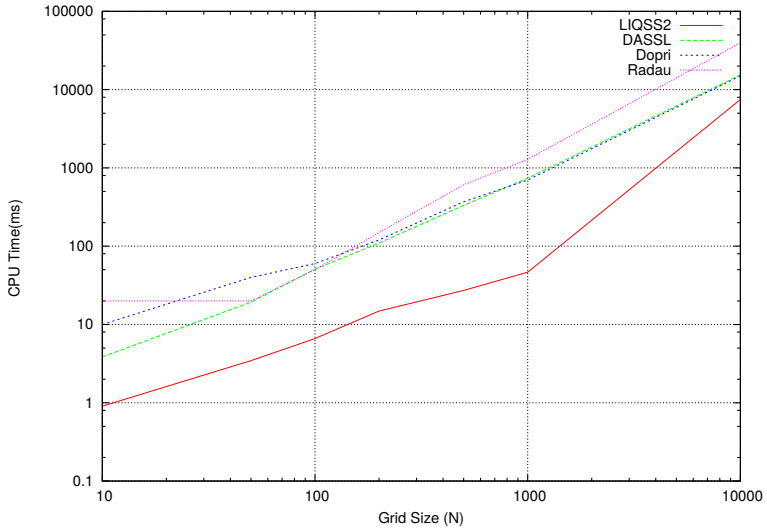


Figura 5.14: CPU time vs. N con $a = 1, d = 1 \cdot 10^{-4}, r = 1000$

En este caso, el método LIQSS2 es el más eficiente en todos los casos. Se puede notar que hasta $N = 1000$, el tiempo de CPU crece de manera casi lineal en N para LIQSS2. Con 1000 LIQSS2 es 15 veces más rápido que DOPRI y DASSL y 27 veces más rápido que Radau.

Tabla 5.8: Tiempo de CPU (ms) y número de evaluaciones de función para diferentes valores de N con $a = 1, d = 1 \cdot 10^{-4}, r = 1000$

N	LIQSS2		**DASSL**		**DOPRI**		**Radau5**	
	tiempo	**eval.**	**tiempo**	**eval.**	**tiempo**	**eval.**	**tiempo**	**eval.**
10	9,04e−1	5,99e3	3,85e0	8,47e3	1,00e1	1,88e5	2,00e1	1,02e4
50	3,45e0	2,79e4	1,93e1	1,02e5	4,00e1	1,06e6	2,00e1	1,74e5
100	6,62e0	5,38e4	5,11e1	3,29e5	6,00e1	2,45e6	5,00e1	5,02e5
200	1,48e1	1,17e5	1,10e2	8,85e5	1,20e2	5,17e6	1,50e2	1,57e6
500	2,73e1	3,16e5	3,33e2	2,61e6	3,70e2	1,73e7	6,10e2	6,06e6
1000	4,66e1	6,05e5	7,41e2	5,64e6	7,00e2	3,54e7	1,29e3	1,23e7
10000	7,49e3	1,07e8	1,54e4	1,08e8	1,50e4	7,41e8	3,97e4	4,04e8

A pesar de que la presencia del término de reacción implica que el sistema es stiff, el algoritmo explícito DOPRI es capaz de obtener tiempos de simulación razonables. De hecho, efectúa muchas evaluaciones de función, pero su costo por paso es bajo por lo que se obtiene un rendimiento similar a DASSL.

Se debe mencionar que los códigos DASPK y Radau5 son adecuados para modelos de gran escala. Más aún, explotan el hecho de la matriz Jacobiana es tridiagonal en este caso en particular. De otra manera, su costo computacional se incrementaría de manera cúbica con el tamaño de N.

En la Tabla 5.9 se muestran los errores máximos y medios obtenidos por los métodos.

Tabla 5.9: Error Máximo y Promedio para diferentes valores de N con $a = 1, d = 1 \cdot 10^{-4}, r = 1000$

N	LIQSS2		**DASSL**		**DOPRI**		**Radau5**	
	Máx.	**Prom.**	**Máx.**	**Prom.**	**Máx.**	**Prom.**	**Máx.**	**Prom.**
10	5,9e−2	2,8e−3	7,4e−1	7,9e−4	3,9e−3	8,7e−4	2,5e−3	2,7e−6
50	8,4e−2	8,1e−4	7,0e−1	6,8e−4	2,2e−2	1,9e−3	3,5e−3	6,7e−6
100	1,2e−1	1,7e−4	6,6e−1	6,1e−4	3,8e−2	2,5e−3	9,1e−3	2,9e−5
200	1,6e−1	1,8e−3	7,5e−1	7,6e−4	9,8e−2	3,0e−3	3,0e−3	1,3e−5
500	1,8e−1	1,1e−3	5,3e−1	4,0e−4	3,9e−2	3,8e−3	1,7e−2	1,4e−5
1000	2,1e−1	1,3e−3	3,4e−2	2,4e−5	5,8e−2	4,8e−3	4,9e−2	3,3e−5
10000	5,9e−1	8,1e−4	1,0e0	1,4e−3	1,9e−1	6,6e−3	3,0e−1	1,3e−4

Los errores promedios de LIQSS2, DASSL y DOPRI son similares, y son consistentes con las tolerancias exigidas. Sin embargo, Radau es aproximadamente dos órdenes de magnitud más preciso. Esto es porque la implementación es extremadamente conservadora con respecto al error exigido.

El error absoluto máximo es alto para todos los algoritmos (excepto para Radau). La razón para esto es que la solución es una onda que viaja con una pendiente larga. Por lo tanto, un error muy pequeño en la velocidad de la onda causa un error muy grande en el valor de u_i cuando la onda pasa a través de el i–ésimo punto de la grilla.

Variación del Tamaño de la Grilla Δx Sin Difusión

En este escenario estudiamos el costo computaciones para distintos número de puntos N en la grilla sin el término de difusión ($d = 0$), i.e. un problema de advección–reacción puro. El resto de los parámetros son: $a = 1, r = 1000$. Los errores no son reportados en este caso dado que son similares a los obtenidos en el escenario previo.

La Figura 5.15 compara los tiempos de CPU de DASSL, Radau5, DOPRI y LIQSS2. La Tabla 5.10 reporta los resultados obtenidos junto al número de evaluaciones de funciones escalares.

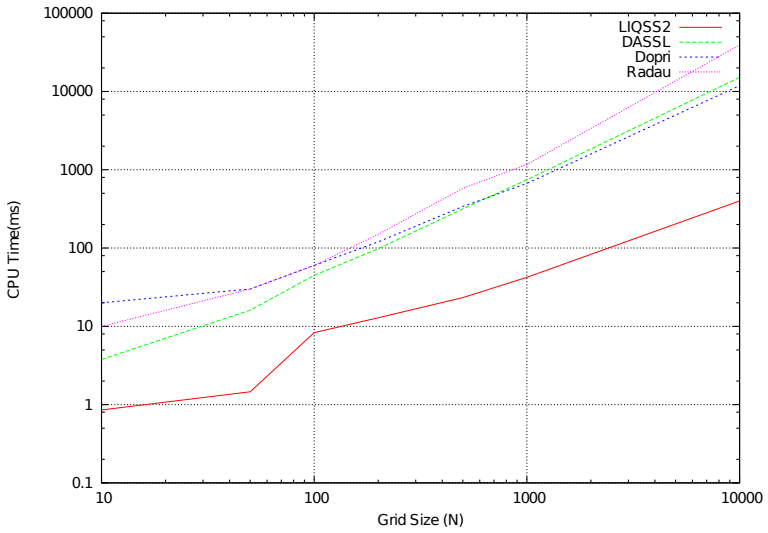


Figura 5.15: Tiempo de CPU vs N con $a = 1, d = 0, r = 1000$

Los resultados en este caso son similares a los obtenidos con $d = 1 \cdot 10^{-4}$, y en este caso para $N = 10000$ LIQSS2 es aproximadamente 30 veces más rápido que DOPRI, 38 veces más rápido que DASSL y 98 veces más rápido que Radau.

Tabla 5.10: Tiempo de CPU (ms) y número de evaluaciones de función para diferentes valores de N con $a = 1, d = 0, r = 1000$

N	LIQSS2		**DASSL**		**DOPRI**		**Radau5**	
	tiempo	**eval.**	**tiempo**	**eval.**	**tiempo**	**eval.**	**tiempo**	**eval.**
10	8,54e−1	6,14e3	3,78e0	8,47e3	2,00e1	1,88e5	1,00e1	1,02e4
50	1,46e0	2,81e4	1,61e1	1,02e5	3,00e1	1,06e6	3,00e1	1,74e5
100	8,30e0	5,92e4	4,49e1	3,12e5	6,00e1	2,46e6	6,00e1	5,02e5
200	1,28e1	1,04e5	9,79e1	8,70e5	1,20e2	5,16e6	1,50e2	1,57e6
500	2,33e1	2,70e5	3,17e2	2,74e6	3,40e2	1,65e7	5,80e2	6,06e6
1000	4,23e1	5,49e5	7,44e2	5,90e6	6,70e2	3,54e7	1,17e3	1,19e7
10000	3,99e2	6,58e6	1,51e4	1,04e8	1,19e4	6,43e8	3,93e4	4,23e8

Variación del Término de Reacción r

En este caso consideramos la variación del término de reacción r con los restantes parámetros definidos como: $a = 1, d = 1 \cdot 10^{-4}, N = 1000$.

La Figura 5.16 compara los tiempos de CPU de DASSL, Radau5, DOPRI y LIQSS2 a medida que crece r. La Tabla 5.11 reporta los resultado obtenidos junto a el número de evaluaciones de funciones escalares. Nuevamente, no reportamos los errores dado que son muy similares a los obtenidos previamente.

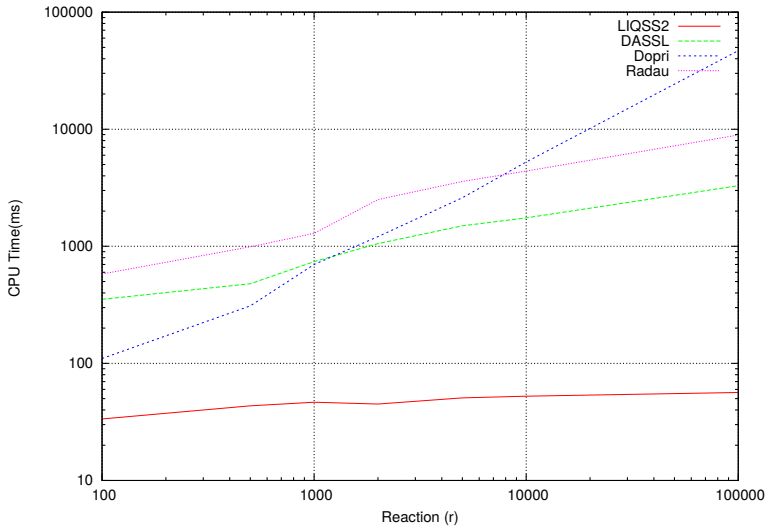


Figura 5.16: CPU time vs. r con $a = 1, d = 1 \cdot 10^{-4}, N = 1000$

En este escenario, LIQSS2 muestra una ventaja significativa sobre el resto de los métodos, su rendimiento no es afectado por el crecimiento del término de reacción r.

Tabla 5.11: Tiempo de CPU (ms) y número de evaluaciones de función para diferentes valores de r con $a = 1, d = 1 \cdot 10^{-4}, N = 1000$

N	LIQSS2		**DASSL**		**DOPRI**		**Radau5**	
	tiempo	**eval.**	**tiempo**	**eval.**	**tiempo**	**eval.**	**tiempo**	**eval.**
100	3,35e1	5,93e5	3,53e2	1,94e6	1,10e2	5,38e6	5,80e2	5,50e6
500	4,34e1	5,45e5	4,79e2	3,68e6	3,10e2	1,61e7	9,90e2	9,18e6
1000	4,66e1	6,05e5	7,41e2	5,64e6	7,00e2	3,54e7	1,29e3	1,23e7
2000	4,49e1	6,51e5	1,05e3	1,00e7	1,21e3	6,37e7	2,51e3	2,41e7
5000	5,08e1	6,84e5	1,50e3	1,71e7	2,60e3	1,41e8	3,58e3	3,52e7
10000	5,25e1	7,04e5	1,75e3	2,14e7	5,25e3	2,78e8	4,39e3	4,49e7
100000	5,64e1	7,68e5	3,29e3	5,12e7	4,68e4	2,71e9	8,93e3	9,43e7

El resto de los métodos presentan varias desventajas en este caso. DOPRI, al ser explícito, limita el tamaño del paso de simulación a la región de estabilidad que se reduce linealmente con r. Por lo tanto, el costo computacional crece linealmente con r.

DASSL y Radau no tienen problemas de estabilidad, pero al crecer r aumenta la no–linealidad del problema y la iteración de Newton necesaria utilizada por estos métodos requiere más pasos para converger.

En conclusión, para el último caso analizado ($r = 100000$), LIQSS es aproximadamente 60 veces más rápido que DASSL, 160 veces más rápido que Radau y 300 veces más rápido que DOPRI.

Variación del Término de Difusión d

En este último escenario estudiamos el costo computacional para distintos valores del término de difusión d mientras que los restantes parámetros se mantienen fijos ($a = 1, N = 1000, r = 1000$). Los errores son similares a los anteriores por lo que no se reportan.

La Figura 5.17 muestra el costo computacional como función de d mientras que la Tabla 5.12 reporta los resultados y el número de las evaluaciones de funciones escalares.

Para valores pequeños de d, el rendimiento de LIQSS2 nuevamente supera al resto de los métodos. Sin embargo, a medida que crece el término de difusión, su eficiencia se degrada rápidamente.

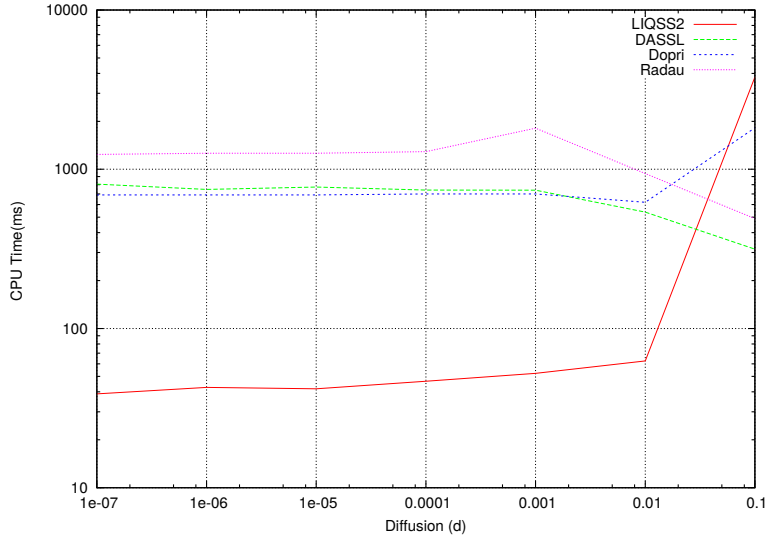


Figura 5.17: Comparación de tiempo de CPU para diferentes valores del término de difusión d - $a = 1, N = 1000, r = 1000$

Tabla 5.12: Tiempo de CPU (ms) y número de evaluaciones de función para diferentes valores de d con $a = 1, N = 1000, r = 1000$

N	LIQSS2		**DASSL**		**DOPRI**		**Radau5**	
	tiempo	**eval.**	**tiempo**	**eval.**	**tiempo**	**eval.**	**tiempo**	**eval.**
$1e{-}1$	$3{,}79e3$	$6{,}19e7$	$3{,}15e2$	$2{,}46e6$	$1{,}82e3$	$9{,}45e7$	$4{,}90e2$	$4{,}76e6$
$1e{-}2$	$6{,}25e1$	$8{,}68e5$	$5{,}38e2$	$4{,}04e6$	$6{,}20e2$	$3{,}16e7$	$9{,}40e2$	$9{,}00e6$
$1e{-}3$	$5{,}23e1$	$6{,}07e5$	$7{,}39e2$	$5{,}26e6$	$7{,}00e2$	$3{,}63e7$	$1{,}81e3$	$1{,}59e7$
$1e{-}4$	$4{,}66e1$	$6{,}05e5$	$7{,}41e2$	$5{,}64e6$	$7{,}00e2$	$3{,}54e7$	$1{,}29e3$	$1{,}23e7$
$1e{-}5$	$4{,}18e1$	$5{,}62e5$	$7{,}73e2$	$5{,}77e6$	$6{,}90e2$	$3{,}54e7$	$1{,}26e3$	$1{,}20e7$
$1e{-}6$	$4{,}27e1$	$5{,}48e5$	$7{,}47e2$	$5{,}45e6$	$6{,}90e2$	$3{,}54e7$	$1{,}26e3$	$1{,}19e7$
$1e{-}7$	$3{,}89e1$	$5{,}22e5$	$8{,}07e2$	$6{,}11e6$	$6{,}90e2$	$3{,}54e7$	$1{,}24e3$	$1{,}19e7$

Modelo Advección–Difusión–Reacción en 2D

Es este último ejemplo analizamos brevemente si los resultados obtenidos anteriormente se mantienen para los casos en dos dimensiones. Con este fin, consideramos un modelo de advección–reacción 2D discretizado dado por las siguientes ecuaciones:

$$\dot{u}_{i,j} = -a_x \frac{(u_{i,j} - u_{i,j-1})}{\Delta x} - a_y \frac{(u_{i,j} - u_{i-1,j})}{\Delta y} + r(u_{i,j}^2 - u_{i,j}^3) \tag{5.34}$$

para $i = 2, \cdots, N, j = 2, \cdots, N$,

$$\dot{u}_{i,1} = -a_x \frac{u_{i,1}}{\Delta x} - a_y \frac{(u_{i,1} - u_{i-1,1})}{\Delta y} + r(u_{i,1}^2 - u_{i,1}^3) \tag{5.35}$$

para $i = 2, \cdots, N$,

$$\dot{u}_{1,j} = -a_x \frac{(u_{1,j} - u_{1,j-1})}{\Delta x} - a_y \frac{u_{1,j}}{\Delta y} + r(u_{1,j}^2 - u_{1,j}^3) \tag{5.36}$$

para $j = 2, \cdots, N$ y finalmente

$$\dot{u}_{1,1} = -a_x \frac{u_{1,1}}{\Delta x} - a_y \frac{u_{1,1}}{\Delta y} + r(u_{1,1}^2 - u_{1,1}^3) \tag{5.37}$$

donde el refinamiento de la grilla es definido por $\Delta x = \Delta y = 10/N$.

Simulamos este modelo para diferentes refinamientos de la grilla, obteniendo los resultados reportados en la Tabla 5.13.

Tabla 5.13: Tiempo de CPU (ms) para diferentes configuraciones posibles de la grilla ($N \times N$) - $a_x = a_y = 1$, $r = 1000$.

N	LIQSS2		**DASSL**		**DOPRI**	
	tiempo	**eval.**	**tiempo**	**eval.**	**tiempo**	**eval.**
10×10	5,70e0	4,86e4	1,37e2	2,39e6	8,64e1	1,28e6
50×50	1,82e2	1,17e6	3,02e3	7,16e7	3,76e4	4,69e8
100×100	8,59e2	4,68e6	1,19e4	2,91e8	–	–
500×500	5,69e4	1,21e8	4,03e5	9,40e9	–	–
1000×1000	4,35e5	4,96e8	1,91e6	4,24e10	–	–

Podemos ver que el método DASSL no puede simular cuando la grilla es $N \times N = 100 \times 100$. En este caso, DASSL debe invertir una matriz muy grande que en este caso no es tridiagonal.

Nuevamente, el método LIQSS2 exhibe un mejor rendimiento que DOPRI y DASSL. Debemos mencionar que mientras que el número de evaluaciones de funciones crece de manera lineal con el tamaño del sistema, el tiempo de CPU escala de manera supra–lineal. Esto es debido al hecho de que el lenguaje μ–Modelica no soporta modelos 2D todavía, por lo tanto en el modelo correspondiente la matriz de tamaño $N \times M$ fue modelada M arreglos. Como consecuencia, el modelo μ–Modelica no es traducido de manera óptima al lenguaje C, donde el lado derecho de la EDO contiene comparaciones innecesarias en este caso. A medida que crece M, esas comparaciones innecesarias afectas el rendimiento global.

Como conclusión podemos decir que:

- El método LIQSS2 es una mejor opción que los métodos clásicos de integración cuando la relación entre la advección y la difusión es grande. Sin embargo, cuando la difusión es alta, los modelos no son tratados de manera apropiada por el método LIQSS2 y los métodos clásicos son más eficientes.
- Cuando el término de difusión se mantiene pequeño, a medida que N crece LIQSS2 muestra ventajas sobre los restantes métodos.
- Al contrario que en los métodos clásicos, el rendimiento de los métodos LIQSS2 no se ve afectado por el crecimiento del término de reacción r.
- En la mayoría de los casos, el método LIQSS2 es al menos 10 veces más rápido que los métodos clásicos.

Capítulo 6

Simulación en Paralelo con el Simulador Autónomo de QSS

Como vimos en el Capítulo 3, el simulador autónomo para métodos QSS está diseñado de manera tal que es posible definir modelos complejos de gran escala, obteniendo la información estructural que necesita el motor de simulación de una manera eficiente. Sin embargo, simular este tipo de modelos usualmente conlleva un costo computacional alto. En los últimos años, debido al desarrollo de procesadores multi–núcleo como así también de clusters de computadoras multi–nodo, la simulación en paralelo de sistemas de tiempo continuo representa la manera usual de reducir los costos de simulación.

La naturaleza asíncrona de los métodos QSS simplifica la paralelización de los cálculos realizados. Como mencionamos anteriormente, los métodos QSS pueden ser representados como sistemas de eventos discretos, por lo tanto, se podrían utilizar en principio las estrategias de paralelización para sistemas de eventos discretos, sin embargo, ninguna de estas estrategias se adapta de manera satisfactoria a los algoritmos de simulación QSS. Al aplicar técnicas de sincronización estricta propuesta en las estrategias conservadoras, no se permite la ejecución de cálculos concurrentes como se muestra en [8]. Por otro lado, aplicar estrategias optimistas requiere el uso de una gran cantidad de memoria para implementar el mecanismo que permite retrotraer la simulación. Finalmente, aplicar la técnica que evita la sincronización entre diferentes PLs podría introducir un error inaceptable en la simulación.

En [4], se proponen dos técnicas de paralelización específicas para métodos QSS, SRTS y ASRTS para una arquitectura multinúcleo de memoria compartida.

Donde la sincronización entre los diferentes submodelos es llevada a cabo por un reloj de tiempo real y adicionalmente se analiza el error introducido por las técnicas de paralelización mostrando que si la diferencia entre los tiempos lógicos de los distintos PLs se mantiene acotada, la paralelización introduce un error numérico adicional acotado. Estas técnicas fueron implementadas en PowerDEVS, por lo que presentan las limitaciones en términos de eficiencia impuestas por el motor de simulación DEVS. Adicionalmente, se requiere uso de un sistema operativo de tiempo real.

Basados en la ideas descriptas, en esta Tesis se desarrollaron dos nuevas técnicas de paralelización específicamente diseñadas para los métodos de QSS que no tienen las limitaciones mencionadas anteriormente. En este capítulo presentamos las nuevas metodologías para la simulación en paralelo de métodos QSS y su implementación basada en el motor de simulación autónomo QSS. En las siguientes secciones introducen la idea básica, los algoritmos y la implementación de los diferentes componentes en el motor de simulación autónomo para métodos QSS.

6.1. Idea Básica

La técnica de paralelización que presentamos requiere como primer paso que el modelo original sea dividido en P submodelos, para que cada submodelo pueda ser simulado en un proceso lógico diferente.

Luego, en cada paso de simulación de un submodelo, el PL correspondiente controla si la variable que cambia debe ser comunicada a otros PLs utilizando información estructural. En caso de tener que comunicar el cambio, se envía un mensaje con el valor nuevo de la variable, indicando además el tiempo lógico del mismo utilizando un mecanismo de comunicación inter–proceso.

Para poder evitar errores inaceptables introducidos por mensajes que son enviados en instantes de tiempo erróneos, la técnica de paralelización propuesta impone una restricción adicional que controla que la diferencia entre los tiempos lógicos de simulación de los diferentes PLs sea acotada. Esta cota es definida por un parámetro llamado Δt dado por el usuario.

De este modo, en esta nueva técnica se propone una sincronización no estricta entre los PLs. Esto se logra calculando el tiempo virtual global gvt (que es igual al mínimo de los tiempos lógicos de todos los PLs) y aplicando la restricción de

que ningún PL puede avanzar su tiempo lógico por encima de la cota impuesta $gvt + \Delta t$. Esto significa que cuando el próximo paso de simulación programado por un PL es mayor a $gvt + \Delta t$, debe esperar hasta que se actualice el gvt o hasta recibir un mensaje.

6.2. Estructura Básica del Simulador Paralelo

Las nuevas técnicas de paralelización propuestas fueron implementadas en el motor de simulación autónomo para métodos QSS en una arquitectura multi–núcleo utilizando memoria compartida en un esquema MIMD (Múltiples Instrucciones Multiples Datos). Para esto se requiere primero dividir el modelo definido por las Ecs.(3.2)–(3.4), donde cada PL simula un submodelo diferente. Cada PL está compuesto por un módulo **Integrador**, un módulo **Cuantificador**, y un módulo **Modelo**, de igual manera que en el caso secuencial descripto en el Capítulo 3.

Teniendo en cuenta que las derivadas de estado calculadas en un PL pueden depender de variables de estado calculadas en otro PL, es claro que los diferentes PLs deben comunicarse durante la simulación. Más aún, el cálculo de una derivada de estado $\dot{x}_i$ en el tiempo t requiere conocer el valor de las variables de estado cuantificadas involucradas en el cálculo de $f_i(\mathbf{q}, \mathbf{d}, t)$ en el tiempo t, por lo tanto, también es necesario un mecanismo de sincronización entre los distintos PLs.

Con este fin, se deben agregar mecanismos permitan la sincronización y la comunicación entre diferentes PLs al módulo **Integrador** secuencial.

El estructura básica del motor de simulación QSS en paralelo se muestra en la Figura **??**[1].

En las siguientes secciones, luego de discutir como generar la partición del modelo, describiremos en detalle los mecanismos de sincronización y comunicación implementados en el motor y el algoritmo de simulación en paralelo propuesto.

[1]Cabe mencionar que esta estructura está desarrollada para una arquitectura de memoria compartida. De todas formas, el mecanismo de comunicación se describe en términos de envío mensajes. Veremos más adelante que la restricción al uso de una arquitectura de memoria compartida está impuesta por el mecanismo de sincronización.

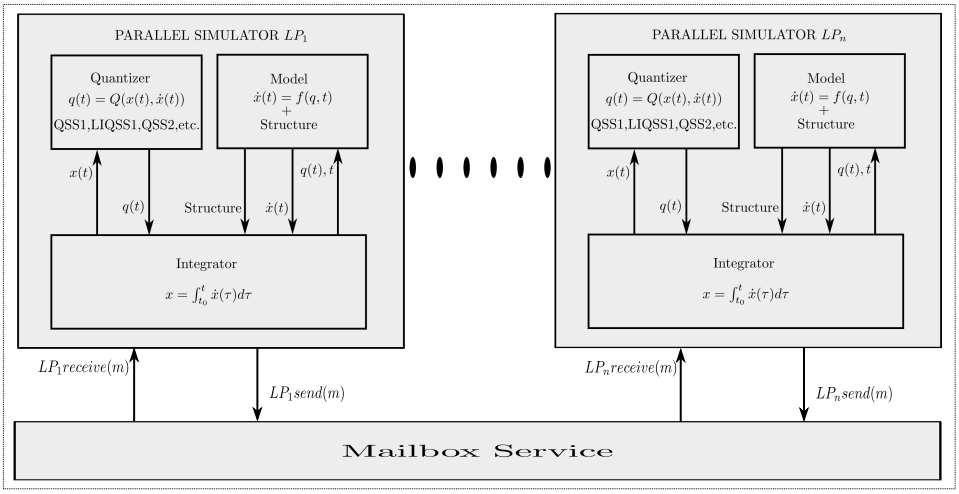


Figura 6.1: Motor de simulación QSS en Paralelo –Esquema Básico de Interacción

6.3. Partición y Estructura Inter–Procesos

Como mencionamos anteriormente, el algoritmo de simulación en paralelo para métodos QSS requiere que el modelo original sea dividido en submodelos que luego pueden ser simulados por diferentes PLs.

Para poder obtener una simulación en paralelo eficiente, la división del modelo debe ser balanceada, y al mismo tiempo la comunicación entre los PLs tiene que ser la mínima posible. En algunos casos simples, cuando el sistema tiene una estructura regular, esta división puede ser realizada de manera manual.

Sin embargo, en el caso general, se deben utilizar algoritmos que dividen el modelo de forma automática. En el motor de simulación en paralelo dispone de un conjunto de algoritmos basados en grafos que permiten generar la partición del modelo de manera automática y fueron desarrollados en [23].

En este trabajo, asumiremos disponible una partición *adecuada* de los modelos presentados.

En el contexto de la implementación, la división del modelo es representada por dos arreglos P^x, y P^z, cuyas componentes toman valores en el conjunto $\{1, \cdots, p\}$. Donde $P_i^x = k$ indica que la variable de estado x_i es calculada por el k–ésimo PL. De manera similar, $P_i^z = k$ indica que la función de cruce por cero ZC_i y su handler correspondiente H_i son calculados por el k–ésimo PL.

Cabe mencionar que en la interfaz gráfica del motor de simulación QSS en paralelo, el usuario final puede seleccionar diferentes algoritmos de división automática de modelos descriptos en [23] o proveer una partición generada manualmente.

Como mencionamos anteriormente, la información estructural del modelo es representada por cuatro matrices de incidencia SD, SZ, HD, y HZ. Estas matrices representan influencias directas desde variables de estado y handlers a derivadas de estado y funciones de cruce por cero.

Una vez que el modelo es dividido en submodelos (de acuerdo a los arreglos P^x y P^z), puede suceder que un cambio en una variable de estado –o la ejecución de un handler– calculado en el k–ésimo submodelo tenga una influencia directa en alguna variable de estado o función de cruce por cero calculada un otro submodelo.

Por lo tanto, es necesario un mecanismo de comunicación inter–proceso que a su vez requiere conocer información estructural con respecto a la comunicación entre los mismos.

En nuestra implementación, esta información es representada por dos matrices de incidencia definidas en cada PL:

- SO^k donde $SO^k_{i,l} = 1$ indica que la i–ésima variable de estado del k–ésimo submodelo influencia alguna derivada de estado o función de cruce por cero calculada en el l–ésimo submodelo.
- HO^k donde $HO^k_{i,l} = 1$ indica que las variables que son modificadas por la ejecución del i–ésimo handler en el k–ésimo submodelo influencia alguna derivada de estado o función de cruce por cero calculada por el l–ésimo submodelo.

Las matrices SO^k y HO^k son calculadas en tiempo de inicialización utilizando la información provista por las matrices de incidencia estructurales SD, SZ, HD, HZ y los arreglos P^x y P^z que describen la partición del modelo. Cabe mencionar que SO^k y HO^k son guardadas en forma *sparse*, al igual que el resto de las matrices que contienen información estructural.

6.4. Algoritmo de Simulación

El algoritmo de simulación en paralelo es implementado localmente en cada proceso lógico. Como explicamos anteriormente, cada PL está compuesto por un

módulo **Integrador**, un módulo **Cuantificador** y un módulo **Modelo**. Tanto el **Cuantificador** como el **Modelo** son idénticos a los módulos secuenciales, mientras que el **Integrador** debe incluir ahora los mecanismos de comunicación y sincronización.

La comunicación entre procesos implica enviar mensajes a otros PLs luego de cambios en variables de estado o la ejecución de un handler, que de acuerdo a la estructura de comunicación inter–procesos, afectan derivadas de estado o funciones de cruce por cero calculadas en otros PLs. Además, cada PL debe poder recibir mensajes de otros PLs con los cambios correspondientes. Con este fin, cada PL mantiene una lista ordenada por los tiempos lógicos de llegada de mensajes sin procesar.

Como ya hemos mencionado, el mecanismo de sincronización limita el avance de la simulación cada proceso para mantener una una diferencia acotada entre los tiempos lógicos de los distintos procesos.

Más adelante, explicaremos en detalle los procedimientos de comunicación y sincronización.

Teniendo en cuenta estas modificaciones, el algoritmo para el k–ésimo **Integrador** paralelo puede describirse de la siguiente manera:

Puede notarse que este algoritmo es similar al algoritmo secuencial (Algoritmo 3.1), con la adición de el cálculo de tiempo global virtual gvt como el mínimo entre los tiempos de simulación lógicos de cada PL y una rutina de sincronización para controlar el avance del tiempo lógico t^k. Finalmente, también se debe tener en cuenta los mensajes recibidos de otros PLs.

Cuando el próximo paso en el k–ésimo PL se debe a un cambio en una variable cuantificada q_i en el tiempo t, el **Integrador** procede de la siguiente manera: Nuevamente, este algoritmo es muy similar al algoritmo secuencial (Algoritmo 3.2). La diferencia en este caso es que solamente son actualizadas las derivadas de estado que son calculadas por el PL, y, eventualmente, se envían mensajes a los otros PLs que calculan las derivadas de estado restantes afectadas por el cambio en la variable cuantificada.

Cabe mencionar que el vector de estados cuantificados $\mathbf{q}(t)$ utilizado para calcular las derivadas de estado y las funciones de cruce por cero pueden contener componentes $q_j(t)$ calculadas en otros PLs. En este caso, el k–ésimo PL tiene una

Algoritmo 6.1: QSS Parallel Integrator Module

```
1  while t^k < t_f //while simulation time of the k--th LP is less than
       the final time tf
2    t_x^k = min(tx_j | P_j^x = k) // time of the next change in a quantized
       variable computed by the k-th LP
3    t_z^k = min(tz_j | P_j^z = k) // time of the next zero--crossing time
       computed by the k--th LP
4    t_m^k = Mailbox.FirstMessage().time() // timestamp of the first
       enqueued message
5    t^k = min(t_x^k, t_z^k, t_m^k) // attempt to advance LP simulation time
6    gvt = min(t^l) // recompute global virtual time
7    Synchronize() // call synchronization procedure
8    if t^k = t_x^k then //state change
9      i = argmin(tx_j) // the i-th quantized state changes first
10     Parallel_Quantized_State_Step(i) //execute a quantized state
         change on variable i
11   elseif t^k = t_z^k then //zero--crossing
12     i = argmin(tz_j) // the i-th event handler is executed first
13     Parallel_Event_Handler_Step(i) //execute the procedure for the
         i-th event handler
14   else
15     Process_External_Message() //process the first enqueued message
16   end if
17   Mailbox.Receive_messages() //check local inbox for incoming
       messages from other LPs
18 end while
```

Algoritmo 6.2: QSS Parallel Integrator Module - Quantized State Change

```
1  Parallel_Quantized_State_Step(i)
2  {
3    integrateState(x_i, ẋ_i, t^k) // integrate i-th state up to time t^k.
4    Quantizer.update(x_i, q_i) // update i-th quantized state q_i
5    tx_i = Quantizer.nextTime(x_i, q_i) //compute next i-th quantized state
       change time
6    for each l such that SO_{i,l}^k = 1
7        Mailbox.Send_state_message(l,i,q_i,t_k) //send new quantized
           state and time stamp to l--th LP
8    end for
9    for each j such that SD_{i,j} = 1 and P_j^x = k
10     integrateState(x_j, ẋ_j, t) // integrate j-th state up to time t.
11     ẋ_j = Model.f_j(q^k(t), d^k(t), t) // recompute j-th state derivative
12     tx_j = Quantizer.nextTime(x_j(t), q_j(t)) //recompute next j-th
         quantized state change time
13   end for
14   for each j such that SZ_{i,j} = 1 and P_j^z = k
15     zc_j = Model.zc_j(q^k, d^k(t), t) // recompute j-th zero--crossing function
16     tz_j = nextEventTime(zc_j) //recompute next j-th zero--crossing time
17   end for
18 }
```

copia *local* (posiblemente desactualizada) de los valores correspondientes. Por lo tanto, el vector que contiene tanto las componentes calculadas por el PL como las copias locales se denota como $\mathbf{q}^k(t)$, una acotación similar puede realizarse sobre

el vector de estados discretos $\mathbf{d}(t)$.

Cuando el próximo paso de simulación es debido a la ejecución de un handler H_i en el tiempo t, el **Integrador** procede como el Algoritmo 3.3 con las mismas modificaciones introducidas anteriormente.

Como mencionamos antes, cuando un PL recibe un mensaje informando un cambio en una variable cuantificada o discreta, el mensaje es guardado en una lista ordenada por el tiempo de simulación lógico de cada mensaje. En el caso de que el primer mensaje de la lista tenga un tiempo lógico ($m_1.t$) menor que el próximo tiempo de simulación local programado, el **Integrador** procesa el mensaje de la siguiente manera (asumiendo que el mensaje corresponde a un cambio en una variable de estado cuantificada):

Algoritmo 6.3: QSS Parallel Integrator Module - External Quantized State Change

```
1  Process_External_Message()
2  {
3    m1=Mailbox.FirstMessage().pop() //dequeue the first message at the
         queue
4    i=m1.index() //index of the quantized variable that changed
5    q_i^k = m1.q //update local copy of quantized state q_i.
6    for each j such that SD_{i,j} = 1 and P_j^x = k
7      integrateState(x_j, ẋ_j, t) // integrate j-th state up to time t.
8      ẋ_j = Model.f_j(q(t), d(t), t) // recompute j-th state derivative
9      tx_j = Quantizer.nextTime(x_j(t), q_j(t)) //recompute next j-th
           quantized state change time
10   end for
11   for each j such that SZ_{i,j} = 1 and P_j^z = k
12     zc_j = Model.zc_j(q, d(t), t) // recompute j-th zero--crossing function
13     tz_j = nextEventTime(zc_j) //recompute next j-th zero--crossing time
14   end for
15 }
```

Cuando el mensaje corresponde a la ejecución de un handler, el procedimiento es similar.

6.5. Sincronización y Comunicación Inter–Procesos

Como mencionamos anteriormente, la implementación en paralelo del motor de simulación QSS utiliza un mecanismo de sincronización no estricta que es necesario para asegurar que los cálculos son efectivamente realizados en paralelo.

Este mecanismo de sincronización requiere que, luego de cada paso de simulación, cada PL calcule:

$$gvt = \min_{1 \leq l \leq P} (t^l) \tag{6.1}$$

como el mínimo tiempo de simulación lógico de todos los PLs. Luego, cada PL limita el avance de su tiempo de simulación local t^k al valor $gvt + \Delta t$, donde Δt es un parámetro definido por el usuario. Cabe mencionar que Δt es una cota superior para la diferencia entre los tiempos de simulación locales de cada PL y define que tan estricta es la simulación.

El procedimiento de sincronización de cada PL implementa una rutina que *espera* mientras se verifica la siguiente condición

$$t^k > gvt + \Delta t \tag{6.2}$$

Cuando el gvt avanza (debido a que otros PLs actualizan su tiempo de simulación local) o cuándo el tiempo local de simulación t^k es actualizado (debido a que el k–ésimo PL recibe un mensaje), y la condición de la Ec. (**??**) no se cumple, el k–ésimo PL puede continuar con la simulación. La rutina de sincronización puede describirse de la siguiente manera:

Algoritmo 6.4: LP Synchronization

```
1  Synchronize()
2  {
3    while(t^k − gvt > Δt)
4      rcv=Mailbox.Receive_messages() //check local inbox for incoming
           messages from other LPs
5      if (rcv)
6        t^k = Mailbox.FirstMessage().time() // timestamp of the first
             enqueued message
7      end if
8      gvt = min(t^l) // recompute the minimum global virtual time
9    end while
10 }
```

Podemos ver que la linea **??** de este algoritmo calcula explícitamente el mínimo tiempo virtual global a partir de los tiempos lógicos t^l de todos los procesos. Este cálculo es la razón que restringe el uso de la estrategia implementada a una arquitectura de memoria compartida. Si se utilizara este algoritmo en una arquitectura distribuida se debería transmitir los diferentes tiempos lógicos de cada procesador en cada paso de simulación, lo que resulta en un tráfico sobre la red muy importante.

La Figura **??** ilustra el funcionamiento del mecanismo de sincronización durante la simulación en paralelo utilizando dos PLs.

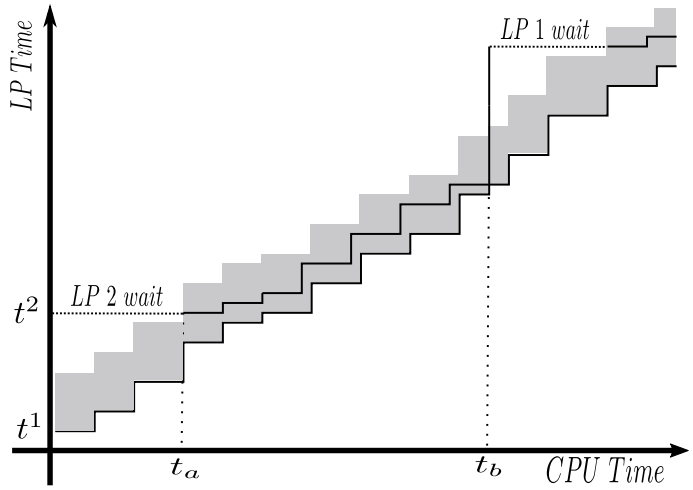


Figura 6.2: Motor de Simulación QSS Paralelo – Ejemplo Esquema de Sincronización

Es este ejemplo, podemos ver que el tiempo de avance máximo para cada PL está limitado por $gvt + \Delta t$, lo cual es representado en la figura por la región gris sobre el gvt.

Al comienzo de la simulación, el proceso lógico (LP2) agenda su próximo cambio para el tiempo t^2, dado que este valor es mayor a $gvt + \Delta t$ debe esperar hasta que avance el gvt (i.e. que avance el tiempo de simulación t^1 del proceso lógico LP1).

Luego, al llegar al tiempo de CPU t_a, el tiempo de simulación t^1 (como así también el gvt) es tal que $t^2 < gvt + \Delta t$ y el proceso LP2 puede reanudar su simulación. Durante el intervalo (t_a, t_b), la distancia entre t^1 y t^2 es menor que Δt por lo que ambos PLs pueden simular en paralelo sin tener que esperar. Luego, en el tiempo de CPU t_b, el proceso LP1 agenda su próximo tiempo de simulación para el tiempo $t^1 > gvt + \Delta t$, y en este caso el proceso LP1 debe esperar.

Es importante notar que en el caso de tomar $\Delta t = 0$, cada PL puede avanzar solamente cuando $gvt = t^k$, lo que lleva a una simulación secuencial y no se realizan cálculos en paralelo. Por otro lado, si tomamos $\Delta t \geq t_f$ (donde t_f es el tiempo final de simulación) no existe sincronización durante la simulación y cada PL simula tan rápido como sea posible, de la misma manera que la estrategia *NOTIME*, pero los errores numéricos introducidos por la simulación en este caso pueden resultar inaceptables.

Como consecuencia, el un buen valor para el parámetro Δt debe ser tal que el error introducido por la simulación sea aceptable y que a su vez permita realizar cálculos en paralelo. Analizaremos en profundidad este problema más adelante, proponiendo también una estrategia apdaptiva para encontrar un valor adecuado para el parámetro Δt.

Como mencionamos anteriormente, los diferentes PLs envían mensajes a otros PLs durante la simulación. Con este propósito, cada proceso lógico tiene una casilla de correo que ofrece los siguientes servicios:

- `Send_state_message(l,i,qi,tk)`, que envía un mensaje desde el k–ésimo PL al l–ésimo PL indicando un cambio en la i–ésima variable de estado cuantificada q_i en el tiempo t_k.

- `Send_event_message(l,i,{dj},{xj},tk)`, que envía un mensaje desde el k–ésimo PL al l–ésimo PL indicando que la ejecución del i–ésimo handler en el tiempo t_k, modifica la lista de variables discretas $\{d_j\}$ y la lista de variables de estado continuas $\{x_j\}$.

- `Receive_messages()`, que controla la existencia de mensajes nuevos y guarda los mensajes recibidos en una lista interna. Adicionalmente, cuando un mensaje con un tiempo lógico $m.t$ es recivido y ese tiempo lógico es menor que el tiempo del último paso de simulación local ejecutado por el PL t^{k-}, el tiempo lógico del mensaje es modificado y se le asigna el valor t^{k-}. De esta manera evitamos recibir mensajes desde el *pasado* durante la simulación.

  Cabe mencionar que la modificación en los tiempos lógicos de los mensajes es el único efecto que tienen los errores de sincronización en los resultados de la simulación. De todas maneras, la diferencia entre $m.t$ y t^{k-} siempre es acotada por Δt.

El mecanismo de comunicación entre los diferentes procesos es híbrido:

- Los mensajes que transmiten cambios de estado se envían de manera asíncrona utilizando los servicios mencionados anteriormente.

- Los mensajes que transmiten cambios discretos utilizan los mismos servicios, pero en este caso de manera sincronizada, i.e. el proceso que envía el mensaje

debe esperar hasta que todos los procesos destinatarios del mensaje procesen el mismo.

Este último caso asegura que cada cambio discreto es procesado por todos los PLs afectados antes de continuar la simulación. Teniendo en cuenta que los cambios discretos pueden provocar grandes cambios en las variables de estado y discretas, esta política previene la introducción de errores numéricos grandes dados por la falta de sincronización luego de discontinuidades.

Cabe mencionar que en la mayoría de los modelos que contienen discontinuidades el número de cambios en las variables continuas es significativamente mayor que el número de cambios discretos. Por lo tanto, en la práctica, la comunicación es en gran parte asíncrona.

6.6. Detalles de Implementación

El modelo de ejecución de cada proceso lógico es implementado utilizando hilos POSIX de Linux. Cada hilo es asociado a un CPU específico utilizando `sched_setaffinity`. La asignación de los hilos a cada CPU está dada por las matrices de partición PX y PZ.

Inicialmente, un proceso principal se encarga de leer la partición del modelo y generar la estructura de comunicación inter–proceso. Luego, se crean P hilos y cada uno de ellos implementa localmente una instancia del **Integrador Paralelo**, el **Cuantificador** y el **Modelo**.

Como mencionamos anteriormente, las instancias del **Cuantificador** y el **Modelo** son idénticas a las implementaciones secuenciales.

La implementación del módulo **Integrador Paralelo** contiene estructuras de datos con la siguiente información:

- Una copia local del tiempo de simulación lógico t^k y el tiempo virtual global gvt.
- Las matrices que indican la comunicación inter–procesos SO^k y HO^k.
- Dos arreglos que contienen los índices de las derivadas de estado y funciones de cruce por cero calculadas por el PL.
- Una instancia de la casilla de correo.

- Una copia local del vector de estados discretos **d**.

Adicionalmente, una opción de compilación permite tener copias locales de:

- Copias locales de los arreglos completos de las variables de estado y cuantificadas **x** y **q**.
- Una copia local de los valores de las funciones de cruce por cero.

En caso contrario esos arreglos son compartidos por todos los PLs junto con el resto de la información global compartida, que consisten en:

- Un arreglo que contiene los tiempos de simulación local de cada PL.
- Un arreglo donde están registradas todas las instancias de las casillas de correo de cada PL.
- Las cuatro matrices de incidencia SD, HD, SZ, y HZ.

Cabe mencionar que la copia completa del arreglo de estados discretos **d** se mantiene para evitar problemas de sincronización entre los PLs, dado que diferentes PLs pueden modificar los valores de cada uno de los componentes. El uso opcional de las copias locales de las variables de estado y las variables cuantificadas permite mejorar el desempeño del motor de simulación en arquitecturas NUMA (Non–Uniform Memory Access). Dado que en este tipo de arquitecturas, el tiempo de acceso a una dirección de memoria de los diferentes PL depende de cada procesador. Por lo tanto, al tener copias locales del estado completo, cada procesador accede a su propio banco de memoria y se evita este problema.

El costo de utilizar copias locales del estado completo es un incremento significativo en el consumo de memoria. Esto puede ser resuelto copiando solamente los subarreglos que son efectivamente utilizados por cada PL y mediante la definición de los mapas correspondientes a los arreglos globales.

A pesar del hecho de que todos los datos son guardados en memoria compartida, los únicos datos que son actualizados por todos los PLs de manera concurrente son los arreglos que contienen los tiempos de simulación local de cada PL (por razones de sincronización) y el arreglo que contiene las casillas de correo (para poder comunicar mensajes)[2].

[2]Adicionalmente, cuando los arreglos de estado no tienen copias locales, pueden ser accedidos de manera concurrente.

Adicionalmente, el arreglo que contiene los tiempos de simulación local es utilizado para calcular el tiempo virtual global gvt. De todas maneras, en la implementación, el gvt solamente es actualizado por el k–ésimo PL cuando el tiempo local t^k es mayor a $gvt + \Delta t$. Esto implica que en la mayoría de los pasos de simulación, no se accede al arreglo compartido.

6.7. Análisis del Error Numérico de Paralelización

El mecanismo de sincronización no estricto adoptado puede provocar que un mensaje enviado por el k–ésimo PL informando un cambio en una variable de estado cuantificada q_i en el tiempo t^k sea recibido por el l–ésimo PL de manera tardía, i.e., cuando el l–ésimo PL ya ha ejecutado uno o más pasos en el tiempo $t^l > t^k$. En este caso, como no existe un mecanismo que permite retrotraer la simulación, el l–ésimo PL modificará el tiempo lógico del mensaje recibido asumiendo que el cambio en la variable cuantificada q_i ocurrió en el tiempo t^l.

Esto implica que, durante el intervalo (t^k, t^l), el l–ésimo PL realiza cálculos utilizando una copia local incorrecta (desactualizada) del estado cuantificado q_i. De todas maneras, tomando en cuenta que la diferencia entre los tiempos lógicos t^l y t^k es acotada por Δt, y conociendo que q_i es una aproximación de una señal que varia de forma continua $x_i(t)$, podemos esperar que la diferencia entre el valor real de q_i (calculada en el k–ésimo PL) y la copia local del l–ésimo PL no sea significativa y que introduzca un error numérico acotado en los cálculos subsiguientes.

El siguiente análisis muestra este hecho de manera formal.

Consideremos la aproximación QSS de una EDO, dada por

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

y supongamos que este sistema es dividido de la siguiente manera:

$$\begin{aligned} \dot{\mathbf{x}}_a(t) &= \mathbf{f}_a(\mathbf{q}_a(t), \mathbf{q}_b(t), \mathbf{v}(t)) \\ \dot{\mathbf{x}}_b(t) &= \mathbf{f}_b(\mathbf{q}_a(t), \mathbf{q}_b(t), \mathbf{v}(t)) \end{aligned} \tag{6.3}$$

donde:

$$\mathbf{x} = [\mathbf{x}_a\ \mathbf{x}_b]^T;\ \mathbf{q} = [\mathbf{q}_a\ \mathbf{q}_b]^T;\ \mathbf{f} = [\mathbf{f}_a\ \mathbf{f}_b]^T$$

Consideremos que implementamos la simulación de este sistema en paralelo de manera tal que $\mathbf{x}_a$ es calculado en un PL y $\mathbf{x}_b$ es calculado por un segundo PL utilizando los mecanismo explicados en las secciones previas.

Luego, el mecanismo de sincronización no estricta implica que el primer PL puede tener copias locales $\mathbf{q}_b^c(t)$ que contengan valores desactualizadas de las componentes de $\mathbf{q}_b(t)$. De manera similar, el segundo PL puede tener copias locales $\mathbf{q}_a^c(t)$, que contengan valores desactualizados de las componentes de $\mathbf{q}_a$.

Como consecuencia, el algoritmo de simulación en paralelo simula la aproximación dada por la Ec. (**??**):

$$\begin{aligned} \dot{\mathbf{x}}_a(t) &= \mathbf{f}_a(\mathbf{q}_a(t), \mathbf{q}_b^c(t), \mathbf{v}(t)) \\ \dot{\mathbf{x}}_b(t) &= \mathbf{f}_b(\mathbf{q}_a^c(t), \mathbf{q}_b(t), \mathbf{v}(t)) \end{aligned} \tag{6.4}$$

Definamos ahora $\Delta_a(t) \triangleq \mathbf{q}_a^c(t) - \mathbf{q}_a(t)$, y $\Delta_b(t) \triangleq \mathbf{q}_b^c(t) - \mathbf{q}_b(t)$. Luego, Eq.(**??**) se puede escribir como

$$\begin{aligned} \dot{\mathbf{x}}_a(t) &= \mathbf{f}_a(\mathbf{q}_a(t), \mathbf{q}_b(t) + \Delta_b(t), \mathbf{v}(t)) \\ \dot{\mathbf{x}}_b(t) &= \mathbf{f}_b(\mathbf{q}_a(t) + \Delta_a(t), \mathbf{q}_b(t), \mathbf{v}(t)) \end{aligned} \tag{6.5}$$

que constituye una versión *perturbada* del sistema original dado por la Ec. (**??**), con términos de perturbación $\Delta_a(t)$ y $\Delta_b(t)$.

Aquí podemos ver que un componente $q_i(t)$ calculado por el primer PL y su copia local $q_i^c(t)$ en el segundo PL pueden tener un retraso máximo dado por el parámetro Δt, i.e., $q_i^c(t) = q_i(t-\tau)$ con $0 \leq \tau < \Delta t$. Por lo tanto, resulta que

$$|q_i^c(t) - q_i(t)| = |q_i(t-\tau) - q_i(t)| \leq |x_i(t-\tau) - x_i(t)| + 2\Delta Q_i$$

donde utilizamos el hecho de que la diferencia entre x_i y q_i es siempre acotada por el quantum ΔQ_i. Luego, asumiendo que la derivada de estado $\dot{x}_i = f_i(\mathbf{q}, t)$ es acotada por una constante M_i mientras que las variables de estado cuantificadas $\mathbf{q}(t)$ se mantiene en cierta región acotada, resulta que

$$|x_i(t-\tau) - x_i(t)| < M_i \cdot \tau \leq M_i \Delta t$$

y como consecuencia

$$|q_i^c(t) - q_i(t)| \leq M_i \cdot \Delta t + 2\Delta Q_i \tag{6.6}$$

Aplicando el mismo análisis en todos las componentes del arreglo $\mathbf{q}_a$ y $\mathbf{q}_b$, resulta que todos las componentes de los términos de perturbación $\Delta_a(t)$ y $\Delta_b(t)$ son

acotados en su valor absoluto por una constante que depende del parámetro Δt y el quantum ΔQ.

Cuando el sistema original dado por Ec. (2.17) es lineal e invariante en el tiempo (LTI) y asintóticamente estable, se puede probar fácilmente que la presencia de perturbaciones acotadas $\Delta_a(t)$ y $\Delta_b(t)$ solamente agregan errores numéricos acotados proporcionales a la cota de perturbación [12, 36]. Para los casos no lineales, asumiendo que la perturbación es suficientemente pequeña, se pueden derivar propiedades similares [39].

En caso de tener más de dos procesadores, el análisis se puede extender fácilmente llegando a los mismos resultados.

En conclusión, la paralelización introduce un error adicional al introducido por los métodos QSS, y este error adicional puede ser acotado por un valor que se incrementa de manera proporcional al parámetro Δt.

A pesar de que esta conclusión es de naturaleza cualitativa (i.e. no provee una cota cuantitativa para el error numérico adicional), en la siguiente Sección derivaremos un algoritmo para el cálculo adaptivo del parámetro Δt para de esta manera asegurar una cota cuantitativa del error introducido.

6.8. Estrategia Adaptiva

La elección del parámetro Δt implica encontrar un balance entre el error numérico introducido por la simulación en paralelo y la cantidad de cálculos que se pueden realizar en paralelo. Obtener un valor adecuado para el parámetro Δt generalmente involucra ejecutar reiteradas simulaciones para poder observar el error introducido por el parámetro y la aceleración obtenida. Adicionalmente, un valor de Δt que es adecuado al comienzo de la simulación puede no serlo luego de un tiempo debido a cambios en la dinámica del modelo.

Para resolver estos inconvenientes, desarrollamos un algoritmo adaptivo que permite calcular de manera dinámica el valor de Δt manteniendo acotado el error numérico.

Como analizamos en la sección previa, la implementación en paralelo con sincronización no estricta provoca una diferencia entre las variables de estado cuantificadas calculadas por un PL y las copias locales utilizadas en otros PLs. Esta

diferencia puede ser vista como una perturbación acotada que introduce un error numérico acotado.

La diferencia entre la variable cuantificada $q_i(t)$ y una copia local $q_i^c(t)$ es acotada de acuerdo a la Ec. (**??**). Por lo tanto, si queremos acotar esta perturbación para que sea proporcional a la tolerancia seleccionada ΔQ_i, tenemos que:

$$|q_i^c(t) - q_i(t)| \leq M_i \cdot \Delta t + 2\Delta Q_i \leq \alpha \cdot \Delta Q_i$$

Donde, M_i es una cota superior para la derivada de estado $\dot{x}_i(t)$ y α es un parámetro definido por el usuario. De esta última inecuación obtenemos

$$\Delta t \leq \frac{(\alpha - 2) \cdot \Delta Q_i}{M_i} \tag{6.7}$$

El límite impuesto a Δt debe ser cumplido por todas las variables de estado cuantificadas que tengan copias locales en otros PLs, i.e., para todas las variables de estado cuantificadas de *salida*.

El conjunto de variables de estado cuantificadas de salida para el k–ésimo PL puede ser definido formalmente como $O_k \triangleq \{i | \exists l \text{ such that } SO_{i,l}^k = 1\}$. Luego, el conjunto de todas las variables de estado cuantificadas puede definirse como $O \triangleq \bigcup_k O_k$.

Como queremos que la Ec. (**??**) se cumpla para todas las variables de estado cuantificadas de salida, el algoritmo adaptivo calcula el parámetro Δt como

$$\Delta t = \min_{i \in O} \frac{(\alpha - 2) \cdot \Delta Q_i}{M_i} \tag{6.8}$$

Con este objetivo, cada PL calcula su mínimo Δt^k como

$$\Delta t^k = \min_{i \in O_k} \frac{(\alpha - 2) \cdot \Delta Q_i}{M_i} \tag{6.9}$$

y luego Δt es calculado como el mínimo de todos los Δt^k.

Las cotas en las derivadas de estado M_i son estimadas numéricamente como[3]

$$M_i \approx \frac{|x_i(t_i) - x_i(t_i^{prev})|}{t_i - t_i^{prev}} \tag{6.10}$$

donde t_i y t_i^{prev} son los tiempos de los últimos dos cambios en la variable de estado cuantificada q_i.

[3] M_i se puede obtener de manera directa dado que $M_i = |\dot{x}_i|$, as $\dot{x}_i$ se guarda en los algoritmos QSS. Sin embargo, cuando una solución oscila cerca de un punto de equilibrio (como suele suceder frecuentemente en las soluciones obtenidas mediante métodos QSS) $\dot{x}_i$ puede ser mayor que la variación real del estado correspondiente, y en este caso, una aproximación numérica representa un mejor resultado.

Cabe mencionar que la estima M_i es actualizada cada vez que cambia la variable de estado cuantificada de salida q_i. Este cambio en M_i puede modificar el mínimo Δt^k del PL calculado por al Ec. (**??**) que puede a su vez modificar valor global del Δt. Como consecuencia, podemos ver que el valor del parámetro Δt es actualizado de manera asíncrona. Este hecho implica además la necesidad de un mecanismo de sincronización que debe ser agregado al módulo **Integrador Paralelo**.

Con este fin, el Algoritmo**??** correspondiente al **Integrador Paralelo** debe ser modificado permitir que cuando un paso de simulación cambia una variable cuantificada de salida, se actualice el mínimo local Δt^k. En caso de ser actualizado, el algoritmo controla si el nuevo valor cambia el valor global Δt. Finalmente, si el valor global es modificado, se debe invocar una rutina de sincronización global para que todos los PLs actualicen el valor del parámetro Δt.

Con este propósito, la estructura de datos de cada PL es extendida con la adición de una copia local del parámetro global Δt. Adicionalmente, un arreglo global es compartido por todos los PLs que contiene el mínimo local de cada proceso Δt^k.

Cabe mencionar que el uso de esta estrategia requiere seleccionar el *coeficiente de cota de error* α, y luego, el parámetro Δt se adapta de manera automática. En principio, seleccionar $\alpha \approx 10$ parece razonable dado que error introducido por la falta de sincronización puede ser como máximo un orden de magnitud mayor que el error introducido por la aproximación QSS.

Podemos notar que, mientras que el análisis de la Sección **??** da una cota cualitativa del error, la estrategia adaptiva para el parámetro Δt da una cota cuantitativa del error numérico, donde podemos asegurar que el mismo no puede ser mayor que α veces el error introducido por la simulación QSS secuencial. Por lo tanto, dado que el quantum ΔQ sea seleccionado de manera adecuada para el caso secuencial, la implementación en paralelo se mantendrá dentro de la tolerancia definida.

6.9. Resultados

En esta sección presentamos los resultados obtenidos en la simulación de cuatro modelos de gran escala. En estos modelos, analizamos y comparamos el ren-

dimiento del motor de simulación QSS en paralelo utilizando las estrategias de sincronización propuestas con distintas configuraciones de parámetros.

Los ejemplos están ordenados por orden de complejidad. El primer modelo, que representa el consumo de energia de una conjunto de aires acondicionados, es un sistema híbrido que no requiere comunicación entre los distintos PLs, por lo tanto resulta útil para medir la carga computacional introducida por la implementación en paralelo. El segundo ejemplo representa la discretización mediante el Método de Líneas de un modelo de Advección–Reacción de dos dimensiones de ecuaciones en derivadas parciales [12]. Este modelo representa un caso puramente continuo que permite también validar los resultados con respecto al error introducido por las estrategias de simulación en paralelo. El tercer modelo es una modificación del primer modelo donde se agrega un control centralizado para el consumo de energía, esta modificación implica que deben comunicarse los cambios discretos entre diferentes PLs. El último modelo, que representa una red de neuronas pulsantes, tiene una estructura de conexiones compleja con eventos que provocan cambios instantáneos en las variables de estado del modelo.

Configuración de los Experimentos

Los resultados presentados en las siguientes secciones fueron obtenidos utilizando un servidor de 64–núcleos (4 procesadores AMD Opteron 6272, con 16 núcleos cada uno) con 32 GB de memoria RAM corriendo un sistema operativo Linux (Ubuntu 14.04) de 64 bits utilizando la implementación del motor de simulación QSS en paralelo versión 3,1 revisión [$c10a14$].

Para poder evaluar el rendimiento de las simulaciones tomamos en cuenta las siguientes métricas:

- Tiempo de Inicialización: Que indica el tiempo consumido por las rutinas de inicialización del motor de simulación en paralelo.
- Tiempo de Simulación: Que indica el tiempo de CPU consumido por el proceso lógico más lento. En este caso no se tiene en cuenta el tiempo de inicialización.
- Consumo de Memoria: Reportamos el consumo de memoria total durante la simulación.

- Aceleración de la Simulación: $Speedup(P) = \frac{T_1}{T_P}$, donde T_1 es el tiempo de simulación consumido por el motor de simulación el paralelo utilizando un proceso lógico y T_P es el tiempo consumido utilizando P núcleos.

- Error medio normalizado[4]

$$error = \frac{mean(|\mathbf{y}^P - \mathbf{y}^{ref})|}{mean(\mathbf{y}^{ref})} \tag{6.11}$$

  donde las trayectorias de referencia $\mathbf{y}^{ref}$ son obtenidas utilizando un solo núcleo y $\mathbf{y}^P$ es la trayectoria obtenida usando P núcleos. Ambas trayectorias son evaluadas en 5000 puntos equidistantes.

  Cabe mencionar que este error no incluye el error numérico introducido por los algoritmos QSS. Solamente evalua el error numérico adicional introducido por la simulación en paralelo.

Para los diferentes modelos, se realizaron los siguientes experimentos:

1. Para poder analizar los efectos de utilizar diferentes valores en el parámetro de sincronización Δt, corrimos varias simulaciones con distintos valores. Estas simulaciones fueron realizadas utilizando 62 núcleos.[5]

2. Luego de seleccionar un valor adecuado para el parámetro Δt (i.e., un valor que permita un buen balance entre la aceleración obtenida y el error introducido) corrimos simulaciones variando el número de núcleos para poder caracterizar la aceleración con respecto al número de núcleos.

3. Para la estrategia de sincronización adaptiva, corrimos varias simulaciones variando el parámetro α con una configuración de 62 núcleos.

4. Luego, utilizando el parámetro $\alpha = 10$ (que es un valor razonable, como analizamos en las secciones previas), corrimos simulaciones variando el número de núcleos para poder observar la aceleración obtenida.

En todos los casos, los modelos fueron particionados de manera manual.

[4] Calculamos el error medio en lugar del máximo error global debido a que en los ejemplos los estados son discontinuos o tienen trayectorias con cambios abruptos. Como consecuencia, un retraso pequeño entre 2 trayectorias provoca que el error máximo sea igual a la amplitud de la trayectoria (independientemente del valor del retraso). Por lo tanto, el error medio captura mejor la diferencia real entre las trayectorias simuladas.

[5] Utilizamos un máximo de 62 núcleos para poder dejar 2 núcleos libres para las tareas del SO.

6.9.1. Conjunto de Aires Acondicionados

Este modelo, tomado de [51], estudia la dinámica de un conjunto de aires acondicionados (AAs) cuando siguen la misma temperatura de referencia. El i–ésimo AA es utilizado para controlar la temperatura de la i–ésima habitación, modelada por:

$$\dot{\theta}_i(t) = -\frac{1}{C_i \cdot R_i}[\theta_i(t) - \theta_a + R_i \cdot P_i \cdot m_i + w_i(t)] \tag{6.12}$$

donde R_i representa la resistencia térmica, C_i es la capacidad térmica, P_i representa el consumo de energía de la unidad de AA cuando está encendido, θ_a es la temperatura ambiente (común a todas las habitaciones), y $w_i(t)$ es una señal de ruido que representa perturbaciones térmicas.

La variable $m_i(t)$ representa el estado del i–ésimo AA que asume el valor 1 cuando está encendido y 0 cuando está apagado. Esta variable sigue la siguiente ley de control encendido–apagado con histéresis:

$$m_i(t^+) = \begin{cases} 0 & \text{if } \theta_i(t) \leq \theta_r^i(t) - 0{,}5 \text{ and } m_i(t) = 1 \\ 1 & \text{if } \theta_i(t) \leq \theta_r^i(t) + 0{,}5 \text{ and } m_i(t) = 0 \\ m_i(t) & \text{otherwise} \end{cases} \tag{6.13}$$

donde $\theta_r^i(t)$ es la temperatura de referencia, que tiene la siguiente forma:

$$\theta_r^i(t) = \begin{cases} 20 & \text{if } 0 \leq t < 1000 \\ 20{,}5 & \text{if } 1000 \leq t < 2000 \\ 20 & \text{if } 2000 \leq t \leq 3000 \end{cases} \tag{6.14}$$

Finalmente, las perturbaciones térmicas $w(t)$ son actualizadas cada minuto, asumiendo valores seudo-aleatorios en el rango $(-1, 1)$.

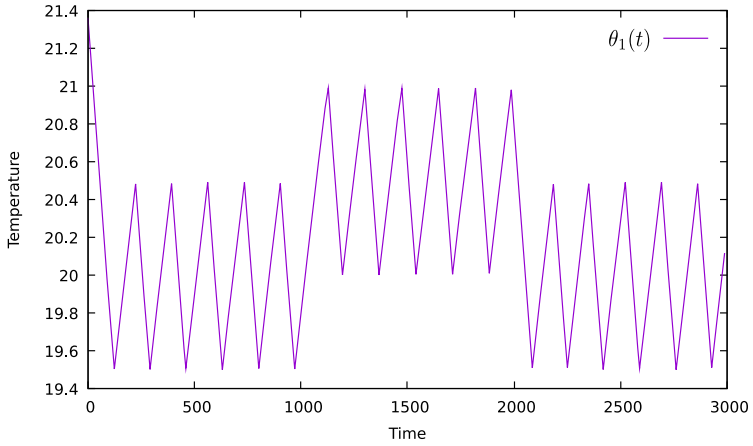
En la Figura **??** se muestra como ejemplo la temperatura del primer AA.

Para este experimento, simulamos un conjunto de 248000 AA y configuramos los parámetros de acuerdo a [51].

Cabe mencionar que cada AA es modelado por una ecuación diferencial (Ec. (**??**)) y tres funciones de cruce por cero: una asociada con la ley de control con histéresis dada por la Ec. (**??**), otra correspondiente a la evolución de la temperatura de referencia dada por la Ec. (**??**), y la última asociada a la actualización de la señal de perturbación $w_i(t)$. Por lo tanto, este modelo consta de 248000 ecuaciones diferenciales y 992000 funciones de cruce por cero.

Podemos observar también que la dinámica de cada AA es independiente de la evolución del resto de los AAs. Por lo tanto, el modelo puede ser dividido de

Figura 6.3: Conjunto de Aires Acondicionados - Trayectoria de Salida



manera tal que no exista comunicación entre los distintos PLs y en consecuencia, la falta de sincronización no introduce errores adicionales en este ejemplo.

Este modelo fue simulado utilizando el método QSS2, una cuantificación de $\Delta Q_{rel} = \Delta Q_{min} = 1e-3$ y el tiempo final de simulación t_f igual a 3000 minutos.

En este caso, el motor de simulación QSS en paralelo detecta automáticamente que no existe comunicación entre los distintos PLs y en este caso asigna al parámetro Δt el valor del tiempo final (3000). De esta manera, en este ejemplo introductorio, no corrimos simulaciones con diferentes valores de los parámetros Δt y α dado que en este ejemplo no tiene efecto alguno. Además, no reportamos el error puesto que son nulos en este caso por las razones explicadas anteriormente (en ausencia de comunicación los resultados secuenciales y paralelo son identicos).

El objetivo de este ejemplo es medir el costo computacional introducido por el mecanismo de simulación en paralelo cuando la sincronización no es necesaria. Este costo incluye el acceso a las matrices estructurales (compartidas por todos los PLs), llamadas adicionales a las rutinas de sincronización y el acceso a la estructura de comunicación inter–proceso.

Corrimos simulaciones variando el número de núcleos desde 1 hasta 62. También probamos las diferentes políticas para la distribución de memoria: utilizando copias completas del estado y utilizando arreglos de estado compartidos. Los resultados obtenidos son reportados en la Tabla **??**.

Tabla 6.1: Conjunto de Aires Acondicionados - Resultados

Núcleos	Tiempo Inicio (ms)	Tiempo Simulación (segundos)	Memoria (MBytes)	Aceleración	Aceleración Estado Compartido
1	580	1132	557	1,00	1,00
2	894	668	753	1,69	1,68
4	948	357	1146	3,17	2,50
8	969	192	1933	5,89	3,69
16	1085	109	3538	10,38	8,02
32	1165	52	6750	21,76	14,51
48	1281	32	9961	35,37	18,55
62	1406	25	12779	45,28	22,19

Como podemos ver en la Tabla **??**, la aceleración obtenida para variando la cantidad de núcleos en este modelo es lineal. Corriendo la simulación con 62 núcleos la simulación se acelera 45 veces utilizando copias completas del estado para cada proceso y 22 veces utilizando arreglos compartidos. Los tiempos de inicialización se incrementan con el número de núcleos (la inicialización incluye calcular y reservar memoria para las matrices de comunicación inter–procesos, que crecen en complejidad con el número de núcleos). De todas maneras, los tiempos de inicialización son despreciables en todos los casos.

El consumo de memoria por otro lado, crece de manera casi lineal con el número de núcleos. Esto es debido al hecho de que cada PL tiene copias locales de los estados, estados cuantificados, etc., y adicionalmente, la estructura de la casilla de correo y las matrices de comunicación inter–procesos.

El bajo rendimiento obtenido al utilizar arreglos compartidos puede ser explicado por la arquitectura NUMA del servidor utilizado para hacer estos experimentos. Por esta razón, en los siguientes modelos utilizamos copias completas del estado para cada proceso.

Adicionalmente, comparamos el tiempo de simulación requerido por la implementación secuencial del motor de simulación QSS. Este experimento llevó 1050 segundos, que es aproximadamente un %8 más rápido que la implementación en paralelo utilizando sólo un núcleo. Esta sobrecarga se debe a los mecanismos de control adicionales del motor paralelo (para comunicación, sincronización y acceso a las matrices inter–procesos). Estos mecanismos no son deshabilitados cuando se corre con un núcleo a pesar de no ser necesarios. Realizamos experimentos similares en el resto de los modelos presentados llegando siempre a una sobrecarga similar.

Por último podemos decir que, en este caso, la simulación utilizando solo un núcleo demora menos de 20 minutos, por lo que es cuestionable tratar este modelo como un modelo de gran escala. De todas maneras, este es sólo un ejemplo introductorio, donde el tiempo final puede ser fácilmente extendido en un orden de magnitud llegando a conclusiones idénticas.

6.9.2. Advección–Reacción

El siguiente modelo, presentado en [6], representa una ecuación de Advección–Reacción en dos dimensiones. Esta ecuación puede describir, por ejemplo, un río transportando un substancia que experimenta una reacción química.

Luego de aplicar el método de líneas en esta EDP, se obtiene el siguiente conjunto de EDOs:

$$\dot{u}_{i,j} = -a_x \frac{u_{i,j} - u_{i,j-1}}{\Delta x} - a_y \frac{u_{i,j} - u_{i-1,j}}{\Delta y} + r u_{i,j}(u_{i,j} - \alpha)(u_{i,j} - 1) \tag{6.15}$$

para $i = 2, \cdots N$, $j = 2 \cdots M$, donde $u_{i,j}(t)$ es la concentración de la substancia transportada en el punto i, j de la grilla del dominio espacial. Los parámetros a_x y a_y representan la velocidad del flujo de transporte en las coordinadas x y y respectivamente, y r es la tasa de cambio de la reacción química. Finalmente, Δx y Δy representan la amplitud de cada sección de la grilla.

En los bordes, la dinámica está definida por

$$\dot{u}_{i,1} = -a_x \frac{u_{i,1}}{\Delta x} - a_y \frac{u_{i,1} - u_{i-1,1}}{\Delta y} + r u_{i,1}(u_{i,1} - \alpha)(u_{i,1} - 1) \tag{6.16}$$

para $i = 2, \cdots N$

$$\dot{u}_{1,j} = -a_x \frac{u_{1,j} - u_{1,j-1}}{\Delta x} - a_y \frac{u_{1,j}}{\Delta y} + r u_{1,j}(u_{1,j} - \alpha)(u_{1,j} - 1) \tag{6.17}$$

para $j = 2, \cdots M$, y,

$$\dot{u}_{1,1} = -a_x \frac{u_{1,1}}{\Delta x} - a_y \frac{u_{1,1}}{\Delta y} + r u_{1,1}(u_{1,1} - \alpha)(u_{1,1} - 1) \tag{6.18}$$

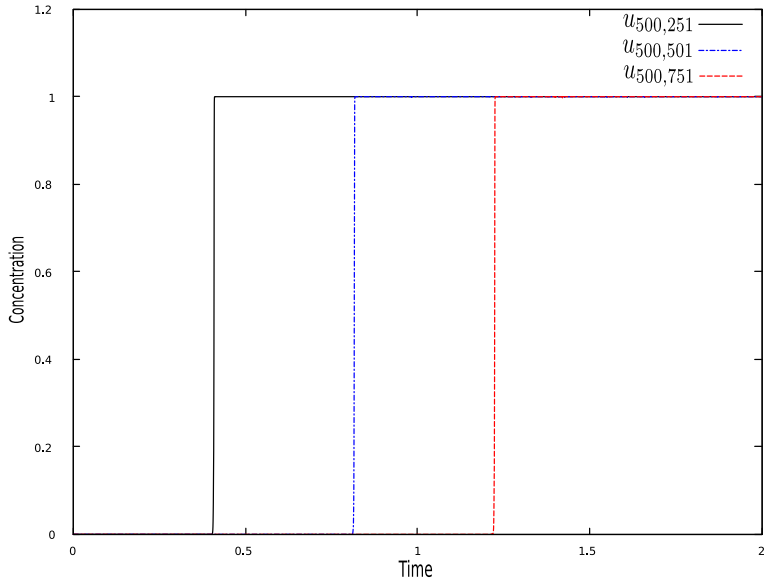
Las condiciones iniciales son las siguientes

$$u_{i,1} = 1 \tag{6.19}$$

para $i = 1, \cdots N$ y $u_{i,j} = 0$ en otro caso.

La Figura **??** muestra como ejemplo la concentración de tres variables en la mitad de la grilla, $u_{750,251}$, $u_{750,501}$ y $u_{750,751}$.

Figura 6.4: Modelo de Advección–Reacción - Trayectorias de Salida



Teniendo en cuenta que el modelo es *stiff* (las reacciones químicas son mucho más rápidas que la dinámica del transporte) el algoritmo linealmente implícito LIQSS2 fue utilizado en esta caso, configurando la tolerancia a $\Delta Q_{rel} = \Delta Q_{abs} = 1e-3$. los parámetros del modelo son $r = 10000$, $a_x = 1$, $a_y = 0{,}1$, $\Delta x = 1/N$, $\Delta y = 1/M$ con $N = M = 1500$. De esta manera, tenemos un dominio espacial de 1×1 con $1500\times = 2250000$ puntos en la grilla, donde cada punto en la grilla representa una variable de estado.

La Tabla **??** reporta los resultados obtenidos para diferentes valores del parámetro Δt utilizando 62 núcleos. Aquí, el error reportado es la media del error obtenido en 20 variables de salida $u_{i,j}(t)$ tomadas en una línea en el medio de la grilla. En cada variable el error fue calculado utilizando la Ec. (**??**).

Tabla 6.2: Modelo Advección–Reacción - Δt Fijo - Utilizando 62 Núcleos

Δt	**Tiempo Simulación (segundos)**	**Aceleración**	**Error**
$1e-07$	1104	2,40	$3{,}61e-06$
$1e-06$	288	9,20	$3{,}45e-06$
$1e-05$	145	18,28	$3{,}66e-06$
$1e-04$	119	22,27	$6{,}22e-06$
$1e-03$	121	21,90	$1{,}38e-05$
$1e-02$	115	23,05	$2{,}12e-05$
$1e-01$	112	23,66	$2{,}39e-04$

Como esperábamos, al aumentar el valor de Δt aumentan la aceleración y el error introducido. Sin embargo, cuando el valor de Δt es muy grande, el rendimiento disminuye. Esto es debido a que la sincronización entre PLs es escasa y esto implica que las listas ordenadas que mantienen los PLs se hacen muy grandes y el costo de ordenarlas es muy alto.

Como podemos ver en los resultados, el mejor rendimiento es obtenido utilizando $\Delta t = 1e-4$. Por lo tanto, asignamos este valor a Δt y calculamos la aceleración obtenida variando el número de núcleos (Tabla **??**).

Tabla 6.3: Modelo Advección–Reacción - $\Delta t = 1e-04$ Fijo

Núcleos	**Tiempo Inicio (ms)**	**Tiempo Simulación (segundos)**	**Memoria (MBytes)**	**Aceleración**	**Error**
1	174	2650	1058	1,00	0
2	699	2248	1626	1,18	$3{,}63e-06$
4	666	1216	2763	2,18	$3{,}66e-06$
8	748	715	5035	3,71	$3{,}33e-06$
16	850	375	9580	7,07	$3{,}14e-06$
32	1100	203	14573	13,05	$5{,}44e-06$
48	1300	153	23647	17,32	$4{,}83e-06$
62	1460	119	31614	22,27	$6{,}22e-06$

Podemos ver que el error siempre tiene el mismo orden de magnitud. Al igual que en el modelo anterior, el consumo de memoria y el tiempo de inicialización crecen de manera lineal con el número de núcleos (los tiempos de inicialización son nuevamente despreciables). Finalmente, la aceleración llega a un valor razonable de 22 veces para 62 núcleos, creciendo de manera casi lineal con el número de núcleos.

También simulamos el modelo utilizando la estrategia adaptiva para el parámetro Δt, variando el parámetro α, los resultados obtenidos en este caso son repor-

tados en la Tabla **??**.

Tabla 6.4: Modelo Advección–Reacción - Δt Adaptivo - Utilizando 62 Núcleos

α	**Tiempo Simulación (segundos)**	**Aceleración**	**Error**
2	169	15,68	$2{,}96e-06$
10	135	19,63	$4{,}50e-06$
20	124	21,37	$4{,}57e-06$
50	122	21,72	$6{,}71e-06$
100	127	20,87	$9{,}33e-06$

Como mencionamos en la Sección **??**, los mejores resultados fueron obtenidos utilizando $\alpha = 10$, logrando una aceleración similar a la obtenida utilizando $\Delta t = 1e-4$, pero con un error menor. Se puede notar que seleccionando α la aceleración y el error son robustos.

Luego, asignando $\alpha = 10$, corrimos simulaciones variando el número de núcleos, obteniendo los resultados reportados en la Tabla **??**.

Tabla 6.5: Modelo Advección–Reacción - Δt Adaptivo - $\alpha = 10$

Núcleos	**Tiempo Inicio (ms)**	**Tiempo Simulación (segundos)**	**Memoria (MBytes)**	**Aceleración**	**Error**
1	174	2650	1058	1,00	0
2	672	2251	1626	1,18	$3{,}63e-06$
4	720	1218	2763	2,18	$3{,}54e-06$
8	733	716	5035	3,70	$3{,}75e-06$
16	871	381	9580	6,96	$3{,}05e-06$
32	1015	207	14573	12,80	$3{,}70e-06$
48	1270	178	23647	14,89	$4{,}99e-06$
62	1468	135	31614	19,63	$4{,}50e-06$

Podemos ver que el error no cambia de manera significativa y que la aceleración sigue una evolución muy similar a la obtenida utilizando el parámetro $\Delta t = 1e-4$.

Cabe mencionar que para obtener el valor óptimo $\Delta t = 1e-4$ debimos ejecutar varios experimentos, mientras que $\alpha = 10$ es la opción por defecto para el algoritmo adaptivo.

Para poder comparar el desempeño del simulador en paralelo con simuladores en paralelo para métodos clásicos, simulamos el mismo modelo utilizando la extensión en paralelo del paquete CVODE llamada PVODE [11] en el mismo servidor y con la misma configuración para la tolerancia. PVODE utiliza la librería

OpenMP y el método numérico de Adams (BDF no funcionó debido al tamaño del problema). Los resultados obtenidos se reportan en la Tabla **??**.

Tabla 6.6: Modelo Advección–Reacción - CVODE - Resultados

Núcleos	**Tiempo Simulación (segundos)**	**Aceleración**
1	12552	1
2	6814	1,84
4	3581	3,50
8	1854	6,77
16	1293	9,70
32	909	13,80
48	741	16,93
62	751	16,71

Podemos ver que la aceleración es casi lineal hasta llegar a 48 núcleos, donde satura. El motor de simulación QSS en paralelo es 4.73 veces más rápido corriendo con un núcleo. Luego, para 62 núcleos, el motor de simulación QSS es aproximadamente 6 veces más rápido que PVODE.

Estos resultados muestran que el simulador QSS es más rápido en una simulación secuencial y la diferencia se vuelve mayor utilizando 62 núcleos. Cabe mencionar que el motor de simulación QSS no está optimizado para tratar arreglos de 2 dimensiones, por lo que el código generado es muy ineficiente.

6.9.3. Control del Consumo de Energía de un Conjunto de Aires Acondicionados

Este modelo, también presentado en [51], extiende el modelo presentado en la Sección **??**, agregando un control centralizado del consumo de energía total. Con este fin, el conjunto de AA es dividido en secciones locales $S = \{s_1, \cdots, s_n\}$ y el consumo de energía de cada sección es calculado como:

$$p_i(t_k) = \sum_{j \in s_k} P_j \tag{6.20}$$

y el consumo total de energía es calculado como:

$$P(t_k) = \sum_{i \in S} p_i \tag{6.21}$$

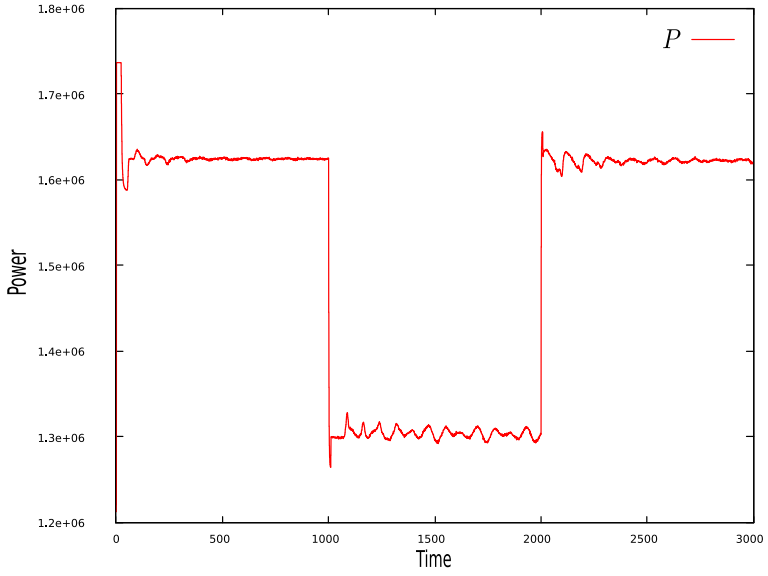
El sistema de control global regula la energía total $P(t_k)$ para poder seguir el control deseado $P_r(t)$. Para esto, se utiliza una ley de control de integral propor-

cional (PI) para calcular la temperatura de referencia común:

$$\theta_r(t_k) = K_P \cdot [P_r(t_k) - P(t_k)] + K_L \cdot \int_{\tau=0}^{t_k} [P_r(\tau) - P(\tau) d\tau \tag{6.22}$$

En la Figura **??** muestra la salida del control en una simulación secuencial del modelo.

Figura 6.5: Control del Consumo de Energía de un Conjunto de Aires Acondicionados - Trayectorias de Salida



Simulamos un conjunto de 248000 AAs con 248 secciones locales, por lo que en este caso el modelo consta de 248000 variables de estado y 496498 funciones de cruce por cero. Utilizamos el método QSS3 con tolerancias de $\Delta Q_{rel} = 1e-4 = \Delta Q_{min} = 1e-4$ y los parámetros del modelo configurados como en la Sección **??**.

El error introducido por la simulación en paralelo fue medido en la variable $(P(t_k))$ que calcula el consumo de energía total de los AAs de acuerdo a la Ec. (**??**).

La Tabla **??** reporta los resultados de simulación obtenidos para diferentes valores del parámetro Δt utilizando 62 núcleos.

Como esperábamos, la aceleración y el error crecen a medida que crece el valor de Δt. A partir de $\Delta t = 1$, no se observan ganancias apreciables en la aceleración, por lo que utilizamos este valor para los siguientes experimentos. Esta vez, la aceleración máxima alcanzada es del orden de 35 veces para 62 núcleos (era de 52 veces en el primer ejemplo, que no requería sincronización). En este ejemplo,

Tabla 6.7: Control del Consumo de Enegía de un Conjunto de Aires Acondicionados - Δt Fijo - 62 Núcleos

Δt	**Tiempo Simulación (segundos)**	**Aceleración**	**Error**
$1e-05$	2009	1,56	$1{,}42e-04$
$1e-04$	761	4,13	$1{,}82e-04$
$1e-03$	432	7,27	$1{,}81e-04$
$1e-02$	250	12,56	$1{,}85e-04$
$1e-01$	200	15,70	$2{,}95e-04$
$5e-01$	94	33,39	$4{,}99e-04$
1	90	34,88	$5{,}50e-04$
2	124	25,32	$5{,}84e-04$

los diferentes PLs deben comunicar cambios discretos (cambios en el consumo de energía de cada sección o cambios en la temperatura de referencia global). Como explicamos en la Sección **??** comunicar cambios discretos fuerza que los PLs que envían estos mensajes a esperar a que todos los PLs que reciben los mensajes procesen el cambio correspondiente, y este hecho aumenta los tiempos de simulación.

Luego, utilizando $\Delta t = 1$, simulamos el modelo variando el número de núcleos obteniendo los resultados reportados en la Tabla **??**.

Tabla 6.8: Control del Consumo de Enegía de un Conjunto de Aires Acondicionados - $\Delta t = 1$ Fijo

Núcleos	**Tiempo Inicio (ms)**	**Tiempo Simulación (segundos)**	**Memoria (MBytes)**	**Aceleración**	**Error**
1	106	3139	458	1,00	0
2	240	1830	655	1,72	$9{,}18e-04$
4	267	1019	1048	3,08	$1{,}02e-03$
8	294	648	1835	4,84	$2{,}02e-04$
16	382	426	3440	7,37	$1{,}25e-03$
32	526	176	6651	17,84	$5{,}46e-04$
48	673	117	9830	26,83	$5{,}72e-04$
62	796	90	12648	34,88	$5{,}50e-04$

Los resultados son muy similares a los obtenidos en el ejemplo previo en término de tiempos de inicialización, consumo de memoria y el error introducido. La aceleración, nuevamente, crece de manera casi lineal.

Luego, simulamos el sistema con el algoritmo adaptivo para diferentes valores de α, obteniendo los resultados reportados en la Tabla **??**.

Tabla 6.9: Control del Consumo de Energía de un Conjunto de Aires Acondicionados - Δt Adaptivo - 62 Núcleos

α	**Tiempo Simulación (segundos)**	**Aceleración**	**Error**
2	128	24,53	$3{,}88e-04$
10	91	34,50	$1{,}47e-03$
20	92	34,12	$1{,}46e-03$
50	89	35,27	$1{,}39e-03$
100	90	34,88	$1{,}59e-03$

Nuevamente, para el valor $\alpha = 10$ se obtiene el mejor rendimiento. Utilizando este parámetro, simulamos el modelo variando el número de núcleos, obteniendo los resultados reportados en la Tabla **??**.

Tabla 6.10: Control del Consumo de Energía de un Conjunto de Aires Acondicionados - Δt Adaptivo - $\alpha = 10$

Núcleos	**Tiempo Inicio (ms)**	**Tiempo Simulación (segundos)**	**Memoria (MBytes)**	**Aceleración**	**Error**
1	106	3139	458	1,00	0
2	195	1799	655	1,75	$1{,}36e-03$
4	209	1032	1048	3,04	$1{,}43e-03$
8	210	651	1835	4,82	$4{,}77e-04$
16	251	426	3440	7,37	$1{,}33e-03$
32	313	172	6651	18,25	$9{,}01e-04$
48	381	117	9830	26,83	$1{,}09e-03$
62	425	91	12648	34,50	$1{,}47e-03$

Otra vez, los resultados obtenidos son muy similares a los obtenidos con el para el parámetro $\Delta t = 1$ reportados en la Tabla **??**.

6.9.4. Red Neuronal Pulsante

El último modelo, adaptado de [62], representa una red de neuronas pulsantes. La i–ésima neurona es modelada por tres ecuaciones diferenciales:

$$\begin{aligned} \tau \cdot \dot{v}_i(t) &= v_{rest} - v_i(t) + g_i^{ex}(t) \cdot (E^{ex} - v_i(t)) + g_i^{inh}(t) \cdot (E^{inh} - v_i(t)) \\ \tau^{ex} \cdot \dot{g}_i^{ex}(t) &= -g_i^{ex}(t) \\ \tau^{inh} \cdot \dot{g}_i^{inh}(t) &= -g_i^{inh}(t) \end{aligned}$$

donde $v_i(t)$ representa el potencial de membrana, $g_i^{ex}(t)$ es la conductancia exitatoria, y $g_i^{inh}(t)$ es la conductancia inhibitoria. La definición y los valores de los restantes parámetros se pueden encontrar en [62].

Cuando el potencial de membrana $v_i(t)$ alcanza el valor límite -50, la neurona se descarga, lo que a su vez cambia su valor de manera instantánea a $v_i(t) = v_{rest} = -60$. La descarga es comunicada a todas las neuronas postsinápticas conectadas, que cambian el valor de las conductancias exitatorias e inhibitorias dependiendo del tipo de neurona que provoca el cambio. Si la i–ésima neurona es exitatoria, las neuronas postsinápticas conectadas cambian su conductancia exitatoria de acuerdo a:

$$g_j^{ex}(t^+) = g_h^{ex}(t) + \Delta g^{ex}$$

para todo $j \in Post_i$ (el conjunto de neuronas postsinápticas de la neurona i). Por otro lado, si la i–ésima neurona es de tipo inhibitorio, la conductancia inibitoria es modificada de la siguiente manera:

$$g_j^{inh}(t^+) = g_h^{inh}(t) + \Delta g^{inh}$$

para todo $j \in Post_i$. Para estas conexiones sinápticas, utilizamos parámetros $\Delta g^{ex} = 0{,}4$ y $\Delta g^{inh} = 1{,}6$.

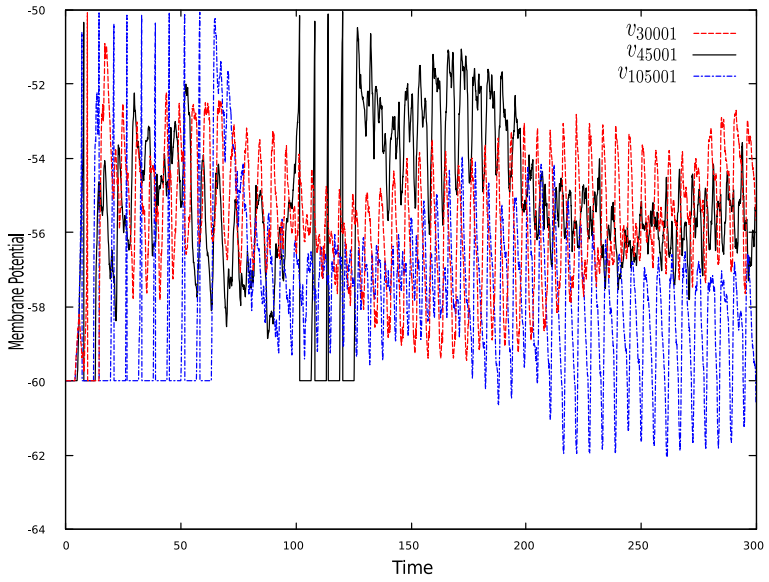
Luego de la descarga de una neurona, la misma entra en un período refractario en el que no cambia su potencial de membrana.

En este ejemplo, consideramos una red de 300,000 neuronas, con el 80 % de tipo exitatorio y el 20 % de tipo inhibitorio distribuidas de manera aleatoria. Cada neurona tiene 200 conexiones postsinápticas, limitadas a las 800 neuronas más cercanas.

Adicionalmente, existe un conjunto de 60,000 neuronas que reciben impulsos de una fuente externa durante los primeros 100 milisegundos.

El sistema tiene un total de 900,000 ecuaciones de estado y 660,000 funciones de cruce por cero. En la Figura **??** se muestra como ejemplo las trayectorias de salida de tres neuronas.

Figura 6.6: Modelo de Red de Neuronas Pulsantes - Trayectorias de Salida



Simulamos este sistema utilizando el método QSS2 con tolerancias de $\Delta Q_{rel} = \Delta Q_{abs} = 1e-3$ y el tiempo final $t_f = 300$.

Este modelo tiene un comportamiento caótico y un pequeño cambio en las tolerancias genera un resultado totalmente diferente si miramos la salida una neurona individual. Por lo tanto, en este caso, en lugar de medir el error cometido, medimos la actividad de la red neuronal contando el número de pulsos generados a lo largo de la simulación.

Debido a este comportamiento caótico, no solo cambia el error sino que también la aceleración se ve afectada entre diferentes simulaciones utilizando la misma configuración de parámetros. Por lo tanto, los resultados reportados en este caso son el promedio de 10 simulaciones.

Como en los ejemplos previos, comenzamos variando el parámetro Δt utilizando 62 núcleos. Los resultados son reportados en la Tabla **??**.

Como esperábamos, la aceleración crece a medida que aumenta el valor de Δt llegando a un máximo de 25 veces. Con respecto al número de pulsos, podemos

Tabla 6.11: Modelo de Red de Neuronas Pulsantes - Δt Fijo - 62 Núcleos

Δt	**Tiempo Simulación (segundos)**	**Aceleración**	**Pulsos Promedio**
$1e-04$	2201	2,45	15,80
$1e-03$	726	7,44	15,92
$1e-02$	410	13,17	15,92
$1e-01$	304	17,76	15,93
$5e-01$	227	23,79	15,87
$1e+00$	214	25,24	15,84
$2e+00$	216	25,00	15,62

observar que no experimenta cambios significativos, por lo que podemos concluir que la actividad de la red es similar.

La Tabla **??** muestra los resultados obtenidos variando el número de núcleos para $\Delta t = 0{,}5$.

Tabla 6.12: Modelo de Red de Neuronas Pulsantes - $\Delta t = 5e-1$ Fijo

Núcleos	**Tiempo Inicio (ms)**	**Tiempo Simulación (segundos)**	**Memoria (MBytes)**	**Aceleración**	**Pulsos Promedio**
1	1244	5402	2555	1,00	15,84
2	1806	3331	2916	1,62	15,86
4	1809	2030	3670	2,66	15,94
8	2071	1349	5177	4,00	15,85
16	2176	1012	8257	5,33	15,93
32	2594	439	14352	12,30	16,16
48	3104	288	20480	18,75	15,81
62	3536	231	25851	23,38	15,80

Los tiempos de inicialización, el consumo de memoria, y la aceleración muestran evoluciones similares a los ejemplos previos, y nuevamente, la aceleración es casi lineal.

Los resultados obtenidos al utilizar al algoritmo adaptivo variando el valor de α son reportados en la Tabla **??**.

Esta vez, los mejores resultados son obtenidos para el valor $\alpha = 100$. Sin embargo, dado el comportamiento caótico del modelo, no podemos estimar los errores introducidos por la falta de sincronización.

Tabla 6.13: Modelo de Red de Neuronas Pulsantes - Δt Adaptivo - 62 Núcleos

α	**Tiempo Simulación (segundos)**	**Aceleración**	**Pulsos Promedio**
2	388	13,92	15,94
10	344	15,70	15,90
20	315	17,14	16,02
50	252	21,43	15,75
100	236	22,88	15,84

Utilizando $\alpha = 10$ y variando el número de núcleos, se obtienen los resultados reportados en la Tabla **??**.

Tabla 6.14: Modelo de Red de Neuronas Pulsantes - Δt Adaptivo - $\alpha = 10$

Núcleos	**Tiempo Inicio (ms)**	**Tiempo Simulación (segundos)**	**Memoria (MBytes)**	**Aceleración**	**Pulsos Promedio**
1	1244	5402	2555	1,00	15,84
2	1805	3367	2916	1,60	16,41
4	1806	2040	3670	2,64	15,50
8	1935	1358	5177	3,97	16,04
16	2196	1049	8257	5,14	15,79
32	2654	486	14352	11,11	15,77
48	3123	354	20480	15,25	15,93
62	3512	344	25851	15,70	15,84

Capítulo 7

Conclusiones

En esta Tesis se presentaron nuevas metodologías y algoritmos que permiten la simulación eficiente de modelos de gran escala utilizando métodos de aproximación numérica QSS. Con este fin, en una primera parte, se desarrolló un entorno de modelado y simulación autónomo para métodos QSS que permite simular modelos híbridos de manera eficiente, donde se pueden describir modelos en un lenguaje de modelado denominado μ–Modelica y a partir de esta especificación se extrae toda la información estructural adicional del modelo que necesita el motor de simulación autónomo QSS de manera automática.

Como parte de este trabajo se muestran luego diferentes ejemplos de aplicación que permiten analizar el rendimiento de la nueva herramienta desarrollada mostrando que supera en más de un orden de magnitud a las implementaciones previas de los métodos QSS basadas en el formalismo DEVS y comparando también su eficiencia contra métodos de integración clásicos.

Dado que la definición de modelos utilizando μ–Modelica puede resultar complejo para el usuario final, como continuación del del trabajo, el entorno de modelado y simulación autónomo para métodos QSS fue integrado con diferentes herramientas de modelado y simulación que permiten ampliar el campo de aplicación de los métodos QSS y a su vez nos permite evaluar su rendimiento.

Luego, se estudio en profundidad el desempeño del simulador en comparación con diferentes métodos de integración numérica clásicos, analizando y mostrando las ventajas que presenta su aplicación a modelos provenientes de diferentes campos de las ciencia y la ingeniería

Como siguiente objetivo en el desarrollo de esta Tesis se propuso poder simular modelos de gran escala de manera eficiente y en una segunda parte de trabajo, se

presentan dos nuevas técnicas de simulación en paralelo específicamente diseñadas para los métodos QSS.

Esta nuevas metodologías están basadas en el uso de un mecanismo de sincronización no estricto entre los diferentes procesos lógicos e introduce un error numérico adicional al de la aproximación QSS. Demostramos formalmente que este error adicional introducido es acotado y que depende de un parámetro dado Δt, diseñando también una estrategia adaptiva que permite actualizar este parámetro de manera dinámica durante la simulación permitiendo de esta manera mantener la cota del error introducida proporcional a la tolerancia seleccionada para el método QSS.

Los resultados obtenidos para modelos de gran escala (uno de ellos puramente continuo y el resto híbridos) muestran una aceleración casi lineal en los tiempos de simulación obtenidos. Para un máximo de 62 núcleos, la aceleración obtenida se mantuvo entre 24 y 52 veces en los diferentes experimentos realizados. Teniendo en cuenta que los algoritmos QSS secuenciales para los problemas presentados son al menos un orden de magnitud más rápidos que los métodos de integración clásicos, los resultados obtenidos con la implementación en paralelo aumentan de manera significativa las diferencias existentes.

Adicionalmente, lo resultados reportados en esta Tesis son los primeros en simular modelos de un tamaño del orden del millón de variables.

Como trabajos a futuro se pueden mencionar:

- Extender el lenguaje μ–Modelica para permitir el uso de arreglos de más de una dimensión en sistemas híbridos de manera eficiente. Esto implica plantear una forma genérica de obtener las matrices de incidencia del sistema en presencia de arreglos multidimensionales y mejorar de manera significativa la calidad del código de simulación generado.

- Integrar al simulador con algoritmos de partición de grafos de manera tal que sea posible obtener particiones del modelo a simular en paralelo de forma automática. Esto implica definir un modelo de grafo correspondiente a una simulación QSS, identificando las diferentes unidades de calculo como así también la comunicación existente entre ellas.

- Investigar la extensión de las técnicas de simulación en paralelo presentadas a arquitecturas distribuidas. Esto implica principalmente desarrollar e implementar de manera eficiente los algoritmos de control de tiempo global de la simulación en paralelo.

Finalmente, como conclusión, el trabajo presentado en esta Tesis muestra el potencial de la aplicación de los algoritmos QSS para simular de manera eficiente problemas complejos planteados en diferentes áreas de la ingeniería y la ciencia en general.

Bibliografía

[1] Pierluigi Amodio and Luigi Brugnano. Parallel solution in time of odes: some achievements and perspectives. *Applied Numerical Mathematics*, 59(3):424–435, 2009.

[2] Tamara Beltrame and François E Cellier. Quantised state system simulation in dymola/modelica using the devs formalism. In *Proceedings of the 5th International Modelica Conference*, pages 73–82, 2006.

[3] F. Bergero, J. Fernández, E. Kofman, and M. Portapila. Quantized State Simulation of Advection–Diffusion–Reaction Equations. In *Mecánica Computacional*, volume XXXII, pages 1103–1119, Mendoza, Argentina, 2013. Asociación Argentina de Mecánica Computacional.

[4] 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.

[5] Federico Bergero. Modelica models download. Internal report, 2012. http://www.fceia.unr.edu.ar/ fbergero/modelica2012.

[6] Federico Bergero, Joaquín Fernández, Ernesto Kofman, and Margarita Portapila. Time Discretization versus State Quantization in the Simulation of a 1D Advection-Diffusion-Reaction Equation. *Simulation: Transactions of the Society for Modeling and Simulation International*, 92(1):47–61, 2016.

[7] Federico Bergero, Xenofon Floros, Joaquín Fernández, Ernesto Kofman, and François E. Cellier. Simulating Modelica models with a Stand–Alone Quantized State Systems Solver. In *9th International Modelica Conference*, pages 237–246, Munich, Germany, 2012.

[8] Federico Bergero, Ernesto Kofman, and FranÃ§ois E. Cellier. A Novel Parallelization Technique for DEVS Simulation of Continuous and Hybrid Systems. *Simulation: Transactions of the Society for Modeling and Simulation International*, 89(6):663–683, 2013.

[9] P. Brown, A. Hindmarsh, and L. Petzold. Using Krylov methods in the solution of large-scale differential-algebraic systems. *SIAM Journal on Scientific Computing*, 15(6):1467–1488, 1994.

[10] Dag Brück, Hilding Elmqvist, Sven Erik Mattsson, and Hans Olsson. Dymola for multi-engineering modeling and simulation. In *Proceedings of modelica*, volume 2002. Citeseer, 2002.

[11] George D Byrne and Alan C Hindmarsh. Pvode, an ode solver for parallel computers. *International Journal of High Performance Computing Applications*, 13(4):354–365, 1999.

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

[13] Mariana C D'Abreu and Gabriel A Wainer. M/cd++: modeling continuous systems using modelica and devs. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 229–236. IEEE, 2005.

[14] Francisco Esquembre. Easy java simulations: A software tool to create scientific simulations in java. *Computer Physics Communications*, 156(2):199–204, 2004.

[15] Charbel Farhat and Marion Chandesris. Time-decomposed parallel time-integrators: Theory and feasibility studies for uid, structure, and fluid–structure applications. *Int. J. Numer. Meth. Engng*, 58:1397–1434, 2003.

[16] J. Fernáfndez and E. Kofman. A Stand–Alone Quantized State System Solver. Part I. In *Proc. of RPIC 2013*, Bariloche, Argentina, 2013.

[17] J. Fernández and E. Kofman. μ–modelica language specificacion. www.fceia.unr.edu.ar/control/modelica/micromodelicaspec.pdf, CIFASIS - CONICET, 2012.

[18] J. Fernández and E. Kofman. A Stand–Alone Quantized State System Solver. Part II. In *Proc. of RPIC 2013*, Bariloche, Argentina, 2013.

[19] Joaquín Fernández and Ernesto Kofman. A Stand-Alone Quantized State System Solver for Continuous System Simulation. *Simulation: Transactions of the Society for Modeling and Simulation International*, 90(7):782–799, 2014.

[20] Andrew Finney and Michael Hucka. Systems biology markup language: Level 2 and beyond. *Biochemical Society Transactions*, 31(6):1472–1473, 2003.

[21] X. Floros, F. Bergero, F. Cellier, and E. Kofman. Automated Simulation of Modelica Models with QSS Methods - The Discontinuous Case. In *Proc. 8th International Modelica Conference*, Dresden, Germany, 2011.

[22] X. Floros, F. Cellier, and E. Kofman. Discretizing Time or States? A Comparative Study between DASSL and QSS. In *Proc. 3rd International Workshop on Equation-based Object-oriented Modeling Languages and Tools*, pages 107–115, Oslo, Norway, 2010.

[23] Xenofon Floros. *Exploiting model structure for efficient hybrid dynamical systems simulation*. PhD thesis, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 21871, 2014, 2014.

[24] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-Interscience, New York, 2004.

[25] Peter Fritzson. *Principles of object-oriented modeling and simulation with Modelica 2.1*. John Wiley & Sons, 2010.

[26] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The openmodelica modeling, simulation, and development environment. In *46th Conference on Simulation and Modelling of the Scandinavian Simulation Society (SIMS2005), Trondheim, Norway, October 13-14, 2005*, 2005.

[27] Richard M Fujimoto. *Parallel and distributed simulation systems*, volume 300. Wiley New York, 2000.

[28] B. H. Gilding and R. Kersner. Travelling waves in nonlinear diffusion-convection-reaction. Memorandum 1585, Department of Applied Mathematics, University of Twente, Enschede, 2001.

[29] G. Grinblat, H. Ahumada, and E. Kofman. Quantized State Simulation of Spiking Neural Networks. *Simulation: Transactions of the Society for Modeling and Simulation International*, 88(3):299–313, 2012.

[30] Ernst Hairer, Syvert Paul Nrsett, and Gerhard Wanner. *Solving Ordinary Differential Equations: Nonstiff problems. v. 2: Stiff and differential-algebraic problems*. Springer Verlag, 2010.

[31] Michael Hucka, Andrew Finney, Herbert M Sauro, Hamid Bolouri, John C Doyle, Hiroaki Kitano, Adam P Arkin, Benjamin J Bornstein, Dennis Bray, Athel Cornish-Bowden, et al. The systems biology markup language (sbml): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.

[32] Willem Hundsdorfer and Jan G. Verwer. *Numerical Solution of Time-Dependent Advection-Diffusion-Reaction Equations*. Springer, 2003.

[33] Shafagh Jafer and Gabriel Wainer. Global lookahead management (glm) protocol for conservative devs simulation. In *Distributed Simulation and Real Time Applications (DS-RT), 2010 IEEE/ACM 14th International Symposium on*, pages 141–148. IEEE, 2010.

[34] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[35] Natalia Kalinnik, Matthias Korch, and Thomas Rauber. Online auto-tuning for the time-step-based parallel solution of odes on shared-memory systems. *Journal of Parallel and Distributed Computing*, 74(8):2722–2744, 2014.

[36] E. Kofman. A Second Order Approximation for DEVS Simulation of Continuous Systems. *Simulation: Transactions of the Society for Modeling and Simulation International*, 78(2):76–89, 2002.

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

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

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

[40] Matthias Korch and Thomas Rauber. Optimizing locality and scalability of embedded runge–kutta solvers using block-based pipelining. *Journal of Parallel and Distributed Computing*, 66(3):444–468, 2006.

[41] Matthias Korch and Thomas Rauber. Locality optimized shared-memory implementations of iterated runge-kutta methods. In *Euro-Par 2007 Parallel Processing*, pages 737–747. Springer, 2007.

[42] J Lions, Yvon Maday, and Gabriel Turinici. A"parareal"in time discretization of pde's. *Comptes Rendus de l'Academie des Sciences Series I Mathematics*, 332(7):661–668, 2001.

[43] Martina Maggio, Kristian Stavåker, Filippo Donida, Francesco Casella, and Peter Fritzson. Parallel simulation of equation-based object-oriented models with quantized state systems on a GPU. In *Proceedings of the 7th International Modelica Conference*, 2009.

[44] G. Migoni, F. Bergero, E. Kofman, and J. Fernández. Quantization-Based Simulation of Switched Mode Power Supplies. *Simulation: Transactions of the Society for Modeling and Simulation International*, 91(4):320–336, 2015.

[45] G. Migoni, M. Bortolotto, E. Kofman, and F. Cellier. Linearly Implicit Quantization-Based Integration Methods for Stiff Ordinary Differential Equations. *Simulation Modelling Practice and Theory*, 35:118–136, 2013.

[46] 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*, 88(4):387–407, 2012.

[47] Gustavo Migoni, Pablo Rullo, Ernesto Kofman, Federico Bergero, and Joaquín Fernández. Simulación eficiente de sistemas híbridos de generación de energía renovable. In *Proc.of RPIC 2015*, Córdoba, Argentina, 2015.

[48] Jayadev Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18(1):39–65, 1986.

[49] Jacques Monod. La technique de culture continue: theorie et applications. 1950.

[50] James Joseph Nutaro. *Parallel discrete event simulation with application to continuous systems*. PhD thesis, UNIVERSITY OF ARIZONA, 2003.

[51] C. Perfumo, E. Kofman, J. Braslavsky, and J.K. Ward. Load Management: Model-Based Control of Aggregate Power for Populations of Thermostatically Controlled Loads. *Energy Conversion and Management*, 55:36–48, 2012.

[52] Dana Petcu. Experiments with an ode solver on a multiprocessor system. *Computers & Mathematics with Applications*, 42(8):1189–1199, 2001.

[53] Linda R Petzold. A differential algebraic system solver, 1982.

[54] Gauthier Quesnel, Raphaël Duboz, Éric Ramat, and Mamadou K Traoré. Vle: a multimodeling and simulation environment. In *Proceedings of the 2007 summer computer simulation conference*, pages 367–374. Society for Computer Simulation International, 2007.

[55] Dhananjai M Rao. Efficient parallel simulation of spatially-explicit agent-based epidemiological models. *Journal of Parallel and Distributed Computing*, 93:102–119, 2016.

[56] D.M. Rao, N.V. Thondugulam, R. Radhakrishnan, and P.A. Wilsey. Unsynchronized parallel discrete event simulation. In *Simulation Conference Proceedings, 1998. Winter*, volume 2, pages 1563–1570, 1998.

[57] Valeriu Savcenco, Willem Hundsdorfer, and JG Verwer. A multirate time stepping strategy for stiff ordinary differential equations. *BIT Numerical Mathematics*, 47(1):137–155, 2007.

[58] SBML. Sbml software tools summary. Internal report, 2016. http://sbml.org/SBML_Software_Guide/SBML_Software_Summary.

[59] Gerard Olivar Tost and Olga Vasilieva. *Analysis, Modelling, Optimization, and Numerical Techniques: ICAMI, San Andres Island, Colombia, November 2013*, volume 121. Springer, 2015.

[60] John J Tyson. Modeling the cell division cycle: cdc2 and cyclin interactions. *Proceedings of the National Academy of Sciences*, 88(16):7328–7332, 1991.

[61] John J Tyson. Modeling the cell division cycle: cdc2 and cyclin interactions. *Proceedings of the National Academy of Sciences*, 88(16):7328–7332, 1991.

[62] Tim P Vogels and Larry F Abbott. Signal propagation and logic gating in networks of integrate-and-fire neurons. *The Journal of neuroscience*, 25(46):10786–10795, 2005.

[63] G Wanner and E Hairer. *Solving ordinary differential equations II*, volume 1. Springer-Verlag, Berlin, 1991.

[64] Bernard P Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. Academic press, 2000.